Ruth E. Goldenberg

dp
Digital
Press

OpenVMS Alpha Internals and Data Structures

Memory Management

OPERATING SYSTEMS

OpenVMS Alpha Internals and Data Structures

Memory Management

This Page Intentionally Left Blank

OpenVMS Alpha Internals and Data Structures

Memory Management

Ruth E. Goldenberg

Digital Press
An imprint of Elsevier Science
Amsterdam • Boston • London • New York • Oxford • Paris • San Diego
San Francisco • Singapore • Sydney • Tokyo

Digital Press™ is an imprint of Elsevier Science.

Library of Congress Cataloging-in-Publication Data

Goldenberg, Ruth E.
OpenVMS Alpha internals and data structures : memory managment / Ruth Goldenberg.
p. cm.
Includes bibliographical references and index.
ISBN 1-55558-159-5
1. Operating systems (Computers) 2. OpenVMS. 3. Memory management (Computer science) I. Title.

QA76.76.O63 G6374 2002
005.4'3--dc21

2002035114

Cover designer: Deborah Dutton and Joseph Sherman Design

Copy editor: Alice Cheyer

Compositor: Sarah Lemaire. This book is set in VAX DOCUMENT.

Index: Sarah Lemaire, Rosemary Simpson, John Mann.

A quotation from the following work appears as an epigraph in this book: Edgar A. Guest, "The Package of Seeds," *Collected Verse of Edgar A. Guest*, copyright 1934, originally published by Contemporary Books, reprinted by permission of The McGraw-Hill Companies.

Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for technical or editorial errors or omissions. Neither is any liability assumed for damages resulting from the information contained herein. The information in this document is provided "as is" without warranty of any kind and is subject to change without notice.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

In memory of Brian Porter, who spent many hours
reading memory management code.
I wish he could have reviewed this book.

And for my three-year-old nephew, Max Goldenberg,
whose memory management is excellent.

This Page Intentionally Left Blank

TABLE OF CONTENTS

EXAMPLES

FIGURES

TABLES

This Page Intentionally Left Blank

Preface

OpenVMS was developed in the 1970s to run on Digital Equipment Corporation's 32-bit VAX architecture. In the early 1990s Digital developed the 64-bit Alpha RISC architecture, and the OpenVMS code base was ported to Alpha. The book *OpenVMS AXP Internals and Data Structures: Version 1.5* (1994) describes key executive components of an early version of OpenVMS Alpha.

The present book describes the memory management subsystem of OpenVMS Alpha Version 7.3 and the system services that create, control, and delete virtual address space and sections. It emphasizes system data structures and their manipulation by paging and swapping routines. It also describes management of dynamic memory, such as nonpaged pool, and support for nonuniform memory access (NUMA) platforms.

This book updates the memory management part of the Version 1.5 volume, as *OpenVMS Alpha Internals: Scheduling and Process Control* (1997), updated the scheduling and process control part. Neither update is wholly independent of the Version 1.5 volume. Thus, to expand on topics mentioned here, chapters in this book refer to chapters in any of the three books. References to chapters within this book are by number, for example, Chapter 1. References to chapters in the preceding books are by title, for example, Chapter *Synchronization Techniques*.

For conceptual background on internals topics not covered in this book, consult the Version 1.5 book and the *Scheduling and Process Control* volume. Although details such as data structure layouts will likely have changed since previous versions, much of the conceptual foundation of OpenVMS is unchanged.

This book describes some features of the Alpha architecture, but presupposes knowledge of other features. The *Alpha Architecture Reference Manual* describes the architecture in detail.

Conventions

A number of conventions are used throughout the text and figures of this volume.

During the life of the VAX VMS operating system, the exact form of its name has changed several times: from VAX/VMS Version 1.0 to VAX VMS Version 5.0 to OpenVMS VAX Version 5.5. In describing the evolution of VMS algorithms and discussing the foundation of the OpenVMS Alpha operating system, this book refers to the OpenVMS VAX operating system by whichever name is appropriate for the version referenced.

The term *executive* refers to those parts of the operating system that are loaded into and that execute from system space. The executive includes the system base images, SYS$BASE_IMAGE.EXE and SYS$PUBLIC_VECTORS.EXE, and a number of other loadable executive images. Because there is no need to distinguish different types of executive image, this book generally shortens the term *loadable executive image* to *executive image*.

The terms *system* and *OpenVMS system* describe the entire OpenVMS software package, including privileged processes, utilities, and other support software as well as the executive itself. The OpenVMS system consists of many different components, each a different file. Components include the system base images, executive images, device drivers, command language interpreters, and utility programs.

The source modules from which these components are built and their listings are organized by facility. Each facility is a directory on a source or listing medium containing sources and command procedures to build one or more components. The facility [DRIVER], for example, contains sources for most of the device drivers. The facility [SYSBOOT] contains sources for the secondary bootstrap program, SYSBOOT. The facility [SYS] contains the sources that make up the base images and many executive images.

This book identifies a [SYS] facility source module only by its file name. It identifies a module from any other facility by facility directory name and file name. For example, [SYSGEN]SYSGEN refers to the source for the system generation utility (SYSGEN).

Closely related routines and modules often have names that differ by only the last few characters. For brevity, this volume refers to two such routines by enclosing the last few characters in square brackets. For example, the name MMG$DALCSTXSCAN[1] refers to routines MMG$DALCSTXSCAN and MMG$DALCSTXSCAN1.

Almost all source modules are built so as to produce object modules and listing files of the same file name as the source module. Thus, a reference in this book to a source module name identifies the file name of the listing file as well. In a case where the two names differ, the text explicitly identifies the name of the listing file. Appendix *Use of Listing and Map Files* discusses how to locate a module in the source listings.

This book identifies a macro from SYS$LIBRARY:LIB.MLB by only its name, for instance, $PHDDEF. The macro library of all other macros is specified.

Ported from VAX VMS, many OpenVMS Alpha executive routines have JSB-type entry points. That is, they were originally written to be entered with a VAX JSB instruction rather than a VAX CALLS or CALLG instruction. Typically this was done for performance reasons at a time when most of the executive was written in VAX MACRO and the rest in VAX BLISS. On an Alpha CPU, however, there is little difference in the code generated for a MACRO-32 JSB instruction and for a MACRO-32 CALLS instruction.

As part of adding support for high-level language device drivers and other system code, a standard call-type entry point has been added for each JSB-type entry point. An added call-type entry point has the string _STD in its name; for example, MMG$CREPAG_64 and MMG_STD$CREPAG_64 are the two entry points of a per-page routine called by system services such as $CRETVA and $CRETVA_64. New routines typically have only one entry point, a standard call-type entry point with a name that does not include the string _STD.

The unmodified terms *process control block* and *PCB* refer to the software data structure used by the executive. The data structure that contains a process's hardware context, the hardware privileged context block (HWPCB), is always called the HWPCB.

The term *inner access modes* means those access modes with more privilege. The term *outer access modes* means those with less privilege. Thus, the innermost access mode is kernel and the outermost mode is user.

The term *SYSGEN parameter* refers to a system cell that can be altered by a system manager to affect system operation. Traditionally, these parameters were altered through either the SYSGEN utility or SYSBOOT, the secondary bootstrap program. Although they can also now be altered through the SYSMAN utility or AUTOGEN command procedure, this volume continues to use the traditional term *SYSGEN parameter.* SYSGEN parameters include both dynamic parameters, which can be changed on the running system, and static parameters, whose changes do not take effect until the next system boot. These parameters are referred to by their parameter names rather than by the symbolic names of the global locations where their values are stored. Appendix *SYSGEN Parameters and Their Locations* relates parameter names to their corresponding global locations.

The terms *byte index, word index, longword index,* and *quadword index* derive from methods of VAX operand access that use context-indexed addressing modes. That is, the index value is multiplied by 1, 2, 4, or 8 (for bytes, words, longwords, or quadwords, respectively) as part of operand evaluation, to calculate the effective address of the operand. Although the Alpha architecture does not include these addressing modes, the concept of context indexing is relevant to various OpenVMS Alpha data structures and tables.

A term in small capital letters refers to the formal name of an argument to an OpenVMS system service, for example, the LOGNAM argument.

A bit field is sometimes described by its starting and ending bit numbers within angle brackets; for example, the interrupt priority level of the processor, in the processor status bits <12:8>, is contained in bits 8 through 12.

Unless otherwise noted, numbers in the text are decimal.

The term *KB* refers to a kilobyte, 1,024 bytes; the term *MB*, to a megabyte, 1,048,576 bytes; the term *GB* to a gigabyte, 1,024 MB; and the term *TB*, a terabyte, to 1,024 GB.

Three conventions are observed for lists:

- In lists like this one, where no order or hierarchy exists, list elements are indicated by leading round bullets. Sublists without hierarchy are indicated by dashes.
- Lists that indicate an ordered set of operations are numbered. Sublists that indicate an ordered set of operations are lettered.
- Numbered lists with the numbers enclosed in circles indicate a correspondence between the list elements and numbered items in a figure or example.

Several conventions are observed for figures. In all diagrams of memory, the lowest virtual address appears at the top of the page and addresses increase toward the bottom of the page. Thus, the direction of stack growth is depicted upward from the bottom of the page. In diagrams that display more detail, such as bytes within longwords, addresses increase from right to left. That is, the lowest addressed byte (or bit) in a longword is on the right-hand side of a figure and the most significant byte (or bit) is on the left-hand side.

Each field in a data structure layout is represented by a rectangle. In many figures, the rectangle contains the last part of the name of the field, excluding the structure name, data type designator, and leading underscore. A rectangle the full width of the diagram generally represents a longword regardless of its depth. A field smaller than a longword is represented in proportion to its size; for example, bytes and words are quarter- and half-width rectangles. A quadword is generally represented by a full-width rectangle with a short horizontal line segment midway down each side. In some figures, a rectangle the full width of the diagram represents a quadword. In these figures, bit position numbers above the top rectangle show numbers from 0 to 63 to indicate that the rectangle represents a quadword.

For example, Figure 2.5 shows the layout of the fixed part of the process header (PHD). The rectangle labeled SIZE represents the word PHD$W_SIZE; the rectangle labeled WSLIST, the longword PHD$L_WSLIST; and the rectangle labeled NEXT_REGION_ID, the quadword PHD$Q_NEXT_REGION_ID.

In almost all data structure figures, the data structure's full-width rectangles represent longwords aligned on longword boundaries. In a few data structures, a horizontal row of boxes represents fields whose sizes do not total a longword. Without this practice, most of the fields in this kind of structure would be split into two part-width rectangles in adjoining rows, because they are unaligned longwords.

Some data structures have alternative definitions for fields or areas within them. A field with multiple names is represented by a box combining the names separated by slash (/) characters. An area with multiple layouts is shown as a rectangle with a dashed line separating the alternative definitions. For example, in Figure 2.18, fields PFN$L_FLINK and PFN$L_SHRCNT are two names for the same field. Figure 2.18 also shows an example of alternative definitions for an area; the quadword at PFN$Q_BAK is also divided into the longword PFN$L_PHD and PFN$L_COLOR_BLINK.

A data structure field containing the address of another data structure in the same figure is represented by a bullet connected to an arrow pointing to the other structure. Where possible, the arrow points to the rightmost end of the field, that is, to bit 0. A field containing a value used as an index into that or another data structure is represented by an x connected to an arrow pointing to the indexed location.

Two conventions indicate elisions in a data structure layout. A specific amount of space is shown as a rectangle whose sides contain dots. Text within the rectangle indicates the amount of space it represents. Field PHD$Q_PAL_RSVD in Figure 2.5, for example, represents 48 bytes.

An indeterminate amount of space, often unnamed, representing omitted and undescribed fields, is indicated by a rectangle whose sides are intersected by short parallel horizontal lines. For example, Figure 2.1, which identifies only the PCB fields related to memory management, contains seven sets of omitted fields among the labeled fields.

In a typical figure that represents a code flow, such as Figure 4.1, time flows downward. Each different environment in which the code executes is represented by a column in the figure. The headings above the columns identify the environment characteristics, for example, "Kernel Thread Context" or "Kernel Mode".

A code flow figure represents only the events in the code most relevant to the current discussion. A description of code within a routine begins with the routine's name in bold-face type followed by text lines describing the routine's actions. When one routine calls another, the routine nesting is shown by indents. A lightning bolt represents an exception or interrupt. A diamond represents a branch test. An arrow indicates a transfer of control, typically from one routine to another.

Acknowledgments

In addition to acknowledging the work of the many contributors to *OpenVMS AXP Internals and Data Structures: Version 1.5,* the precursor of the present series, I would like to acknowledge the contributors to the present volume.

A number of people reviewed substantial portions of the book and made suggestions that improved its quality. I am particularly grateful to John Gillings, Mike Harvey, Andy Kühnel, Arthur Lampert, Karen Noel, and Guy Peleg. I'd also like to thank Nitin Karkhanis and Bobbi Ketelsen for their review.

I was delighted to work again with Alice Cheyer and Sarah Lemaire. Alice has been editing previous versions of this book since 1987 and shepherding them through production. Her punctilious attention to detail has greatly improved the book's consistency and readability. She challenges me to do my best work and keeps me honest.

Sarah Lemaire contributed immeasurably to the composition, figures, and index of this book. She remained unflappable and cheerful in the face of seemingly endless problems in every aspect of the production.

I am grateful to Bryan Jones, Group Engineering Manager for OpenVMS Engineering, for funding my work on this volume. I would also like to thank my managers for their support: Verell Boaen, Jim Janetos, and Joe Schuster. Carl Gallozzi and Margie Sherlock provided critical supplemental support.

Pam Chester and Theron Shreve guided the book through its publication stages at Digital Press.

Alan Rose aptly managed the prepress production; his wry sense of humor lightened the process. Lauralee Reinke capably dealt with font mysteries. Lazarus Guisso greatly improved the format of the book's pages.

This Page Intentionally Left Blank

Chapter 1

Fundamentals and Overview

One must have a good memory to be able to keep
the promises one makes.

Friedrich Wilhelm Nietzsche, *Human, All Too Human*

Virtual memory support for the OpenVMS Alpha operating system is based upon Alpha architectural features. It is designed to provide

- Maximum compatibility with OpenVMS VAX memory management
- Access to the larger Alpha address space
- Support for memory shared among multiple OpenVMS instances running on a Galaxy platform
- Support for efficient operation of platforms with nonuniform memory access

This chapter describes the Alpha memory management architecture and provides an overview of OpenVMS Alpha memory management. Sections 1.1–1.11 describe the fundamental concepts of memory management and the architectural mechanisms that underlie it. Sections 1.12–1.14 give an overview of OpenVMS management of virtual and physical memory.

1.1 Overview

Physical memory is the real memory supplied by the hardware. A virtual memory environment supports software that has memory requirements greater than the available physical memory. An individual process can require more physical memory than is available, or the total requirements of multiple processes can exceed available physical memory. A virtual memory system simulates real memory by transparently moving the contents of memory to and from block-addressable mass storage, usually disks.

An Alpha processor and the executive cooperate to support virtual memory. As used here, the term *processor* includes both the CPU hardware and its privileged architecture library (PALcode) address translation code.

In normal operation, the processor interprets all instruction and operand addresses as virtual addresses (addresses in virtual memory) and translates virtual addresses to physical addresses (addresses in physical memory) as it executes instructions.

This execution time translation capability enables the executive to execute an image in whatever physical memory is available. It also enables the executive and an Alpha processor in combination to restrict access to selected areas of memory, a capability known as memory protection.

The term *memory management* describes not only virtual memory support but also the ways in which the executive exploits this capability. Memory management is fundamentally concerned with the following issues:

- Movement of code and data between mass storage and physical memory as required to simulate a virtual memory larger than the physical one
- Support of memory areas in which individual processes can run without interference from others, areas in which system code can be shared but not modified by its users, and areas in which application code and data can be shared
- Arbitration among competing uses of physical memory to optimize system operation and allocate memory equitably

1.2 Physical Memory Configurations

On a uniprocessor system, all the physical memory is associated with one CPU. Some of the physical memory is permanently occupied by executive code and data, and the rest is available for system processes and user applications.

A symmetric multiprocessing (SMP) system is a hardware platform with two or more CPUs. Each can access all the physical memory and execute instructions independently of the others. As in a uniprocessor system, some of the physical memory is permanently occupied by a single copy of executive code and data, and the rest is available for other uses. Each CPU executes a different thread of execution, for example, an interrupt service routine or a kernel thread of some process. Executive code allocates physical memory, coordinates scheduling of the CPUs, and when necessary, synchronizes their operations.

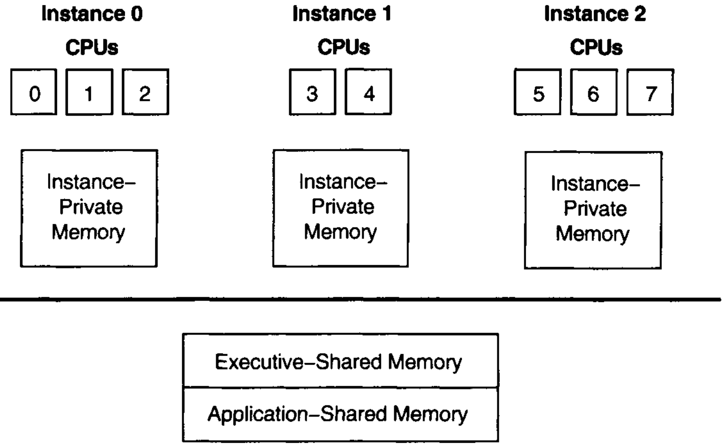
OpenVMS Alpha introduced support for Galaxy systems in Version 7.2. In a Galaxy system, multiple copies of OpenVMS execute within one multiprocessor computer. Each copy is called an instance. The system manager assigns each instance some of the computer's resources, in particular, memory, CPUs, and I/O peripherals. The system manager can reassign resources among the instances as needs change. With OpenVMS Alpha Version 7.3, only CPUs can be reassigned.

The term *soft partitioning* describes this type of software-controlled separation of computing resources. In contrast, *hard partitioning* is a physical separation of computing resources by hardware-enforced barriers.

An instance can be a uniprocessor or an SMP system. In each instance, some physical memory is occupied by executive code and data, and the rest is available for other uses. Some physical memory is shared among all the instances for Galaxywide system data structures. Applications running on multiple instances can create Galaxywide global sections in shared memory. Executive code synchronizes its own access to executive Galaxywide data and provides mechanisms for applications to synchronize their access to Galaxywide sections.

Figure 1.1 shows a simple representation of a Galaxy platform's CPUs and memory, which have been divided into three instances. Instance 0, for example, is a three-member SMP system. The executive occupies some of its private memory, and the rest is available for other uses. The three instances all share memory for executive data, and applications running on them can share global sections in shared memory.

Figure 1.1 Example Galaxy Configuration

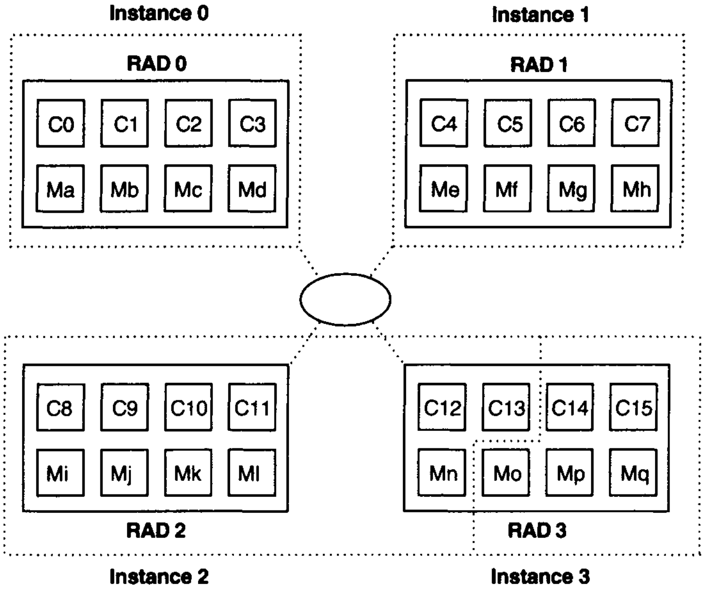


Some newer platforms are made up of hardware components called system building blocks (SBBs). Each SBB can have CPUs, memory, and I/O adapters. The components within an SBB have similar access characteristics. On Alphaserver GS160 and GS320 systems, for example, the SBB is called a quad building block (QBB). On these systems a CPU can access physical memory in its own QBB more quickly than other memory. This phenomenon is called nonuniform memory access (NUMA).

The system manager can configure the CPUs into a single SMP system running one OpenVMS instance or into a Galaxy system running multiple instances. In either case, if a single instance runs on multiple SBBs, the executive differentiates between memory local to a SBB and nonlocal memory to improve performance. For example, if a process is assigned to a particular SBB, the executive attempts to allocate its physical memory from memory local to that SBB. Section 1.7 describes another way in which OpenVMS supports NUMA platforms.

A software grouping of hardware components with similar access characteristics is called a resource affinity domain (RAD). For example, on an Alphaserver GS160 or GS320 system, a RAD corresponds to a QBB. Figure 1.2 shows the CPUs and memory of an example GS160 configuration. The system manager has configured it as a Galaxy system running four OpenVMS instances. Instances 0 and 1 each correspond to a QBB. Instance 2, however, has CPUs and memory from QBB 2 and QBB 3 and thus makes use of two RADs. Instance 3's CPUs and memory are all from QBB 3.

Figure 1.2 Example GS160 Configuration



1.3 Virtual Memory Concepts

Virtual memory is implemented so that each process has its own address space. Some of the address space is private to a process, and some of it is common to all processes. Executive code and data occupy the common virtual memory, which is called system space. Virtual memory can be larger than physical memory.

Support for virtual memory enables a process to execute an image that only partly resides in physical memory at any given time. Only the portion of virtual address space actually in use need occupy physical memory. This enables the execution of images larger than the available physical memory. It also makes it possible for parts of different processes' images and address spaces to be resident simultaneously even when they are in the same address range. Address references in an image built for a virtual memory system are independent of the physical memory in which the image actually executes.

Physical memory consists of byte-addressable storage locations eight bits (one byte) long. Physical address space is the set of all physical addresses that identify unique physical memory storage locations and I/O space locations. A physical address can be transmitted by the processor over the processor-memory interconnect, typically to a memory controller.

During normal operations, an instruction accesses memory using the virtual address of a byte. The processor translates the virtual address to a physical address using information provided by the operating system. The set of all possible virtual addresses is called virtual memory, or virtual address space.

Virtual address space and physical memory are divided into units called pages. Virtual and physical pages are the same size. Each page is a group of contiguous bytes that starts on an address boundary that is a multiple of the page size in bytes. The page is the unit of address translation. An entire virtual page is always mapped to an entire physical page; addresses within one virtual page translate to addresses within the same physical page. The virtual page is also the unit of memory protection. Each virtual page has protection attributes specifying which access modes can read and write that page.

Memory management is always enabled on an Alpha processor. The CPU hardware treats all instruction-generated addresses as virtual. Note, however, that kernel mode code can access a physical address directly by executing the instruction CALL_PAL STQP or CALL_PAL LDQP.

The CPU hardware attempts to translate virtual addresses to physical addresses using a hardware component called a translation buffer (TB). A TB is a cache of previously translated addresses. Because a TB can be accessed and searched faster than a page table, address translation is first attempted through a TB lookup. If the TB does not include this particular translation, PALcode must access a set of software data structures called page tables to load the translation into the TB.

Each process has its own set of page tables. Page tables provide a complete association of virtual to physical pages. A page table consists of page table entries (PTEs), each of which associates one page of virtual address space with its physical location, either in memory or on a mass storage medium.

A PTE contains a bit called the valid bit, which, when set, means that the virtual page currently occupies some page of physical memory. A PTE whose valid bit is set contains the number of the physical page occupied by the virtual page. The physical page number, called a page frame number, consists of all the bits of the physical page's address except for those that specify the byte within the page. When a reference is made to a virtual address whose PTE valid bit is set, the processor uses the page frame number to transform the virtual address into a physical address. This transformation is called virtual address translation.

When a reference is made to a virtual address whose PTE valid bit is clear, the processor cannot perform address translation and instead generates a translation-not-valid exception, more commonly known as a page fault. The page fault exception service routine, called the page fault handler, examines the PTE to determine the physical location of the invalid page.

- If the page is on disk, the page fault handler obtains an available page of physical memory, stores its page frame number in the PTE, and initiates I/O to read the virtual page into it from mass storage. When this occurs, the process is said to be faulting the page in.

  When the I/O completes successfully, the page fault handler sets the PTE valid bit and dismisses the exception. With the virtual page now valid, control returns to the instruction whose previous execution triggered the page fault, and it is reexecuted.
- If the page is a demand zero page, a data page that is initialized to all zeros, the page fault exception service routine allocates a page of memory, zeros it, stores its page frame number in the PTE, sets the PTE valid bit, and dismisses the exception.
- If the invalid page is still cached in physical memory, the page fault handler simply updates the PTE.

Reading a virtual page into memory or creating a demand zero page in response to an attempted access is called demand paging.

The set of a process's valid virtual pages is called its working set. The executive limits the number of pages of physical memory a process can use at once by setting a maximum size for its working set. When this limit has been reached and the process incurs a page fault, the page fault handler selects one of the process's virtual pages to remove from its working set. When this occurs, the process is said to be faulting the page out. Removing one virtual page from a process's working set to make room for another is called replacement paging.

The mass storage location from which a virtual page is read is called its backing store. A common example of backing store is a set of blocks in an image file. If the virtual page is guaranteed not to change (that is, it contains code or read-only data), the page fault handler need not write the page to mass storage when it is faulted out (thus saving the I/O) and can reread it from the image file as often as required. Thus, the backing store file remains the image file. If, however, the virtual page contains writable data, the page is faulted in once from the image and later faulted out to page file backing store, from which any subsequent faults will be satisfied.

The sections that follow discuss how these concepts are implemented by the Alpha architecture and OpenVMS Alpha.

1.4 Virtual and Physical Pages

The Alpha architecture supports a page size of 8 KB, 16 KB, 32 KB, or 64 KB. Each of the CPUs supported by OpenVMS Alpha Version 7.3, however, has a page size of 8 KB (8,192 bytes). For simplicity, therefore, this volume describes virtual addresses and address translation in terms of a page size of 8 KB.

In contrast, the VAX architecture supports a single page size, 512 bytes. To distinguish the two architectures' pages, the term *pagelet* identifies a VAX page, or a 512-byte unit of memory.

Each page is a group of 8 K contiguous bytes starting on an 8 KB address boundary. The first page starts at address 0, the second at address 2000_{16} (or 8192_{10}), the third at address 4000_{16} (or 16384_{10}), and so on.

Each physical page has an identifying number called a page frame number (PFN). A PFN is simply the portion of the physical address that specifies the physical page, namely, all but the low-order bits that specify the byte offset within the page. Typically, PFNs start at 0 and increase toward higher numbers. OpenVMS, however, does not require that PFNs start at 0 or that physical memory be contiguous.

Each virtual page has an identifying number called a virtual page number (VPN). A VPN is the portion of the virtual address that specifies the virtual page, namely, all but the low-order bits that specify the byte offset within the page.

Although all Alpha CPUs to date have a page size of 8 KB, OpenVMS memory management is adaptable to other page sizes. The console subsystem passes the page size to the executive during system initialization, and the executive defines various global cells accordingly. The following is a list of the most common of these cells and their contents for a page size of 8 KB:

- MMG$GL_PAGE_SIZE—Size of page in bytes (00002000_{16})
- MMG$GL_VPN_TO_VA (also known as MMG$GL_BWP_WIDTH) —Number of bits to shift left to derive the virtual address from a VPN ($0000000D_{16}$)
- MMG$GL_VA_TO_VPN—Number of bits to shift right to derive the VPN from a virtual address (expressed as a negative number to indicate a right shift) ($FFFFFFF3_{16}$)
- MMG$GL_BWP_MASK—Mask of set bits corresponding to the byte offset, or byte within page, field in a virtual address ($00001FFF_{16}$)

Executive routines use these and similar cells as parameters for page size dependent code. An application program can determine the page size by requesting the Get System Information ($GETSYI) system service to return information about item SYI$_PAGESIZE.

1.5 Virtual Addresses and Page Tables

In the Alpha architecture, a virtual address is represented as a 64-bit unsigned integer. An Alpha virtual address is divided into five parts whose sizes and starting bit positions can vary with page size. Figure 1.3 shows the parts of a virtual address on a system with a page size of 8 KB.

The processor translates a virtual address to a physical address using a three-level hierarchy of page tables. Each level field diagramed in Figure 1.3 indexes a different level of page table, and each is used in translating the virtual address. An Alpha page table of any level is one page long, and each PTE in it is eight bytes long. The value in a level field is thus multiplied by 8 to select a PTE.

Figure 1.3 Parts of an Alpha Virtual Address

63 43	42 33	32 23	22 13	12 0
Sign extension bits	Level 1	Level 2	Level 3	Byte within page

A level 3 page table (L3PT) contains 1,024 L3PTEs (8,192/8 = 1,024), each of which can map a page of code or data. A level 2 page table (L2PT) contains 1,024 L2PTEs, each of which can map an L3PT. A level 1 page table (L1PT) contains 1,024 L1PTEs, each of which can map an L2PT. Figure 1.4 shows the relations among the three levels of page table.

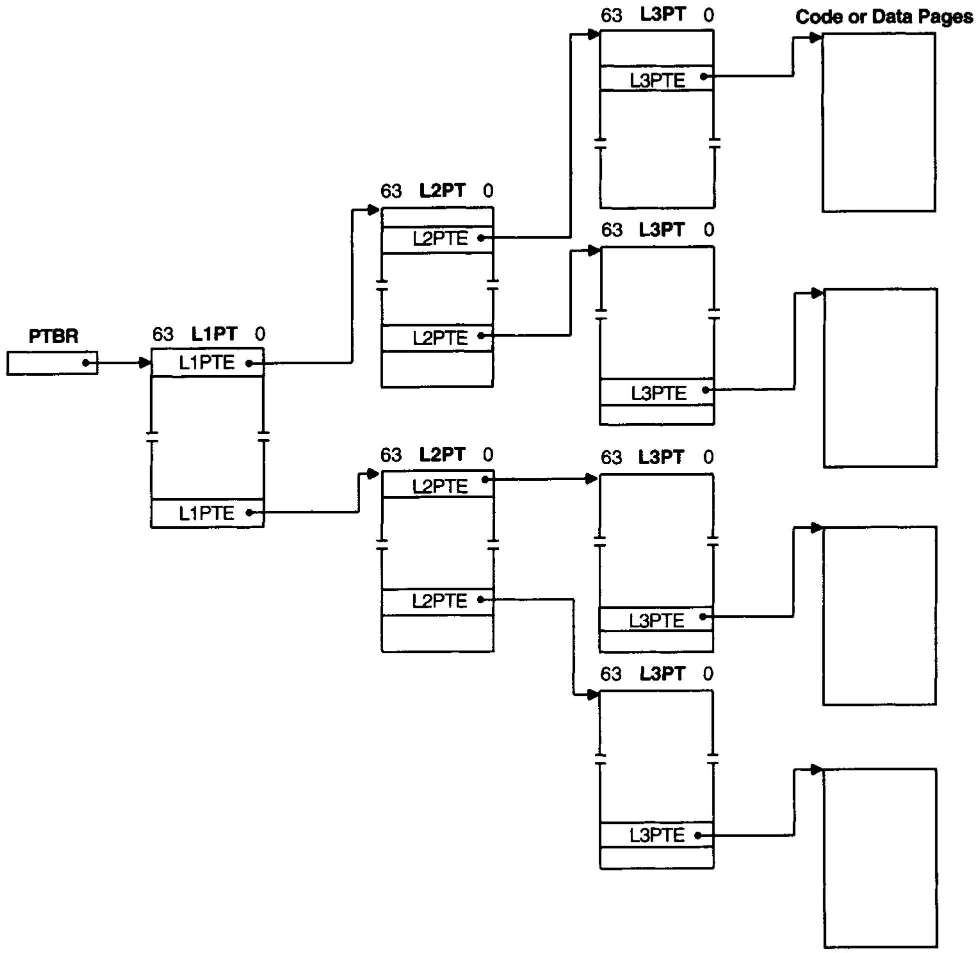
Each process has its own L1, L2, and L3 page tables; it shares some L2 and L3 page tables with other processes. The page table base (PTBR) processor register contains the PFN of the L1PT associated with the current process. The PTBR is part of the hardware privileged context and is swapped with process context.

The Alpha architecture supports a sparse virtual address space. Whether a particular virtual page is defined is independent of the state of its neighboring pages. Unlike the VAX architecture, the Alpha architecture has no page table length registers and does not require multiple physically contiguous page tables. Moreover, holes in the virtual address space need not be represented by page tables, and the architecture permits those L2PTs and L3PTs that exist to be pageable. These characteristics reduce the memory needed for page tables.

Figure 1.5 illustrates the basic steps of address translation for an example virtual address whose three level fields contain the values L1, L2, and L3. These steps are as follows:

1. The PTBR points to the L1PT.
2. The contents of the level 1 field in the virtual address, L1, index the L1PT to select an L1PTE, which contains the PFN of an L2PT.

Figure 1.4 Page Table Hierarchy

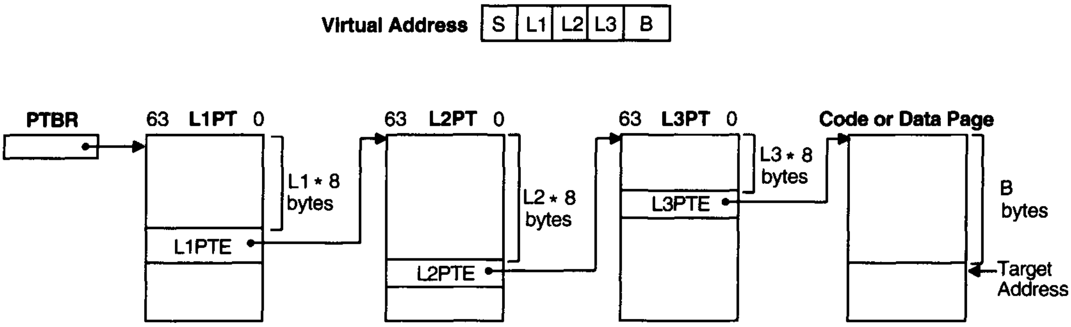


3. The contents of the level 2 field in the virtual address, L2, index the L2PT to select an L2PTE, which contains the PFN of an L3PT.
4. The contents of the level 3 field in the virtual address, L3, index the L3PT to select an L3PTE, which, if its valid bit is set, contains the PFN of the page containing the code or data at that virtual address.
5. If the L3PTE is valid, the contents of the byte within page field, B, are concatenated with the PFN to form the target physical address.

   Otherwise, a memory management exception occurs to notify the operating system that the translation cannot be completed.

Section 1.10 covers virtual address translation in more detail.

Figure 1.5 Example of Address Translation



Although instruction execution generates 64-bit virtual addresses, a particular processor implements a smaller virtual address whose size is a function of the processor's page size. A page table on a system with a page size of 8 KB contains 1,024 PTEs. On such a system, each level field in a virtual address identifies one of 1,024 PTEs and thus is ten bits wide (1,024 = 2^{10}). Byte offset, or byte within page, is 13 bits wide (8,192 = 2^{13}). Thus, a virtual address on a system with an 8 KB page with a three-level page table has only 43 bits of significance; the high 21 bits are simply sign extension bits.

Table 1.1 shows the sizes of the virtual address parts for the possible page sizes. Each number in the column Virtual Address Bits is calculated as the byte offset bits plus three times the number of level bits. Each number in the column Physical Address Bits is calculated as the byte offset bits plus 32, the size of a page frame number.

Table 1.1 Effects of Page Size on Alpha Virtual Addresses

Page Size	Byte Offset Bits	Level Bits	Maximum Size of Virtual Address Space	Virtual Address Bits	Physical Address Bits
8 KB	13	10	8 TB	43	45
16 KB	14	11	128 TB	47	46
32 KB	15	12	2,048 TB	51	47
64 KB	16	13	32,768 TB	55	48

On a system with an 8 KB page, a correct 64-bit virtual address therefore has identical values in bits <63:43>, and the values of these sign extension bits are the same as the value of bit 42, the high bit of the level fields (see Figure 1.3). Thus, correct virtual addresses must be either in the range 0 to 000003FF FFFFFFFF_{16} or the range FFFFFC00 00000000_{16} to FFFFFFFF FFFFFFFF_{16}.

Addresses in the range 00000400 00000000_{16} to FFFFFBFF $FFFFFFFF_{16}$ are illegal. This address range is known as the gap. The processor generates an access violation if an attempt is made to access an address in this range. The gap represents an inherently unreachable portion of the virtual address space, addresses in which bits <63:43> are not the same as bit 42. The gap exists because there are only 43 significant bits in a virtual address, given an 8 KB page and a three-level page table, yet there are 64 bits available for expressing a virtual address.

1.6 Virtual Address Space

Each process has its own virtual address space and page tables. Virtual address space is divided into the following parts:

- System space
- Process-private space
- Page table space

The executive is mapped into the same address range in each process's address space and is shared among all processes. That address range is called system space. The page tables that map system space are shared in each process's page table space. When the executive builds L1PTs for new processes, it initializes the L1PTEs that map system space to the same value in each L1PT. As a result, the L2PTs and thus the L3PTs that map system space are the same physical pages for every process.

The nonshared part of virtual address space is called process-private space. Each process can access only system space and its own process-private space; it is thereby protected against references to its process-private space from other processes.

All the kernel threads within a process share the same address space. One kernel thread at a time executes on a processor. (On a symmetric multiprocessing system, each processor can be executing a different kernel thread from the same process.) When a kernel thread is placed into execution, its process's page tables become the working page tables for that processor.

Process page tables are mapped into the same virtual address range, called page table space, in each process's address space. Some of this range maps process-private space, and some maps system space. The page tables that map process-private space are typically process-private, and the page tables that map system space are shared.

In early versions of OpenVMS Alpha, page table space was used only by PALcode. The executive mapped each process's page tables into system space as well and accessed them using system space addresses.

OpenVMS Alpha Version 7.0 and subsequent releases access page tables only through page table space. This change removes prior limits to the growth of process address space, enabling a process to use a much larger process-private virtual address space. The change, however, removes the double mapping by which the executive could access process-private page tables from outside the context of that process. The executive must now use another mechanism in those cases where such access is required.

Section 1.6.1 describes the organization of virtual address space in more detail, and Section 1.6.2 discusses page table space.

1.6.1 OpenVMS Alpha Virtual Address Space Layout

Figure 1.6, not to scale, shows OpenVMS Alpha virtual address space.

A key initial design goal of OpenVMS Alpha memory management was maximum compatibility with VAX VMS. OpenVMS Alpha virtual address space was initially based upon VAX VMS virtual address space. The VAX architecture defines a 32-bit virtual address space.

The low half of the VAX virtual address space (addresses between 0 and $7FFFFFFF_{16}$) is called process-private space. This space is further divided into two equal pieces called P0 space and P1 space. Each is 1 GB long. The P0 space range is from 0 to $3FFFFFFF_{16}$. P0 space starts at 0 and expands toward increasing addresses. The P1 space range is from 40000000_{16} to $7FFFFFFF_{16}$. P1 space starts at $7FFFFFFF_{16}$ and expands toward decreasing addresses.

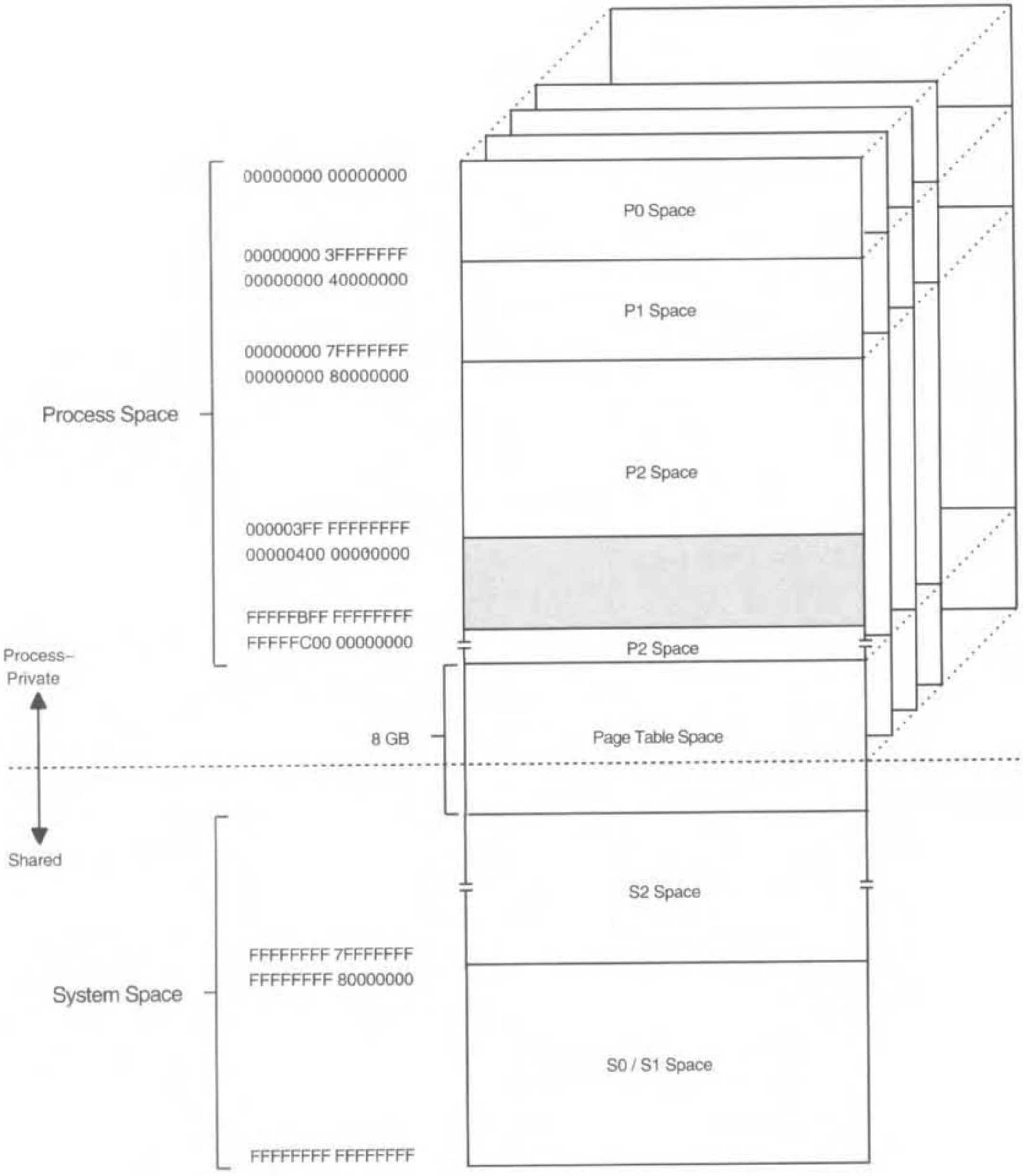
The upper half of the VAX virtual address space is called system space and is shared by all processes. The lower half of system space (the addresses between 80000000_{16} and $BFFFFFFF_{16}$) is called S0 space. S0 space begins at 80000000_{16} and expands toward increasing addresses. Although the original VAX architecture specified that the upper half of system space, S1 space, was undefined and reserved, the architecture has since been modified to permit S0 space to expand to $FFFFFFFF_{16}$. The expanded address range results in 2 GB of system space.

OpenVMS Alpha P0 and P1 virtual address space ranges are identical to their VAX counterparts. OpenVMS Alpha defines the combined S0/S1 space as FFFFFFFF 80000000_{16} to FFFFFFFF $FFFFFFFF_{16}$.

The Alpha 64-bit addresses for P0, P1, and S0/S1 ranges are sign-extended versions of the VAX 32-bit ones. Because the Alpha LDL instruction sign-extends in loading a longword from memory into a quadword register, P0, P1, and S0/S1 addresses can be stored as longwords in memory. (An Alpha instruction requires its address operands to be in registers.) These three virtual address ranges are 32-bit addressable spaces. Although they are part of the overall 64-bit Alpha address space, because of their addressability they are known as 32-bit space.

Defining system space at the high end of address space rather than at 00000000 80000000_{16} had the advantage of maximizing compatibility of the initial OpenVMS Alpha port. Moreover, it leaves the low end of address space free for process-private use, and it enables the L3PTEs that map system space to be mapped by a shared L2PT rather than by a process-private one. (On a system with an 8 KB page, an L2PT maps 8 GB. Thus a single L2PT would map P0, P1, and system space if system space were mapped at 00000000 80000000_{16}.)

Figure 1.6 Virtual Address Space



OpenVMS Alpha Version 7.0 added support for additional virtual address space ranges to exploit Alpha's 64-bit addressing capability and to enable a process to map a very large process-private address space:

- A shared virtual address range called S2 space is adjacent to S0/S1 space and extends toward lower addresses.
- A process-private virtual address range called P2 space is adjacent to P1 space and extends toward higher addresses.

The S2 and P2 ranges can only be accessed with 64-bit addresses. Although the entire 64-bit address space includes P0, P1, and S0/S1 space as well, the S2 and P2 ranges in particular are referred to as 64-bit space because of their distinctive 64-bit addressability.

The size of S2 space is determined at boot time rather than being fixed; its size depends on its contents, which include the PFN database and the global page table. The base address of S2 space can therefore vary from boot to boot. MMG$GQ_SYSTEM_VIRTUAL_BASE contains the S2 space base address.

Because the upper bound of P2 space is the lower bound of S2 space, constraining S2 space to be no larger than necessary leaves the maximum amount of virtual space for application use in P2 space. The maximum size of P2 space is a function of the size of S2 space. MMG$GQ_PROCESS_SPACE_LIMIT contains a value 1 higher than the highest possible P2 space address.

Page table space, described further in Section 1.6.2, consists of both process-private and shared memory. Its address range varies. Accessing it requires 64-bit addressing.

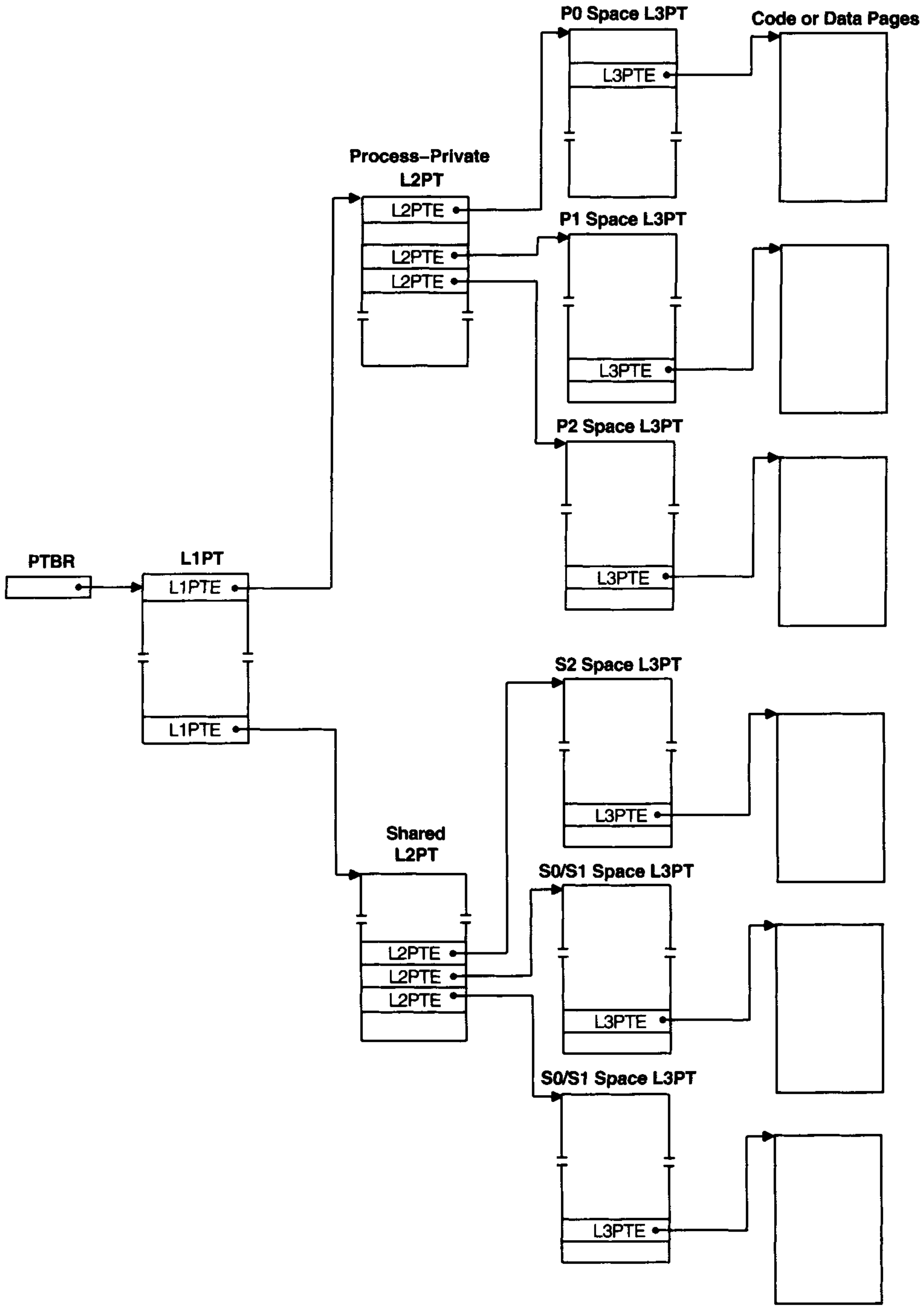
Figure 1.7 shows the relations among the levels of page table that map process-private and system space. For simplicity, it omits page table address space.

1.6.2 Page Table Space

In early versions of OpenVMS Alpha, a process's process-private page tables were part of a data structure called a process header (see Section 1.12.3) and double-mapped in 32-bit system space as well as in page table space. The maximum size of a process header is computed during system initialization, and enough system space is reserved for each resident process to have a maximum-sized process header. Mapping process-private page tables in this way does not scale to support many processes with a 64-bit address space. The maximum possible page table space for a single process is 8 GB long, clearly dwarfing the combined S0/S1 space in which all processes' process-private page tables had been double-mapped.

In OpenVMS Alpha Version 7.0, therefore, the process-private page tables were removed from system space and mapped only in page table space. The shared page tables were also mapped only in page table space. A consequence of removing this double mapping is that the page tables are no longer accessible via 32-bit system space addresses or from outside the owning process's context. Although most references to process-private page tables are made by the memory management subsystem from the context of that process, additional support was required to minimize the impact on kernel mode code referring to page tables from outside process context or using 32-bit addressing. Section 1.12.3 describes one aspect of this support, the system page table window.

Figure 1.7 Page Table Hierarchy Mapping Process-Private and System Space



Page table space is unique in not only being mapped by page tables but also consisting of them. The operating system creates page table space by selecting its virtual address range and initializing the corresponding L1PTE to map the L1PT itself. During system initialization the operating system selects a virtual address range for this space that meets the following constraints:

- It is a virtual address range not otherwise in use.
- It is as large as the number of bytes that are mapped by one L1PTE. On a system with an 8 KB page size this address space is 2 00000000_{16} bytes, or 8 GB, long.
- It begins on a boundary that is a multiple of its size in bytes.
- It is at an address range higher than process-private space and lower than system space so that there is a single dividing line, within page table space, between the process-private and shared address spaces.

The operating system initializes the L1PTE corresponding to the base of this virtual address range with the PFN of the L1PT, valid bit set, kernel mode read and write enabled, and all other bits zero. The operating system then records the base of the corresponding virtual address in MMG$GQ_PT_BASE and also loads it into the virtual page table base (VPTB) processor register. The VPTB contains the virtual (not physical) address of the base of page table space.

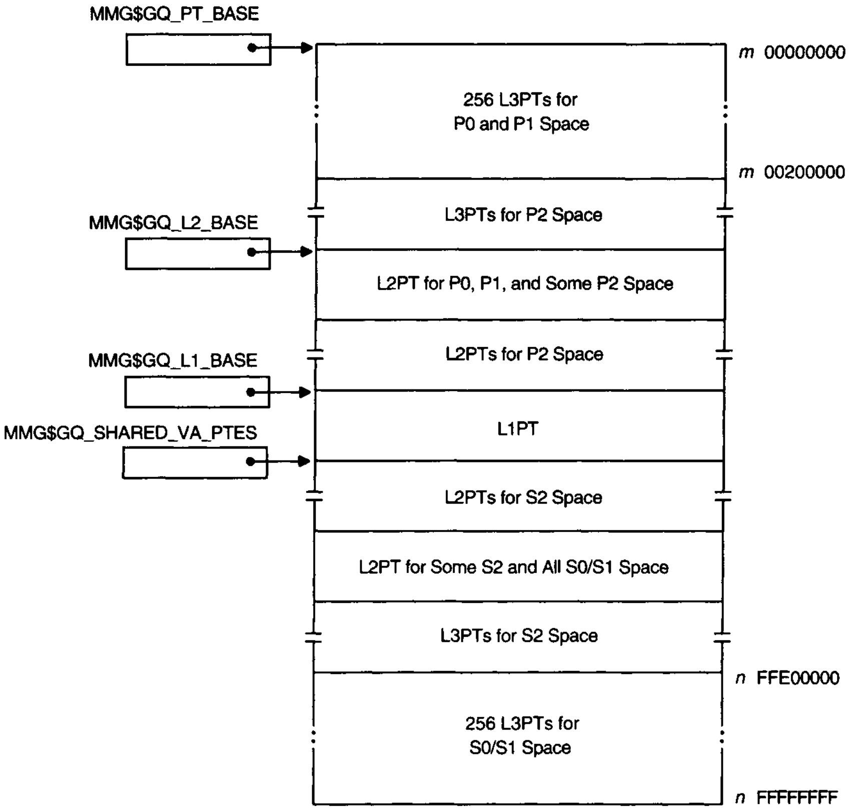
Whenever a new process is created, the operating system allocates and initializes an L1PT for it. The system then initializes the appropriate L1PTE in the process's L1PT to map the L1PT and thereby the page table virtual address space. Because page table virtual address space is mapped by the same L1PTE in each process's L1PT, page table space occupies the same address range in each process's virtual address space.

Figure 1.8 shows the organization of page table space on a system with an 8 KB page. In the figure, *m* is a function of the L1PTE selected for self-mapping and represents the high-order part of the resulting virtual address; *n* is equivalent to $m + 1$. MMG$GQ_SHARED_VA_PTES contains the address of the dividing line between process-private and system space.

The self-mapped L1PT creates a page table hierarchy that is shifted one level up from its use in a normal page table. Because the self-mapped L1PT is pointed to by an L1PTE, and an L1PTE maps an L2PT, the self-mapped L1PT becomes an L2PT as well. Because the self-mapped L1PT is pointed to by an L2PTE, it is also an L3PT. Mapped by an L2PTE in the shifted hierarchy, each normal L2PT becomes an L3PT, and the normal L3PTs become the data pages, the target of virtual address translation. The shifted hierarchy maps page table space.

Page table space contains a linear array of all possible L3PTEs for a given process address space, including process-private, page table, and system address space. In this address space, any L3PTE can be located by using the concatenated level 1, level 2, and level 3 parts of a virtual address as a single linear index. The linear layout and single index eliminate the necessity for the L1PTE and L2PTE accesses in many cases.

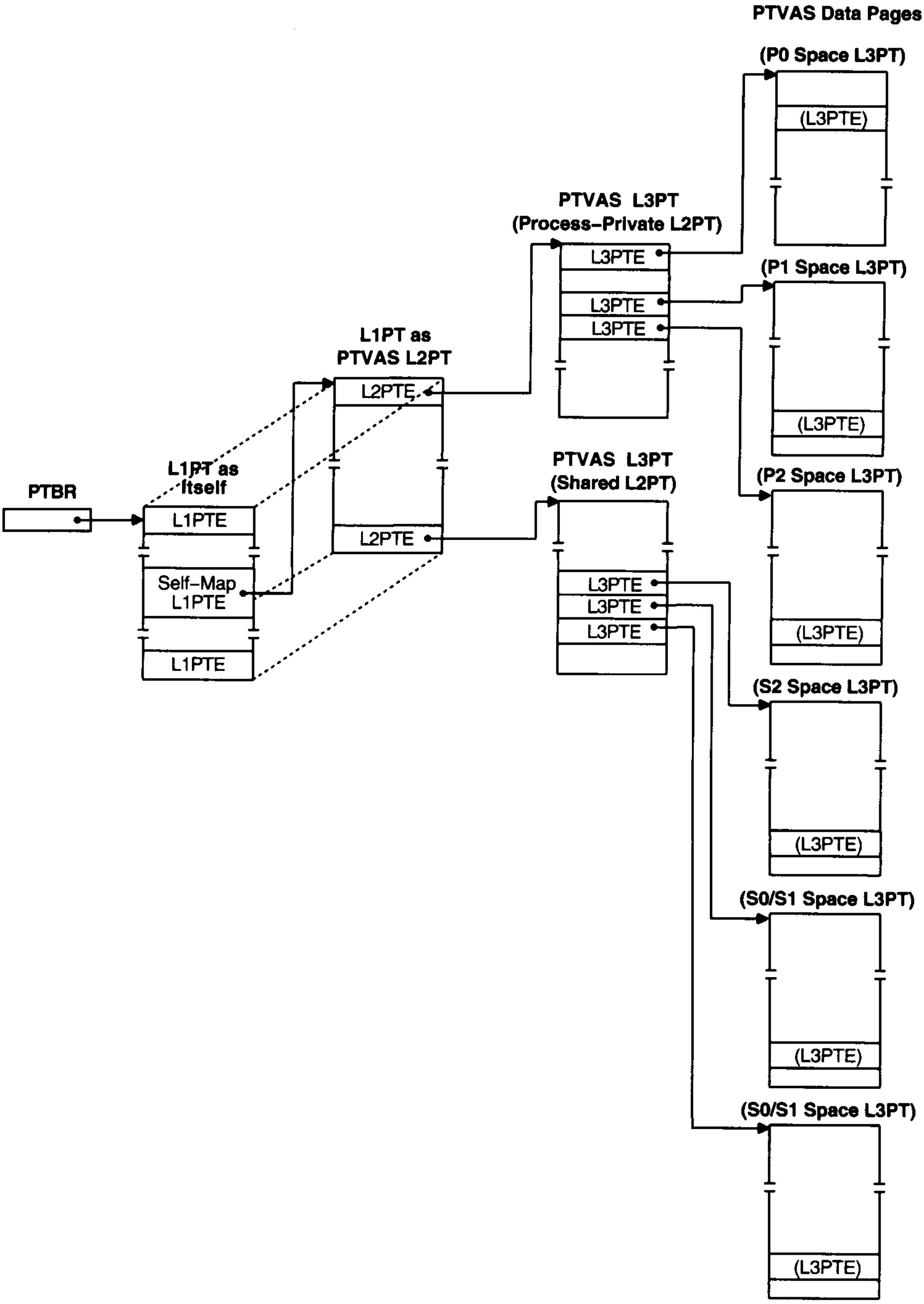
Figure 1.8 Page Table Space



Seen as a linear array of L3PTEs, on a system with an 8 KB page size, page table space consists of 1,024 * 1,024 L3PTs. Figure 1.8 consists of L3PTs that map P0, P1, and P2 space, followed by L3PTs that map page table space, and finally by L3PTs that map S2 space and S0/S1 space. Seen as the standard page table hierarchy, page table space includes 1 L1PT, 1,023 L2PTs, and 1,023 * 1,024 L3PTs.

Figure 1.9 shows the transformation of the page table hierarchy of Figure 1.7 into page table space. Each page table in Figure 1.9 has two labels: one identifying its use in mapping page table virtual address space (PTVAS), and the other, in parentheses, identifying its normal use in mapping any other virtual address. The page table containing the L1PTE is not only the L1PT but is now also an L2PT (and an L3PT and a data page). The page table virtual address space L3PTs it maps are normally used as L2PTs. The page table space data pages are normally used as L3PTs.

Figure 1.9 Transforming the Page Table Hierarchy into Page Table Space



1.7 Virtual Addressing on a NUMA System

In the case of an SMP system with multiple RADs, OpenVMS optionally replicates system space code in physical memory local to each RAD. Having multiple copies improves the performance of executive code and images installed resident. This replication requires RAD-specific system PTEs: each RAD's system space PTEs must map its own copy of the code and data.

To facilitate this mapping, the Alpha architecture has been enhanced to support separate physical page table structures for system and process-private space. A new processor register, the virtual address boundary (VIRBND) register, contains the lowest virtual address in shared address space, the address of the self-mapping L1PTE. Another new processor register, the system page table base register (SYSPTBR), contains the PFN of the L1PT that maps system space addresses.

Each RAD has its own L1PT to map system space and its own L2PT and L3PTs to map the RAD-specific copy of system code. An L1PTE mapping system space data that is not RAD-specific is identical to its counterparts in all the other RADs' L1PTs. Similarly, an L2PTE that maps system space data is identical to its counterparts in all the other RADs. L2PTs and L3PTs that map system space data are not replicated. Chapter 2 contains further details.

On a NUMA system, the basic steps of address translation, described in Section 1.5, include a comparison of the virtual address to be translated and the contents of VIRBND:

- If the address is less than VIRBND, the address of the L1PT to be used is contained in the PTBR.
- If the address is equal to or greater than VIRBND, the address of the L1PT to be used is contained in the SYSPTBR.

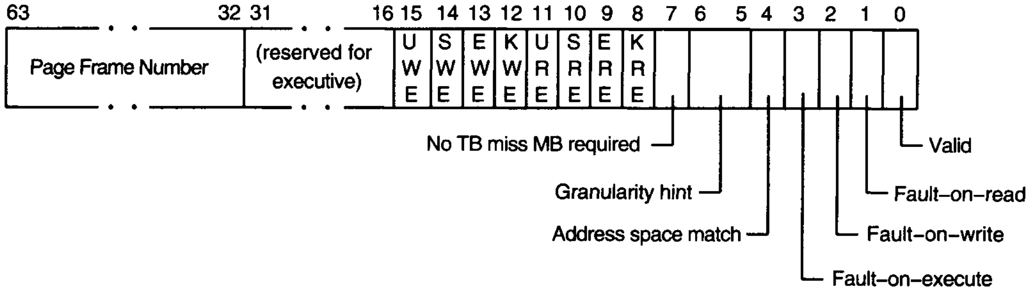
System space replication is controlled at boot time through SYSGEN parameter RAD_SUPPORT: when bit 0 (RIH$V_RAD_ENABLE) is set to enable general RAD support, bit 2 (RIH$V_SYSTEM_REPL) must also be set to enable system space replication.

1.8 PTE Contents

Figure 1.10 shows the architectural definition of a valid PTE, which contains the following fields:

- Bit 0 in the PTE is set to indicate that the virtual page is valid and that the processor can interpret bits <63:32> as a PFN.
- Each of the fault-on bits, when set, causes the hardware or PALcode to trigger an exception when the page is referenced in a particular way. Chapter 2 describes how OpenVMS uses the fault-on bits.

Figure 1.10 Valid Page Table Entry



- Bit 4, the address space match bit, is set in a PTE that maps a page shared at the same address range in all processes' address spaces. System space address ranges are mapped at the same place in all processes' address spaces and are shared. The executive therefore sets the address space match bit in the L2PTEs and L3PTEs that map system space. Section 1.11.1 explains the purpose of this bit.

- The Alpha architecture includes support for a feature called a granularity hint region. A granularity hint region is made up of a number of physically and virtually contiguous pages that are treated as a unit during address translation. A nonzero value in bits <6:5>, the granularity hint bits, identifies the page as belonging to a granularity hint region. Section 1.11.3 provides further information about granularity hint regions.

- Bit 7, if set, indicates that no TB miss memory barrier is required. This bit is architecturally reserved for hardware and is currently used on EV6 platforms when bit NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter. Under these circumstances, the operating system sets bit 7 in certain PTEs, for example, PTEs for process-private pages accessed from a single-threaded process and PTEs for pages permanently mapped into system space. Section 1.10.2 explains how setting this bit in a PTE can improve performance.

- Bits <15:8> of the PTE are the protection bits for the virtual page. The Alpha architecture provides two bits for each access mode: a read enable (*m*RE, where *m* symbolizes access mode) and a write enable (*m*WE). The first, if set, enables read references to the page from that mode. The other enables write references. If a write enable bit is set but the corresponding read enable is not, the operation of the processor is undefined. In other words, access to the page may be allowed or may cause an access violation, depending on the particular system.

  The OpenVMS executive uses only a subset of the possible combinations of protections that the architecture provides. For compatibility with OpenVMS VAX applications, it uses only those combinations that implement protections consistent with the VAX architecture. These combinations obey the following rules:

  — If a given access mode has write access to a particular page, then that access mode also has read access to that page.

  - — If a given access mode can read a particular page, then all more privileged access modes can read the same page.
  - — If a given access mode can write a particular page, then all more privileged access modes can write the same page.

- The architecture reserves bits <31:16> for use by the executive, which maintains a modify bit in one of them. (Because the architecturally defined Alpha PTE has no modify bit, the operating system is responsible for recording whether a page has been modified.)

  OpenVMS Alpha use of the reserved bits is explained further in Chapters 2 and 4.

- The maximum amount of space addressable on any processor, whether physical memory or I/O space, is limited by the 32-bit PFN field. Thus, the maximum architecturally defined physical address space is 2^{32} pages. The architectural maximum number of bytes of physical address space varies with page size, as shown in Table 1.1. On processors with 8 KB pages, the architectural maximum is 32 TB.

1.9 Translation Buffer

A translation buffer (TB) is a CPU hardware component that caches the results of recent successful virtual address translations of valid pages. Each TB entry has an associated valid bit; when the bit is set, the entry represents a valid translation that the CPU can use. Each TB entry caches one translation: a VPN and, minimally, its corresponding PFN, address space match, and protection bits. To simplify the hardware and software, only information from valid PTEs is cached. An attempted translation that results in a page fault is not cached; however, after the page is read in from backing store, the faulting instruction will be reexecuted and then the valid PTE will be cached.

Like a physical memory cache, a TB is a relatively small amount of memory that the CPU can access more quickly than physical memory.

In the course of fetching instructions and the operands they reference, the CPU accesses the TB to get mapping information for a particular page containing an instruction or an operand. For an instruction to be fetched and then executed, the TB must contain mapping information for the code and all data pages referenced by that instruction.

The CPU's access to the TB is purely associative and does not involve the page table hierarchy or page table space. Thus, for the CPU to access a code or data page, the TB must contain mapping information from L3PTEs. The contents of L1PTEs and L2PTEs are not directly relevant to the CPU's operation.

Consider, however, an instruction that refers to an L2PT or L3PT as data using its page table space address. For such an instruction to execute, the TB must contain mapping information for the page of page table space. Although this situation is more complex to describe, the concept is the same as accessing a code or data page not in page table space: the CPU needs mapping information in the TB from an L3PTE that maps the code or data page in order to execute the instruction. In terms of the standard page table hierarchy, that mapping information comes from an L1PTE or L2PTE. In page table space terms, the mapping information comes from one of the quadwords in the page table space, each of which is an L3PTE.

The size and organization of a TB are CPU-specific. Some CPUs have both an instruction stream TB (ITB) and a data stream TB (DTB). The ITB caches translations performed as the result of instruction fetches. The DTB caches translations performed as the result of loading or storing memory operands. The information in each type of TB entry can be different. For example, on some CPUs, the ITB does not include the fault-on bits. On such CPUs, therefore, no TB entry is made for a page whose fault-on-execute bit is set. Instead, it is always the PALcode's responsibility to generate this fault.

Because there are considerably fewer TB entries than virtual pages, a one-to-one mapping between virtual pages and TB entries is impossible. When all the TB entries are in use and another translation must be cached, one of the entries must be replaced with the new translation.

The architecture defines a processor register related to TB use called translation buffer check (TBCHK). The operating system can execute the instruction CALL_PAL MFPR, specifying the TBCHK register and a virtual address to determine whether the translation for a particular virtual page is cached in the TB. The presence of a TB entry for a page indicates the page has been referenced recently and may therefore not be a good candidate to remove from a process working set. OpenVMS makes this check as part of its page replacement algorithm.

Section 1.10 describes the TB's role in address translation, and Section 1.11, additional TB features.

1.10 Virtual Address Translation

The sections that follow build on the simplified description of address translation in Section 1.5.

1.10.1 Translation Using the Translation Buffer

Section 1.5 describes indexing the page table hierarchy with a virtual address's level fields to locate the L3PTE that maps that virtual address. In practice, performing physical memory references is too slow to do every address translation that way. Instead, an Alpha CPU translates addresses through TB lookups.

If a virtual address to be translated is represented in the TB (a hit), the CPU hardware tests the cached PTE information to determine whether the reference should be allowed:

1. The CPU tests the access mode and intended type of reference against the protection bits to determine whether the reference is legal. For this purpose, an instruction fetch is considered a read.

   If the protection on the page prohibits the access, the CPU generates an exception called an access violation.

2. If the protection bits allow the access, the CPU checks the intended reference against the fault-on bits. For this purpose, an instruction fetch is considered an attempted execution. If the corresponding fault-on bit is set, the CPU invalidates the TB entry and generates a fault-on-read, fault-on-write, or fault-on-execute exception, as appropriate. (The TB entry is invalidated on the presumption that the exception service routine will alter the PTE to clear the fault-on bit so that the instruction can reexecute without faulting. Having altered a valid PTE, the exception service routine would otherwise have to request the invalidation explicitly.)

3. If the access is allowed and no fault-on exception need be generated, the CPU forms the physical address by concatenating the PFN in the TB entry with the low-order bits of the virtual address, the byte offset within page.

If a virtual address to be translated is not present in the TB (a miss), the CPU dispatches to the PALcode TB miss routine, described in the next section.

1.10.2 TB Miss PALcode Routine

After a TB miss, the CPU hardware extracts the concatenated level 1, level 2, and level 3 fields of the address and multiplies them by 8, the size of a PTE, to form an offset into the array of L3PTEs in the page table virtual address space. It adds the offset to the contents of the VPTB to form the page table space virtual address of the L3PTE that maps the virtual address to be translated. It then dispatches to the TB miss PALcode routine.

In general, the routine tries to access the L3PTE using its page table space virtual address, but if that results in another TB miss, the page table hierarchy is used instead. The routine then creates a TB entry for the original virtual address so that the CPU can execute the instruction.

In detail, the TB miss PALcode routine proceeds as follows:

1. It tries to fetch the contents of the L3PTE using its page table space virtual address.

2. If the fetch causes another TB miss, the PALcode routine continues with step 8. Otherwise, it now has the L3PTE contents.

3. The routine tests the valid bit in the L3PTE contents, and if the bit is set, it continues with step 5.

4. If the valid bit is clear, the routine tests the intended reference to the target virtual address and the mode from which it is being made against the protection bits in the L3PTE. If the page is protected against the intended reference, the routine generates an access violation. This test enables the legality of an intended reference to an invalid page to be checked without having to fault the page into memory.

   If the protection permits the intended access, the routine generates a page fault exception and exits.

5. For an L3PTE whose valid bit is set, the routine loads a TB entry with information from the L3PTE. The CPU hardware will check the intended access against the protection when the instruction is reexecuted.

6. The routine must then execute a memory barrier (MB) instruction or take some other CPU-specific action to ensure ordering between its fetching the PTE and other software's fetching data from the page. Without this ordering, it is possible for stale data to be prefetched from the page under certain circumstances. Consider, for example, a page accessed by multiple threads on an SMP system that has just been read into memory by one CPU to satisfy a page fault. When the I/O completes on that CPU, an MB is done, after which the operating system sets the valid bit in the PTE. Without the ordering, a thread on another CPU could access stale data using the newly valid PTE. See Chapter *Synchronization Techniques* for information on read and write ordering.

   The TB miss PALcode routine on EV6 tests bit 7 in the PTE, prior to taking action to ensure ordering. If the bit is set, indicating that no ordering is necessary, the routine performs no ordering action. By default the bit is clear and the routine executes an MB.

   Use of bit 7 is not implemented in earlier CPUs. Instead their TB miss PALcode routines always perform a CPU-specific ordering action.

7. The routine exits.

8. If the fetch of the L3PTE caused a TB miss, a double TB miss PALcode routine is entered to load the TB so that the fetch in step 1 can be completed. In other words, the double TB miss routine must load the TB with mapping information for the page table space page containing the L3PTE that maps the original virtual address.

   In terms of the shifted page table hierarchy, the entry corresponds to the L3PTE that maps the page table page containing the target L3PTE (recall that in the shifted page table hierarchy, the target L3PTE is a quadword in a data page). In terms of the standard page table hierarchy, the entry to be loaded corresponds to the L2PT that maps the target L3PTE.

   The double TB miss routine takes the following steps:

   a. It indexes the page table hierarchy using the level fields in the target L3PTE address. The level 1 field is irrelevant because of the self-mapping in page table space: that field simply selects an L1PTE that contains the PFN of the

L1PT itself. The routine gets the PFN of the L1PT more quickly from either the PTBR register or the SYSPTBR register, depending on the faulted virtual address and the contents of the VIRBND register.

It extracts the level 2 field from the target L3PTE address, multiplies by 8, and adds the result to the PTBR contents to calculate the physical address of the L2PTE. (Recall that the L1PT and the L2PT are one and the same for page table space.) This L2PTE maps the L3PT page containing the target L3PTE.

b. The routine physically fetches the L2PTE and tests its valid bit. Physically fetching the L2PTE without checking for residence is safe because the L1PT must be resident.

   — If the L2PTE valid bit is clear, the routine also tests that the L2PTE permits kernel mode read access. If it does not, the routine generates an access violation. If it does, the routine generates a page fault exception.

   — If the L2PTE valid bit is set, the PFN in it identifies the page containing the L3PT. The routine calculates the physical address of the L3PTE that maps the page table space page containing the target L3PTE.

c. It physically fetches that L3PTE and makes the tests just described to determine whether the L3PTE is valid or whether a page fault or access violation exception should be generated.

d. If the L3PTE valid bit is set, the PFN in it identifies the page table page containing the target L3PTE. The routine loads a TB entry with information from the L3PTE, thereby loading the target L2PTE.

   The double TB miss routine exits and returns control to step 1, having loaded the TB with the information (the target L2PTE) necessary for the fetch in step 1 to succeed. That TB entry enables any of the 1,024 L3PTEs in the same page table space page to be fetched with one virtual memory reference.

When the PALcode TB miss routine exits, the CPU retries its translation of the address that incurred the TB miss. This time the TB contains an entry representing the virtual address. Using data from the entry, the CPU checks the intended reference against the fault-on bits, calculates the target physical address, and executes the instruction.

The *Alpha Architecture Reference Manual* contains further details of the architecturally defined address translation mechanism.

1.10.3 Address Translation Exceptions

Before dispatching to any memory management exception service routine (access violation, translation-not-valid, or fault-on), PALcode loads the following exception parameter information into registers:

- R4—The exact virtual address whose attempted reference caused the exception

- R5—The memory management flag quadword, whose possible values are
  - 00000000 00000000_{16} for a faulting data read
  - 00000000 00000001_{16} for a faulting instruction fetch
  - 80000000 00000000_{16} for a faulting data write

The saved program counter (PC) field in the exception stack frame (see Chapter *Interrupts, Exceptions, and Machine Checks*) contains the address of the instruction the fetching of which failed or the address of the instruction that incurred the fault.

1.11 Translation Buffer Features

The following sections provide additional information on the translation buffer.

1.11.1 Invalidating TB Entries

The contents of a TB entry that represents a valid translation can remain valid until they are superseded by a later translation of a different virtual address that maps to the same TB entry. The operating system is responsible, therefore, for flushing no longer correct entries from the TB. For example, it must invalidate a TB entry corresponding to a no longer valid PTE that maps a page being deleted or removed from a process's working set.

Because all processes have the same virtual address range, all TB entries are process-specific. In theory, the entire TB would have to be invalidated when process context is swapped. However, in practice, a TB entry that represents a physical page shared at the same virtual address in all processes need not be invalidated. The L3PTE mapping such a page has the address space match bit set to indicate it maps a virtual address whose translation is the same in any process context. When process context is swapped, the swap privileged context (SWPCTX) PALcode routine invalidates only entries whose address space match bits are clear. Moreover, as described in Section 1.11.2, the use of address space numbers further reduces the need for TB invalidations.

On a multiprocessor system, each CPU has its own TB. Although each CPU executes a different process, it is possible for a particular page to be represented in multiple processors' TBs, for example, a system space page shared by all processes. When the operating system changes the L3PTE of a valid page whose address space match bit is set, it is responsible for invalidating the page in all processors' TBs.

More precisely, each CPU executes a different kernel thread. On a multiprocessor system, multiple CPUs could be executing multiple kernel threads of the same process. Thus, even a process-private page accessed from different kernel threads could be in multiple processors' TBs. When the operating system changes the L3PTE of a valid page in a multithreaded process, it is responsible for invalidating the page in all processors' TBs.

The operating system can invalidate one or more TB entries by executing the CALL_PAL MTPR instruction with one of the following possible processor registers specified:

- TBIA—TB invalidate all
- TBISD—TB invalidate a single DTB entry
- TBISI—TB invalidate a single ITB entry
- TBIS—TB invalidate a single TB entry from both the ITB and DTB
- TBIAP—TB invalidate all process entries (those whose address space match bits are clear)

A CPU implementation is allowed to flush more entries than the register specifies.

Chapter 5 describes OpenVMS use of these processor registers.

OpenVMS uses the fault-on-execute bit to minimize TB invalidations on a system with both an ITB and a DTB. It sets the fault-on-execute bit in a page faulted as the result of anything but an instruction fetch. Any later attempt to execute an instruction from the page results in a fault-on-execute exception. The exception service routine clears the fault-on-execute bit and returns. If no instruction is executed from the page, the fault-on-execute bit remains set. If, when the page is removed from the working set, the fault-on-execute bit is still set, there cannot be a TB entry for the page in the ITB, and the executive needs to invalidate only the DTB. If, however, the fault-on-execute bit is clear when the page is removed from the working set, the executive must invalidate both the DTB and the ITB.

1.11.2 Address Space Numbers

The architecture includes support for a feature called address space number (ASN). Each TB entry is tagged with a number identifying the address space whose address translation the TB entry represents. (TB entries for pages whose address space match bits are set are not tagged in this way.) The processor register ASN is part of hardware privileged context on a CPU that supports this feature. The current ASN is an implicit input for all TB lookups, invalidation of single TB entries, and examination of the TBCHK register.

On a CPU that supports ASNs, the SWPCTX PALcode routine does not invalidate TB entries. Instead, the operating system ensures that unique ASNs are assigned to different kernel threads and invalidates all process-private TB entries if it must recycle ASNs. Use of ASNs can increase the usefulness of the TB as a cache by making it possible for entries to remain cached across multiple executions of a kernel thread on a particular CPU.

ASN is a CPU-specific designation; although each member of an SMP system uses the same numeric range of ASNs, the members do not use the same set of ASNs: a particular ASN on one CPU typically does not represent the same virtual address space as that ASN on another CPU. Because ASNs are CPU-specific and because multiple kernel threads of a process can execute concurrently on multiple CPUs, an ASN is associated with each hardware privileged context block (HWPCB) rather than with each process or address space. Chapter *Scheduling* contains further information on the executive's handling of ASNs.

1.11.3 Granularity Hint Regions

An Alpha translation buffer optionally supports a feature called a granularity hint, by means of which one TB entry can represent a group of physically and virtually contiguous pages with identical PTE characteristics (protection, validity, and fault-on bits), known as a granularity hint region. Use of granularity hints improves performance by increasing the number of apparent TB entries and thus reducing TB misses. The number of pages in a group specified by one TB entry is specified in bits <6:5> of the PTE of each page in the group. The bits are as follows:

- Bit value 00—1-page region
- Bit value 01—8-page region
- Bit value 10—64-page region
- Bit value 11—512-page region

If the TB holds an entry for any virtual page in the group, the CPU uses that entry to translate any virtual address in the entire group. The details of a particular TB, such as how many entries support granularity hints, vary with CPU type.

A granularity hint region must be on a naturally aligned boundary. For example, a granularity hint region of 64 pages must be on a physical and virtual 64-page boundary. For the operating system to make use of granularity hints, it must reserve blocks of physical memory and virtual address space early in system initialization to ensure that the address constraints can be met.

By default OpenVMS allocates one or more granularity hint regions in system space for each of the following purposes:

- Base and executive images' nonpaged code
- Base and executive images' nonpaged data
- S0/S1 space executive data (see Section 1.6.1 for a description of the various address spaces)
- Resident images' code
- Resident images' data

It creates additional granularity hint regions wherever possible, such as for the PFN database that describes memory.

The appropriate granularity hint bits as well as the address space match bit are set in each of the L3PTEs that map these regions.

Applications can also create granularity hint regions (see Chapter 3).

Chapters 2 and *The Modular Executive* provide more information on granularity hint regions.

1.12 Virtual Memory

This section summarizes OpenVMS Alpha use of each of the virtual address spaces and the data structures associated with those uses.

1.12.1 Use of Virtual Address Spaces

The various address spaces are used differently, created at different times, and have different protections.

Virtual address space is created (and recreated) at different times during system operation. System space is formed once and mapped in each process's address space. Process-private address space is created for each process and mapped only when that process is current.

S0/S1 space contains the executive, systemwide data structures, and any images installed permanently resident by the system manager. The highest virtual page in S0/S1 space is left invalid for error detection. For example, to differentiate 64-bit calling standard descriptors from 32-bit descriptors, the former have a longword of $\mathrm{FFFFFFFF}_{16}$, or -1, at offset 4, the address field location in a 32-bit descriptor. If a 64-bit descriptor is passed to a routine expecting a 32-bit descriptor, the routine's attempt to access FFFFFFFF $\mathrm{FFFFFFFF}_{16}$ will cause an access violation rather than incorrect results or corrupted data. Consult the *OpenVMS Calling Standard* for more information on calling standard descriptors.

Actually, the highest 64 KB of S0/S1 space are left invalid regardless of page size. The system page table window (see Section 1.12.3) occupies the part of S0/S1 space just below the invalid 64 KB. Leaving the high 64 KB of S0/S1 space invalid aligns the system page table window on a granularity hint boundary.

S2 space contains systemwide data structures accessed by 64-bit pointers, in particular, the PFN database, global page table, and lock management lock ID and resource hash tables.

SYSBOOT and other initialization routines load executive images into S0/S1 space, form the dynamic memory pools, and initialize the other parts of system space. Chapters *Bootstrap Processing* and *Operating System Initialization and Shutdown* describe the formation and initialization of system space in detail. Even after initialization is over, S0/S1 space can expand upward to its maximum size of 2 GB, and S2 space can expand downward to the address in MMG$GQ_SYSTEM_VIRTUAL_BASE. The architecture does not require that page tables be physically contiguous, and it permits a sparse mapping. Expansion of system space during normal operation is described in Chapter *The Modular Executive*. Individual PTEs can be altered to create, delete, or modify particular pages of system space.

P1 space contains the process stacks and permanent process control information maintained by the executive. It also contains address space used on the process's behalf by inner access mode components such as Record Management Services, the file system, and a command language interpreter.

When a process is created, its P1 space is created in several stages, as described in Chapters *Process Creation* and *Process Dynamics.* The global cell CTL$GL_CTLBASVA contains the address that is the boundary between the permanent and temporary portions of P1 space. The parts of P1 space below this address, namely, the user stack and a possible replacement image I/O section, are recreated by the image activator when it activates an executable image. P1 space can expand toward lower addresses during image execution as a result of system services requested explicitly by the image or implicitly on its behalf.

P0 space maps whatever images the user activates. By default an image is linked to a base address of 10000_{16}. 10000_{16}, or 64 KB, is the size of the largest potential Alpha page. This leaves VPN 0 invalid, regardless of page size, to help catch errors such as references through uninitialized pointers.

The image activator creates address space for the image and every shareable image that it references. As the image activator processes images, it creates process sections for the image sections it encounters. (A process section can also be created dynamically in response to a system service request.) A process section is a group of contiguous virtual pages with the same characteristics, such as writability and shareability.

During image execution the image activator creates additional P0 address space as necessary to activate images requested through the Run-Time Library procedure LIB$FIND_IMAGE_SYMBOL.

P0 and P1 space can be dynamically allocated at run time through various system services and Run-Time Library routines.

P2 space can contain user data accessed through 64-bit pointers. It can be dynamically allocated at run time through various system services and Run-Time Library routines. In addition, an image can contain demand zero data sections based in P2 space. Although code can theoretically execute in P2 space, OpenVMS does not currently support activating code sections there.

P0 space and the nonpermanent parts of P1 and P2 space are deleted at image rundown and recreated with each new image run.

Chapter *Image Activation and Exit* describes the image activator and the memory management system services it requests to map the sections of an image. Chapter 3 describes those system services as well as others an image can request to create, map, and delete address space.

Appendix *Size of System and P1 Virtual Address Spaces* describes the layout of system and P1 space in more detail.

Different areas of virtual address space have different protections. The protection codes on most system space data pages prohibit access from all but kernel and executive modes. S0/S1 space pages occupied by executive code allow any access mode to execute code in them. The protection on P2 space pages is specified indirectly to the system service that creates it. It usually allows read and write access from user mode. Certain parts of P1 space are protected against access from outer access modes. The protection on P0 space pages is specified indirectly to the system service that creates it. It usually allows read access from user mode and sometimes write access as well.

1.12.2 Virtual Memory Regions

From the first release of VAX/VMS, process-private virtual address space was divided into regions. Initially, regions corresponded one-for-one to the page tables defined by the VAX architecture:

- The program region, corresponding to P0 space
- The control region, corresponding to P1 space

A significant aspect to this division is that a section in virtual memory must be contained within a single region; a section cannot cross a region boundary. Address space within a region can be expanded contiguously to existing address space. In the program region, address space expands upward to higher addresses; in the control region, downward to lower addresses.

In OpenVMS Alpha Version 7.0, the concept of virtual region was extended and formalized. OpenVMS defines three process-permanent regions—program, control, and 64-bit program (P2) regions—and enables an application to define additional regions within them. A user-defined region must be within a single one of the process-permanent regions and cannot overlap another user-defined region. A region is identified by its ID and has attributes such as size, protection, and expansion direction.

Defining a region is a low-overhead operation that enables an application to reserve contiguous virtual address space for a given region's maximum needs without having to create all the address space for that region at once. When the application requests a system service to create or expand address space, it identifies the region. This gives the application better control of a virtual address region, with no conflicting allocations and deallocations by code such as run-time libraries running in the same process. With the traditional 32-bit system services, the application can only implicitly identify the program or control region; the newer 64-bit system services accept a region ID.

1.12.3 Virtual Address Space Data Structures

The major data structures that describe virtual address space are

- Process section table (PST)
- Region descriptor entries (RDEs)
- Page tables
- System page table window

The executive builds a data structure called a process header (PHD) to record memory management data about the process. The PHD contains the PST, which has one process section table entry (PSTE) to describe each process section created in that process's address space. A PSTE contains information necessary to resolve a page fault for a page in the section. The PTE for an invalid page that is part of a process section contains a pointer to the section's PSTE.

Chapter 2 contains more information on the PST.

An RDE describes a region. RDEs for the process-permanent regions are defined within the PHD. RDEs for user-defined regions are allocated dynamically. Chapter 2 contains more information on RDEs, and Chapter 3 describes the services that create and delete regions.

During system initialization, the shared page tables that map system space are created. OpenVMS double-maps the L3PTs that map S0/S1 space so that those L3PTs can be accessed using 32-bit pointers. These double-mapped page tables are referred to as the system page table (SPT) window. The SPT window provides compatibility for a device driver ported from OpenVMS VAX that allocates a system page table entry and accesses it using a 32-bit pointer.

Chapter 2 describes page tables in further detail.

1.13 Physical Memory

The OpenVMS Alpha system allocates some pages of physical memory permanently, for example, the pages that contain the SPT or the system base images. More typically, the system allocates a physical page of memory for a particular need, such as a virtual page in a process's address space, and deallocates the page when it is no longer needed.

This section summarizes the OpenVMS Alpha management of physical memory and its associated data structures.

1.13.1 Physical Memory Data Structures

OpenVMS Alpha Version 7.1 and later versions support noncontiguous physical memory. OpenVMS records what memory is present in several forms. A structure called the SYI memory map lists the starting PFN and size of each memory segment. A user can access this information through the $GETSYI system service, which returns it in a form called a physical memory map (PMM) structure.

A database called the PFN database records significant information about each physical page, such as whether it is currently in use and for what purpose.

Chapter 2 contains detailed descriptions of data structures related to physical memory.

The pages of physical memory allocated to a process are called its working set. A structure within the PHD called the working set list represents just those pages in a compact form. (In contrast, L3PTEs describing valid pages are scattered among those describing invalid pages in process-private L3PTs.) The working set list is briefly described in Chapter 2 and in more detail in Chapter 5.

Physical pages available for allocation are linked together into a list called the free page list. A page is allocated from the front of the list and generally deallocated to the back of the list. At allocation a physical page is associated with a virtual page: the PFN of the physical page is placed in the PTE corresponding to the virtual page, and the contents of the virtual page are read into the physical page from mass storage. The physical page retains its virtual contents until it is allocated for a new use. Even when the physical page is removed from a process's working set and the valid bit in the virtual page's PTE is cleared, the PTE still contains the physical page's PFN. Until the physical page is reused, it is possible to resolve a fault for the virtual page by removing the physical page from the free page list and setting the PTE valid bit again. A page fault resolved in this manner without the need for mass storage I/O is sometimes called a soft page fault.

When a physical page that has been modified is removed from a process's working set, the page is inserted at the back of another list, called the modified page list. The modified page list differs from the free page list in that a physical page on the modified page list cannot be reused until its contents are written to backing store, for example, a page file or the section file to which the virtual page belongs. Once the swapper has written the contents of the modified page to backing store, the swapper moves the page to the back of the free page list. (Acting in this capacity, the swapper is referred to as the modified page writer.)

While a physical page is on either the modified or free page list, a page fault for its virtual page can be resolved as a soft page fault without I/O. Thus these lists act as systemwide caches of recently used virtual pages.

When the system has no current process to run, the executive removes a page from the free page list that has no more ties to a virtual page, for example, a page whose contents have been deleted, and zeros it. Afterward, it inserts the page into a list of similar pages called the zeroed page list, from which demand zero pages and certain other types of virtual page are allocated. Zeroing an 8 KB or larger page when the system would otherwise be idle reduces the overhead incurred to allocate a page of all zeros. Chapter *Scheduling* provides further details.

1.13.2 Sharing Physical Memory

The page is the unit of sharing. Because system space addresses are mapped into each process's address space, the physical memory occupied by system pages is shared by all processes. However, on a NUMA system with replicated system space, only processes in the same RAD share replicated system space code pages; all processes in all RADs share system space data pages.

In addition, multiple processes' PTEs can map the same physical pages to enable the processes to share physical memory. For example, multiple processes using the same command language interpreter can share the read-only pages of the image. (However, each process needs a private copy of its writable data pages.) Sharing physical pages makes more efficient use of memory and reduces the number of page faults that require mass storage I/O.

Multiple processes share physical memory through a mechanism called a global section. All the pages of a global section have the same attributes. A global section resembles a process section and is dealt with similarly by the page fault handler.

Several data structures are associated with global sections:

- Global section table
- Global section descriptors
- Global page table

The global section table (GST) is analogous to a process section table and contains a global section table entry (GSTE) for each global section. Like a PSTE, a GSTE has information necessary to resolve a page fault for a page in the section.

A global section descriptor (GSD) identifies a particular global section by name and associates the name with a GSTE. A GSD contains information used to determine whether a particular process is allowed to access the global section.

The global page table (GPT) contains global PTEs (GPTEs) that serve as templates for the process PTEs that map global pages. Unlike other PTEs, GPTEs are not accessed in the course of translating virtual addresses; they are only accessed by memory management routines.

When multiple processes are mapped to a global section, all processes can potentially benefit from each other's page faults. When process A incurs a page fault for a global page not in its working set, if the page is not valid it is read in from its backing store. After the page fault completes, the GPTE is modified to show that the global page is valid. If process B then incurs a page fault for that page, the page fault handler copies the information from the GPTE to B's PTE and resolves the fault as a soft fault without the need for I/O.

OpenVMS Alpha also supports memory-resident global sections. Once made valid, all the pages of a memory-resident global section are permanently resident. Permanently resident pages are not listed in the process's working set, and they do not require backing store. Optionally, the page tables that map a global section can be shared as well, saving memory and backing store.

Chapter 2 contains more details on these data structures as well as those that describe memory-resident sections. Chapter 3 describes the system services that create, map, and delete global sections. Chapter 4 discusses global page faults.

In an SMP system, multiple CPUs share all of physical memory, executing one copy, or instance, of OpenVMS. Processes running on the different CPUs can share physical memory through global sections just as they can on a single CPU.

On a Galaxy platform, multiple instances of OpenVMS can execute cooperatively. On current platforms, all instances have access to the same physical memory. Software partitions the memory and assigns it to individual instances of the operating system. Each instance thus has private physical memory for its copy of the operating system and for the processes that run on it.

The system manager can apportion some of the physical memory to be shared by all the instances. Processes running on the multiple instances can map memory-resident Galaxywide shared sections and share access to application data. Like memory-resident sections, Galaxywide section pages are not listed in a process's working set and do not require backing store. Optionally, the page tables that map a Galaxywide section can be shared as well, saving memory and backing store.

1.13.3 Managing Physical Memory

Physical memory is used in the following ways:

- Permanently, by pages occupied by the system base images and the nonpageable sections of executive images (may be replicated on a NUMA system)
- Permanently, by systemwide nonpageable data structures (for example, system context stacks, the PFN database, and nonpaged pool)
- Permanently, by images other than executive images that have been installed resident in system space, for example LIBOTS.EXE and LIBRTL.EXE
- Permanently, by pages within memory-resident global sections
- Permanently, by pages reserved for memory-resident global sections
- Dynamically, by pages on the free, modified, and zeroed page lists
- Dynamically, by pages in processes' working sets
- Dynamically, by pages in the system working set, namely, pageable sections of executive images and pageable system data (although much of the executive is nonpageable, some executive images have pageable image sections)

The executive apportions physical memory among these uses based on

- SYSGEN parameters that specify various minimum and maximum limits, such as the sizes of the free and modified page lists and the systemwide maximum process working set size
- Process quotas and limits that specify process-specific minimum and maximum working set sizes
- Statistics and measurements that describe the current environment, such as the size of the free page list and the rate at which a particular process has been page faulting recently

Hewlett-Packard Company recommends that memory-resident sections be registered in the Reserved Memory Registry so that AUTOGEN can tune the system to exclude permanently resident pages from its SYSGEN parameter calculations. Chapter 2 has more information on the Reserved Memory Registry.

1.14 Software Memory Management Mechanisms

This section provides an overview of the mechanisms by which physical and virtual memory are managed. OpenVMS Alpha memory management is based upon VAX VMS memory management. Recent OpenVMS Alpha releases, however, have added mechanisms designed to improve performance of applications using very large amounts of memory.

VAX VMS memory management mechanisms are best introduced from a historical perspective. Historically, the system has had two basic mechanisms to control its allocation of physical memory to processes: paging and swapping. Several auxiliary mechanisms, such as automatic working set limit adjustment and swapper trimming, supplement these fundamental ones.

1.14.1 Comparison of Paging and Swapping

The executive uses both paging and swapping to make efficient use of available physical memory. The page fault handler executes in the context of the process that incurs a page fault. It supports programs with virtual address spaces larger than physical memory. The swapper enables a system to support more active processes than can fit into physical memory at one time. The swapper's responsibilities are more global and systemwide than those of the page fault handler. Table 1.2 compares the page fault handler and the swapper in its role as working set swapper.

1.14.2 Original Design

An important goal of the initial release of the VAX/VMS operating system was to provide an environment for a variety of applications, including real-time, batch, and time-sharing, on a family of VAX processors with a wide range of performance and capacity. The memory management subsystem was designed to adjust to the changing demands of time-sharing loads and to meet the more predictable performance required by real-time processes.

The major problems common to virtual memory systems that concerned the original designers were the following:

- The negative effect that one heavily paging process has on others' performance
- The high cost of starting a process that has to fault all its pages into memory
- The high I/O load imposed by paging

VAX/VMS support of virtual memory was designed to address these problems. With some modifications, the original design remains intact in the OpenVMS Alpha operating system.

Table 1.2 Comparison of Paging and Swapping

Differences

Paging	Swapping
The page fault handler moves pages in and out of process working sets.	The swapper moves entire processes in and out of physical memory.
The page fault handler is an exception service routine that executes in the context of the process incurring the page fault.	The swapper is a separate process that is awakened from hibernation periodically and as needed by components that detect a need for swapper action.
The unit of paging is the page, although the page fault handler attempts to read more than one page (a page cluster) with a single disk read.	The unit of swapping is the process or, actually, the pages of the process currently in its working set.
Page read requests for process pages are queued to the driver according to the base priority of the process incurring the page fault.[1]	Swapper I/O requests are queued according to the value of the SYSGEN parameter SWP_PRIO. Modified page write requests are queued according to the SYSGEN parameter MPW_PRIO.[1]
Paging supports images with very large address spaces.	Swapping supports a large number of concurrently active processes.

Similarities

The page fault handler and swapper work from a common database. The most important structures used for both paging and swapping are the process page tables, the working set list, and the PFN database.

The page fault handler and swapper do conventional I/O, using a shortcut into the normal Queue I/O Request ($QIO) system service mechanism.

Both attempt to maximize the number of blocks read or written with a given I/O request. The page fault handler reads clusters of pages. The swapper attempts to inswap or outswap the entire working set in one or a few I/O requests. The modified page writer writes clusters of pages.

[1] This consideration has meaning for few mass storage device drivers. The priority at which an I/O request is queued to many drivers is largely irrelevant because they handle most requests immediately by queuing them to the device, which is likely to reorder them based on considerations such as disk head position.

The original VAX/VMS designers chose to have a process page against itself, for the most part, rather than against other processes. This minimizes the risk of page fault thrashing among processes and also enables more predictable performance for a real-time process.

A process is created with a working set quota that limits its maximum use of physical memory. The default and maximum sizes of each process's working set are specified at process creation. As a process executes and faults pages, they are read into memory from backing store and added to the process's working set. When the process's working set grows to its maximum size, a subsequent page fault must be a replacement page fault, requiring that a page first be removed from the working set. In this manner, the process pages against itself.

Unlike some virtual memory architectures, neither the VAX nor the Alpha architecture includes a reference bit in each PTE by means of which less recently referenced pages can be identified. Instead, the executive uses the order of working set list entries to determine length of residence. The working set list, which describes the pages in the process's working set, is a ring buffer with a pointer to the entry most recently added to the working set. In general, the page most likely to be removed from the working set is the one following the most recently added, that is, the oldest.

Although this working set replacement algorithm is simple to implement and has low CPU overhead, its selection of a page to be removed is not optimal and may cause more page faults. For those reasons, the original algorithm has been enhanced. Chapter 5 describes the current algorithm.

To minimize the performance impact of this algorithm, the executive caches pages removed from a working set so that they can be faulted back into it without the need for mass storage I/O; the executive inserts a page removed from a working set at the tail of the free page list or the modified page list, depending on whether the page had been modified. When a process needs a physical page of memory, for example, to fault a nonresident page, the executive allocates the physical page at the head of the free page list. Thus an unmodified page is cached for a length of time proportional to the size of the free page list and the frequency with which pages are allocated from it. When the modified page list grows beyond a certain size or the free page list shrinks below a certain size, the executive writes modified pages to their backing store, typically a page file, and then inserts them at the tail of the free page list. A modified page is thus cached while it is on both the modified and free page lists.

As previously noted, the working set replacement algorithm typically removes the oldest page in the working set rather than the least recently used. The page list caches, however, make it possible to fault the page back in as the newest page in the working set with little overhead. Because the working set list thereby tends to become somewhat ordered by use, the page list caches considerably improve the performance of the working set list replacement algorithm, bringing it close to that possible with a least-recently-used algorithm but with less overhead. (Note that a heavily paging process can affect others indirectly by causing the page lists to turn over more rapidly, thus reducing their effectiveness as caches for the other processes.)

The executive provides services by which a process can exercise some control over its working set list: it can lock and unlock selected pages into its working set and purge its working set of pages in a specified address range. At image exit, the executive deletes P0 space and the nonpermanent parts of P1 and P2 space, thereby removing these pages from the working set. Before a process executes a new image, the executive purges the working set of no longer needed pages, such as command language interpreter code and data.

The VAX/VMS system was designed to manage memory by both paging and swapping. Paging occurs in response to process page fault exceptions and results in moving virtual pages into and out of physical memory. Swapping, which occurs in response to events detected by the executive, results in moving whole working sets into and out of physical memory. Swapping all of a process's working set minimizes the time to reactivate the process and the number of I/O operations required to remove its pages from memory and to read them back in. Swapping makes it possible for more processes to coexist even when their working sets cannot all fit into memory at once.

Processes in certain long-lasting wait states are more likely to be outswapped than computable processes. When an outswapped process becomes computable, it is eventually inswapped. Chapter 6 describes the relation between process scheduling states and the swapper's selection of inswap and outswap candidates. A privileged process can prevent itself from being swapped.

To reduce the I/O overhead of paging, the executive reads and writes multiple pages at a time in units called clusters. A page fault cluster size is defined for each pageable entity, for example, an image section or a process page table. When a page is faulted, the executive tries to read a cluster's worth of pages. It writes modified pages in clusters also, to reduce I/O overhead. A SYSGEN parameter specifies the number of modified pages written to a page file at once. Within this larger cluster, the modified page writer groups related virtual pages so that they can be faulted back in as a cluster. Chapter 4 describes both types of clustering.

Simply deferring the writing of modified pages reduces I/O overhead to some extent: some pages are deleted before they are written; some pages are faulted in from the modified page list and modified again before they are written.

In VAX/VMS Version 1, the following characteristics of the memory management subsystem could be controlled by SYSGEN parameters and process authorization limits:

- The minimum sizes of the free and modified page lists
- The maximum size the modified page list could grow before the system began to write its pages to a page file
- The maximum number of concurrently resident processes
- For each process, a default and maximum working set size

As processes were created, used free pages, and faulted pages, the free page list would shrink and the modified page list would grow. If the free page list shrunk too low, the swapper would write modified pages and, if necessary, outswap a process. If the modified page list grew too large, the swapper would write modified pages. Occasionally, the swapper would have to write the entire modified page list, or flush it, in order to force specific pages out of memory. A process could alter its working set size from its default to its maximum through a system service to use that many more pages. Its working set size would be reset to its default at image exit.

1.14.3 Auxiliary Mechanisms

VAX/VMS Version 2 added a mechanism called automatic working set limit adjustment, by which a process's working set size was altered in response to its page fault rate. The working set of a heavily faulting process grew so as to reduce its page fault rate. The working set of a process that incurred very few page faults was shrunk. With expansion considered the more significant part of the mechanism, it was triggered at quantum end, based on the idea that a process that could not execute even for a quantum did not need its working set limit adjusted. Chapter 5 describes automatic working set limit adjustment.

VAX/VMS Version 2 also employed an enhancement to the VAX architecture that made it possible to test whether a page had been referenced recently enough so that its PTE was in the TB cache and thus not a candidate to be removed from the working set. The Alpha architecture also supports this capability through the TBCHK processor register.

In VAX/VMS Version 3, automatic working set limit adjustment was enhanced to permit a heavily faulting process to grow beyond its normal maximum working set if the free page list was sufficiently large. An alternative mechanism for reclaiming physical pages was added, called swapper trimming. The basic idea was that when the swapper process detected that the free page list had shrunk too low, it could reclaim memory from the working sets of processes expanded in times of plenty. If more memory was needed, it could either outswap a process or shrink a process working set as low as the SYSGEN parameter SWPOUTPGCNT. This added considerable flexibility to the original design; by altering this and several other parameters, a system manager could tune the system to favor swapping over paging, or vice versa.

VAX/VMS Version 4 refined swapper trimming, correcting a failure to reclaim memory from a low-priority compute-bound process whose working set had expanded when the system was lightly loaded. As a result of the pixscan mechanism, described in Chapter *Scheduling,* the refinement was not always effective.

In VAX VMS Version 5 there were several changes to the modified page writer, the most significant being that it no longer flushed the modified page list to force specific pages out of memory. Instead, it could be requested to search the list for selected pages and write them, leaving the rest of the pages as cache. Swapper trimming was further refined to reclaim memory more quickly from certain kinds of processes, in some cases by outswapping rather than trimming them. Chapters 5, 6, and *Scheduling* give further details of these proactive memory reclamation mechanisms.

Based on VAX VMS Version 5, the OpenVMS Alpha operating system uses these same mechanisms.

1.14.4 Very Large Memory Support

As the cost of physical memory has dropped, systems have been configured with more and more memory. OpenVMS Alpha has been extended to improve performance for applications such as database applications that can benefit by using very large amounts of memory (VLM).

In systems with limited physical memory, the mechanisms that limit per-process use ensure fair and equal access to a scarce resource. On a memory-rich system, however, intended to service VLM applications, such memory limits constrain performance of the VLM applications.

Extensions in support of VLM include

- Support for a 64-bit virtual address space
- Memory-resident global sections for large caches, to decrease time to access data
- Shared page tables for memory-resident global sections, to reduce application startup and shutdown time as well as memory needs
- Larger working set lists
- Reserved Memory Registry, to reserve and preallocate memory for memory-resident global sections so that they may occupy granularity hint regions

Chapters 2, 3, and 5 provide additional information.

1.15 Further Information

- Chapter 2, for a description of the data structures used by the memory management subsystem
- Chapter 3, for a description of the system services that an image requests to alter a process's virtual address space
- Chapter 4, for a discussion of the translation-not-valid fault (page fault) handler, the exception service routine that responds to page faults and brings virtual pages into memory
- Chapter 5, for a description of the working set list and the mechanisms that alter, shrink, and expand it
- Chapter 6, for an examination of the swapper process, a system process that manages physical memory by writing modified pages, shrinking process working sets, and swapping processes
- Chapter 7, for a description of the various pools from which virtual memory is allocated for transient needs, such as creation of dynamic data structures

- *Alpha Architecture Reference Manual*, Part II A, Chapter 3, for information on OpenVMS Alpha memory management support (http://www.digitalpressbooks.com/)
- *Digital Technical Journal* 8, no. 2 (1996), "OpenVMS for 64-Bit Addressable Virtual Memory" (http://www.research.compaq.com/wrl/DECarchives/DTJ/)
- *Digital Technical Journal* 9, no. 4 (1997), "OpenVMS Alpha 64-Bit Very Large Memory Design" (http://www.research.compaq.com/wrl/DECarchives/DTJ/)
- *OpenVMS Alpha Partitioning and Galaxy Guide* (http://www.openvms.compaq.com:8000/, order no. AA-REZQC-TE)
- *OpenVMS Alpha Guide to Upgrading Privileged-Code Applications* (http://www.openvms.compaq.com:8000/, order no. AA-QSBGD-TE)
- *OpenVMS Calling Standard* (http://www.openvms.compaq.com:8000/, order no. AA-QSBBD-TE)

Chapter 2

Memory Management Data Structures

. . . but there's one great advantage in it, that one's memory works both ways.

Lewis Carroll, *Through the Looking Glass*

This chapter describes data structures used by the memory management subsystem. These include the following:

- Structures that describe process virtual memory
- Page tables that help implement virtual memory
- Structures that describe granularity hint regions
- Structures that describe the state of physical memory
- Structures that enable processes to share memory through global pages and sections
- Structures that describe the state of page and swap files

The other memory management chapters discuss how the routines that compose the memory management subsystem use these structures.

2.1 Process Data Structures

Much memory management information about a process is maintained in its process header (PHD). The PHD includes a list of valid virtual process pages (the working set list), a description of the sections that make up the process-private address space (process section table), and a description of the permanent regions in P0, P1, and P2 space.

The process control block (PCB) is the key data structure that represents a process. The kernel thread block (KTB) is the key data structure that represents a kernel thread. A process is created with an initial kernel thread, that is, an execution context, and optionally can create additional kernel threads. The PCB and KTB contain some information related to memory management.

The PHD, PCB, and KTB are all allocated in system space. When a process is created, a PCB and KTB are allocated for it from nonpaged pool. A region of system space called the balance set slots contains space for the PHDs of the maximum number of resident processes. When a process is created, a slot is reserved for its PHD. If the process is outswapped, its PHD may be outswapped as well, but the PCB and KTB remain resident.

Region descriptor entries (RDEs) describe reserved regions of process-private virtual memory.

The PCB, KTB, RDEs, and PHD are described in the sections that follow.

A process's page tables describe its address space and the state of its virtual pages. Section 2.3 discusses page tables.

2.1.1 Process Control Block and Kernel Thread Block

A PCB is allocated for the life of the process and remains in nonpaged pool whether the process is resident or outswapped. When a process is outswapped, the PCB remains as the representation of the existence of that process and must contain all information that the swapper requires to inswap the process.

Figure 2.1 shows the PCB and KTB fields related to memory management. The KTB was designed to overlay the PCB. That is, each structure is sparse, with some fields designated KTB fields and others PCB fields. This overlay enables the initial thread's KTB to occupy the same memory as the PCB. In Figure 2.1 the KTB fields are shaded.

The longwords STS and STS2 are unusual in that they each represent fields in both the PCB and the KTB. Some STS and STS2 bits are processwide and are part of PCB$L_STS and PCB$L_STS2. Others are kernel-thread-specific and are part of KTB$L_STS and KTB$L_STS2. All the status bits related to memory management are part of the PCB status fields.

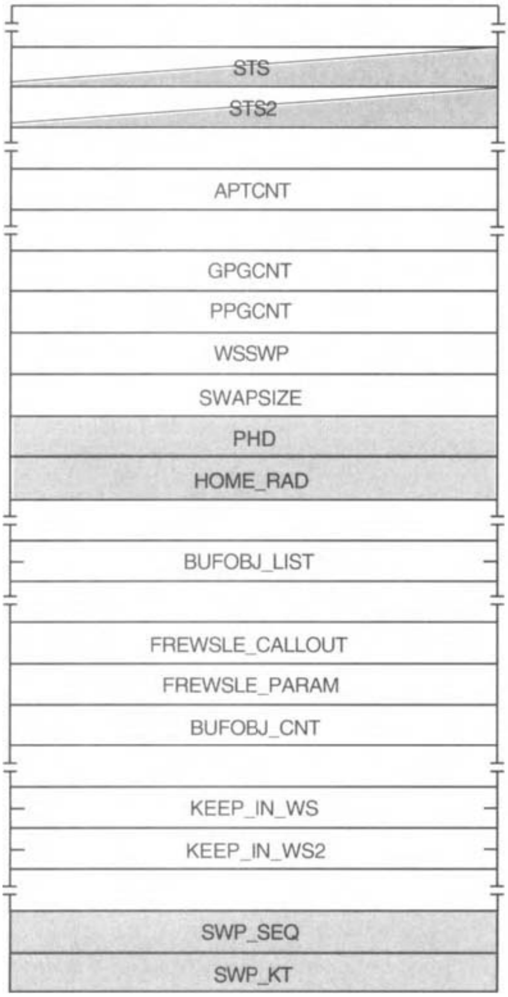
PCB$L_STS contains several status bits relevant to memory management:

- PCB$V_RES, when set, means that the process (that is, its PHD and its working set) is resident in memory.
- PCB$V_PSWAPM, when set, means that the process has disabled outswapping of itself.
- PCB$V_PHDRES, when set, means that the process's PHD is resident. (When a process is outswapped, its header may remain in memory.)
- PCB$V_DISAWS, when set, means that the process has disabled automatic working set limit adjustment.

PCB$L_STS2 contains two status bits related to memory management:

- PCB$V_PHDLOCK, when set, means that the process has one or more pages locked in memory through one of the Lock Pages in Memory ($LCKPAG or $LCKPAG_64) system services. The PHD of such a process may not be outswapped.

Figure 2.1 PCB and KTB (Shaded) Fields Related to Memory Management



- PCB$V_FREDLOCK, when set, means that the process has created more than 16 additional kernel threads and its PHD has been expanded by more than one FRED page to accommodate them. Because the additional floating-point register and execution data structure (FRED) pages are accessed physically by code that does not hold the MMG or SCHED spinlock, the PHD of such a process may not be outswapped. Chapter *Kernel Threads* describes FRED structures.

PCB$L_APTCNT only has meaning for an outswapped process; the swapper records in it the number of PHD and page table pages outswapped in the process's swap slot. Page table pages that map buffer objects (see Section 2.6) are not outswapped and thus not included in this count.

PCB$L_GPGCNT contains the number of global pages in the process's working set, and PCB$L_PPGCNT, the number of process-private pages. The sum of these two fields is the number of physically resident pages, the size of the process's working set. Note that this sum does not include pages of memory-resident global sections, PFN-mapped sections, and Galaxywide sections to which the process is mapped.

When a process is newly created, PCB$L_WSSWP is cleared to signal the swapper that the process's initial pages come from the shell (see Chapter *Process Creation*). The field has a different use later in the life of the process: when a process is outswapped, PCB$L_WSSWP contains its mass storage location. If the process has been outswapped in one extent, PCB$L_WSSWP contains a page file index (see Section 2.9.2) identifying the swap file and the starting virtual block number. The high bit of PCB$L_SWAPSIZE is set to indicate such a process; the low 31 bits of PCB$L_SWAPSIZE contain its outswapped size in pages. If the process is outswapped in more than one extent, PCB$L_WSSWP contains the address of a page and swap file mapping window block (PFLMAP), a data structure that lists the locations and sizes of the extents. Chapter 6 describes the PFLMAP and process swapping.

PCB$L_PHD, and KTB$L_PHD contain the address of the PHD, if PCB$V_PHDRES in PCB$L_STS is set.

On a nonuniform memory access (NUMA) platform with resource affinity domain (RAD) support enabled (RIH$V_RAD_ENABLE set in SYSGEN parameter RAD_SUPPORT), PCB$L_HOME_RAD and KTB$L_HOME_RAD record the number of the RAD associated with the process, the one to which most of its physical memory belongs. The default value, –1, means no RAD is associated with the process.

PCB$Q_BUFOBJ_LIST is the listhead for buffer object descriptors. Each buffer object descriptor describes a buffer object, a piece of address space used for certain kinds of I/O. Section 2.6 contains further information on buffer objects and their descriptors. PCB$L_BUFOBJ_CNT is the number of buffer object and PFN-locked pages left in memory after the process has been outswapped. Chapter 6 contains further information.

PCB$A_FREWSLE_CALLOUT, if nonzero, is the procedure value of a procedure to be called when a page is selected for removal from the process's working set. The procedure is called with arguments identifying the process and the page, and with the contents of PCB$L_FREWSLE_PARAM. Chapter 5 gives further information.

PCB$Q_KEEP_IN_WS and PCB$Q_KEEP_IN_WS2 delimit the virtual address range whose pages should not be removed from the working set list. Executive code uses these fields to keep particular pages in the working set temporarily. Chapter 5 contains further details.

KTB$L_SWP_SEQ and KTB$L_SWP_KT are used by the code that selects a process to outswap (see Chapter 6).

2.1.2 Region Descriptor Entries

A virtual region is a reserved range of process-private virtual address space. (The term *virtual region* is usually shortened to *region.*) Identified by its ID, a region has attributes such as size, protection, owner access mode, permanence, expansion direction, and whether address space within it can be mapped by shared page tables.

OpenVMS defines three process-permanent regions: the program region in P0 space, the control region in P1 space, and the program region in P2 space. The P0 and P1 space regions can be accessed with 32-bit addresses sign-extended to 64 bits. The P2 space region can only be accessed with 64-bit addresses. An application can create additional regions. It can also create address space within a region and later expand within that region. If an application has not explicitly deleted regions it created, they are typically deleted at image rundown. Permanent regions, however, which can be created by inner access mode code, survive image rundown.

The three process-permanent regions occupy all the P0, P1, and P2 address space unused by application-defined, or dynamic, regions. Thus they may shrink as dynamic regions are created and expand as dynamic regions are deleted.

Each region is described by an RDE. RDEs are process-private data structures; each process has its own set of RDEs. The RDEs for the process-permanent regions are created within the PHD, as shown in Figure 2.2. The RDE for the program region, for example, begins at offset PHD$Q_P0_RDE.

When a user requests the Create Virtual Region ($CREATE_REGION_64) system service to create a new virtual region, a dynamic RDE for it is allocated from the P1 space variable-length pool. As shown in Figure 2.2, RDE$PS_VA_LIST_FLINK and RDE$PS_VA_LIST_BLINK link a dynamic RDE into a list of all dynamic RDEs within the same part of process-private address space.

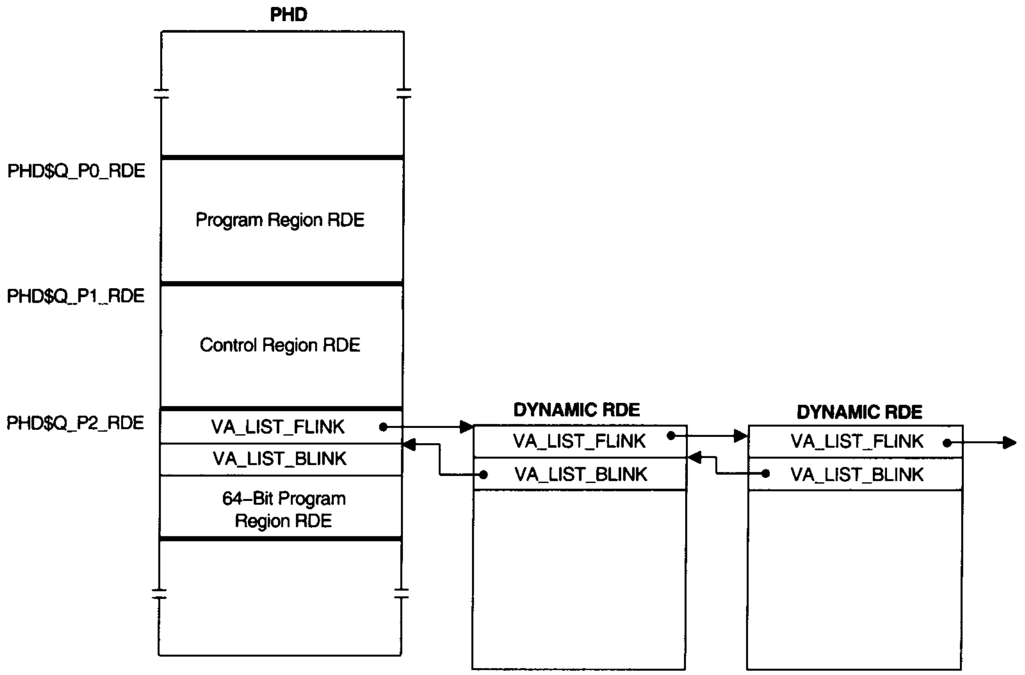
RDE$PS_VA_LIST_FLINK and RDE$PS_VA_LIST_BLINK in each of the three permanent RDEs form the listheads for these lists of dynamic RDEs. The P0 and P2 space lists are ordered in ascending order by starting virtual address. The P1 space list is ordered in descending order.

Figure 2.3 shows the layout of an RDE and the array of RDE listheads.

RDEW_SIZE, RDEB_TYPE, and RDE$B_SUBTYPE form the standard dynamic data structure header.

A dynamic RDE is linked through RDE$L_TABLE_LINK into an RDE list corresponding to the low-order four bits of its region ID. A 16-longword array of such listheads begins at CTL$A_REGION_TABLE. This array speeds the lookup of a dynamic RDE with a particular region ID. Figure 2.3 shows an example of a process that has created two dynamic regions.

Figure 2.2 Process-Permanent RDEs in the PHD



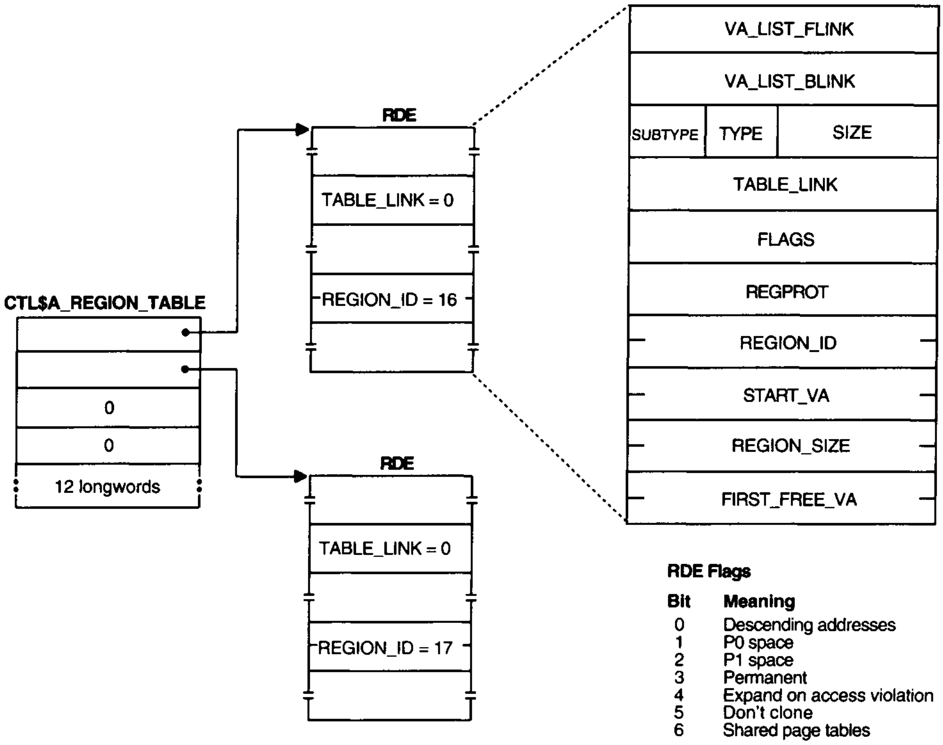
RDE$Q_REGION_ID contains the ID associated with the region. The process-permanent regions have IDs VAC_P0, VAC_P1, and VA$C_P2. PHD$Q_NEXT_REGION_ID contains the ID of the next dynamic region to be created. It is initialized to 16 at process creation. When a new dynamic region is assigned the ID stored in PHD$Q_NEXT_REGION_ID, the latter is incremented. The application identifies the region by its ID in subsequent memory management system service requests.

RDE$L_FLAGS describes various characteristics of the region, for example, whether it expands toward ascending or descending addresses, in which virtual address space it exists, whether its page tables are shared, and whether to expand it automatically after an access violation. A shared page table region can map only memory-resident or Galaxywide global sections.

RDE$R_REGPROT contains a structure that identifies the access mode that created the region and the access mode that owns it. Only the owner and more privileged modes can delete a region. The low-order four bits specify the access mode of the owner. The next four bits specify the least privileged mode allowed to create address space.

RDE$PQ_START_VA contains the lowest possible address in the region. RDE$PS_START_VA is the name for the low 32 bits of this field. RDE$Q_REGION_SIZE (alias RDE$L_REGION_SIZE) contains the size of the region. RDE$PQ_FIRST_FREE_VA (alias RDE$PS_FIRST_FREE_VA) contains the virtual address of the next available page in the region.

Figure 2.3 Layout of an RDE



RDE Flags

Bit	Meaning
0	Descending addresses
1	P0 space
2	P1 space
3	Permanent
4	Expand on access violation
5	Don't clone
6	Shared page tables

In previous versions PHD$L_FREP0VA contained the virtual address corresponding to the first available unmapped page in P0 space. PHD$L_FREP0VA is now an alias for the field RDE$PS_FIRST_FREE_VA in the program region RDE. Similarly, PHD$L_FREP1VA is now an alias for RDE$PS_FIRST_FREE_VA in the control region RDE.

RDEs are accessed only from process context. Accesses to them are synchronized with the inner mode semaphore (see Chapter *Kernel Threads*) and IPL 2 execution.

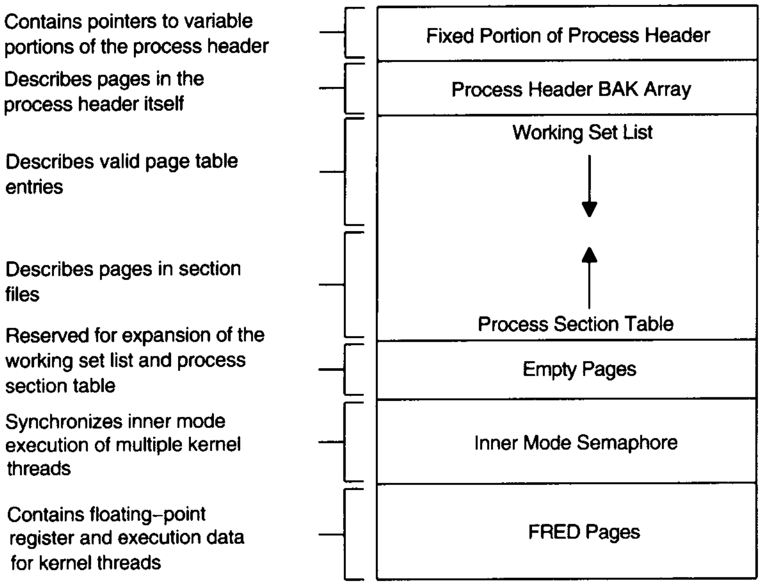
2.1.3 Process Header

Much important process-specific memory management information about a process is contained in its PHD. Shown in Figure 2.4, a PHD consists of a fixed part and several variable-length substructures:

- The working set list describes the subset of process-private pages that are currently valid. It also describes global pages that are valid in the process's page tables.
- The process section table (PST) contains entries that associate the process sections created in the process's address space with the corresponding sections in the files where the pages originate.

- The BAK array contains information about the pages of the PHD itself, which the swapper uses when it outswaps the PHD.
- In the case of a process with multiple kernel threads, an inner mode semaphore data structure synchronizes inner mode execution of the threads and an array containing FREDs.

Figure 2.4 Discrete Portions of the Process Header


Contains pointers to variable portions of the process header
Describes pages in the process header itself
Describes valid page table entries
Describes pages in section files
Reserved for expansion of the working set list and process section table
Synchronizes inner mode execution of multiple kernel threads
Contains floating-point register and execution data for kernel threads
Fixed Portion of Process Header
Process Header BAK Array
Working Set List
Process Section Table
Empty Pages
Inner Mode Semaphore
FRED Pages

The maximum sizes of these substructures are fixed by SYSGEN parameters, but their actual sizes vary in response to process needs. Pointers or indexes in the fixed portion of the PHD locate each substructure. Although the substructures vary in size, the balance set slots in which PHDs reside are of fixed size to simplify memory management code, as described in Section 2.8.3. The size of a balance set slot in pages is stored in global location SWP$GL_BSLOTSZ.

The dynamic growth area of the PHD must accommodate the growth of both the PST and the working set list. Expansion in either of these can result in moving the PST to higher addresses in the PHD. Section 2.1.3.3 describes PST/working set list expansion.

The PHD has several unusual characteristics that distinguish it from other data structures:

- The PHD is swappable.

  When a process is outswapped, its PHD may be outswapped as well. When later inswapped, the PHD is likely to be placed in a different balance set slot at a different system space address. Section 2.8.1 describes balance set slots.

- The PHD is referenced using addresses in two different address regions.

The PHD is located in system space so that the swapper and other memory management code can access it.

The PHD is also mapped in P1 space and accessed through global pointer CTL$GL_PHD. This P1 window to the PHD is at a fixed virtual address range and remains the same across outswaps and inswaps. The exact location of the window varies with system version; its size varies with several SYSGEN parameters. Chapter 6 contains more information on the double mapping of the PHD.

- The PHD is described by the process's working set list and is, in fact, locked into the working set because none of the PHD is pageable. PHD pages are the only pages with system virtual addresses that are part of a process working set.

The swappability of the PHD results in several different methods for synchronizing access to fields within it. Because a PHD can be inswapped to a different balance set slot than it last occupied, accesses to a PHD that use its system space address must be synchronized against swapper interference. Accesses from a kernel thread to its own PHD can be made with the SCHED spinlock held to block any rescheduling and possible swapping of the process. Holding the MMG spinlock is another way to block swapping.

Alternatively, executive code that runs in process context can access the PHD through the P1 window and thus avoid the need for blocking possible movement of the PHD to a different balance set slot.

The sections that follow describe the fixed part of the PHD and its memory management substructures.

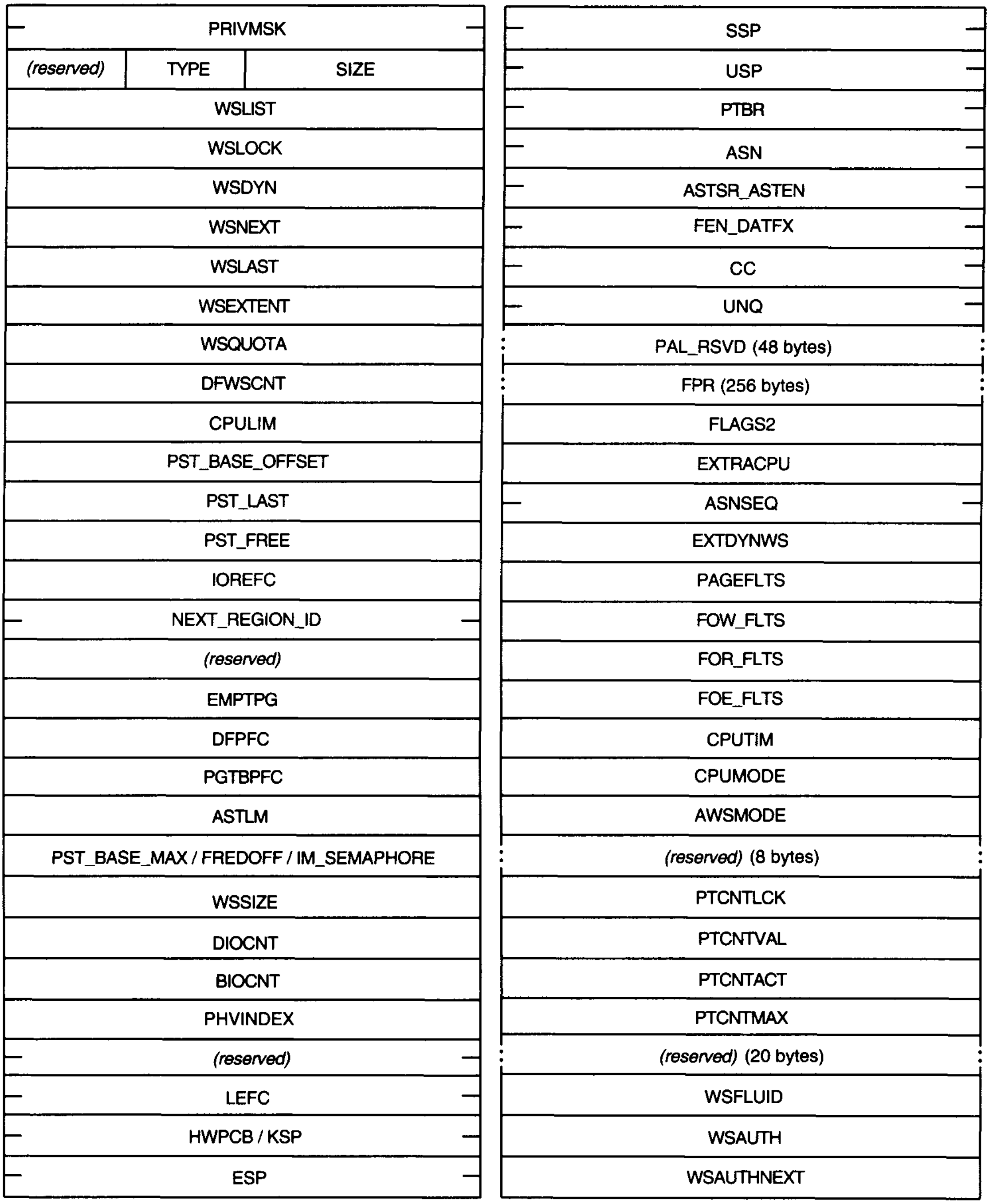
2.1.3.1 Fixed Part of the PHD

In addition to the pointers and indexes that locate variable-length parts of the PHD, the fixed area contains cells for process accounting information and several process quotas and limits. As described in Section 2.1.2, the fixed area of the PHD contains the RDEs for the three process-permanent regions.

Figure 2.5 shows the detailed layout of the fixed part of the PHD. Specific fields in the PHD are described, where appropriate, in this and the other memory management chapters.

The hardware privileged context block (HWPCB), the area in which the privileged register context of the initial kernel thread is saved, is in the fixed part of the PHD. This part of the PHD also contains space to save the contents of the initial kernel thread's floating-point registers when it is not current. In effect, it is a FRED page for the initial kernel thread.

Figure 2.5 Layout of Fixed Part of the Process Header (PHD)



PRIVMSK

(reserved) | TYPE | SIZE

WSLIST

WSLOCK

WSDYN

WSNEXT

WSLAST

WSEXTENT

WSQUOTA

DFWSCNT

CPULIM

PST_BASE_OFFSET

PST_LAST

PST_FREE

IOREFC

NEXT_REGION_ID

(reserved)

EMPTPG

DFPFC

PGTBPFC

ASTLM

PST_BASE_MAX / FREDOFF / IM_SEMAPHORE

WSSIZE

DIOCNT

BIOCNT

PHVINDEX

(reserved)

LEFC

HWPCB / KSP

ESP

(continued)

SSP

USP

PTBR

ASN

ASTSR_ASTEN

FEN_DATFX

CC

UNQ

PAL_RSVD (48 bytes)

FPR (256 bytes)

FLAGS2

EXTRACPU

ASNSEQ

EXTDYNWS

PAGEFLTS

FOW_FLTS

FOR_FLTS

FOE_FLTS

CPUTIM

CPUMODE

AWSMODE

(reserved) (8 bytes)

PTCNTLCK

PTCNTVAL

PTCNTACT

PTCNTMAX

(reserved) (20 bytes)

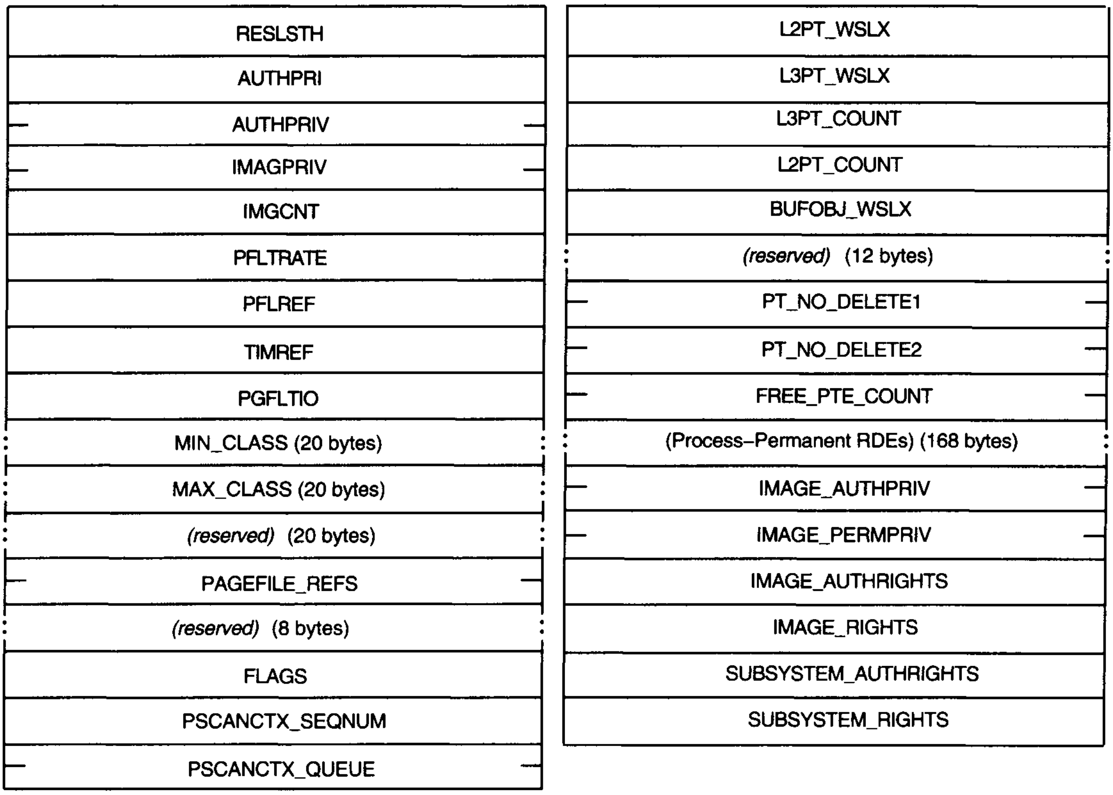
WSFLUID

WSAUTH

WSAUTHNEXT

(continued)

Figure 2.5 ***(continued)*** **Layout of Fixed Part of the Process Header (PHD)**



RESLSTH
AUTHPRI
AUTHPRIV
IMAGPRIV
IMGCNT
PFLTRATE
PFLREF
TIMREF
PGFLTIO
MIN_CLASS (20 bytes)
MAX_CLASS (20 bytes)
(reserved) (20 bytes)
PAGEFILE_REFS
(reserved) (8 bytes)
FLAGS
PSCANCTX_SEQNUM
PSCANCTX_QUEUE

L2PT_WSLX
L3PT_WSLX
L3PT_COUNT
L2PT_COUNT
BUFOBJ_WSLX
(reserved) (12 bytes)
PT_NO_DELETE1
PT_NO_DELETE2
FREE_PTE_COUNT
(Process-Permanent RDEs) (168 bytes)
IMAGE_AUTHPRIV
IMAGE_PERMPRIV
IMAGE_AUTHRIGHTS
IMAGE_RIGHTS
SUBSYSTEM_AUTHRIGHTS
SUBSYSTEM_RIGHTS

(continued)

2.1.3.2 Working Set List

Another memory management data structure located in the PHD is the working set list. The working set list describes the subset of process-private and global pages that are currently valid. Pages described in a process's working set list are P0, P1, P2, page table, or PHD pages. Its capacity to describe pages limits the number of physical pages the process can occupy with the exception of memory-resident and Galaxywide global section pages or PFN-mapped section pages, which are not included in the working set list.

The page fault handler and swapper use the working set list to determine which virtual page to discard (to mark invalid) when it is necessary to remove a physical page from the process. The swapper also uses the working set list to determine which virtual pages need to be written to the swap file when the process is outswapped.

Although the working set list currently remains in the PHD, it may move in a future release. For that reason, a process's working set list is generally located through the pointer CTL$GQ_WSL, which currently points to the working set list within the P1 space mapping of the PHD.

Chapter 5 describes the organization and use of the working set list and the layout of a working set list entry (WSLE).

2.1.3.3 Process Section Table

The process section table (PST) is also located in the PHD. It contains process section table entries (PSTEs).

A PSTE describes the association between a contiguous portion of virtual address space and a contiguous portion of a file. Both these portions are known as sections and consist of pages with identical characteristics, for example, protection, owner access mode, writability, and file location. Virtual address space is largely managed in units of sections.

When an image is activated (see Chapter *Image Activation and Exit*), the file containing the image is opened and a process section is created for each process-private image section. Although each image section is mapped separately, the image file is opened only once, and the image's sections page using the same assigned channel and window control block.

A process section is also created when

- A process opens a file and requests a system service that creates and maps a process-private section, for example, the Create and Map Private Disk File Section ($CRMPSC_FILE_64) system service or the Create and Map Section ($CRMPSC) system service, to map the file or some part of it into its address space
- A shareable image is activated that is not shared (that is, one that has not been installed with the /SHARED qualifier through the Install utility)
- A shared image is activated that has a copy-on-reference section

PSTEs enable the memory management subsystem to keep track of process pages in different sections, potentially in different files on different mass storage devices.

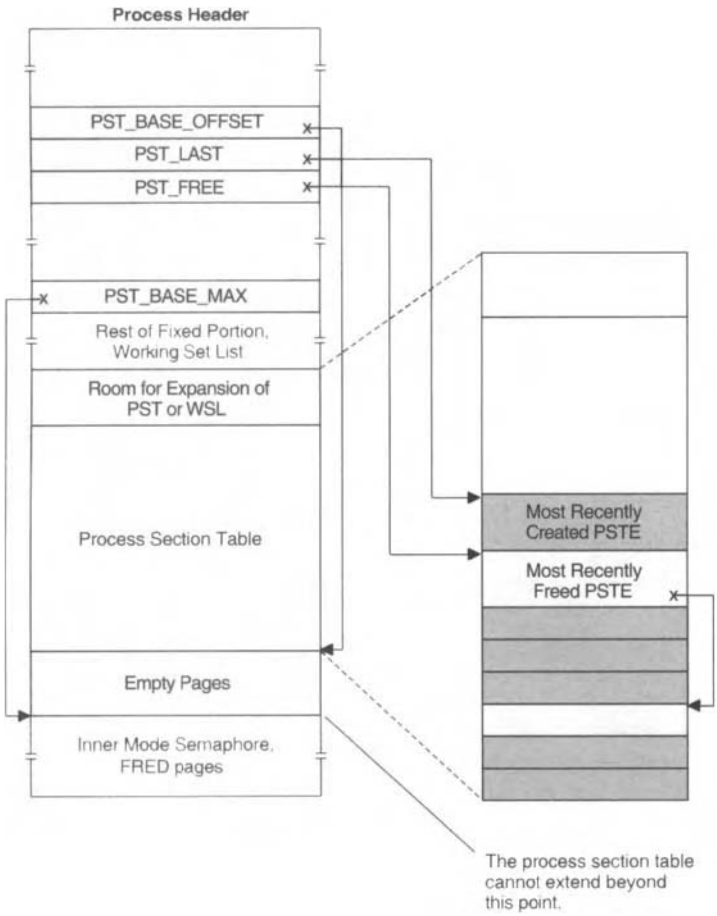
Figure 2.6 shows the location of the PST within the PHD. PHD$L_PST_BASE_OFFSET contains the byte offset from the beginning of the PHD to the base of the PST, its high-address end.

Each PSTE within the table is 40 (symbolically, SEC$C_LENGTH) bytes long and is located through a longword index from the base of the PST. The first PSTE has an index of 1, and the second an index of 2. Successive PSTEs are at lower addresses. Since all references to a PSTE are relative to PHD$L_PST_BASE_OFFSET, the PST can be moved within the PHD without requiring changes in process PTEs that contain process section table indexes or in PSTEs.

The following operations compute the address of a particular PSTE:

1. Add the contents of PHD$L_PST_BASE_OFFSET to the address of the PHD. The result is the address of the base of the PST.
2. Multiply the process section table index by the length of a PSTE.

Figure 2.6 Process Section Table


Process Header
PST_BASE_OFFSET
PST_LAST
PST_FREE
PST_BASE_MAX
Rest of Fixed Portion, Working Set List
Room for Expansion of PST or WSL
Process Section Table
Empty Pages
Inner Mode Semaphore, FRED pages
Most Recently Created PSTE
Most Recently Freed PSTE
The process section table cannot extend beyond this point.

3. Subtract the result from the address of the base of the PST.

Allocating or deleting a PSTE is synchronized by executing at IPL 2 and holding the inner mode semaphore (see Chapter *Kernel Threads*).

A PST is organized into a variable number of linked lists of PSTEs. Figure 2.6 shows a typical PST with free and allocated PSTEs; the allocated PSTEs are shaded. The index in PHD$L_PST_LAST is the largest index of any entry ever allocated and is thus a “high-water mark.”

All the process sections that page from the same section file using the same assigned channel are linked together. The entries are linked together through the backward and forward link index fields of each entry.

When a section is deleted, the PSTE that mapped the section is placed on the list of free entries so that it can be reused. The index PHD$L_PST_FREE points to the most recent addition to the free list. If no entry has been deleted, PHD$L_PST_FREE contains zero. The first longword in a PSTE on the free list contains the index to the previous element on the free list. When a section is created, the PSTE allocation routine first checks the free list. If there is no free PSTE, a new one is created from the expansion region between the working set list and the PST, and PHD$L_PST_LAST is modified.

The executive attempts to keep the working set list and PST virtually adjacent, partly to simplify and shorten manipulation of the PHD during outswap and inswap and partly to minimize the chances of wasting physical memory for partial pages of both. When the executive must expand the working set list into the area already occupied by the PST or vice versa, it allocates space from the existing empty page area (see Figure 2.6). Then it moves the entire PST into the allocated space at higher addresses and stores the byte offset of the new base address in PHD$L_PST_BASE_OFFSET.

The longword at PHD$L_PST_BASE_MAX specifies the maximum size of the PST. This longword points to the high-address end of the empty page area. It contains a byte offset from the beginning of the PHD.

Room is reserved in the PHD for the maximum PST and working set list, specified by the SYSGEN parameters PROCSECTCNT and WSMAX. It is possible for the PST to grow larger than PROCSECTCNT specifies, at the expense of the working set list.

Figure 2.7 shows the layout of a section table entry. A section table entry in the system header describes a global section and is called a global section table entry (GSTE; see Section 2.7.2). Field names within a section table entry are defined by the STARLET.MLB macro $SECDEF and begin with SEC$.

The first longword in the PSTE has two names: in a PSTE, SEC$L_CCB contains the address of the channel control block (CCB) on which the section file has been opened; in a GSTE, SEC$L_GSD contains the address of the global section descriptor (GSD) for that section.

SEC$L_SECXFL and SEC$L_SECXBL contain indexes of the previous and next section table entry. These link an entry in use into a list of others that page using the same CCB. They also link all free entries together.

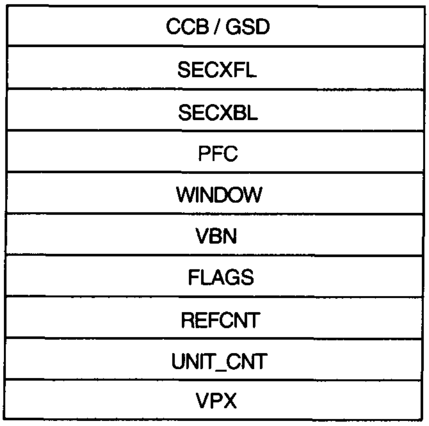
SEC$L_PFC contains the page fault cluster for this section, the number of section pages that the page fault handler attempts to read in together when a page fault occurs.

SEC$L_WINDOW is the address of the window control block (WCB) that describes the locations of the section file on a mass storage volume. The WCB points to the unit control block (UCB) for the volume.

SEC$L_VBN specifies the starting virtual, or file-relative, block number (VBN) of the section file at which the pages in this section begin.

SEC$L_FLAGS contains flag bits that describe the section.

Figure 2.7 Layout of a Process/Global Section Table Entry (PSTE/GSTE)



CCB / GSD
SECXFL
SECXBL
PFC
WINDOW
VBN
FLAGS
REFCNT
UNIT_CNT
VPX

PSTE Flags

Bit	Meaning
0	Global
1	Copy on reference
2	Demand zero
3	Writable
4	Shared memory–resident section
5	*(reserved)*
6–7	Access mode for writing
8–9	Owner access mode
10	*(reserved)*
11	Shared read–only page tables
12	Shared page tables
13	Memory–resident section
14	Permanent
15	0 = Group global 1 = System global

SEC$L_REFCNT contains the number of PTEs that refer to the section.

SEC$L_UNIT_CNT contains the number of units in the section. A PFN-mapped section is measured in units of physical pages. Any other type of section is measured in 512-byte pagelets. A pagelet is the size of a mass storage block. Note that a section file can occupy an arbitrary number of blocks or pagelets but a section must be created as a number of pages. If the number of blocks in a section file is not an integral multiple of blocks per page, the last page in the section is said to be partial.

For a process-private section, SEC$L_UNIT_CNT is initially related to SEC$L_REFCNT. If the section has no partial pages, then SEC$L_UNIT_CNT is initialized as an integral multiple of SEC$L_REFCNT. On a system with an 8 KB page size, SEC$L_UNIT_CNT would be SEC$L_REFCNT multiplied by 16. If the section ends with a page not completely backed up by section file blocks, SEC$L_UNIT_CNT is less than an integral multiple of SEC$L_REFCNT. For a global section, SEC$L_REFCNT is the number of PTEs that refer to the section's units from all the processes that have mapped it. For either type of section, SEC$L_REFCNT is reduced when a process deletes pages in its address space that map the section.

SEC$L_VPX contains the starting virtual page number at which the section's pages are mapped in the address space.

Most fields in a PSTE are initialized when the section is created and not modified subsequently. SEC$L_REFCNT is modified as the process deletes section pages from its address space. It is modified with the MMG spinlock held, since it can be accessed from process context and also by I/O postprocessing code.

The following steps locate a virtual page in a section file through information in the PSTE:

1. Subtract the section's starting virtual page number from the virtual page number of the faulting page to get the page offset into the section.

2. Multiply the page offset by the number of pagelets per page.
3. Add the contents of SEC$L_VBN to the block offset computed in step 2 to get the VBN of the virtual page within the file.

   In page faulting from a section file or writing a modified page back to a section file, the executive checks whether the section file has a page's worth of blocks beginning at that VBN. It compares the contents of SEC$L_UNIT_CNT, the number of pagelets (and therefore blocks) in the section file, to the sum of that VBN and the number of blocks in a page. If the section does not have enough blocks, the executive transfers only as many blocks as exist in the section file for that virtual page and zeros the rest of the page.
4. Use the mapping information in the WCB to transform the VBN to a logical block number on a mass storage volume.

2.1.3.4 Process Header Page BAK Array

In OpenVMS versions prior to Version 7.0, information about each PHD page was stored in four PHD page arrays.

Two of the arrays contained reference counts for each L3PT page. Sizing these arrays as a function of the size of the PHD, which previously contained the page tables, was straightforward. Now that the page tables have been removed from the confines of the PHD and their maximum lengths are not fixed, describing them with a fixed-size array is not viable. Instead, the reference counts have been moved to the PFN database (see Sections 2.5.3.16 and 2.5.3.17).

Two of the arrays saved information about each PHD page in the working set, namely, their working set list position and backing store. Information was recorded in these two arrays at outswap of a process and was used during inswap. The working set list information is reconstructed at inswap (see Chapter 6).

The BAK array is the only one left in the PHD. While a PHD is resident, the backing store location of each of its pages is stored in the PFN database. When the PHD is outswapped, both the physical pages and the balance set slot it occupied are released for other uses. The PHD BAK array records the backing store information for each PHD page, which would otherwise be lost. Note that it does not include information about page table pages, which are no longer part of the PHD. The backing store location of a page table page in a page file is stored in the PTE that maps that page table page.

The BAK array begins at offset PHD$Q_BAK_ARRAY, following the fixed part of the PHD. It has a quadword element for each of the maximum number of pages in the PHD.

2.1.3.5 Array of FREDs

A FRED consists of a HWPCB followed by space in which the floating-point registers can be saved at context switch, plus several other fields.

The FRED for the initial kernel thread in a process is contained in the fixed part of the PHD. The FREDs of subsequent kernel threads are contained in a part of the PHD expanded when multithreading is initiated in a process. Each FRED occupies 512 bytes.

The PHD is expanded by one or more physically contiguous pages, depending on the number of kernel threads requested. The maximum number of expansion pages possible, stored in SWP$GW_FREDPTE, is based on the maximum number of kernel threads supported, PCB$K_MAX_KT_COUNT. In Version 7.3, PCB$K_MAX_KT_COUNT is 256.

PHD$L_FRED_OFF contains the longword offset from the beginning of the PHD to the FRED array in the expansion pages. The array is indexed by the low part of the kernel thread ID. The process's inner mode semaphore occupies the first 512-byte block. Chapter *Kernel Threads* contains further information.

2.2 System Header and System PCB

The executive maintains two data structures for itself that are analogous to process structures: the system PCB and system header. Using these, the page fault handler can treat page faults of system pages almost identically to page faults for process pages.

The system PCB, whose address is in MMG$AR_SYSPCB, contains a base priority used for I/O requests for page faults of system space pages and global pages. It also has a pointer to the system header, parallel to the PHD pointer in any process PCB.

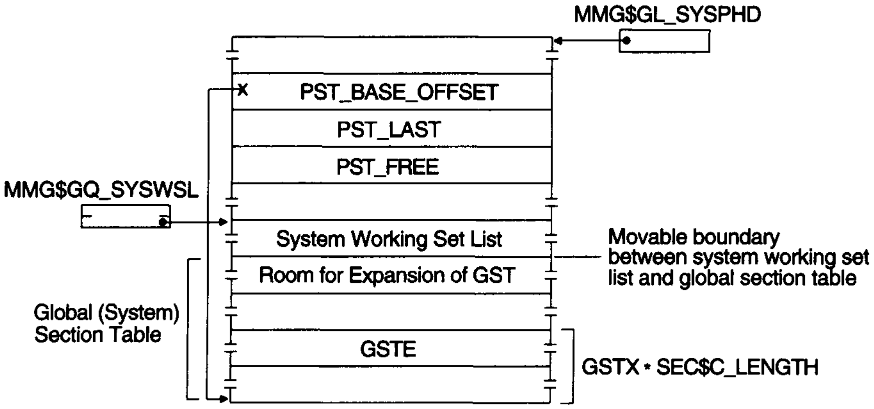
The system header, whose address is in MMG$GL_SYSPHD, occupies part of the granularity hint region for systemwide writable data (see Section 2.4). As shown in Figure 2.8, the system header contains a working set list and a section table.

Its working set list governs page replacement for pageable system pages from pageable sections in executive images and paged pool. (Although much of the executive is nonpageable, some executive images contain pageable image sections.) These are all described in the system working set list. Its size in pagelets is determined by the SYSGEN parameter SYSMWCNT. Unlike other working set lists, the system working set list does not expand or contract in response to system page fault rate. Once the system working set fills, replacement paging is required. Changes to the system working set list are synchronized by the MMG spinlock.

For consistency with the process working set list, the system working set list is also located through a pointer, which is named MMG$GQ_SYSWSL.

The backing store for pageable writable executive data and page file global sections is within page files.

Figure 2.8 System Header Containing the System Working Set List and the Global Section Table


MMG$GL_SYSPHD
PST_BASE_OFFSET
PST_LAST
PST_FREE
MMG$GQ_SYSWSL
System Working Set List
Room for Expansion of GST
Movable boundary between system working set list and global section table
Global (System) Section Table
GSTE
GSTX * SEC$C_LENGTH

The section table in the system header contains entries for sections in files that contain pageable system pages and for global sections. The SYSGEN parameter GBLSECTIONS specifies the number of entries in the section table.

Although the system header, like any other PHD, has space to describe three process-permanent regions, it does not describe any process-private space. The P1 and P2 RDEs are unused. The P0 space RDE represents allocatable S0/S1 space, insofar as PHD$PQ_P0_FIRST_FREE_VA contains the virtual address of the next available unmapped page of S0/S1 space.

2.3 Page Tables

As shown in Figure 1.7, each process has its own page table hierarchy, beginning with its own level 1 page table (L1PT). The hierarchy includes process-private level 2 page tables (L2PTs) that map process-private level 3 page tables (L3PTs) and shared L2PTs that map shared L3PTs. A process that maps a memory-resident global section with shared page tables potentially has a process-private L2PT that maps both process-private and shared L3PTs.

The sections that follow describe process-private and shared page tables, system space page tables on a platform with replicated system space, and page table entry (PTE) formats.

2.3.1 Process-Private Page Tables

When a process is created, the executive allocates and initializes a page of physical memory for use as the process's L1PT. It zeros most of the L1PTEs and initializes several valid L1PTEs:

- An L1PTE at offset 0 to map a process-private L2PT for P0 and P1 space and some of P2 space

- An L1PTE to map the page table virtual address space
- One or more L1PTEs, at the end of the L1PT, to map one or more shared L2PTs for system space

A process's L1PT is permanently locked into the process's working set list and is outswapped and inswapped with the process header.

The executive also creates a process-private L2PT for the process. This L2PT maps the L3PTs that map the process's P0, P1, and some of P2 space. On a system with an 8 KB page size, the first 256 L2PTEs are sufficient to map the entire 1 GB of P0 space and 1 GB of P1 space. This L2PT is nonpageable and permanently locked into the process's working set list.

The executive also creates some P1 space for the process and an L3PT to map the P1 space. This L3PT is permanently locked into the process's working set because some of the pages it maps are nonpageable.

As additional P0, P1, or P2 virtual address space is created for the process, the executive creates additional pageable L3PTs as necessary. If the process creates P2 space that cannot be mapped by the L2PT that maps P0 and P1 space, the executive creates additional pageable L2PTs as necessary.

2.3.2 System Space Page Tables

During system initialization, console software allocates and initializes page tables for the primary bootstrap program, APB. The secondary bootstrap program, SYSBOOT, uses the same page tables during its execution.

SYSBOOT is responsible for initializing system space. It sizes system space based on the sum of the maximum size of S0/S1 space and the following:

- The value of the GBLPAGES SYSGEN parameter
- The value of the S2_SIZE SYSGEN parameter
- The value of the MAXBOBS2 SYSGEN parameter
- The size of the PFN database
- The size of the lock management database

SYSBOOT allocates a physical page to become the shared L2PT that maps S0/S1 space and part of S2 space and zeros it. It initializes the last L1PTE in the current L1PT with the L2PT's PFN. The ASM bit is set in the L1PTE.

SYSBOOT creates S0/S1 space beginning at location FFFFFFFF 80000000_{16} and going toward higher addresses. It creates as much as is needed for executive images; data structures such as the balance set slots and error log message buffers; and nonpaged pool.

Beginning at the lowest possible S2 space address and going toward higher addresses, SYSBOOT creates as much S2 space as is needed for the PFN database and global page table. Subsequent expansion of S2 space begins at the high end of S2 space and goes toward lower addresses.

SYSBOOT also double-maps the current L1PT into S2 space, storing its address in MMG$GQ_SYSTEM_L1PT. This will be the system context L1PT to be used when no thread is executable.

SYSBOOT allocates physical pages of memory for the page tables that map the system space it is creating and stores their PFNs in the appropriate PTEs. Because the Alpha architecture supports a sparse address space and does not require page tables that map virtually contiguous address regions to be physically contiguous, additional L3PTs and L2PTs can be allocated after system initialization as additional S0/S1 or S2 system space is needed.

Chapters *Bootstrap Processing* and *Operating System Initialization and Shutdown* describe the bootstrap sequence in detail.

System space is shared by all processes. The ASM bit is set in all L1PTEs, L2PTEs, and L3PTEs that map system space. When a new process is created, the shared PTEs in the system L1PT are copied to the new process's L1PT so that the new process maps the shared L2PTs that map S0/S1 and S2 space.

Figure 2.9 shows part of the page table hierarchies of two independent processes. Each L1PT maps a process-private L2PT and process-private L3PTs for P0 and P1 space, but each maps a shared L2PT and shared L3PTs for system space. For simplicity, the figure omits the shared L2PT and L3PT that map the lower end of S2 space.

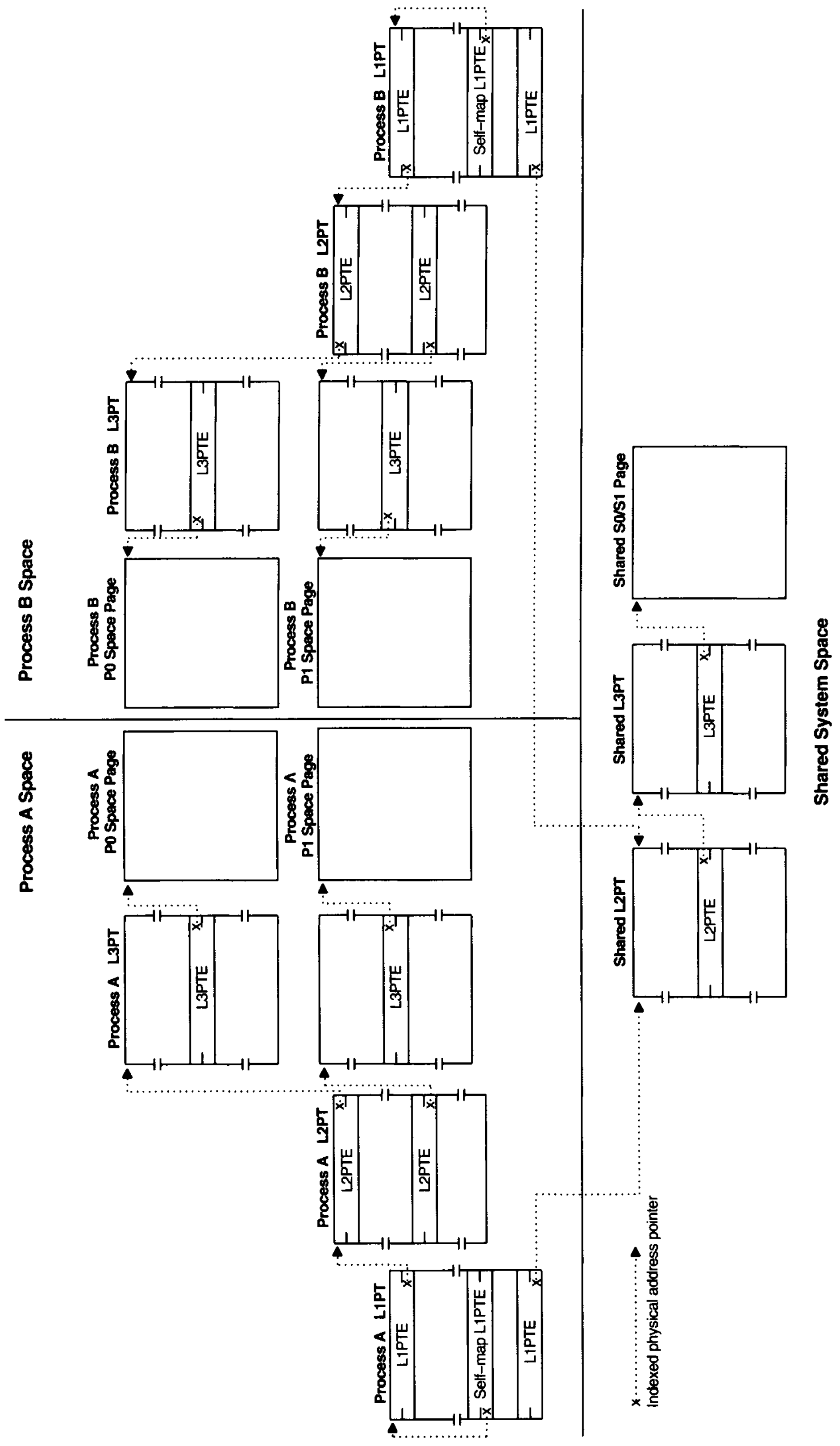
When a shared L3PT is allocated to accommodate system space expansion, a PTE in a shared L2PT is updated to map the new L3PT. Because each process maps that shared L2PT, each process automatically has access to the new system space.

When, however, a shared L2PT is allocated to accommodate S2 space expansion, a PTE in each process-private L1PT must be updated to map the new L2PT. (S0/S1 expansion cannot exceed a single L2PT.) The update is performed asynchronously, as needed: when an access violation accessing system space occurs, the exception handler checks whether the attempted access was to S2 space not currently mapped by the current process's L1PT. If so and if the space is mapped by the system L1PT, the exception handler updates the current process's L1PT and dismisses the exception.

The page tables that map system space are not pageable.

Section 2.3.4 summarizes how system space page tables differ on a system with replicated system space.

Figure 2.9 Process-Private and System Page Tables



]

2.3.3 S0/S1 Page Table Window

In OpenVMS versions prior to Version 7.0, the S0/S1 page table self-mapped itself to be accessible through system space addresses. For example, memory management routines altering S0/S1 PTEs in response to system space page faults accessed them through system space virtual addresses. That self-mapping had the following disadvantage: because it self-mapped through an L2PTE, it wasted 6 MB of system space on a system with an 8 KB page.

In OpenVMS Version 7.0 and later versions, the level 2 self-mapping has been eliminated. Memory management routines access S0/S1 and S2 page tables using 64-bit page table space addresses.

OpenVMS, however, also double-maps into S0/S1 space the L3PTs that map S0/S1 space. The double mapping enables pre-Version 7.0 device drivers that referenced S0/S1 space L3PTEs with 32-bit addresses to execute on Version 7.0 and later versions without source changes.

The global cell MMG$GL_SPTBASE continues to contain the system virtual address of the S0/S1 L3PTs. In releases of OpenVMS prior to Version 7.0, in which the only system space was S0/S1 space, the L3PTs that mapped system space were called the system page table (SPT), a name taken from the VAX architecture. Thus, the term *SPT window* refers to the double-mapped S0/S1 L3PTs.

Figure 2.10 shows the SPT window. Basically, up to 256 S0/S1 L3PTEs are copies of the L2PTEs that map S0/S1 space L3PTs. This enables the S0/S1 L3PTs to be accessed using S0/S1 addresses. The shaded areas in the figure represent those two sets of PTEs.

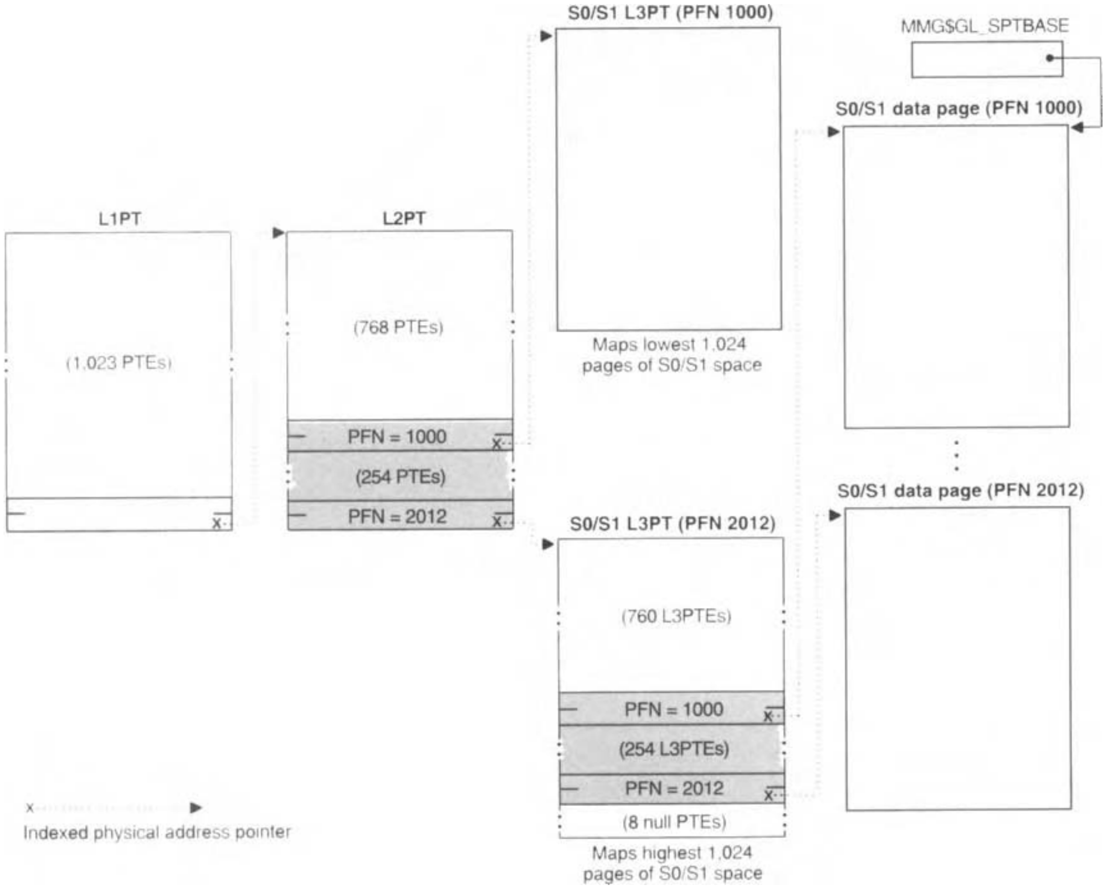
The left-hand part of the figure shows a process's L1PT. The last L1PTE maps 8 GB that includes S0/S1 space. That L1PTE points to an L2PT whose last 256 L2PTEs map S0/S1 space L3PTs. The figure shows the 768th and the last L2PTE each pointing to an S0/S1 L3PT.

The next part of the figure shows those two L3PTs. The upper one, in this example contained in PFN 1000, maps the lowest 1,024 pages of S0/S1 space. The lower one, in this example, PFN 2012, maps the highest 1,024 pages of S0/S1 space.

The highest page of S0/S1 space is deliberately made not accessible (see Chapter 1). To make the size of the no-access space consistent regardless of page size, 64 KB is made inaccessible. The L3PTEs that map the rest of S0/S1 space are simply copied from the L2PTEs that map S0/S1 space L3PTs.

As a consequence of this double mapping, any reference to an S0/S1 space L3PTE made using its S0/S1 space address accesses the real L3PTE. Whenever memory management code expands S0/S1 space such that a new L3PT is allocated, it copies the contents of the newly used L2PTE to the corresponding L2PTE that maps the SPT window.

Figure 2.10 SPT Window



2.3.4 Replicated System Space Page Tables

As discussed in Chapter 1, on a NUMA platform OpenVMS optionally replicates a part of system space in physical memory local to each RAD. The system manager enables replication by setting bits 0 and 2 (RIH$V_RAD_ENABLE and RIH$V_SYSTEM_REPL) in SYSGEN parameter RAD_SUPPORT.

During system initialization, after executive and resident images have been loaded, OpenVMS calculates the number of physical pages needed for each RAD's replication as the sum of the following:

- 1 for an L1PT
- 1 for an L2PT to map some of S0/S1 space
- SYSGEN parameter GH_EXEC_CODE, the size of the executive image code huge page (see Section 2.4)

- SYSGEN parameter GH_RES_CODE, the size of the resident image code huge page
- L3PTs to map the executive and resident image code huge pages

OpenVMS checks that there is enough physical memory to accommodate system replication for each RAD and allocates that many physically contiguous pages from memory local to each RAD. It double-maps each RAD's L1PT into S2 space. It initializes each RAD's L1PT by copying the base RAD's L1PT and replacing the self-map L1PTE and the L1PTE that maps the two huge pages.

It initializes each RAD's L2PT by copying the base RAD's S0/S1 L2PT and replacing the L2PTEs that map the two huge pages. It initializes the L3PTs that map the two huge pages and copies the contents of the two code huge pages to the pages allocated for this RAD's use. The PFN of the RAD's L1PT is stored in the SYSPTBR processor register of each CPU in that RAD.

When OpenVMS creates a new process's L1PT on such a system, it initializes only the L1PTEs that map process-private space. The RAD-specific L1PT will map system space.

Figure 2.11 shows a slightly simplified version of the page table hierarchy of Figure 2.9 on a NUMA platform with replicated system space. Each process has its L1PT to map process-private space. Each RAD has its own L1PT to map system space. Each L1PTE that maps S0/S1 space points to a RAD-specific L2PT. Each valid RAD-specific L2PTE points to a RAD-specific L3PT that maps RAD-specific copies of system space code.

Expansion of system space has to be reflected in each RAD's page tables. When a shared L3PT is allocated to accommodate system space expansion, as described in Section 2.3.2, each RAD's L2PT is updated to map the new L3PT. If a shared L2PT is allocated to accommodate system space expansion, each RAD's L1PT is updated to map the new L2PT.

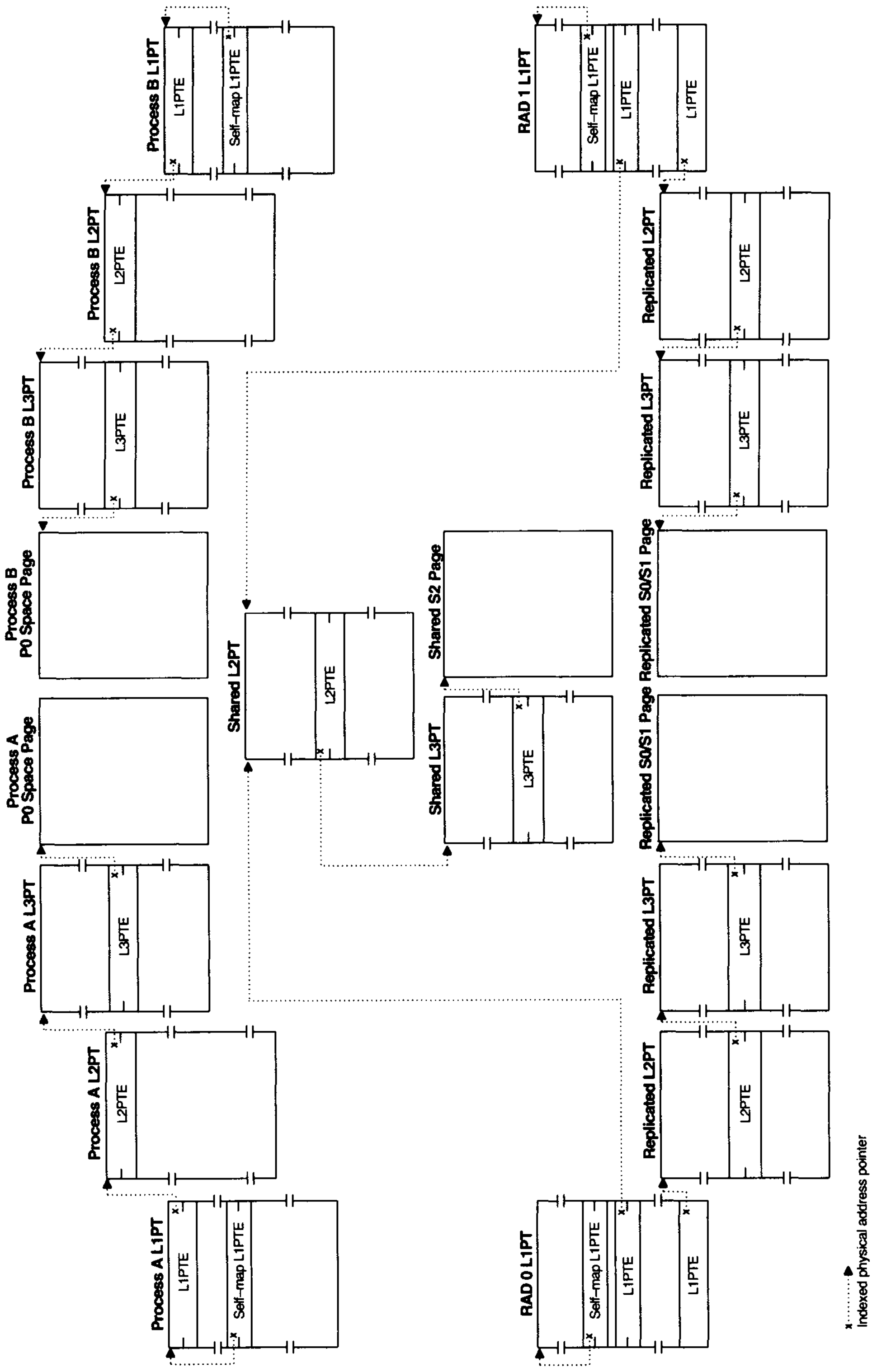
2.3.5 PTE Formats

Chapter 1 describes the architecturally defined bits in a valid PTE: bits <15:0> and bits <63:32>. This section describes the bits reserved to software and the various formats possible for a PTE that describes an invalid page. Although the three fault-on bits are architecturally defined, their use is operating-system-specific and is described here.

Figure 2.12 shows the various forms of valid and invalid PTE that can appear in an L3PT. The shaded bits in each PTE are either reserved or bits whose contents are irrelevant for that form of PTE.

The fault-on-write bit enables OpenVMS to maintain a modify bit for a writable virtual page. The state of the modify bit determines on which transition list a page is placed when it is faulted out of the working set. OpenVMS sets the fault-on-write bit in the L3PTE of a writable page when it is faulted with read intent. After the page becomes valid, if any attempt is made to write it, the processor generates a fault-on-write

Figure 2.11 Process-Private and System Page Tables with Replicated System Space



exception. The exception service routine clears the fault-on-write bit and sets the modify bit in the L3PTE, which is within the bits reserved to software. In contrast, when a writable page is faulted with write intent, the modify bit in its L3PTE is set when the page fault I/O completes. Chapter 4 describes in more detail how the modify bit is maintained.

The executive uses fault-on-execute to restrict access to translated image pages that the Translated Image Environment (TIE) facility identifies as no-execute. These are image pages that contain VAX instructions. Any attempt to execute instructions from such a page results in a fault-on-execute exception. The exception service routine signals an access violation to the TIE's condition handler. Chapter *Translated Image Environment* describes the use of this mechanism in more detail. Chapter 1 describes an additional use of the fault-on-execute bit.

Section 2.4.3 describes the use of the fault-on-read bit.

Bits <31:16> are reserved for software. The executive defines a number of them:

- Bit 16 in a valid PTE is the window bit. When set in an L3PTE, it means that the virtual page is a double mapping of a physical page. When the virtual page is deleted, the PFN database for the physical page should not be altered. This bit is also set in an L2PTE for an L3PT that is a shared page table or that maps memory-resident global sections or window pages. This bit is set in an L1PTE for an L2PT that maps shared page tables.
- Bit 20 in a valid PTE is the modify bit. When set, it means that the virtual page has been modified and not yet been written to backing store.
- Bits <29:28> specify how the page should be copied when a process's address space is cloned during a Portable Operating System Interface (POSIX) fork operation.
- Bit 30, when set, specifies that no execution access to the page is permitted. The TIE facility identifies a VAX image page containing untranslated code as no-execute. The executive sets the fault-on-execute bit as well so that an attempted instruction fetch from such a page triggers an exception.
- Bit 31 always contains zero. In previous versions, it was used to distinguish PTE contents from the system space virtual address of a PTE or GPTE, whose bit 31 was set.

Note that the valid bit, protection bits, owner access mode bits, copy characteristic bits, and bits 7 (no TB miss MB required) and 31 have the same meaning in all forms of PTE.

The owner access mode bits record the access mode that owns that page. The executive allows a process to modify the characteristics of a virtual page or delete it from an access mode equal to or more privileged than the page's owner access mode.

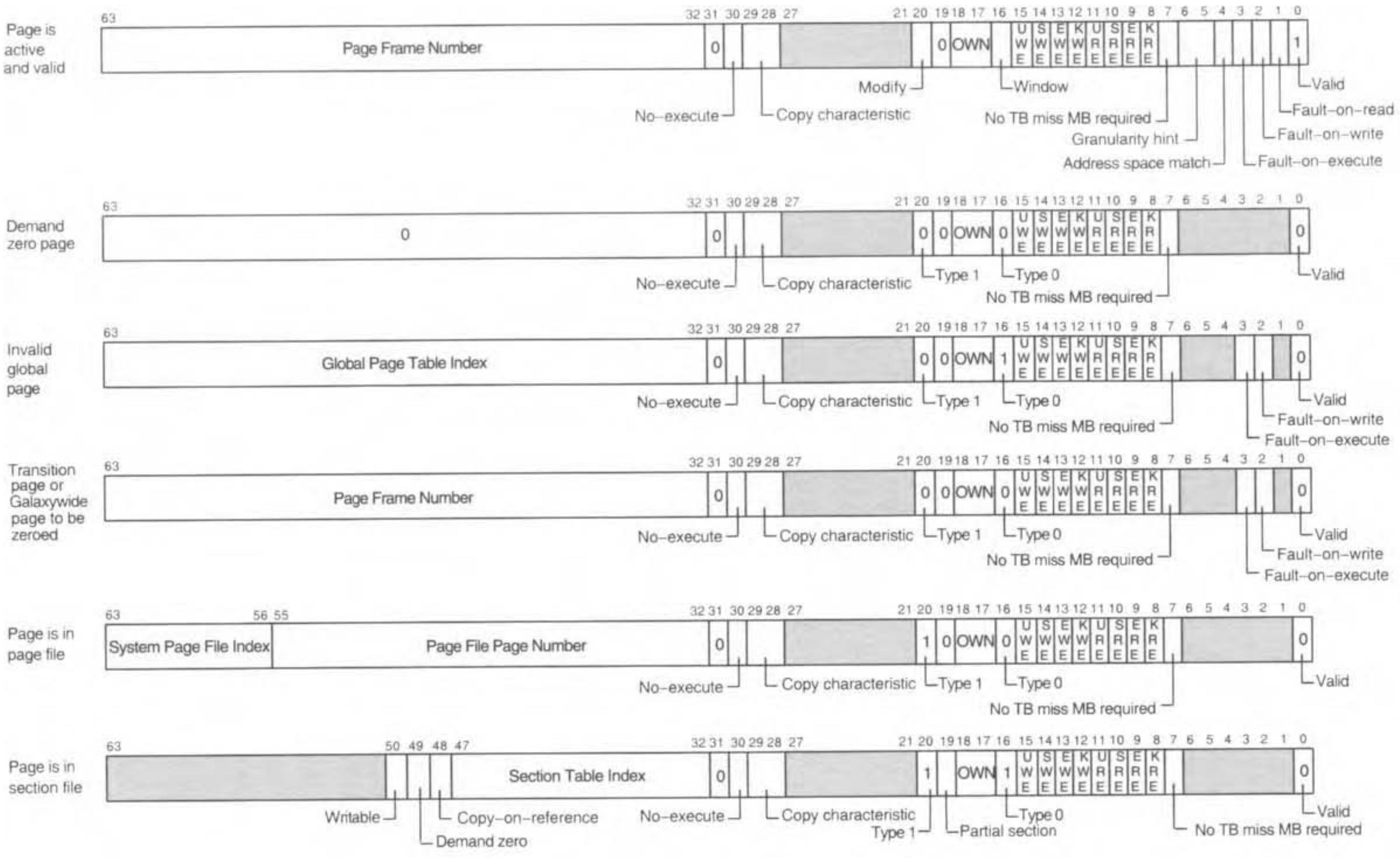


Figure 2.12 L3PTE Formats

A PTE for an invalid page contains either the location of the page or a pointer to further information about the page. The page fault handler uses the type bits, bits 16 and 20, in the invalid PTE to distinguish the different forms of invalid PTE. These are described in the sections that follow. Chapter 4 describes the processing of page faults for various types of invalid PTE.

One form of invalid PTE not pictured in Figure 2.12 is a null page, a quadword of zero. A PTE with a zero protection code disallows any access to the page by any mode. This form of PTE describes an unmapped page of address space.

2.3.5.1 PTE Containing a Process Section Table Index

The PTE of each page in a process section contains the index of the PSTE describing that section. The PSTE has information about the location of the file mapped into the process address space and about the mapping between virtual file blocks and section pages.

The PSTE also contains control bits that are copied to the PTE of each page in the section:

- Bit PTE$V_CRF (bit 48) is set to indicate the page is copy-on-reference.
- Bit PTE$V_DZRO (bit 49) is set to indicate the page is demand zero.
- Bit PTE$V_WRT (bit 50) is set to indicate the page is writable.

In addition, bit PTE$V_PARTIAL_SECTION (bit 19) is set in a PTE that maps a page not entirely backed by a section file. With page size not equal to disk block size, any section file whose block count is not an integral multiple of pages has a last page with this attribute.

Section 2.1.3.3 describes the PST organization and the layout of the PSTE.

2.3.5.2 PTE Containing a Page File Page Number

With OpenVMS Version 7.3, the system manager can install up to 254 page files. A process can page in any or all of them and is no longer limited to four page files, as it was in prior versions.

When a virtual page has been faulted out to a page file, its PTE contains the number of the page within the page file and an eight-bit number starting at bit PTE$V_PGFLX (bits <63:56>) indicating the page file in which the page is located. The eight-bit number is an index into the page-and-swap-file vector. Section 2.9.2 contains more information on the page-and-swap-file vector, and Chapter 4 discusses how page file backing store is assigned.

2.3.5.3 PTE Containing a Global Page Table Index

The PTE of an invalid process page mapped to a global page contains an index into the global page table, where an associated global PTE contains the information used to locate the page. Section 2.7.4 describes the contents of global PTEs.

2.3.5.4 PTE of a Page in Transition

When a physical page is removed from a process working set, it is not discarded but put on the free or modified page list. The invalid virtual page, still associated with the physical page, is called a transition page. Its PTE contains a PFN, but the valid bit is clear. The two type bits are also clear. Retaining the connection to a physical page enables the executive to fault the virtual page back into the working set with minimal overhead until the physical page is reallocated for another use.

Another type of transition page is a virtual page in transit between mass storage and physical memory. When a process faults a page not in memory, the page fault handler allocates a physical page and requests an I/O operation to read the virtual page from its backing store. While the I/O request is in progress, the virtual page has a transition PTE.

Yet another type of transition page is a page in a Galaxywide demand zero memory-resident section that has not yet been zeroed. When a process first accesses a page mapped by such an invalid L3PTE, it will trigger a page fault. The page fault exception service routine will recognize that this is an uninitialized page in a Galaxywide global section and zero it.

A transition page is described further by its physical page's record in the PFN database (see Section 2.5.3). In particular, the PFN$L_PAGE_STATE field in the PFN database record (see Section 2.5.3.6) identifies the state of the page and distinguishes among the different types of transition page.

2.3.5.5 PTE of a Demand Zero Page

One form of transition PTE has a zero in the PFN field. This zero indicates a special form of page called a demand-allocate, zero-fill page, or demand zero page for short. A demand zero page is a writable page of address space, created on demand instead of being read in from backing store, and zeroed. When a page fault occurs for such a page, the page fault handler first tries to allocate a physical page from the zeroed page list. If the zeroed page list is empty, the page fault handler must allocate a physical page from the free page list and fill the page with zeros. In either case, it then inserts the PFN into the PTE, sets the valid and modify bits, and dismisses the exception.

2.3.5.6 System Space PTEs

For the most part, system space L3PTEs can take on the same formats as valid and invalid process-private L3PTEs (see Figure 2.12). One exception is that an invalid system space L3PTE cannot have the global page table index format.

Additionally, invalid system space L3PTEs that are unused and available for allocation are linked together in a list. The L3PTEs themselves contain information: a pointer to the next group of free L3PTEs and the number of free L3PTEs in this group. Section 2.3.6 shows the contents of L3PTEs used in this way.

The MMG spinlock synchronizes changes to system space PTEs.

2.3.6 Available System Space

As previously described, SYSBOOT defines the initial size and layout of S0/S1 and S2 space, based on SYSGEN parameters and other values. Additional space is created during normal system operation by creating additional L3PTEs, a page's worth at a time. Code running in kernel mode calls various executive routines to allocate and map pages of system space.

Unused and available L3PTEs that map system space are kept in either of two lists: the one for S0/S1 space or the one for S2 space. These lists facilitate a search for available system space PTEs: examining list elements is quicker than scanning all the PTEs for adjacent free ones. The elements on each list are groups of system space L3PTEs. The listheads are LDR$GQ_FREE_S0S1_PT and LDR$GQ_FREE_S2_PT. Each points to the first element on its list. Figure 2.13 shows the form of the list, with free L3PTEs shaded.

Each element on the list represents a group of adjacent available system space L3PTEs. The smallest group contains one L3PTE. A single available L3PTE contains, in bits <63:19>, a pointer to the next group. Bit 16 is set to identify the L3PTE as the sole member of its group.

Two L3PTEs are required to describe an element consisting of a group of two or more adjacent available L3PTEs. The first L3PTE points to the next group of free L3PTEs; the second contains the number of L3PTEs in this group.

The low-order 16 bits of each free L3PTE, which include protection code bits and the valid bit, must be zero so that the L3PTE appears to map an invalid page with all access prohibited.

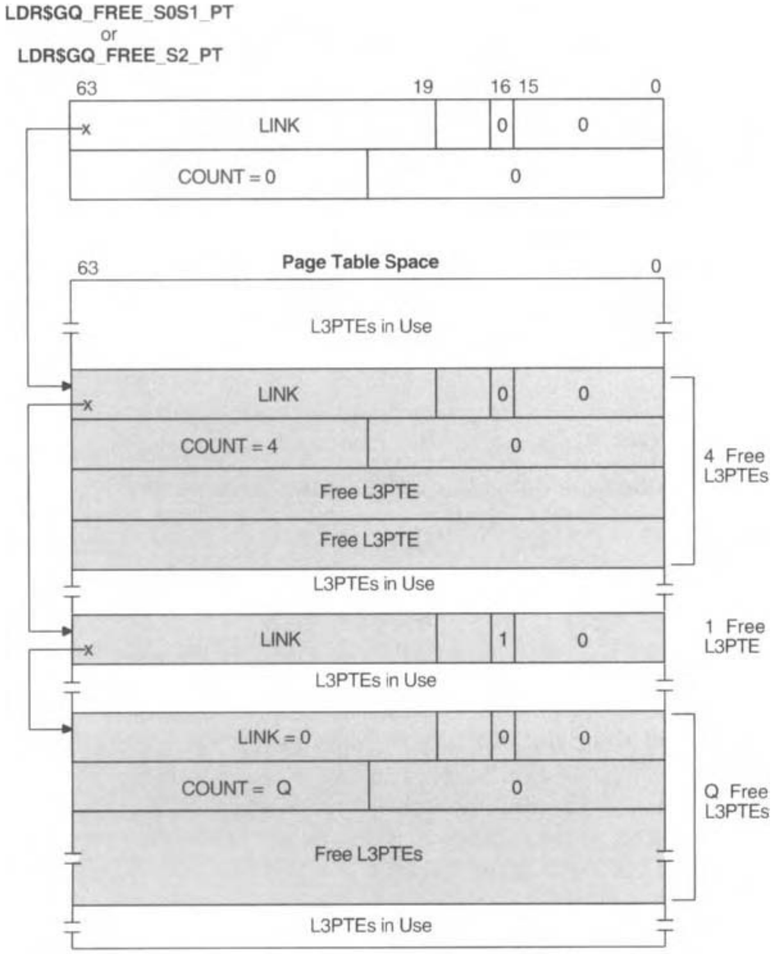
A group of free L3PTEs is identified by its quadword index from the beginning of page table space (that is, its PTE number). The quadword index of the next element is stored in bits <63:19> of the L3PTE. For example, if LDR$GQ_FREE_S0S1_PT<63:19> contains 100_{16}, the first L3PTE available for allocation is at offset (100_{16} * 8) from the base of page table space. The number of L3PTEs in that group is at offset (100_{16} * 8) + C_{16}.

The MMG spinlock synchronizes allocation from and deallocation to the lists of available system space L3PTEs.

This list format is also used for free global page table entries (see Section 2.7.4).

Chapter *The Modular Executive* describes some of the routines by which system space L3PTEs are allocated and deallocated.

Figure 2.13 List of Available System Space L3PTEs



2.4 Granularity Hint Regions and Huge Pages

An Alpha translation buffer (TB) supports granularity hints, by which a single TB entry can represent a group of pages that are virtually and physically contiguous. The total number of TB entries and the number that can represent a group of pages are CPU-dependent. Chapter 1 describes the role of the TB in address translation. During system initialization, physical memory and system address space are reserved for granularity hint regions.

2.4.1 Uses of Granularity Hint Regions

During system initialization, OpenVMS creates one or more granularity hint regions for several specific uses. The term *huge page* refers to the granularity hint region or regions associated with each of these particular uses. A huge page consists of more than one granularity hint region if it requires more physical pages than can be mapped by one TB entry. Each huge page is described by a data structure called a loader huge page descriptor (see Section 2.4.2).

By default OpenVMS creates the following huge pages:

- Base and executive images' nonpaged code (LDRHP$K_CODE)
- Base and executive images' nonpaged user-read data (LDRHP$K_DATA)
- Nonpaged dynamically allocated executive-read S0/S1 space data (LDRHP$K_EXEC_DATA)
- Code of images installed resident (LDRHP$K_RES_CODE)
- User-read data of images installed resident (LDRHP$K_RES_DATA)

One granularity hint region is always created for nonpaged dynamically allocated system data. Whether other regions are created depends on various SYSGEN parameters, including the flags in the SYSGEN parameter LOAD_SYS_IMAGES.

In conjunction with nonzero values for SYSGEN parameters GH_EXEC_CODE and GH_EXEC_DATA, bit 1 of LOAD_SYS_IMAGES, SGN$V_EXEC_SLICING, when set, specifies that base and executive images should be loaded with their nonpaged code sections in the code huge page and their data in a data huge page. The SYSGEN parameter GH_EXEC_CODE specifies the size of the code huge page, and GH_EXEC_DATA, the size of the data huge page. The default value for GH_EXEC_CODE is 512 pages, and for GH_EXEC_DATA, 128 pages. Chapter *The Modular Executive* describes how executive images are loaded sliced into huge pages.

Regardless of whether executive image slicing is enabled, a data huge page is allocated. Its size is based on various SYSGEN parameters, including the initial size of nonpaged pool, the size of the error log allocation buffers, and the size of the system header. At the end of system initialization, unused space in the executive image code and data huge pages is released.

The code sections of images installed resident, such as LIBOTS and LIBRTL, are loaded into another huge page, whose size is based on the SYSGEN parameter GH_RES_CODE. Their data is stored in a huge page whose size is based on GH_RES_DATA. The default value for GH_RES_CODE is 512 pages, and for GH_RES_DATA, zero pages. Chapter *Image Activation and Exit* describes the use and installation of resident images.

By default, at the end of system initialization, some or all unused pages in the resident image huge page code region are released to the free page list. The contents of the SYSGEN parameter GH_RSRVPGCNT specify how many pages are to be left available for mapping images installed resident after system initialization is complete. By default its value is zero.

On a NUMA platform with RAD support and system space replication enabled (bits RIH$V_RAD_ENABLE and RIH$V_SYSTEM_REPL set in SYSGEN parameter RAD_SUPPORT), each RAD has its own copy of the executive and resident image code huge pages (see Section 2.3.4). When system space replication is enabled, unused space in both the executive and resident image code sections is not released.

Provision is made for a huge page in S2 space containing nonpaged dynamically allocated kernel data. In OpenVMS Versions 7.0 and 7.1, the PFN database occupied this page. In Version 7.2 and later versions, the PFN database is mapped by its own granularity hint regions (see Section 2.5.3), and this huge page is currently not created.

In addition to these huge pages, one or more granularity hint regions are created to map each memory-resident global section. Furthermore, granularity hint regions can be created dynamically, for example, in response to a request to create a PFN-mapped section (see Chapter 3). The number of granularity hint regions possible is limited only by the availability of contiguous physical memory and virtual address space with the required alignment.

2.4.2 Loader Huge Page Descriptors

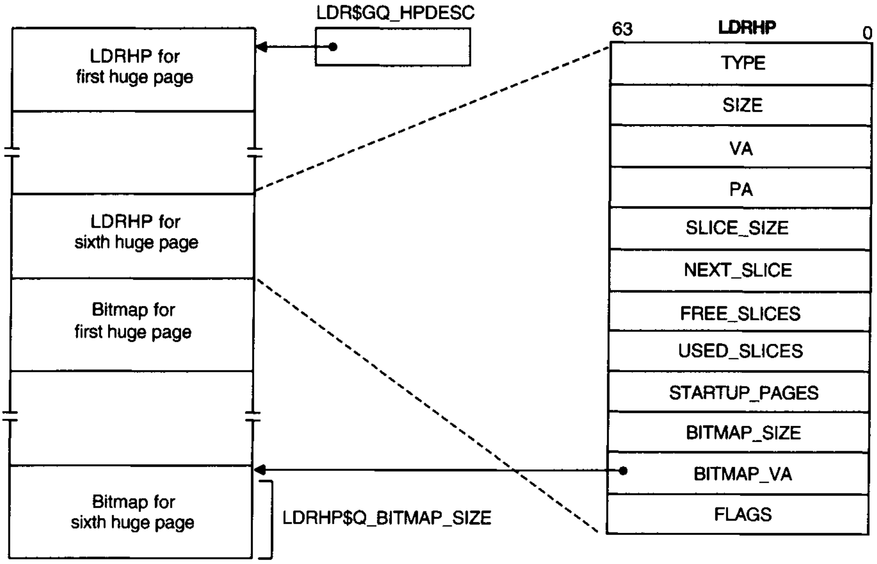
Each huge page is described by a nonpaged pool data structure called a loader huge page descriptor (LDRHP) and a bitmap that reflects allocations within the huge page. Six LDRHPs are allocated together, followed by six bitmaps. The LDRHPs are accessed by page type, defined symbolically by the $LDRHPDEF macro, for example, LDRHP$K_EXEC_DATA.

Each bitmap begins on a quadword boundary. The starting address of these structures is recorded in LDR$GQ_HPDESC. Once SYSINIT begins to execute, access to these structures and bitmaps is synchronized with the base image mutex, EXE$GQ_BASIMGMTX. Figure 2.14 shows the layout of these structures.

LDRHP$Q_TYPE identifies the type of huge page: read-only image sections, writable image sections, or systemwide writable data. LDRHP$Q_SIZE contains the size of the huge page in bytes; LDRHP$Q_PA, its starting physical address; and LDRHP$Q_VA, its starting virtual address.

LDRHP$Q_SLICE_SIZE contains the granularity of allocation, or slice, from this huge page. On current Alpha implementations, the granularity of allocation is 8 KB for the image code page and systemwide data page and 512 bytes for the image data page. LDRHP$Q_FREE_SLICES contains the number of available slices left in the huge page. LDRHP$Q_USED_SLICES contains the number of slices in the page that are in use. LDRHP$Q_STARTUP_PAGES contains the number of pages in use in the huge page at the end of system initialization. The contents of this field in the code huge page are used to determine how many pages to release to the free list.

Figure 2.14 Layout of Huge Page Data Structures



The bitmap, whose starting virtual address is in LDRHP$Q_BITMAP_VA, has one bit per slice. If the value of the bit is 1, the slice is available; if 0, it has been allocated. The size in bytes of the bitmap is in LDRHP$Q_BITMAP_SIZE. LDRHP$Q_NEXT_SLICE contains the number of the first free slice.

LDRHP$Q_FLAGS describes the state of the huge page. Currently, two flags are defined:

- LDRHP$V_ALLOC_FAIL, when set, means an attempt to allocate a slice from this huge page has failed.
- LDRHP$V_RELEASED, when set, means that unused pages of the huge page have been released to the free list.

2.4.3 Contents of PTEs Mapping Granularity Hint Regions

As described in Chapter 1, a nonzero value in bits <6:5>, the granularity hint bits, identifies the page as belonging to a granularity hint region and specifies the number of pages in the region.

On a system without replicated system space, OpenVMS sets the fault-on-read bit in each L3PTE that maps pages in the granularity hint regions containing executive nonpaged code and resident image code sections. The executive sets all *m*RE bits for these pages, which can contain mode of caller system service procedures and Run-Time Library procedures. That the protection bits enable read access means that any mode can fetch and execute instructions from these pages. The set fault-on-read bit, however, causes data fetches to fault. This mechanism blocks undesirable read accesses to these pages.

On a system without replicated system space, after all resident images have been installed, some or all of the unused physical pages that were part of a code huge page are released to the free page list. The system space L3PTEs that mapped them are zeroed. Even though the L3PTE that mapped such a page is zeroed, the granularity hint feature permits virtual addresses to be translated to physical addresses within the released page. If the TB holds an entry for any valid virtual page in the granularity hint region, the CPU uses that entry to translate any virtual address in the entire region, which still includes the released pages.

Once on the free page list, a released page can be reallocated for another use and mapped by some other L3PTE. Such a page can have two mappings: one, for example, in process-private space, and the other through the system space address range for the granularity hint region. If one process tries to read from a page reallocated to another process by using its former system virtual address, the set fault-on-read bit causes the CPU to generate a fault-on-read exception. If the system space L3PTE is null, the exception service routine advances the program counter in the exception stack frame past the instruction that incurred the exception and dismisses the exception. If the L3PTE is not null, the service routine interprets the instruction, fetches the data, and returns it. Chapter 3 describes another method by which the contents of these pages can be accessed as data.

If a process tries to execute from such a page, neither the protection nor fault-on bits will block its execution of whatever random contents the page might have.

On a system with replicated system space, unused pages are not released.

2.5 Data Structures Describing Physical Memory

OpenVMS maintains information about physical memory in several kinds of data structures:

- Configuration of the memory in physical memory maps and page frame number (PFN) memory maps (see Section 2.5.1)
- Reservations of physical memory for specific application purposes in reserved memory descriptors (see Section 2.5.2)
- Current state of each page of memory available to OpenVMS in the PFN database (see Section 2.5.3)

The actual amount of memory under OpenVMS control and available for it to allocate is not necessarily all the physical memory present on the system. Several factors can reduce this amount:

- Memory reserved by the console software for its code and data and for PALcode
- The PHYSICAL_MEMORY SYSGEN parameter
- Memory reserved for application use through the Reserved Memory Registry (see Section 2.5.2)

The PHYSICAL_MEMORY SYSGEN parameter specifies the amount of available memory that OpenVMS is allowed to use. Its default value, –1, allows OpenVMS to use all available memory, apart from that reserved by the console and the Reserved Memory Registry. A value lower than the actual amount of memory present enables smaller memory configurations to be tested on a larger system and also enables custom applications to use the upper part of memory.

In addition to the data structures previously listed, OpenVMS records some information about physical memory in system global cells. Following are some global cells that describe physical memory sizes and their contents:

- MMG$GL_MEMSIZE—the actual number of physical pages in the system, including console pages, any pages reserved through the PHYSICAL_MEMORY SYSGEN parameter, and any pages reserved through the Reserved Memory Registry file
- MMG$GL_PHYPGCNT—the number of physical pages in the system, excluding any reserved through the PHYSICAL_MEMORY SYSGEN parameter
- PFN$GL_PHYPGCNT—the number of fluid pages in the system, those not committed to permanent or long-term use

Section 2.5.3 describes several cells related to sizing the PFN database.

2.5.1 Memory Configuration

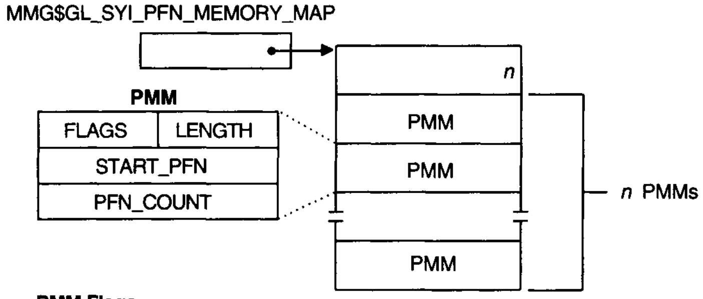
At boot time, the console determines the memory configuration and passes its findings as well as its own memory use to OpenVMS through a data structure called the hardware restart parameter block (HWRPB). Because physical memory is not required to start at PFN 0 or be contiguous, the console describes each group of contiguous pages by its starting page number and number of pages. Chapter *Bootstrap Processing* provides more details on the HWRPB and associated structures.

Using the information provided by the console, SYSBOOT initializes system cells and builds data structures to describe the memory accessible to this system:

- On every system, it describes memory in an array of physical memory map (PMM) descriptors.
- On a system with noncontiguous physical memory or a Galaxy node, SYSBOOT describes memory in an array of PFN memory map (PMAP) structures.

On every system, memory is described by an array of PMMs. Each PMM describes a contiguous set of pages with a common owner. PMM contents are returned to application code in response to a Get System Information ($GETSYI) request that specifies the item code SYI$_PFN_MEMORY_MAP. Through this information, an application can determine, for example, which pages, if any, have been excluded from OpenVMS use through the PHYSICAL_MEMORY SYSGEN parameter.

Figure 2.15 $GETSYI Physical Memory Map (PMM) Array



PMM Flags

Bit	Meaning
0	Allocated to console
1	Allocated to OpenVMS
2	Available for other uses
3	Galaxy base memory
4	Galaxy shared memory

Figure 2.15 shows the PMM array.

MMG$GL_SYI_PFN_MEMORY_MAP points to the longword that immediately precedes the nonpaged pool PMM array. The longword contains the number of PMMs that follow. PMM$L_START_PFN and PMM$L_PFN_COUNT have the same meaning as the corresponding PMAP fields. PMM$W_LENGTH contains 12, the size of each PMM, and PMM$W_FLAGS describes the state of the memory.

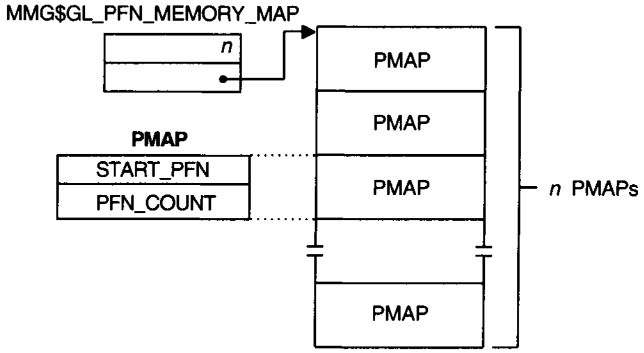
On a Galaxy instance or a system with noncontiguous physical memory, a PMAP array is allocated from nonpaged pool. MMG$GL_PFN_MEMORY_MAP contains the number of valid PMAPs in the array. The longword following it contains the address of the array. MMG$GL_MAX_MEM_FRAGMENTS contains the maximum number of PMAPs in the array. The valid PMAPs are at the beginning of the array; the invalid PMAPs are zeroed. On a non-Galaxy system with contiguous physical memory, MMG$GL_PFN_MEMORY_MAP contains zero.

The macro VALID_PFN tests whether a given PFN represents a page of memory or the number of a nonexistent page. The macro generates code that scans the PMAP array to determine whether the page exists. The macro returns TRUE if the page has a PFN database entry.

Figure 2.16 shows the PMAP array. PMAP$L_START_PFN contains the number of the first PFN in the group, and PMAP$L_PFN_COUNT, the number of PFNs.

On a Galaxy system, PMAPs are also used to describe I/O space that is private to this instance. MMG$GL_IO_MEMORY_MAP and the following longword describe the array of I/O PMAPs.

Figure 2.16 PFN Memory Map (PMAP) Array



2.5.2 Reserved Memory Descriptors

A system manager or similarly privileged user runs the System Management (SYSMAN) utility to build a Reserved Memory Registry file, called SYS$SYSTEM:VMS$RESERVED_MEMORY.DATA. The file describes physical memory to be reserved for use by demand zero memory-resident global sections and by privileged kernel mode applications.

Memory reserved through entries in this file is not included in AUTOGEN's calculation of fluid page count, ensuring a better-tuned system. AUTOGEN uses fluid page count to size the system page file, maximum number of processes, and maximum working set size. Additionally, the Reserved Memory Registry enables a memory-resident global section to be created from one or more chunks of contiguous aligned memory suitable for granularity hints. This kind of allocation is possible only during system initialization, while large chunks of contiguous physical pages are still unused. Note that physical contiguity is not guaranteed: bad pages of memory or gaps in physical memory, for example, may result in a reservation that occupies several noncontiguous segments of physical memory.

Each entry in the file is identified by a name and, optionally, a UIC group number. Memory for an entry can be preallocated at system initialization or allocated later when the section is actually created. Preallocated memory can be zeroed on demand when each page is first accessed or through a combination of the idle loop and code that creates memory-resident global sections. An entry describing a memory-resident global section can specify that the section is mapped by shared page tables, in which case memory is also reserved for the page tables.

An entry can request allocation in the memory of a particular RAD; a reservation can be extended into multiple RADs.

During system initialization, OpenVMS reads the Reserved Memory Registry file and attempts to act on each record in it. First, it builds one or more reserved memory descriptors (RMDs) for each record. A record that describes a section mapped by shared page tables requires a second RMD for the page table memory. A reservation that requested allocation in multiple, specific RADs has a record and thus an RMD for each requested RAD.

OpenVMS then attempts to process the RMDs. For each RMD, it reduces the system's fluid page count, PFN$GL_PHYPGCNT, by the size of the reservation. It checks whether the sum of the fluid page count and the minimum number of modified and free pages is large enough to accommodate a maximum-size outswapped process. If not, it sets RMD$V_RESERVE_ERROR in RMD$L_FLAGS and writes the value SS$_INSFLPGS in RMD$L_ERROR_STATUS. It outputs the RMD name and size with the error message

```
%RESMEMINIT-I-ALLOCFAIL, Fluid page check failed on reservation
```

It restores the previous value of PFN$GL_PHYPGCNT. Then it continues with the next RMD because there may be sufficient memory on the system to satisfy other, smaller requests.

If an RMD describes a section to be preallocated, OpenVMS tries to allocate contiguous aligned memory of the specified size from the RAD requested, if any. If that RAD has no memory, OpenVMS tries to allocate memory without regard to RAD.

- If it could allocate no contiguous pages anywhere, it sets RMD$V_RESERVE_ERROR in RMD$L_FLAGS and writes the value SS$_INSFRPGS in RMD$L_ERROR_STATUS. It outputs the RMD name and size with the error message

  ```
  %RESMEMINIT-I-ALLOCFAIL, Failed to allocate PFNs for reservation
  ```

  and restores the previous value of PFN$GL_PHYPGCNT.

- If it was partially successful, it records the number of pages allocated in RMD$L_PFN_COUNT and creates a new RMD and inserts it in the list after the current one. The new RMD is identical to the current one except that the number of pages already allocated is subtracted from the new RMD$L_PFN_COUNT.

  While processing the new RMD, OpenVMS may create another new RMD if it is unable to allocate enough contiguous physical memory.

- If it was partially or completely successful, it initializes the PFN database record for each page, setting its type to PFN$C_UNKNOWN.

In either case, it continues with the next RMD.

Figure 2.17 shows the layout of an RMD. As shown in the figure, RMDs are linked together into a list whose head is MMG$GL_RES_MEM_FLINK and MMG$GL_RES_MEM_BLINK. The list is ordered by UIC group and name. Access to the list is synchronized with the MMG spinlock. RMDs are allocated from nonpaged pool.

RMD$PS_FLINK and RMD$PS_BLINK link the RMD into the list. RMDW_SIZE, RMDB_TYPE, and RMD$B_SUBTYPE form the standard dynamic data structure header.

RMD$L_FLAGS records choices made when the Reserved Memory Registry entry was created, for example, whether the reservation is for a group or system global section. It also describes the state of the entry, for example, whether pages have been zeroed and whether they are in use.

In the case of an allocated reservation, RMD$L_FIRST_PFN contains the starting PFN.

RMD$L_PFN_COUNT contains the number of PFNs in this reservation, and RMD$L_IN_USE_COUNT, the number currently in use.

If RMD$V_RESERVE_ERROR is set, RMD$L_ERROR_STATUS records the error status resulting from the last attempt to allocate pages for this reservation.

During the zeroing of a preallocated section, RMD$L_ZERO_PFN records the number of the next page to be zeroed.

RMD$T_NAME contains an ASCII counted string identifying the reserved memory. RMD$L_GROUP contains the UIC group code.

RMD$L_RAD contains the number of the RAD requested for allocation.

For further information on the Reserved Memory Registry, consult the *OpenVMS System Manager's Manual, Volume 1: Essentials.*

2.5.3 PFN Database

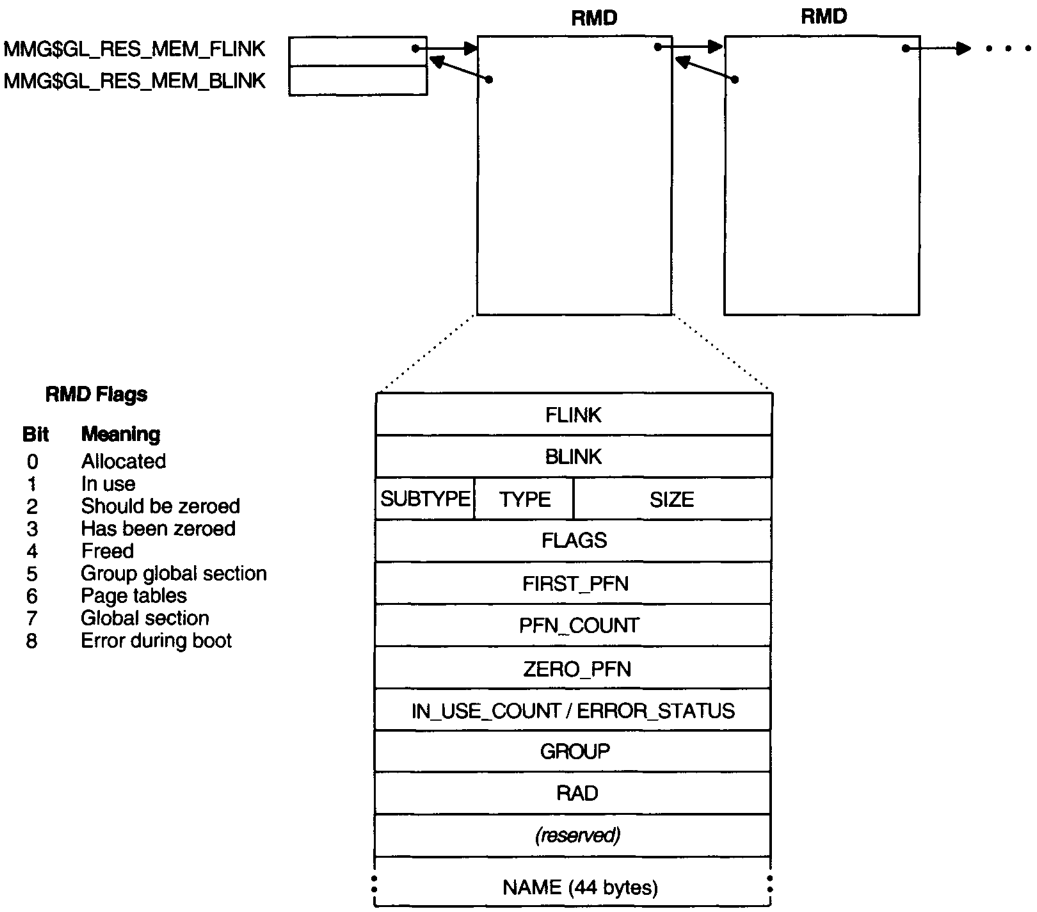
The PFN database contains information about each page of physical memory. It includes pages reserved for use by the console subsystem or for memory-resident global sections but excludes memory reserved through the PHYSICAL_MEMORY SYSGEN parameter.

The fact that this information must be accessible while the page is in use means that it cannot be stored in the page itself. In addition, the caching strategy for the free and modified page lists requires physical page information to be accessible even when pages are not currently active and valid.

During system initialization, SYSBOOT determines the range of PFNs present on the system and records the highest PFN in MMG$GL_MAX_NODE_PFN. It also determines the maximum PFN possible on the system, because on a Galaxy platform there may be shared memory not yet known to this system, and records it in MMG$GL_MAXPFN. If the PHYSICAL_MEMORY SYSGEN parameter has been used to reserve memory for customer use, the highest PFNs are reserved for customer use and subtracted from MMG$GL_MAX_NODE_PFN.

It calculates the size of the PFN database needed to describe pages from PFN 0 to the contents of MMG$GL_MAXPFN.

Figure 2.17 Layout of a Reserved Memory Descriptor (RMD)



RMD Flags

Bit	Meaning
0	Allocated
1	In use
2	Should be zeroed
3	Has been zeroed
4	Freed
5	Group global section
6	Page tables
7	Global section
8	Error during boot

SYSBOOT creates enough S2 space for the entire database, allocates physical memory for PFN records to describe instance-local memory (pages from PFN 0 to MMG$GL_MAX_NODE_PFN), and zeros that memory. To the extent possible, it allocates the physical memory as granularity hint regions. In each PFN record for a PFN that exists, SYSBOOT initializes the type to PFN$C_UNKNOWN.

Because physical memory can be noncontiguous, the database may have physical gaps but can still be accessed virtually as an array of records from PFN 0 to MMG$GL_MAXPFN.

The OpenVMS Alpha PFN database consists of one 40-byte record, or structure, for each page of physical memory. Its starting address is stored in cell PFN$PQ_DATABASE, and access to it is synchronized by the MMG spinlock. Each field in the record contains a specific item of information about that physical page of memory.

Figure 2.18 shows the layout of the Alpha PFN database record. To save space, many bytes within it have multiple uses. The quadword name PFN$Q_BAK, for example, is also called PFN$Q_BAK_PRVPFN to represent an alternative use. Furthermore, as shown, that quadword can also be made up of two longwords, one with two uses.

Table 2.1 summarizes the information in each PFN database record. In listing the names of the fields in each record, the table omits the prefix PFN$*x*_, where *x* identifies the data type.

Although the OpenVMS VAX PFN database contains the same basic information, its organization is quite different: it consists of multiple arrays, each containing a different type of information with an element for each page.

Typically, executive code accesses more than one kind of information about a particular page when it accesses the PFN database. Thus, to make cache hits more likely and to improve performance, the OpenVMS Alpha PFN database is organized as a set of records, each one holding different types of information about the same page.

The page frame number of a physical page is the index of its record in the PFN database; that is, information about a particular page is located by indexing the PFN database with the PFN of that page. To transform a PFN into the address of its PFN database record, the OpenVMS Alpha system provides a macro called PFN_TO_ENTRY for use by kernel mode code. This transformation currently consists of multiplying the size of each record by the page's PFN and adding that offset to the base address of the PFN database.

An example of the use of this macro in MACRO-32 code follows:

```
PFN_TO_ENTRY -                          ;Get PFN database record address
          PFN = R0,-                    ;PFN of interest (input)
          ENTRY = R15                   ;Address of its record (output)
EVAX_LDQ  R2,PFN$Q_BAK(R15)             ;Get backing store information
MOVL      PFN$L_PAGE_STATE(R15),R3 ;Get page state information
```

An example of the use of this macro in C code follows:

```
/* Get MMG protected info about PFN */

pfn = pte_contents.pte$v_pfn;
entry = pfn_to_entry (pfn);
wslx = entry->pfn$l_wslx_qw;
wsle = *(ctl$gq_wsl + wslx);
```

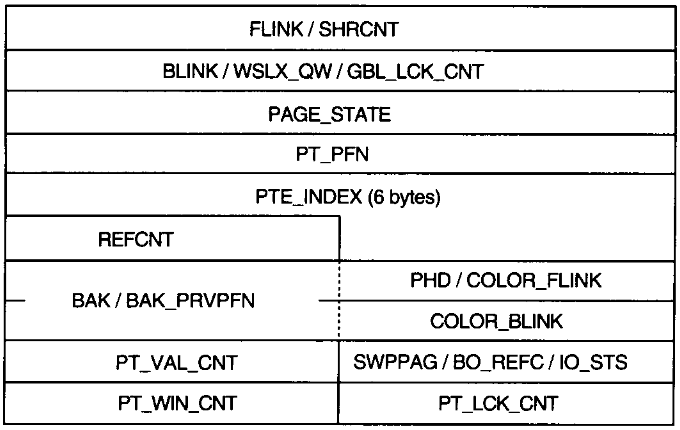
Most of the information in a PFN record for a page relates to the current virtual use of that physical page. For a physical page that has no connection to a virtual page, the only meaningful information is found in the PFNL_FLINK, PFNL_BLINK, PFNL_PAGE_STATE, PFNL_COLOR_FLINK, and PFN$L_COLOR_BLINK fields.

The sections that follow describe the various lists on which a PFN can be found and the fields that make up each PFN record.

Table 2.1 PFN Database Record Fields

Contents	Name	Size	Comments
Forward link	FLINK	Longword	Figure 2.19; overlays SHRCNT
Global share count	SHRCNT	Longword	Overlays FLINK
Backward link	BLINK	Longword	Figure 2.19; overlays WSLX_QW, GBL_LCK_CNT
Working set list index	WSLX_QW	Longword	Overlays BLINK, GBL_LCK_CNT
Global lock count	GBL_LCK_CNT	Longword	Overlays BLINK, WSLX_QW
Physical page state and type	PAGE_STATE	Longword	Figure 2.21
PFN of mapping page table	PT_PFN	Longword	
Page table space index of PTE	PTE_INDEX	Quadword	
Reference count	REFCNT	Word	Partially overlays PTE_INDEX
Address of associated PHD	PHD	Longword	Overlays COLOR_FLINK, BAK, BAK_PRVPFN
Forward link in page color list	COLOR_FLINK	Longword	Overlays PHD, BAK, BAK_PRVPFN
Backward link in page color list	COLOR_BLINK	Longword	Overlays BAK, BAK_PRVPFN
Backing store address	BAK	Quadword	Figure 2.22; overlays BAK_PRVPFN, PHD, COLOR_FLINK, COLOR_BLINK
Private PFN listhead link	BAK_PRVPFN	Quadword	Overlays BAK, PHD, COLOR_FLINK, COLOR_BLINK
Swap file page number	SWPPAG	Word	Overlays BO_REFC, IO_STS
Buffer object reference count	BO_REFC	Word	Overlays SWPPAG, IO_STS
I/O error status	IO_STS	Word	Overlays SWPPAG, BO_REFC
Page table count of valid WSLEs	PT_VAL_CNT	Word	
Page table count of locked WSLEs	PT_LCK_CNT	Word	
Page table count of window pages	PT_WIN_CNT	Word	

Figure 2.18 Layout of a PFN Database Record



2.5.3.1 PFN Lists

A physical page that is available to OpenVMS and that is not occupied by a valid virtual page is commonly in one of five lists: the free, modified, bad, untested, or zeroed page list. Note that preallocated memory registered in the Reserved Memory Registry is not on any list, nor is memory reserved to the console.

The heads of these lists are in an array of longwords that begins at global location PFN$AL_HEAD. Their list tails are in the array PFN$AL_TAIL. Each array has eight elements: the first for the free page list, the second for the modified page list, the third for the bad page list, the fourth for the untested page list, and the last for the zeroed page list. The fifth, sixth, and seventh elements are unused. The arrays are indexed by the PFN$V_LOC bits in the PFN$L_PAGE_STATE field.

A third longword array, PFN$AL_COUNT, is also indexed by page type. An entry typically contains the number of pages in the corresponding list.

These page lists must all be doubly linked because a page is often removed from the middle of the list. The links cannot exist in the pages themselves because the contents of each page must be preserved. The forward link (FLINK) and backward link (BLINK) fields in a PFN database record implement the links for each page. The PFN$L_FLINK field contains the PFN of the successor page, and the PFN$L_BLINK field that of the predecessor page.

A zero in one of the link fields indicates the end of the list rather than being a pointer to physical page 0. This is one reason that physical page 0 cannot be used in any dynamic function. Another reason is that the representation of invalid demand zero PTEs assumes that a PFN of zero can never appear in an invalid PTE (see Figure 2.12). However, it can be used by a system virtual page that is always resident. Physical page 0 is usually in an area of memory reserved for the console subsystem.

Figure 2.19 shows an example of pages on the free page list, along with their corresponding PFN$L_FLINK and PFN$L_BLINK fields. The PFN$L_PAGE_STATE location bits for each page contain zero, indicating that the physical page is on the free page list. The PFNs are hexadecimal.

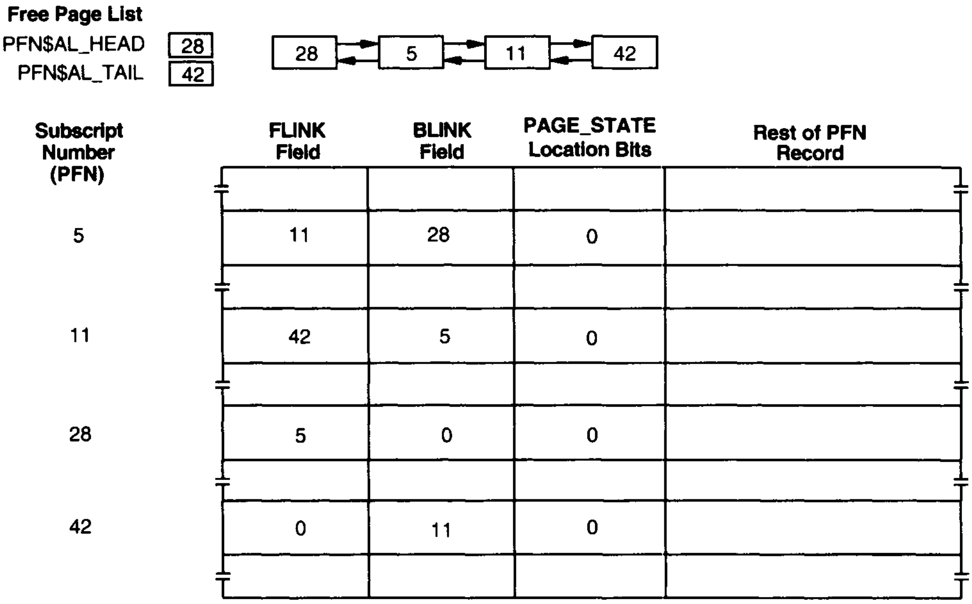
The number of pages on the zeroed list is in cell MMG$GQ_ZEROED_LIST_COUNT. The SYSGEN parameter ZERO_LIST_HI specifies the maximum number of pages on this list. The list serves as a source of demand zero pages that have already been zeroed. When there is no computable kernel thread to execute, the idle loop removes from the free page list a page that has no connection to any virtual page and clears it. After clearing the entire page, the idle loop inserts it on the zeroed page list.

2.5.3.1.1 Colored and RAD-Specific Page Lists

Since OpenVMS Version 6.1, there may be multiple free and zeroed page lists. That release added support for a feature known as page coloring.

Historically, OpenVMS has allocated PFNs randomly in response to processes' demand paging. Consequently, program execution results in random access to physical memory as references cross page boundaries. For some applications, less random references improve performance. Performance can suffer, for example, if a loop crosses page boundaries and addresses in two of the pages in the loop have the same cache index: executing code in one of those sections would cause the other to be removed from cache.

Figure 2.19 Example of Free Page List Showing Linkage Method



Subscript Number (PFN)	FLINK Field	BLINK Field	PAGE_STATE Location Bits	Rest of PFN Record
5	11	28	0	
11	42	5	0	
28	5	0	0	
42	0	11	0	

Page coloring is a technique for addressing such cache thrashing problems and the resulting performance loss. Page coloring classifies allocatable pages by the low-order bits of their PFN. The SYSGEN parameter PFN_COLOR_COUNT, whose default value is 1, specifies how many classifications should exist. The number of classifications determines the number of low-order PFN bits used to classify pages. The default value of 1 effectively disables page coloring.

The only pages classified this way are unencumbered free pages (pages with no ties to virtual pages, that is, with no backing store connections) and zeroed pages. The classification is used when a physical page is being allocated in response to demand paging: instead of allocating the next available page from the free page list, a page whose color matches the faulting virtual address is allocated.

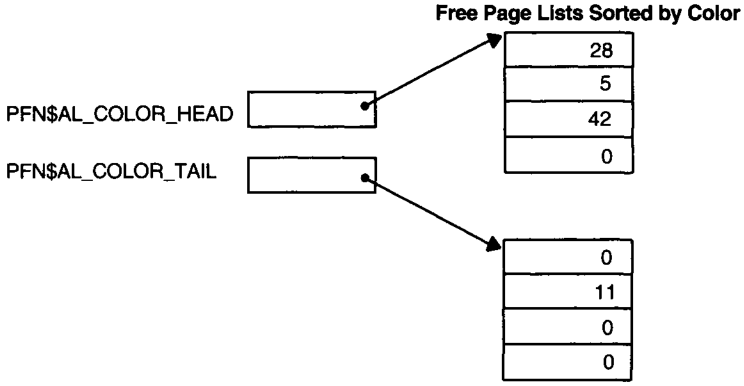
On a NUMA platform with RAD support enabled (RIH$V_RAD_ENABLE set in SYSGEN parameter RAD_SUPPORT), free and zeroed pages are classified by RAD instead of by color. The classification is used when a physical page is being allocated in response to demand paging: instead of allocating the next available page from the free page list, a page may be allocated from the RAD associated with the process, for example. The number of classifications is the maximum number of RADs present on the platform.

At system initialization, on a non-NUMA platform, the value of PFN_COLOR_COUNT is rounded up, if necessary, to a power of 2. On a NUMA platform, the maximum number of RADs is rounded up, if necessary, to a power of 2. That many free and zeroed page listheads and tails are allocated from nonpaged pool. PFN$AL_COLOR_HEAD and PFN$AL_COLOR_TAIL are eight-longword arrays corresponding to PFN$AL_HEAD and PFN$AL_TAIL for pages to which coloring has been applied. The arrays must be large enough to cover all possible values of the page state location bits (see Section 2.5.3.6.1) to which page coloring might be applied. Currently only the entries corresponding to free and zeroed pages are used. Rather than containing PFNs, each entry contains a pointer to an array of longwords indexed by page color.

An unencumbered free page is inserted not only at the head of the standard free page list but also onto the free page list corresponding to its color or its RAD, through PFN$L_COLOR_FLINK and PFN$L_COLOR_BLINK. Similarly, a zeroed page is inserted onto both the zeroed page list and the zeroed page list corresponding to its color.

Figure 2.20 shows the free page list of Figure 2.19 sorted onto free page lists corresponding to their page colors. The figure assumes that all pages are unencumbered, that the value of PFN_COLOR_COUNT is 4, and that therefore the low-order two bits of the PFN are the color value. The PFNs are hexadecimal.

Figure 2.20 Example of Colored Page Lists



2.5.3.1.2 Untested Page List

On some Alpha systems, the console tests all of memory before passing control to the OpenVMS bootstrap program. The time to test all pages on a system that supports a very large memory may be prohibitively long. On such systems some of the testing is therefore left to the operating system, to enable the system to become operational sooner.

On such systems, a flag in the SYSGEN parameter MMG_CTLFLAGS controls when OpenVMS performs testing of memory beyond that needed to boot the system. If MMG$V_BOOTIME_MEMTEST is 1, all previously untested memory is tested in EXE$INIT (see Chapter *Operating System Initialization and Shutdown*). By default the flag is 0 and memory testing is deferred; untested memory is put on the untested page list. The idle loop performs deferred memory testing, placing a tested page on either the free page list or the bad page list. Deferred memory testing is also performed when necessary, for example, when allocating a physical page that has not yet been tested.

2.5.3.1.3 Private PFN Lists

In addition to the lists previously described, a page can be on a private PFN list. This mechanism enables a kernel mode application to manage a list of PFNs, perhaps for a system space cache that must occupy a fixed amount of physical memory. Such an application would call MMG_STD$ESTABLISH_FREEPFN_LIST, in module MEM_ALLOC, to create the list and populate it with free PFNs. The routine allocates a private PFN (PRVPFN) listhead from nonpaged pool and links it into a list of such listheads at MMG$GL_PRVPFN_FLINK and MMG$GL_PRVPFN_BLINK. The kernel mode application is responsible for synchronizing access to its private PFN list and for returning PFNs from the list in response to a request from OpenVMS when memory is scarce.

2.5.3.2 PFN$L_FLINK and PFN$L_BLINK Fields

These fields link a page into the master free, modified, master zeroed, bad, untested, or a private PFN page list. The PFN$L_FLINK field contains the PFN of the successor page, and the PFN$L_BLINK field that of the predecessor page.

2.5.3.3 PFN$L_SHRCNT Field

PFN$L_SHRCNT, the share count field in a PFN database record, counts the number of process-private PTEs that are mapped to a particular global page. When the share count for a particular page goes from 0 to 1, the PFN$W_REFCNT field is incremented. Further additions to the share count do not affect the reference count.

As the global page is removed from the working set of each process mapped to the page, the share count is decremented. When the share count finally reaches zero, the PFN$W_REFCNT field for the page is also decremented.

In the case of a global page mapped only by a shared page table, the share count is 1, regardless of how many processes are mapped to the global section.

Because a physical page with a nonzero share count cannot be on one of the page lists, the forward and backward link fields are not needed for such a page. The PFN$L_SHRCNT field overlays the PFN$L_FLINK field.

Process-private page table pages also use the PFN$L_SHRCNT field as a reference count for the page table page. The count includes all valid or transition PTEs in the page, excluding window pages. When this count goes from zero to nonzero, the page table page is dynamically locked into the process working set. Chapter 4 describes the share count in further detail.

2.5.3.4 PFN$L_WSLX_QW Field

The working set list index field, PFN$L_WSLX_QW, for a valid page contains a quadword index from the beginning of the working set list to the WSLE for that page. The PFN$L_WSLX_QW field is used, for example, during the deallocation of a page of memory. If the virtual page is valid, the WSLE that describes it must be altered. Without the contents of the PFN$L_WSLX_QW field, it would be necessary to search the working set list to locate the WSLE.

In OpenVMS versions prior to Version 7.0, the WSLX was a longword index from the beginning of the PHD. The meaning of this field changed to reflect the fact that a WSLE is now a quadword. Basing the index on the beginning of the working set list rather than the beginning of the PHD facilitates the possible removal of the working set list from the PHD in some future release.

Because a physical page in a working set is not on one of the page lists, the PFN$L_FLINK and PFN$L_BLINK fields are not needed for such a page. The PFN$L_WSLX_QW field overlays the PFN$L_BLINK field.

2.5.3.5 PFN$L_GBL_LCK_CNT Field

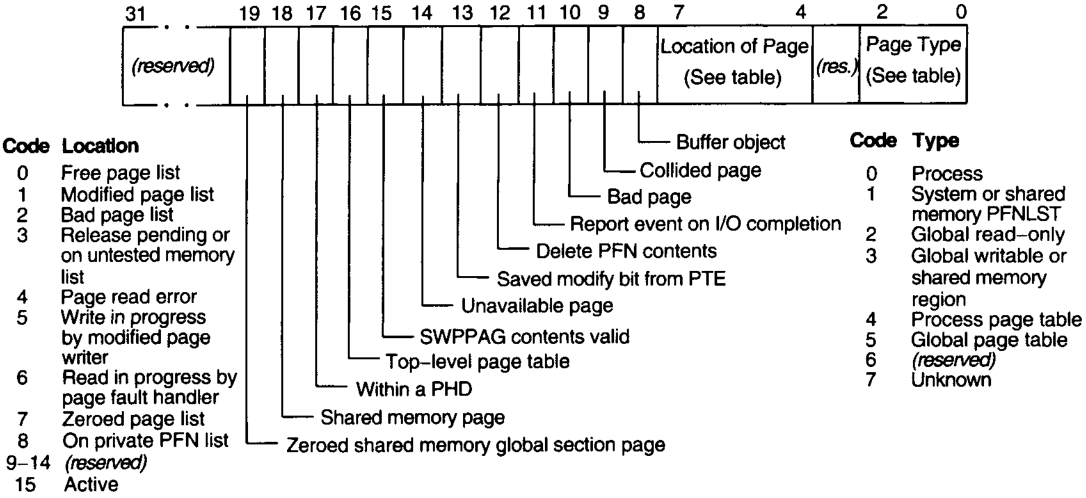
The PFN$L_GBL_LCK_CNT field for a global page counts the number of times the page has been locked into memory. The field is initialized to 0.

In prior versions, this information was kept in the PFN$L_WSLX field. In OpenVMS Version 7.0, the PFN$L_GBL_LCK_CNT field was added as an overlay of the PFN$L_WSLX_QW field to formalize the additional use of the field.

2.5.3.6 PFN$L_PAGE_STATE Field

The PFN$L_PAGE_STATE field, shown in Figure 2.21, indicates the state, type, and location of a physical page.

Figure 2.21 Contents of PFN$L_PAGE_STATE Field



Code	Location
0	Free page list
1	Modified page list
2	Bad page list
3	Release pending or on untested memory list
4	Page read error
5	Write in progress by modified page writer
6	Read in progress by page fault handler
7	Zeroed page list
8	On private PFN list
9–14	*(reserved)*
15	Active

Code	Type
0	Process
1	System or shared memory PFNLST
2	Global read–only
3	Global writable or shared memory region
4	Process page table
5	Global page table
6	*(reserved)*
7	Unknown

As shown in the figure, bits <2:0> of this field identify the type of virtual page that occupies the corresponding physical page, for example, whether it is a process or system page or page table page. The page fault handler, swapper, and other parts of the executive take actions dependent on page type.

The sections that follow describe the location codes and status bits in the page state field.

2.5.3.6.1 Page State Location Codes

Bits <7:4> contain the page location code, indicating, for example, whether the page is on the free page list or valid in a working set.

Several page location codes require further explanation:

- Release pending means that the virtual page has been removed from a working set but still has a nonzero reference count. When the reference count is decremented to zero at I/O completion, the physical page will be placed on the free or modified page list.

An untested page is one not yet tested by console or operating system. Because there is no overlap in the code that deals with release pending and untested pages, PFN$C_RELPEND and PFN$C_UNTESTED have the same numeric value. This conserves space in the page state field. Section 2.5.3.1 has further information on untested pages.

- Page read error means that a nonrecoverable I/O error occurred during an attempt to read the virtual page from its backing store into the physical page. During postprocessing of the I/O request, when the error is noted, this code is stored in the PFN$L_PAGE_STATE field, and the I/O error status is stored in PFN$W_IO_STS. Consequently, when the page is later refaulted, the page fault handler will signal a page read error exception, passing the I/O error status in bits <15:0> of the first argument of the signal array.
- Write in progress means that the modified page writer has initiated I/O to write the page to its backing store.
- Read in progress means that the page fault handler has initiated I/O to read the page from its backing store.
- A page on the zeroed page list is a free page that was completely zeroed when the system would otherwise have been idle. Such a page can be allocated as a demand zero page, as a page most of whose contents are zero (for example, an L1PT page), or as a section page only partly represented on disk.
- A page on a private PFN list is being managed by a kernel mode application independently of OpenVMS. Section 2.5.3.1 has further information.

2.5.3.6.2 Page State Status Bits

The PFN$L_PAGE_STATE field has a number of status bits.

The buffer object bit (PFN$V_BUFOBJ), when set, means the page is part of a buffer object or is a page table page that maps a buffer object (see Section 2.6).

The collided page bit (PFN$V_COLLISION) is set when a page fault occurs for a virtual page that is already being read in from its backing store (one whose location bits show it as read in progress). This can happen, for example, if multiple kernel threads from the same or different processes fault the same page. It can also happen if a kernel thread in a page fault wait is interrupted for asynchronous system trap (AST) delivery and then reexecutes the instruction that triggered the page fault. When I/O completes for a page with this bit set, I/O postprocessing code clears the bit and reports the system event collided page available for all kernel threads in the collided page wait state. Chapter *Scheduling* describes system events. Collided pages are discussed briefly in Chapter 4.

The bad page bit (PFN$V_BADPAG) is set when an uncorrectable memory error occurs trying to access the page in memory. The page will be put onto the bad page list when it is deallocated.

The report event bit (PFN$V_RPTEVT) is set when an attempt is made to delete a virtual page that cannot be deleted immediately, for example, because the modified page writer is writing the page to its backing store. The executive places the kernel thread into a page fault wait. When the modified page writer's I/O completes, it reports a page fault completion system event. When the kernel thread is placed back into execution, the page deletion proceeds.

The delete contents bit (PFN$V_DELCON) is set to indicate that the connection between a physical page and its virtual contents should be severed. When the reference count of a physical page whose delete contents bit is set becomes zero, the PFN$L_PT_PFN and PFN$Q_PTE_INDEX fields in its PFN database record are cleared. The physical page is then put at the front of the free page list, where it will be reused before pages that are still associated with virtual pages. Such a page is also put on the free page list corresponding to its color.

The saved modify bit (PFN$V_MODIFY) is set to indicate a modified page that has not yet been written to its backing store. It determines whether a physical page is put on the free page list or the modified page list when the page's reference count reaches zero. The modify bit is set under a number of circumstances, including the following:

- On the first attempt to write to a writable virtual page, the executive sets the modify bit in its PTE. When a virtual page is removed from a working set, the modify bit in its PTE is logically ORed into the saved modify bit in the PFN$L_PAGE_STATE field for the physical page. The modify bit must be recorded in the PFN$L_PAGE_STATE field because that bit in an invalid PTE has another use as the TYP1 bit.
- When a page is used as a direct I/O read buffer, the executive routine that locks down pages, MMG$IOLOCK, in module IOLOCK, sets the modify bit in its PTE. When the page is removed from the process's working set, the OR operation described in the previous item sets the modify bit in PFN$L_PAGE_STATE.
- When a copy-on-reference page is faulted into a working set, the executive sets the modify bit in the PFN$L_PAGE_STATE field of the physical page. Thus, even if the virtual page is not modified while it is valid, when the page is removed from the working set, the physical page is inserted into the modified list. This ensures that it will be written to page file backing store, from where it will be read on a subsequent page fault.
- When a demand zero page is faulted into a process's working set, the modify bit in PFN$L_PAGE_STATE is set.
- When a buffer object is created, the modify bit is set in PFN$L_PAGE_STATE for each of its pages.

The unavailable page bit (PFN$V_UNAVAILABLE), when set, means the page is not available for the operating system to use. Typically, it means that the page is in a memory region reserved for the console subsystem's use.

The swap page valid bit (PFN$V_SWPPAG_VALID) is set by the swapper to indicate that the contents of PFN$W_SWPPAG represent a swap file page number. Chapter 6 has further details.

The top-level page table bit (PFN$V_TOP_LEVEL_PT) is set to indicate that the page is the most significant page table in the hierarchy and further that its PFN$L_PHD field identifies the PHD of the process associated with this page table hierarchy. Currently, with a three-level hierarchy, this bit is set for a PFN containing an L1PT. Memory management code tests this bit to determine whether it has reached the top of the hierarchy following the pointer (PFN$L_PT_PFN) from one page table to the page table that maps it.

The balance slot (PFN$V_SLOT) bit is set to indicate that the page is part of some process's PHD. Historically, all PHD pages had a page type of process page table (PFN$C_PPGTBL), even those that were not page tables. To minimize code changes, PHD pages continue to have this page type, as do actual page table pages. The value of this bit distinguishes the two types.

The shared memory page bit (PFN$V_SHARED) and the zeroed shared memory page bit (PFN$V_ZEROED) describe pages in a Galaxywide section.

2.5.3.7 PFN$L_PT_PFN and PFN$Q_PTE_INDEX Fields

When assigning a physical page to a new use, the executive examines the PTE that maps it to determine whether the page is a transition page and still pointed to by a PTE associated with its previous use. If the field contents are not zero, the executive must take steps to sever the connection between the physical page and its previous use. The term *back pointer* is used to refer to the location of the PTE that maps a physical page.

In versions prior to OpenVMS Version 7.0, PFN$L_PTE contained the system virtual address of the PTE mapping the PFN. Furthermore, with process-private page tables mapped in system space, the PTE address itself was enough to locate the PHD, given fixed-size balance set slots.

With page tables mapped only in page table space, unique specification of a PTE is more complex: the process associated with the PTE may not be current at the time memory management code needs to examine the PTE. PFN$Q_PTE_INDEX and PFN$L_PT_PFN replace PFN$L_PTE and enable the PTE to be located.

PFN$L_PT_PFN contains the PFN of the page table page with the PTE that maps this PFN. The PFN$Q_PTE_INDEX field contains the quadword index from the base of page table space to the PTE containing this PFN.

To locate the PTE when the process is current, memory management code simply uses the contents of PFN$Q_PTE_INDEX to index the current page table space. To locate the PTE when the process is not current, memory management code takes the following steps:

1. It maps the PFN in PFN$L_PT_PFN into system space.
2. It multiplies PFN$Q_PTE_INDEX by 8.

3. It indexes the mapped page using the low-order bits of the product in step 2 as a byte offset into the page.

There are several instances, most notably within the modified page writer, when it is necessary to obtain a PHD address from a physical page's PFN. To do this, memory management code iterates the preceding steps, traversing the PFN$L_PT_PFN links from a lower level page table up to the next level page table, until it reaches one with the PFN$V_TOP_LEVEL_PT bit set. That page table page's PFN database field PFN$L_PHD contains the address of the PHD.

If no virtual page is mapped to a physical page, its PFN$Q_PTE_INDEX field contains zero. The PFN$Q_PTE_INDEX field for a non-copy-on-reference global page contains the index of the global PTE from the beginning of the global page table. The PFN$L_PT_PFN for a global page contains the number of the global page table page.

A page that is part of a buffer object (see Section 2.6) is not assigned to another use until the buffer object is deleted. The page remains associated with the buffer object across process outswap and inswap. For that reason, no attempt is made to keep correct PFN$L_PT_PFN information. Instead, the PCB of the process that created the buffer object is stored in its PFN$L_PT_PFN field. If the page is used for direct I/O, I/O completion code will be able to locate the PHD address from the PCB to unlock the PHD.

2.5.3.8 PFN$L_PHD Field

This field is used only for a PFN that is a top-level page table. It contains the system virtual address of the PHD belonging to the process whose top-level page table it is.

During deletion of the PHD, the swapper uses this field to record the balance set slot index (see Section 2.8.2) corresponding to the PHD.

This field overlays PFNQ_BAK, PFNQ_BAK_PRVPFN, and PFN$L_COLOR_FLINK, which are unused for a top-level page table page.

2.5.3.9 PFN$W_REFCNT Field

The PFN$W_REFCNT field counts the number of reasons a physical page should retain its current contents. For instance, the count is incremented if a page is in a process working set; is part of a direct I/O buffer with I/O in progress; or is part of a buffer object or a page table page mapping a buffer object. The field is initialized to zero.

I/O completion and working set replacement use the same mechanism to decrement the reference count. When the reference count goes to zero, the physical page is released to the free or modified page list, depending on the saved modify bit in its PFN$L_PAGE_STATE field. Manipulations of the reference count are illustrated and described in greater detail in Chapter 4.

2.5.3.10 PFN$L_COLOR_FLINK and PFN$L_COLOR_BLINK fields

These fields link an unencumbered free page (one with no connection left to a virtual page) into a colored free or zeroed page list. Such a page has no valid BAK contents; thus these links overlay the PFN$Q_BAK field as well as the PFN$BAK_PRVPFN field.

On a NUMA platform with RAD support enabled, these fields link an unencumbered free page into a RAD-specific free or zeroed page list.

2.5.3.11 PFN$Q_BAK Field

The PFN$Q_BAK field contains the backing store location for the virtual page occupying a physical page. When a physical page is assigned to another use, the PTE, if any, that currently maps the page must be updated. The executive replaces information about the location of the virtual page in memory (the PFN of the physical page that contains it) with information about its location in mass storage copied from the PFN$Q_BAK field.

Figure 2.22 shows the possible contents of a PFN$Q_BAK field. The shaded bits in each form are either reserved or bits whose contents are irrelevant for that form of backing store information.

Before a demand zero or copy-on-reference page is assigned actual page file backing store, the system page file index field contains FF_{16} to indicate no assigned page file. In addition, the field can contain zero.

2.5.3.12 PFN$Q_BAK_PRVPFN Field

PFN$Q_BAK_PRVPFN is used only for a page managed through a private PFN list. It contains the address of the PRVPFN listhead used to manage the page.

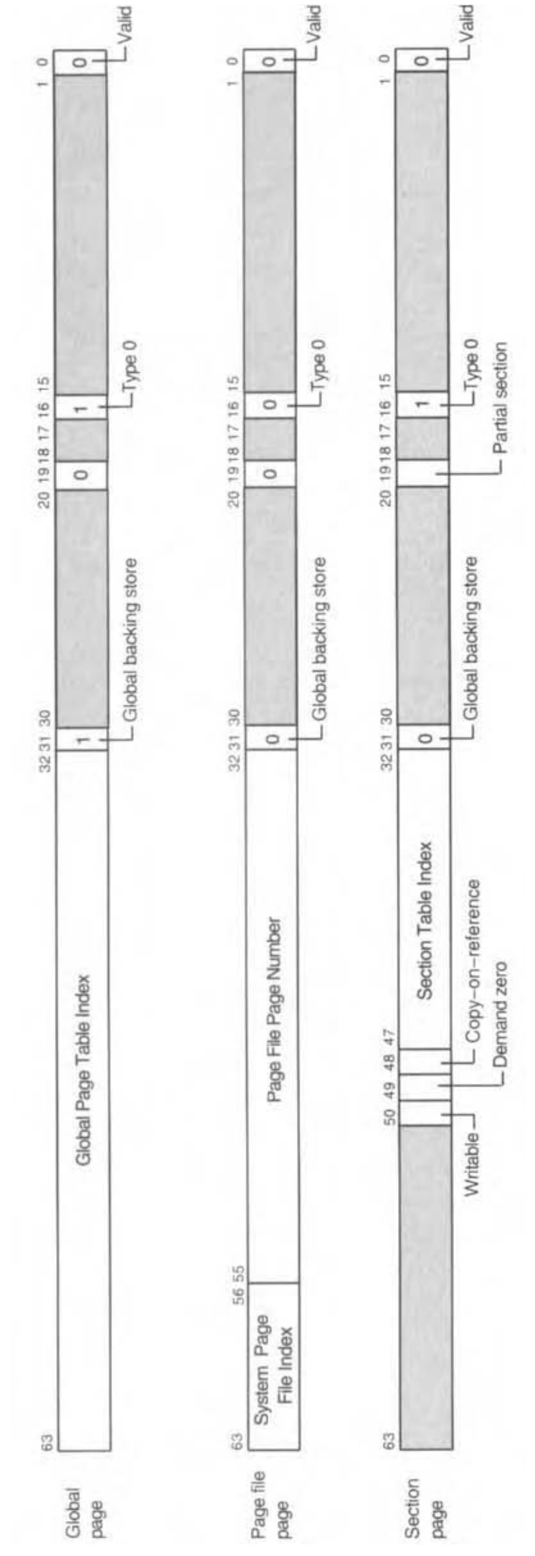
2.5.3.13 PFN$W_SWPPAG Field

The swap file page number field, PFN$W_SWPPAG, supports the outswap of a process with read I/O in progress. When such an outswap occurs, the swapper sets bit PFN$V_SWPPAG_VALID in PFN$L_PAGE_STATE and records in PFN$W_SWPPAG the page offset in the process body part of the swap slot into which the locked down page should be written.

When the swapper I/O is completed, the locked page is marked release pending. When the original I/O is completed, the I/O postprocessing routine sees that the page is in the release pending state and has the saved modify bit set, and inserts the page on the modified page list. The modified page writer checks the PFN$V_SWPPAG_VALID bit and, if it is nonzero, diverts a modified page from its normal backing store address to the designated location in the swap file.

Because a physical page in a buffer object or a page table page that maps a buffer object cannot be outswapped, this field is not needed to describe such a page. The PFN$W_BO_REFC field and PFN$W_IO_STS overlay the PFN$W_SWPPAG field.

Figure 2.22 Possible Contents of PFN$Q_BAK Field



2.5.3.14 PFN$W_BO_REFC Field

Another form of reference count is kept for buffer object pages (see Section 2.6). The buffer object reference count field, PFN$W_BO_REFC, counts the number of buffer objects mapping the page. The field is initialized to –1. When the reference count for a particular buffer object page goes from –1 to 0, its PFN$W_REFCNT field is incremented. Further additions to the buffer object reference count do not affect the PFN reference count. For a page table page that maps one or more buffer objects, PFN$W_BO_REFC contains the number of buffer object pages mapped by the page table page.

Because a physical page in a buffer object or a page table page that maps a buffer object cannot be outswapped, the PFN$W_SWPPAG field is not needed to describe such a page. The PFN$W_BO_REFC field overlays the PFN$W_SWPPAG and PFN$W_IO_STS fields.

2.5.3.15 PFN$W_IO_STS Field

This field is used only for a page that incurs an I/O processing error during an attempt to fault it in from backing store or inswap it. The I/O error status is recorded in it. Section 2.5.3.6.1 has additional information.

The PFN$W_IO_STS field overlays the PFN$W_SWPPAG and PFN$W_BO_REFC fields.

2.5.3.16 PFN$W_PT_VAL_CNT Field

This field is used only for page table pages. It contains the number of valid working set list entries mapped by that page table page. A value of –1 for this field means the page maps no such pages or is not a page table.

Prior to OpenVMS Version 7.0, this information was kept in the PHD PTWSLEVAL array.

Chapter 4 contains further information.

2.5.3.17 PFN$W_PT_LCK_CNT Field

This field is used only for page table pages. It contains the number of locked pages mapped by that page table page. A value of –1 for this field means the page table page maps no such pages or is not currently in use as a page table.

Prior to OpenVMS Version 7.0, this count was included in the PHD PTWSLELCK array.

Chapter 4 contains further information.

2.5.3.18 PFN$W_PT_WIN_CNT Field

This field is used only for page table pages. It contains the number of window pages and memory-resident global section pages mapped by that page table page. A window page is a virtual page that is a double mapping of a physical page. For example, a virtual page in a section mapped by PFN is a window page. A value of –1 for this field means the page table page maps no such pages or is not currently in use as a page table.

In the case of a shared L3PT mapping shared global section pages, PFN$W_PT_WIN_CNT counts the number of pages of global section mapped. In the case of a process-private L2PT mapping shared L3PTs, PFN$W_PT_WIN_CNT counts the number of shared L3PTs mapped by this L2PT.

Prior to OpenVMS Version 7.0, this count was included in the PHD PTWSLELCK array.

Chapter 4 contains further information.

2.6 Buffer Objects

A buffer object is a special kind of I/O buffer. When a buffer object is created, the pages that compose it are locked into physical memory and may be mapped in process-private space, system space, or both, depending on how the buffer object was created. A system buffer object, new with OpenVMS Version 7.3, is mapped only in system space. An ordinary buffer object consists of process-private pages. A global buffer object consists of global pages. Typically, a very large global buffer object is not double-mapped in system space.

I/O can be initiated to or from the buffer with minimal overhead using the Perform Fast I/O ($IO_PERFORM) system service; in particular, there is no need to probe the buffer or lock its pages into memory. The body, PHD, and page tables of a process with I/O in progress to a buffer object can all be swapped. Although an L3PT page table page that maps a buffer object is locked in memory, it is not locked into the process's working set.

A buffer object is created when a process requests the Create Buffer Object ($CREATE_BUFOBJ_64) system service (see Chapter 3), specifying an existing process-private address range to be mapped as a buffer object.

Each buffer object is described by a nonpaged pool data structure called a buffer object descriptor (BOD), shown in Figure 2.23. All the BODs for buffer objects created by a particular process are linked together in a list whose head is in the process's PCB$Q_BUFOBJ_LIST field. The listhead for system buffer object BODs is in the system PCB.

BODs enable the memory management subsystem to keep track of the buffer objects the process created and their associated system virtual addresses. When an image exits, the executive examines the process's BOD list and deletes buffer objects that still exist.

BOD$L_FLINK and BOD$L_BLINK link a BOD into the PCB list of others by the same process.

BOD$W_SIZE and BOD$B_TYPE are the standard dynamic data structure header fields. A BOD has a type of DYN$C_BOD.

BOD$L_ACMODE contains the owner access mode of the buffer object.

Figure 2.23 Layout of a Buffer Object Descriptor (BOD)

FLINK		
BLINK		
(reserved)	TYPE	SIZE
ACMODE		
SEQNUM		
REFCNT		
FLAGS		
PID		
BASEPVA		
BASESVA		
VA_PTE		
PAGCNT		

BOD$L_SEQNUM contains a sequence number identifying the buffer object.

BOD$L_REFCNT contains the number of references to the buffer object and the number of reasons the buffer should not be deleted. Creating a buffer object establishes the reference count as 1. The reference count is incremented when an I/O request is processed that uses the buffer and decremented when the I/O completes.

BOD$L_FLAGS contains flag bits that describe the section:

- BOD$V_DELPEN, when set, means that a request to delete the buffer object has been made and that its deletion is pending.
- BOD$V_NOQUOTA, when set, means that the buffer object creation was requested from an inner access mode with flag CBO$V_EXMAXLIM set to specify that limit checking is to be bypassed.
- BOD$V_NOSVA, when set, means the buffer is not double-mapped in system space.
- BOD$V_S2_WINDOW, when set, means the buffer is double-mapped in S2 space.
- BOD$V_SYSBUFOBJ, when set, means the buffer is mapped only in system space and is a system buffer. Record Management Services (RMS) uses a system buffer object to map global buffer descriptors and the structures that synchronize access to them.

BOD$L_PID contains the internal ID of the process that created the buffer object.

BOD$L_PAGCNT contains the number of pages in the buffer object.

BOD$PQ_BASEPVA contains the process virtual address at which the buffer object is mapped.

If BOD$V_NOSVA is clear, the buffer is double-mapped into system space. BOD$PQ_BASESVA contains the system virtual address at which the buffer object is mapped.

If BOD$V_S2_WINDOW is clear, the buffer is mapped into S0/S1 space, and BOD$PQ_VA_PTE contains the system virtual address of the SPT window PTE that maps the first page of the buffer object. If BOD$V_S2_WINDOW is set, the buffer is mapped into S2 space, and BOD$PQ_VA_PTE contains the page table space address of the PTE that maps the first page of the buffer object.

SYSGEN parameter MAXBOBMEM limits the amount of physical memory buffer objects can consume.

2.7 Data Structures for Global Pages

The treatment of global pages is somewhat different from that of process-private pages; the executive must keep additional systemwide data to describe global pages and sections. The sections that follow describe these data structures.

2.7.1 Global Section Descriptor

Global sections are created by various OpenVMS system services, for example, Create and Map Section ($CRMPSC), Create Permanent Global Demand Zero Section ($CREATE_GDZRO), and Create Permanent Global Disk File Section ($CREATE_GFILE). Such services can be requested directly from a user image or indirectly through the Install utility.

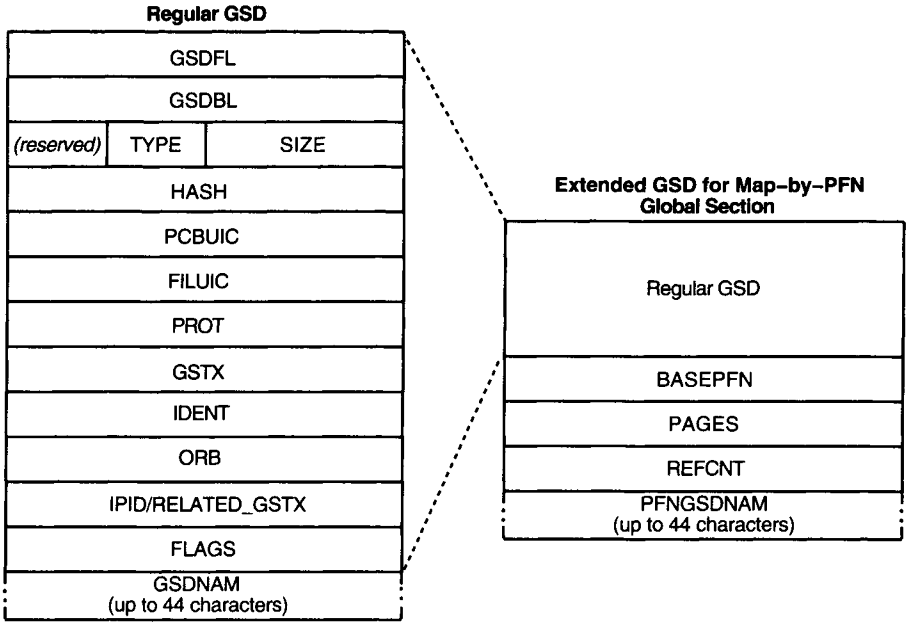
A special type of global section, new with OpenVMS Version 7.1, is a memory-resident global section. The pages of such a section do not page and are not backed up by a section file. The global pages are permanently valid. Like any other global section, a memory-resident global section is described by a GSD.

Optionally, a process can map memory-resident global sections with shared page tables, thereby using the same L3PTs as other processes to map the global section. Those shared page tables themselves make up a type of global section called a global page table section, which like any other global section, is described by a GSD. Thus a memory-resident global section with shared page tables is described by two GSDs.

Figure 2.24 shows the layout of a GSD. A GSD associates the global section name to its GSTE. The information in the GSD is only used when some process attempts to map to or delete the section. The page fault handler does not use this data structure.

GSD$L_GSDFL and GSD$L_GSDBL link a GSD into one of several GSD lists maintained by the system. All system global section descriptors are linked into one list, whose listhead is formed by global cells EXE$GL_GSDSYSFL and EXE$GL_GSDSYSBL. Group global section descriptors (independent of group number) are linked into the other list, at EXE$GL_GSDGRPFL and EXE$GL_GSDGRPBL. Note

Figure 2.24 Layout of a Global Section Descriptor (GSD)



that a GSD for a global page table section is not linked to any of these GSD lists except immediately prior to its deletion.

When a request is made to delete a global section to which processes are still mapped, its GSD is removed from its current list and inserted into a list of delete-pending GSDs, the listhead of which is at EXE$GL_GSDDELFL and EXE$GL_GSDDELBL.

GSDs representing Galaxywide shared sections are linked to separate lists: EXE$GL_GLXGRPFL and EXE$GL_GLXGRPBL for group sections, and EXE$GL_GLXSYSFL and EXE$GL_GLXSYSBL for system sections.

The mutex EXE$GL_GSDMTX (see Chapter *Synchronization Techniques*) serializes access to all these GSD lists.

GSD$W_SIZE and GSD$B_TYPE are the standard dynamic data structure fields.

GSD$L_HASH contains a hashed representation of the global section name. Comparing hash values rather than section names speeds up a search for a global section with a particular name.

GSD$L_PCBUIC is the user identification code (UIC) from the software PCB of the creating process. GSD$L_FILUIC is the UIC of the owner of the section file.

GSD$L_PROT is currently unused. Information about the protection on the global section is stored in the object rights block associated with the global section.

GSD$L_GSTX contains the global section table index (GSTX) for the section's GSTE.

GSD$L_IDENT contains the version identification of the global section. The value is specified by the requestor of the system service that created the global section. In the case of a global section created for image installed /SHARE, the Install utility gets the information from the image header of the image being installed.

GSD$L_ORB contains the address of the associated object rights block (ORB). In the case of a section that maps a file, the global section shares the ORB associated with the open file.

When a process requests that a global section be deleted, its internal process ID is copied to GSD$L_IPID. If the global section is writable, when all its modified pages have been written, the modified page writer queues an AST to that process to perform the cleanup and deletion of the global section.

For a memory-resident global section with shared page tables, GSD$L_RELATED_GSTX contains the index of the GSTX of the related global page table section.

GSD$L_FLAGS contains flags that describe the section. They are based on the ones in the GSTE (see Figure 2.7).

GSD$T_GSDNAM contains a counted ASCII string that is the section's name.

A PFN-mapped global section has no associated GSTE; its pages are not paged. Such a section has an extended GSD, as shown in Figure 2.24. In the extended GSD, GSD$L_BASEPFN contains the starting PFN of the section. GSD$L_PAGES specifies its size in pages. GSD$L_REFCNT specifies how many PTEs map to this section. GSD$T_PFNGSDNAM, rather than GSD$T_GSDNAM, contains the section name.

2.7.2 Global Section Table Entries

The section table in the system header serves a second purpose. When a global section is created, a section table entry that describes the global section file is allocated from the section table in the system header. Because of this use, the system header's section table is usually called the global section table (GST).

The layout of a GSTE is nearly identical to the layout of a PSTE. Figure 2.7 illustrates both kinds of section table entry.

A GSTE is accessed in a similar way to a PSTE, with a positive index from the bottom of the GST (see Section 2.1.3.3). The GSTX in the GSD is such an index, associating a GSD with a GSTE.

When a memory-resident global section with shared page tables is created, two GSTEs are created: one for the global section itself, and one for the related global page table section.

Allocation and deletion of GSTEs are synchronized by the MMG spinlock.

2.7.3 Global Page Table

Like other L3PTs, the global page table (GPT) describes the state of the pages it maps. Unlike the others, the GPT is not accessed by the TB miss PALcode routine to load an entry into the translation buffer. It is only accessed by OpenVMS Alpha memory management routines. The GSD mutex synchronizes allocation and deallocation of global page table entries (GPTEs). The MMG spinlock synchronizes modification of GPTEs allocated to a global section.

The global page table is located at the low-address end of S2 space, allocated during system initialization. Its initial size depends upon the SYSGEN parameter GBLPAGES, the number of expected global *pagelets*. (Historically, the parameter was defined in terms of VAX pages. To facilitate application porting and maximize cross-platform compatibility, the units of the parameter continue to be pagelets.) MMG$GQ_MAX_GPTE contains the address of the highest GPTE.

Free GPTEs are maintained in a list whose structure is described in Section 2.3.6. The listhead for free GPTEs is MMG$GQ_FREE_GPT.

Each global page is mapped by one GPTE. When a process maps a portion of its address space to a global section, its process-private PTEs that map the section are initialized to the GPTX form of PTE (see Figure 2.12). A global section is mapped by a set of contiguous GPTEs, one for each global page plus two additional GPTEs. The two additional GPTEs, one at the beginning of the set and one at the end, are cleared and serve as stoppers to limit modified page write clustering (see Chapter 4).

During system operation, GPTEs are allocated when an image is installed /SHARE or an application creates a global section. If there are insufficient GPTEs to map a new global section, the system manager can increase the value of GBLPAGES, which is dynamic. A subsequent attempt to create a global section would result in expanding the GPT if all the following are true:

- Expanding it by the necessary number of GPTEs doesn't increase its size over the value of GBLPAGES.
- There is sufficient free S2 address space contiguous with the existing GPT.
- Decreasing fluid pages by the growth of the GPT leaves a result larger than four times the largest swap image.

GPT pages are created as demand zero pages. Once faulted, each remains resident unless the GPT is contracted by that page. (This is a change from earlier versions, in which global page table pages were pageable.) The GPT can be contracted by one or more pages under the following circumstances:

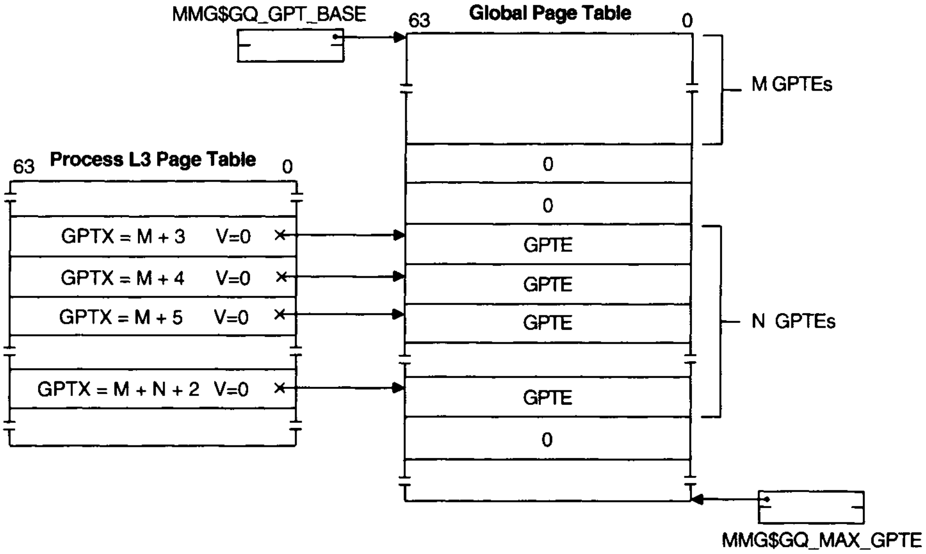
- Global pages are deleted and GPTEs thus deallocated.
- One or more pages of GPT at the high-address end map no global pages.
- The system manager has reduced GBLPAGES, and the current size of the GPT is larger.

The executive locates a specific GPTE in the GPT using a GPTX as a quadword context index from the contents of MMG$GQ_GPT_BASE, the cell that contains the starting address of the GPT.

The process-private PTE that maps the first page of a global section contains the GPTX of the GPTE that maps the first page in the global section. Each successive process-private PTE contains the next higher GPTX, so that each PTE effectively points to the GPTE that maps that particular page in the global section.

The relation between process-private PTEs and GPTEs is shown in Figure 2.25. In the figure, the first M GPTEs are in use for other sections, and the global section shown is mapped by N + 2 GPTEs beginning with GPTE M + 2. GPTE M + 1, GPTE M + 2, and GPTE M + 2 + N + 1 are stoppers.

Figure 2.25 Relation Between Process-Private PTEs and GPTEs



When a process first accesses an invalid global section page, it incurs a page fault. Determining that the invalid page is a global page, the page fault handler indexes the GPT with the GPTX to locate the GPTE that describes the global page.

2.7.4 Global Page Table Entries

Each page in every type of global section is described by a GPTE. Even pages in memory-resident global sections with shared page tables are described by GPTEs. This simplifies the memory management code and enables a process to map the global section using process-private page tables, perhaps to obtain read-only access. Moreover, each global page table section is itself described by GPTEs.

GPTEs are restricted to the following forms of PTE, illustrated in Figure 2.26. The shaded bits in each GPTE are either reserved or bits whose contents are irrelevant for that form of GPTE.

- The GPTE can be valid, indicating that the global page is in at least one process working set or that it is a valid page in a memory-resident global section.
- The GPTE can indicate a page in some transition state. The corresponding PFN$L_PAGE_STATE field identifies the transition state.
- For a global page in a global section file, the GPTE contains a global section table index.
- The GPTE can indicate a demand zero page in a global page-file section.
- The GPTE can indicate a global page-file section page that has been created and is in use.

Note that there are no protection bits in a GPTE. When a global section is mapped, the executive determines values for the protection bits based on section flags, the access mode from which the section is mapped, and the FLAGS argument to the system service that maps the section.

When a global page is faulted in, the bits shown in Figure 2.26 labeled Global and Global Write are incorporated into the PFN$L_PAGE_STATE field for the physical page and the entry corresponding to the page in the working set lists of processes that have mapped to it.

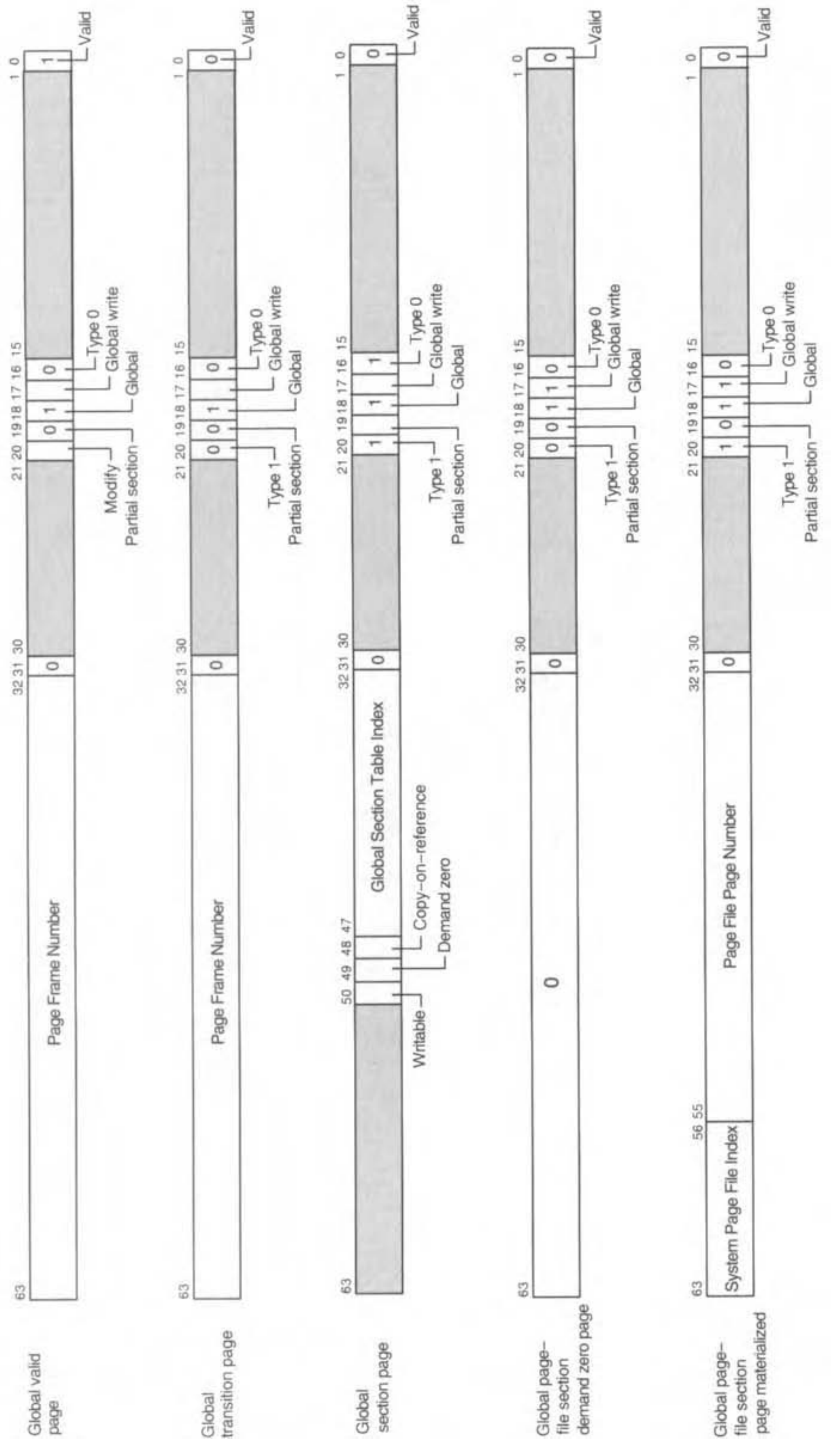
Invalid GPTEs that are unused and available for allocation are linked together in a list from listhead MMG$GQ_FREE_GPT. The organization of the list is the same as that of free system page table entries (see Section 2.3.5.6).

2.7.5 Relations among Global Section Data Structures

Figure 2.27 shows the relations among the GSD, GSTE, and GPTEs for a given section on a system with a page size of 8 KB (for simplicity, the figure omits the stopper GPTEs):

- The central shaded structure is the GSTE (see Figure 2.7 for its layout) within the GST. The first longword in the GSTE points to the GSD.
- The virtual page number field (which contains J in Figure 2.27) contains the GPTX of the first GPTE that maps this section.
- The global section consists of K pages and, in this example, none of them is partial. That is, the number of mass storage blocks in the section is an integral multiple of the number of blocks per page. Given a system with a page size of 8 KB, the SEC$L_UNIT_CNT field in the GSTE therefore contains the number of pages in the section multiplied by 16, the number of mass storage blocks per page.
- The GSD contains a GSTX that locates the GSTE.

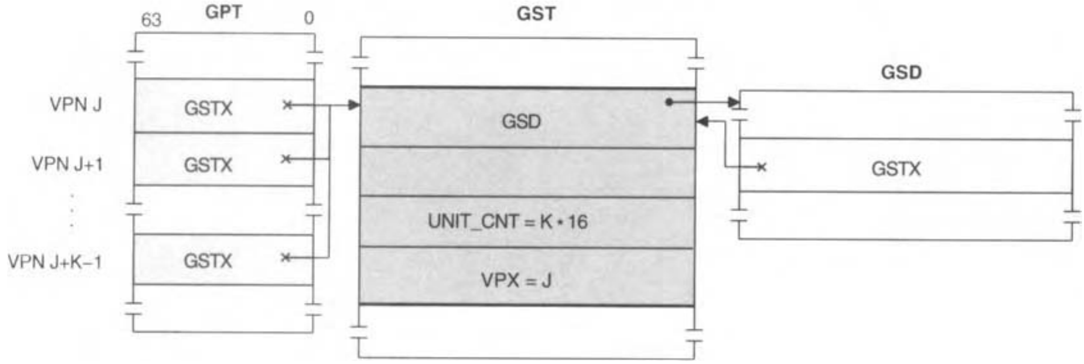
Figure 2.26 GPTE Formats



- The original form of each GPTE contains the same GSTX found in the GSD. When any given GPTE is either valid or in transition, the GSTX is stored in the corresponding PFN database record PFN$Q_BAK field. Note that a GPTE for a global page-file section contains a page file backing store address.

The allocation and initialization of global section data structures are described along with the create and map global section system services in Chapter 3.

Figure 2.27 Relations among Global Section Data Structures



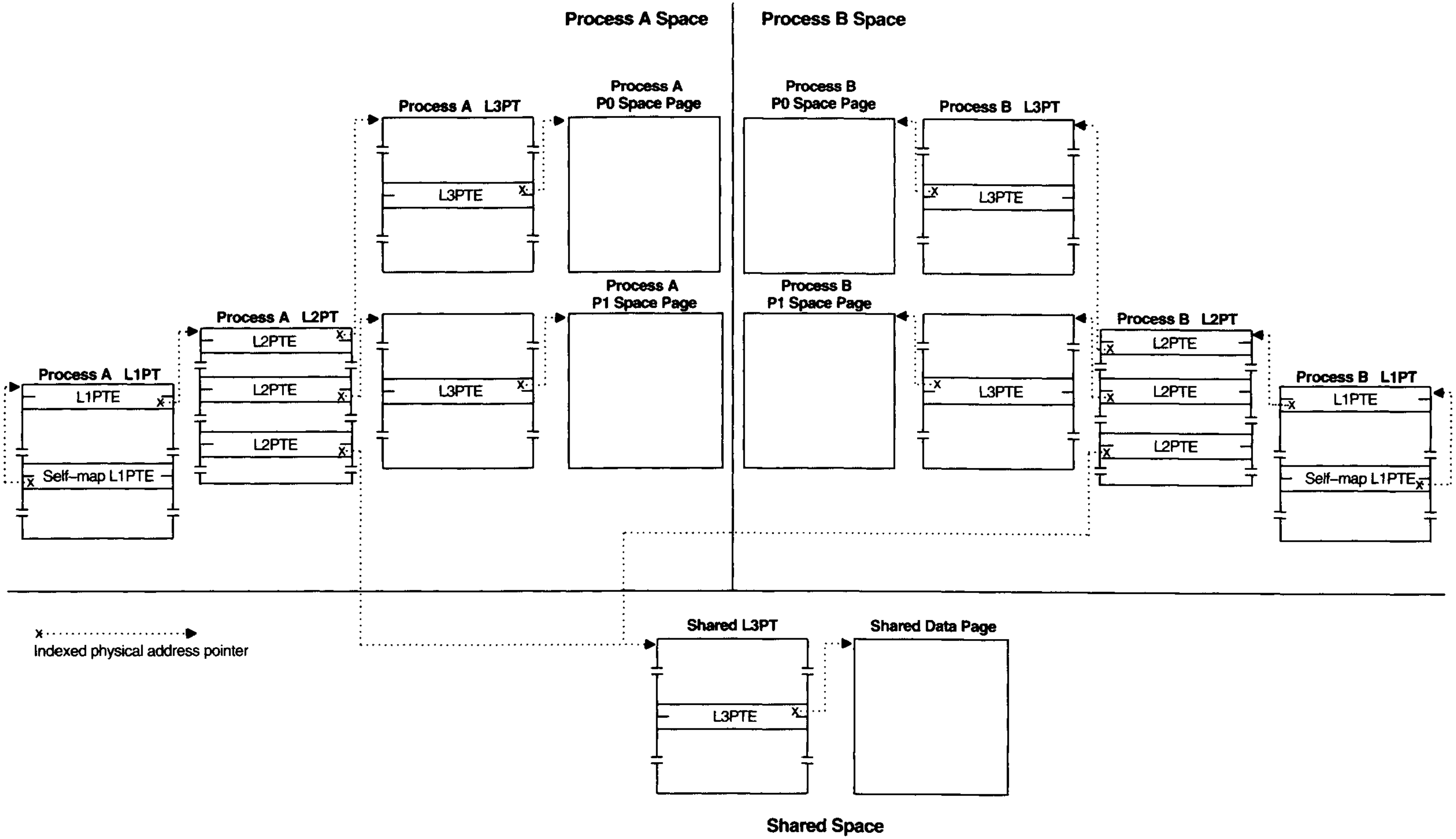
2.7.6 Global Shared Page Table Sections

To map a memory-resident global section using shared page tables, a process first creates a shared page table region by requesting the $CREATE_REGION_64 service. This ensures that the global section can be mapped starting at a suitable virtual address, one mapped by the first L3PTE in an L3PT.

If the memory-resident global section has not already been created, the process then requests either the Create Permanent Global Demand Zero Section ($CREATE_GDZRO_64) system service or the Create and Map to Global Demand Zero Section ($CRMPSC_GDZRO_64) system service to create and map the section as well as the global page table section. To map the sections, another process would request either the Map to Global Section ($MGBLSC_64) system service or the $CRMPSC_GDZRO_64 system service.

Figure 2.28 shows part of the page table hierarchies of two independent processes. Each L1PT maps a process-private L2PT and process-private L3PTs for P0 and P1 space, and each also maps a shared L3PT that maps a memory-resident global section. For simplicity, the figure shows only one shared data page and omits the shared L2PTs and L3PTs that map system space.

Figure 2.28 Process-Private and Shared Page Tables



2.8 Data Structures for Swapping

The swapper and page fault handler both reference page tables, described in Section 2.3. In addition, the following three data structures are used primarily by the swapper but also indirectly by the page fault handler:

- Balance set slots
- PHD reference count array
- Process index array

The SYSGEN parameter BALSETCNT, whose global cell name is SGN$GL_BALSETCT, specifies the number of elements in each array.

2.8.1 Balance Set Slots

A balance set slot is a piece of system virtual address space reserved for a PHD. The number of balance set slots defines the maximum number of concurrently resident processes.

When the system is initialized, an amount of system virtual address space equal to the size of a PHD times BALSETCNT is allocated. The location of the beginning of the balance set slots is stored in global cell SWP$GL_BALBASE. The size of a PHD in pages is stored in global location SWP$GL_BSLOTSZ.

Figure 2.29 shows this area. Appendix *Size of System and P1 Virtual Address Spaces* describes the calculations performed by SYSGEN to determine the size of the PHD.

Figure 2.29 Balance Set Slots Containing Process Headers

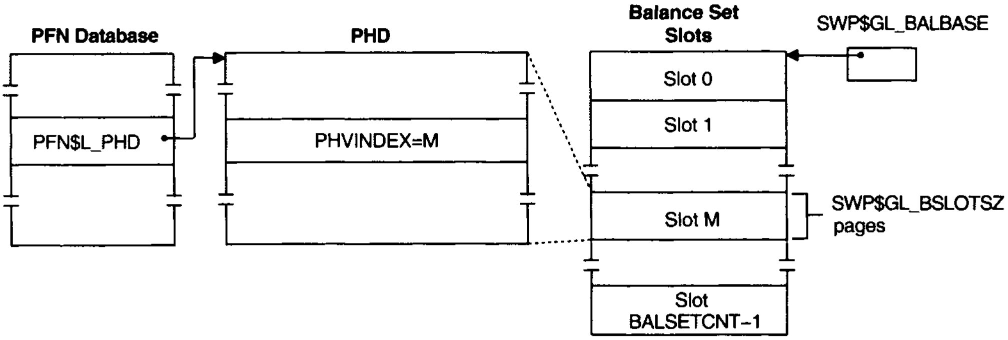
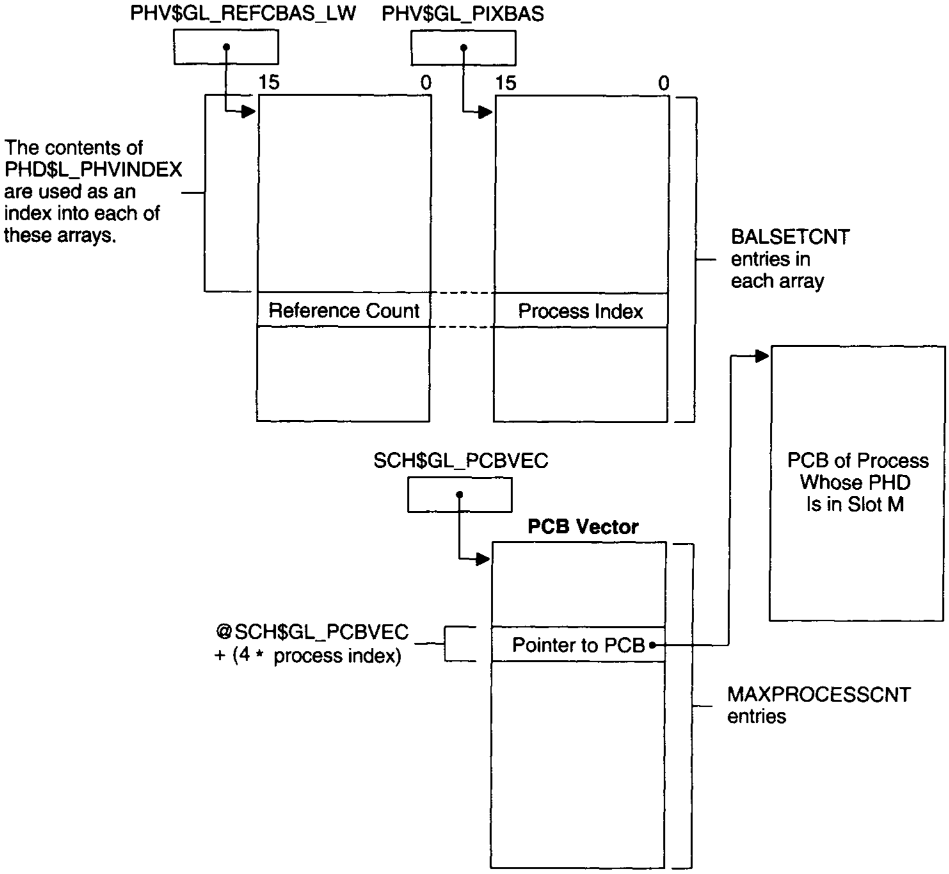


Figure 2.30 Balance Set Slot Arrays


PHV$GL_REFCBAS_LW
PHV$GL_PIXBAS
15
0
15
0
The contents of PHD$L_PHVINDEX are used as an index into each of these arrays.
BALSETCNT entries in each array
Reference Count
Process Index
SCH$GL_PCBVEC
PCB Vector
PCB of Process Whose PHD Is in Slot M
@SCH$GL_PCBVEC + (4 * process index)
Pointer to PCB
MAXPROCESSCNT entries

2.8.2 Balance Set Slot Arrays

As shown in Figure 2.30, the system maintains two arrays describing each process with a PHD stored in a balance set slot. Both arrays are indexed by the balance set slot number occupied by the resident process. The balance set slot number is stored in the fixed portion of the PHD at offset PHD$L_PHVINDEX. Entries in the first array contain the number of references to each PHD. Entries in the second array contain an index into a longword array that points to the PCB for each PHD.

Global cell PHV$GL_REFCBAS_LW contains the starting address of the longword reference count array. Each of its elements counts the number of reasons that the corresponding PHD cannot be removed from memory. Chapter 4 lists the circumstances under which an element is incremented and decremented. A value of –1 in a reference count array element means that the corresponding balance set slot is not in use.

Global cell PHV$GL_PIXBAS contains the starting address of the process index word array. Each of its elements contains an index into the longword array, based at the global pointer SCH$GL_PCBVEC. An element in the longword PCB vector contains the address of the PCB of the process with that process index. Figure 2.30 illustrates how the address of a PHD is transformed into the address of the PCB for that process, using the entry in the process index array.

A value of 0 in the process index array entry means that the corresponding balance set slot is not in use. A value of –1 in a process index array entry means that the process whose PHD used that balance set slot has been deleted and its PHD can be deleted to reclaim physical memory as well as the balance set slot.

If the PHD address is known, the balance set slot index can be calculated or obtained from PHD$L_PHVINDEX. By using this as an index into the process index array, the longword index into the PCB vector is found. The array element in the PCB vector is the address of the PCB, whose PCB$L_PHD entry points back to the balance set slot. Chapter *Process Creation* contains a more detailed description of the PCB vector and its use by the Create Process ($CREPRC) system service.

2.8.3 Comment on Equal-Size Balance Set Slots

In the original VAX/VMS design, a fixed amount of virtual address space was reserved for each balance set slot, despite the fact that PHDs would vary in size a great deal because of differences in section count, working set list size, and virtual address size.

This design simplified memory management code and ensured that if a free balance set slot were available, its size would be sufficient to inswap any process. It also simplified keeping track of the state of PHD pages with fixed-length PHD page arrays.

Another reason for this choice was that it enabled easy calculation of an associated PHD address from a PFN for a private page. PFN$L_PTE contained the system space virtual address of the PTE mapping that page. (Recall that page tables were part of the PHD.) From that, it was easy to identify which PHD contained the PTE.

Although the last reason no longer holds because the page tables have been removed from the PHD, the balance set slots remain equal-size.

2.9 Data Structures Describing the Page and Swap Files

Page and swap files are used by the memory management subsystem to save physical page contents and process working sets. Page files are used to save the contents of modified pages that are not in physical memory. With OpenVMS Version 7.3, the use of page files is recommended, but not absolutely necessary, on a system with enough memory to accommodate all modified pages. The crash dump is written to the primary page file when the system crashes and there is no dump file (see Chapter *Error Handling*).

Swap files save the working sets of processes that are not in the balance set. On today's large memory systems, there is typically little or no swapping. However, on a system that allows very large working sets and that occasionally has a load of many processes, swapping is likely to occur. When it does, having at least one swap file is desirable, because a swap file is typically less fragmented than a page file. Moreover, after several large processes are outswapped into a page file, the page file may be sufficiently full to hinder modified page write clustering.

If there is insufficient space in the swap files, or if there are no swap files, a process can be outswapped to a page file unless SYSGEN parameter NOPGFLSWP is set, inhibiting swapping to page files.

OpenVMS keeps track of total page and swap file usage in the following global cells:

- MMG$GQ_PAGEFILE_PAGES, total number of pages in all installed page files
- MMG$GQ_PAGEFILE_ALLOCS, total number of pages allocated from all installed page files
- MMG$GQ_PAGEFILE_REFS, count of all pages to be backed by page files
- MMG$GQ_SWAPFILE_PAGES, total number of pages in all installed swap files
- MMG$GQ_SWAPFILE_ALLOCS, total number of pages allocated from all installed swap files

The subsections that follow discuss the data structures that describe page and swap files, except for the page file map (PFLMAP) structure, described in Chapter 6.

2.9.1 Page File Control Blocks

Each page and swap file in use is described by a data structure called a page file control block (PFL). A page or swap file can be placed in use either automatically during system initialization or manually through SYSGEN commands. In either case, code in module [SYSINI]INITPGFIL allocates a PFL from nonpaged pool and initializes it.

In addition, each page and swap file is described by two bitmaps:

- The storage bitmap has one bit for each page in the file. A value of 1 means the mass storage blocks equivalent to one page of memory are free; a value of 0 means the page is in use. (Although the unit typically associated with a file is a mass storage block, page and swap files are also described in terms of their capacity to hold pages of memory.)
- Each bit in the directory bitmap represents 16 bits in the storage bitmap that are aligned on a 16-bit boundary. A value of 1 in a directory bit means all the corresponding bits in the storage bitmap are 1; a value of 0 means some or all of them are 0.

Scanning the directory bitmap first to allocate pages, rather than the storage bitmap, improves performance.

Initializing the PFL includes the following operations:

1. The file is opened and a special window control block is built to describe all the file's extents. The special WCB, called a cathedral window, ensures that the memory management subsystem does not have to take a window turn (see Chapter *I/O System Services*), which could lead to a system deadlock.
2. The address of the WCB is stored in the PFL.
3. The sizes of the storage and directory bitmaps are calculated. If the combined sizes of the bitmaps are less than a page of memory, they are allocated from nonpaged pool. Otherwise, they are allocated from S2 space. Both bitmaps are initialized to all 1's.
4. The address of the PFL is stored in an available slot in the page-and-swap-file vector. The slot number is the index that identifies that page or swap file. If the PFL represents a page file, it is also linked into one of four page file lists. Section 2.9.2 contains more details.

Figure 2.31 shows the layout of a PFL.

PFL$W_SIZE and PFL$B_TYPE are the standard dynamic data structure fields. PFL$L_POOLBYTES contains the size of the nonpaged pool request.

PFL$L_PFC is the number of pages to try to cluster on a page read; it sets an upper limit on modified page writer clustering (see Chapter 4). It is the minimum of 1,024 and SYSGEN parameter MPW_WRTCLUSTER.

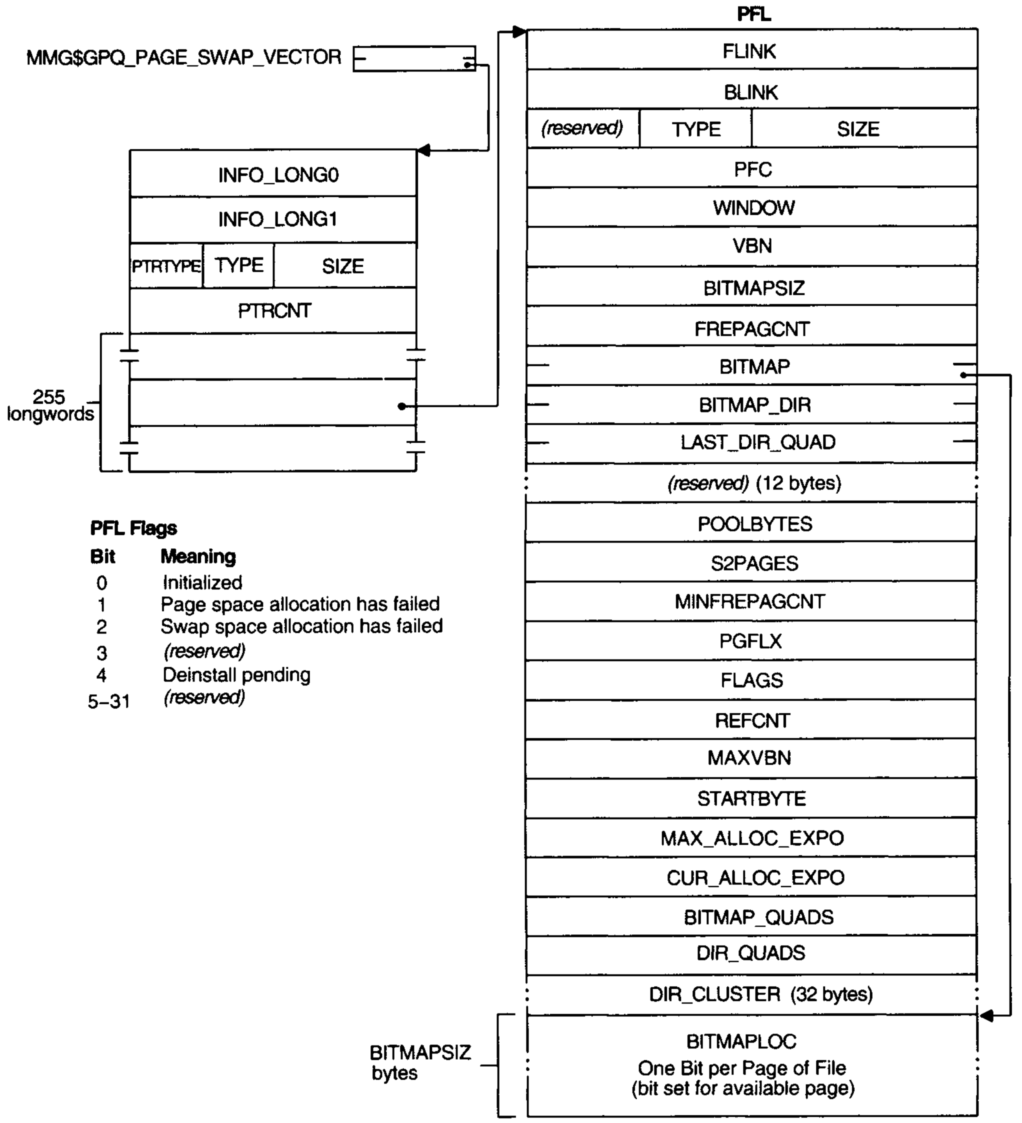
PFL$L_WINDOW is the address of the WCB that describes the mapping extents of the file, which enable file-relative, or virtual, block numbers to be converted to volume-relative, or logical, block numbers.

Generally, PFL$L_VBN contains zero; in the case of a primary page file in use as a crash dump file, it contains a number that reserves enough blocks in the page file to contain the dump. If the dump has already been analyzed, one page's worth of blocks is reserved. If there is a valid unanalyzed dump in the file, PFL$L_VBN contains the size of the dump in blocks rounded up to the next multiple of one page's worth of mass storage blocks. Chapter *Error Handling* discusses use of the primary page file as a dump file.

PFL$L_VBN has an additional use for a page file larger than $FFFFFF_{16}$ pages. When installing such a file, SYSGEN divides it into segments of $FFFFFF_{16}$ blocks. It initializes a PFL for each segment, plus one for the last partial segment. PFL$L_VBN indicates the starting virtual block number of each segment. A page in a segment is represented by the combination of page file index and a page number relative to the start of the segment. The page number is thus small enough to fit into the page file page number portion of a page file backing store PTE. To calculate the actual backing store address, the page file page number is multiplied by the blocks per page and then added to the contents of the associated PFL$L_VBN.

When installing a swap file larger than $FFFFFF_{16}$ pages, SYSGEN similarly divides it into segments of $FFFFFF_{16}$ pages.

Figure 2.31 Page and Swap File Database



PFL Flags

Bit	Meaning
0	Initialized
1	Page space allocation has failed
2	Swap space allocation has failed
3	*(reserved)*
4	Deinstall pending
5-31	*(reserved)*

Note that the PFL contains a WCB field, virtual block number field, and page fault cluster factor field at the same relative offsets they are in a section table entry. Because all fields are present and at the same offsets, page file and section file I/O requests can be processed by common code, independent of the data structure that describes the file being read or written.

PFL$L_FREPAGCNT is the actual number of pages that can be accommodated by the free blocks in the page file. This field is not decremented until the modified page writer actually assigns space to a particular virtual page. It is incremented whenever a page file page is released, either because its virtual page is being deleted or its contents are known to be obsolete. (That is, when a page previously assigned space in a page file is placed into the modified page list, its backing store copy can no longer be regarded as good.)

PFL$L_BITMAPSIZ is the length of the storage bitmap in bytes, and PFL$L_ BITMAP_QUADS, its length in quadwords. If the bitmap is allocated from S2 space, its size in pages is stored in PFL$L_S2PAGES; otherwise, the field contains 0. PFL$Q_ BITMAP contains the address of the start of the storage bitmap.

PFL$Q_BITMAP_DIR contains the address of the start of the directory bitmap. PFL$Q_DIR_QUADS contains its size in quadwords. PFL$L_STARTBYTE identifies the directory bitmap quadword at which the next scan for free blocks should begin. A negative value means that the directory bitmap is all zeros, indicating a fragmented page file, and that the storage bitmap has to be scanned, starting at the quadword that is the complement of the negative value.

PFL$Q_LAST_DIR_QUAD contains the initial value of the last directory quadword.

PFL$L_DIR_CLUSTER is an eight-longword array that describes the directory bitmap and indicates the degree of fragmentation of available space in the file. The first element counts the number of set bits in the directory bitmap. The second element records the number of pairs of bits that are set within each directory quadword. The third element counts the number of groups of four adjacent set bits, and so on through the seventh element, which counts the number of quadwords equal to –1. The eighth element is unused. The counts are initialized when the file is installed and updated as pages are allocated and deallocated.

A directory quadword of –1, for example, has 64 single bits, 32 pairs, 16 groups of four, eight groups of eight, four groups of 16, two groups of 32, and one group of 64. Such a quadword would be represented in each of the seven different counters.

PFL$L_MAX_ALLOC_EXPO contains the largest index into this array of the nonzero element whose group size represents a cluster less than or equal to PFL$L_PFC. For example, a PFL$L_PFC value of 64 pages is represented by four adjacent directory bits, and thus PFL$L_MAX_ALLOC_EXPO is 2.

PFL$L_CUR_ALLOC_EXPO contains the index currently being used for this page file. Initially, it has the same value as PFL$L_MAX_ALLOC_EXPO. When the file becomes full or fragmented and there are no more groups of the size corresponding to PFL$L_MAX_ALLOC_EXPO, PFL$L_CUR_ALLOC_EXPO is decremented. If the file becomes more full or fragmented and there are no more groups corresponding to that size, PFL$L_CUR_ALLOC_EXPO is decremented again. As pages are deallocated and the directory counters updated, PFL$L_CUR_ALLOC_EXPO can be incremented to reflect availability of the next larger group. Keeping track of the largest group in this manner prevents fruitless scans of the directory bitmap.

PFL$L_MINFREPAGCNT is the "low-water mark" for the file and represents the smallest number of pages free during the use of the file.

PFL$L_PGFLX is the index number of the page-and-swap-file vector entry that contains the address of the PFL.

PFL$L_FLAGS contains bits describing the state of the file.

PFL$L_REFCNT contains the number of pages used in the file for paging or swapping.

PFL$L_MAXVBN is the mask applied to a PTE with a page file backing store address. For either type of file, it contains the value FFFFFF_{16}.

If the bitmaps were allocated from nonpaged pool, the storage bitmap begins at offset PFL$L_BITMAPLOC. The directory bitmap follows it.

Chapter 4 describes the use of page files, and Chapter 6, of swap files.

2.9.2 Page-and-Swap-File Vector

Pointers to the PFLs are stored in a nonpaged pool array called the page-and-swap-file vector. The array contains 255 longwords and can accommodate pointers to 254 files, the maximum number of page and swap files that can be in use on the system. SYSGEN parameters SWPFILCNT and PAGFILCNT are obsolete as of OpenVMS Version 7.3; a maximum-size array is always allocated. A header precedes the array. The macro $PTRDEF defines symbolic names for the fields in the header.

A page or swap file is identified by an index number indicating the position of its PFL address in this array. Addresses of swap file PFLs are stored at the beginning of the array, and addresses of page file PFLs, at the end of the array.

During system initialization, the routine EXE$INIT, in module INIT (see Chapter *Operating System Initialization and Shutdown*), allocates and initializes the page-and-swap-file vector.

It initializes PTR$L_INFO_LONG0 to 0 and PTR$L_INFO_LONG1 to 255 to indicate no page or swap files have been installed. When a swap file is installed, OpenVMS increments PTR$L_INFO_LONG0; when a page file is installed, OpenVMS decrements PTR$L_INFO_LONG1. Each longword identifies the most recently used slot number for files of that type. The third longword of the header contains the size of the data structure, a type value of DYN$C_PTR, and a subtype value of DYN$C_PFL. PTR$L_PTRCNT contains the number of elements in the array. The array begins at the next longword.

EXE$INIT stores the address of the structure in MMG$GPQ_PAGE_SWAP_VECTOR. Figure 2.31 shows the use of the page-and-swap-file vector data area to point to PFLs. EXE$INIT initializes each pointer with the address of the null page file control block, the contents of MMG$AR_NULLPFL. For the most part, this address serves as a zero value, indicating that no page or swap file with this index is in use.

A PFL for a page file is also linked into one of four circular lists, in descending order of its free space:

1. Page files that have at least one cluster's (PFL$L_PFC) worth of adjacent pages
2. Page files that do not have even one cluster's worth of adjacent pages but do have set bits in the directory bitmap
3. Page files that do not have set bits in the directory bitmap but do have set bits in the storage bitmap
4. Page files that are full or being deinstalled

A four-longword array at MMG$GA_PAGE_FILES contains the current positions in each list. Pages are allocated from the first PFL found, beginning with the first list. After pages are allocated from a PFL, the pointer into its list is moved to the next PFL in the list, to enable more even page file use. Chapter 4 has additional details.

The SYSINIT process (see Chapter *Operating System Initialization and Shutdown*) places in use SYS$SPECIFIC:[SYSEXE]PAGEFILE.SYS, the primary page file, if it exists. (Any page file installed at a later stage of system initialization or operation is not considered a primary page file, even if it is the first page file installed.) SYSINIT builds a PFL and places its address in the page-and-swap-file vector. The first page file installed has an index value of 254. Additional page files have decreasing index values.

SYSINIT also installs SYS$SPECIFIC:[SYSEXE]SWAPFILE.SYS, if it exists, as the primary swap file. (A swap file installed at a later stage is not a primary swap file, even if it is the first one.) The first swap file installed has index 1. If there is no swap file, index 1 points to the null PFL. Additional swap files have increasing index values.

Any additional page and swap files are placed in use by SYSGEN in response to the commands INSTALL/PAGEFILE and INSTALL/SWAPFILE. Installing page files other than the primary one on different disks allows for balancing the paging load. A system with alternative swap files can support a greater number of processes or processes with larger working sets.

An inactive page or swap file can be removed from use. After a privileged user enters the SYSGEN command DEINSTALL to initiate the removal of a page or swap file, no new allocations are made from it. However, the actual removal from use is deferred until the file is inactive and PFL$L_REFCNT has gone to zero.

2.10 Swapper and Modified Page Writer Page Table Arrays

The I/O subsystem enables an image to make a direct I/O request (direct memory access transfer) to a virtually contiguous buffer. There is no requirement that pages in a buffer be physically contiguous, only virtually contiguous. This capability is called scatter-read/gather-write or, more simply, scatter/gather.

2.10.1 Direct I/O and Scatter/Gather

A combination of hardware and I/O subsystem software supports I/O to and from physically noncontiguous pages. The manner in which this is supported varies with processor type and I/O adapter type.

Regardless of the manner of the support, a direct I/O request typically involves locking the pages of a virtually contiguous buffer into memory. The I/O locking mechanism brings each page into the working set of the requesting process, makes it valid, and increments that page's reference count in its PFN database record to reflect the pending read or write. The buffer is generally described in the I/O request packet (IRP) through three fields:

- IRP$L_SVAPTE has traditionally contained the system virtual address of the first PTE that maps the buffer.
- IRP$L_BOFF and IRP$L_BCNT are used to calculate how many PTEs are required to map the buffer.

A driver processes this I/O request in a manner suitable to the processor and I/O adapter. Typically, the PFNs of the buffer are copied into I/O adapter map registers. After the requested I/O operation is complete, I/O postprocessing code accesses these fields to decrement each PFN's reference count.

As of OpenVMS Version 7.0, process PTE addresses are no longer in system space and are thus inaccessible when that process is not current. Consequently, the exact meaning of IRP$L_SVAPTE has changed. In order for existing device drivers to work with minimal or no change, however, IRP$L_SVAPTE continues to point to a nonpageable system space address at which the PTE contents are accessible.

The I/O subsystem implements two techniques:

- PTE copy method
- PTE window method

The former is used for relatively small buffers, the latter for larger buffers. These techniques make the change in page table location transparent to most device drivers.

In the PTE copy method, PTE contents are copied into a data structure called a direct I/O buffer map (DIOBM). DIOBMs vary in length. An embedded DIOBM in the IRP can accommodate the contents of nine PTEs. If necessary, a secondary DIOBM can be allocated instead.

In the PTE window method, the page table pages that map the buffer are double-mapped into 32-bit system space. This method has the relatively high overhead of PTE allocation and translation buffer invalidation of the system space addresses but can support very large buffers.

The I/O subsystem chooses between the two techniques based on the contents of IOC$GL_DIOBM_PTECNT_MAX, which contains a value derived from performance testing. If the buffer has more pages than the contents of that cell, the PTE window method is used. Otherwise, the PTE copy method is used.

The *OpenVMS Alpha Guide to Upgrading Privileged-Code Applications* contains further information.

2.10.2 Swapper I/O

The swapper is presented with a more difficult problem. It must write a collection of process pages to disk that are not virtually contiguous.

During system initialization, a piece of nonpaged 32-bit system space is allocated for the swapper's use. The space contains one quadword for each entry in the largest possible working set that could be swapped (the minimum of WSMAX and 64 K pages). This system space is known as the swapper map. The starting address of the swapper map is stored in global cell SWP$GL_MAP.

As described in Chapter 1, the swapper is an independent process, scheduled like any other. Its code, however, is part of an executive image loaded into system space. The swapper temporarily adopts the address space of the process being swapped when it needs to access the process's page tables. This enables it to keep running from system space while accessing the page table space of the process being swapped.

When the swapper scans the working set list of the process being outswapped, it copies the PFNs in every valid PTE to successive entries in its swapper map. The swapper stores the address of the base of the swapper map in the field IRP$L_SVAPTE before the IRP is passed to the driver. (The swapper can exercise this control because it builds a portion of its own IRP.) The swapper map looks just like any other page table to the hardware/software combination that implements scatter/gather I/O.

What the swapper has succeeded in doing is making pages that were not virtually contiguous into pages that appear to be virtually contiguous. At the same time that each PTE is processed, any special actions based on the type of page are also taken care of. The whole operation of outswap and the complementary steps taken when the process is swapped back into memory are discussed in Chapter 6.

The swapper map supports only one use at a time. When an inswap or outswap operation is in progress, the swap-in-progress flag (SCH$V_SIP), in location SCH$GL_SIP, is set to indicate its use.

2.10.3 Modified Page Writer PTE Arrays

The modified page writer, in its attempt to write many pages to backing store with a single write request (so-called modified page write clustering), is faced with a problem similar to that of the swapper. The modified page writer must build a table of PTEs just as the swapper does.

Unlike the swapper, which can perform only one swap operation at a time, the modified page writer can perform concurrent multiple modified page writes. The SYSGEN parameter MPW_IOLIMIT specifies its maximum number of concurrent I/O operations.

When the modified page writer is building an I/O request, it can encounter three different types of page:

- Pages bound for a swap file (PFN$V_SWPPAG_VALID set) are written individually.
- Pages bound for a section file are not necessarily virtually contiguous; these pages will be written as a group only if they are virtually contiguous.
- Pages on the modified page list that are to be written to a particular page file may not only be noncontiguous within one process address space but may also belong to several processes. It is these pages that the modified page writer must cluster so they appear virtually contiguous.

During system initialization, the modified page writer's initialization routine, MPW$INIT in module WRTMFYPAG, allocates nonpaged pool to build I/O maps. It allocates MPW_IOLIMIT number of structures and links them into a lookaside list. Each structure is large enough for an IRP and two arrays, each of MPW_WRTCLUSTER elements. One array is a quadword array, and the other is a longword array.

When modified pages are written, the first array is filled with PTEs containing PFNs in a manner analogous to the way in which the swapper map is used. The longword array contains an index into the PHD vector for each page in the map. In this way, each page that is put into the map and written to its backing store location is related to the PHD containing the PTE that maps this page. The operation of the modified page writer, including its clustered writes to a page file, is discussed in detail in Chapter 4.

2.11 Relevant Source Modules

Source modules described in this chapter include

[LIB]BODDEF.SDL
[LIB]GSDDEF.SDL
[LIB]LDRHPDEF.SDL
[LIB]PCBDEF.SDL
[LIB]PFLDEF.SDL
[LIB]PFNDEF.SDL
[LIB]PHDDEF.SDL
[LIB]PTEDEF.SDL
[LIB]RDEDEF.SDL
[LIB]RMDDEF.SDL
[LIB]SYSPARDEF.SDL
[STARLET]PMMDEF.SDL
[STARLET]SECDEF.SDL
[STARLET]VADEF.SDL
[SYS]ALLOCPFN.MAR
[SYS]INIT.C
[SYS]INIT_PFL_BITMAP.C
[SYS]INITPGFIL.MAR

[SYS]LDR_INIT_MEM.B64
[SYS]PAGE_FILE.C
[SYS]RES_MEM_INIT.C
[SYS]SYSTEM_REPLICATE.C

Chapter 3

Memory Management System Services

A place for everything and everything in its place.

Isabella Mary Beeton, *The Book of Household Management*

This chapter describes those system services that affect process-private virtual address space and several related others:

- $CREATE_REGION_64, which assigns characteristics to an area of a given size
- $CRETVA and $CRETVA_64, which create demand zero pages in P0, P1, and P2 space
- $EXPREG and $EXPREG_64, which create demand zero pages at the next available address within a specified virtual address region
- Various create and map section services that create a process-private or global section that maps the blocks of a file or particular pages of physical address space to a portion of process-private address space
- Various create and map section services that create and map memory-resident or Galaxywide global sections
- $MGBLSC, $MGBLSC_64, and $MGBLSC_GPFN_64, which map to an existing global section
- $DELTVA and $DELTVA_64, which delete P0, P1, or P2 pages
- $CNTREG, which deletes the upper end of P0 space or the lower end of P1 space
- $DGBLSC, which marks a global section for deletion when no more processes are mapped to it
- $DELETE_REGION_64, which deletes a given region
- $CREATE_BUFOBJ and $CREATE_BUFOBJ_64, which create a buffer object
- $DELETE_BUFOBJ, which deletes a buffer object
- Services that return information about address space, such as $GET_REGION_INFO, $GETSECI, and $FIND_GPAGE_64

- $SETSWM, which enables or disables process swapping
- $SETPRT and $SETPRT_64, which change the protection on pages of virtual address space
- $SETFLT and $SETFLT_64, which set the fault-on-execute bit for a page
- $COPY_FOR_PAGE, which reads data from a page with fault-on-read set

Chapter 4 describes the Update Section File on Disk ($UPDSEC) and Update Global Section File on Disk ($UPDSEC_64) system services, which write the contents of all modified pages in a section to their backing store. It also describes the Fault Page ($FAULT_PAGE) system service, requested to fault a set of pages prior to their use. Chapter 5 describes the system services that control a process's working set list.

3.1 Common Characteristics of Memory Management System Services

This chapter describes several types of memory management system services. The original system services accept only 32-bit address arguments and have been supplemented with 64-bit services that take 64-bit address arguments. The latter have names ending in _64 to indicate that they accept 64-bit addresses by reference. For example, $CRETVA is the original system service requested to create virtual address space. It continues to be used, but to create P2 space, an application must request $CRETVA_64. This chapter uses the term *32-bit services* to refer to the original services and the term *64-bit services* for the services whose names end in _64 or services that can affect P2 space.

When 64-bit support was provided for $CRMPSC, that complex system service was split into a number of new services. $CRMPSC both creates and maps various types of process and global sections. The new services deal with either process or global sections, but not both. Generally, three new services are provided for each type of section: one to create the section if it does not already exist and then to map it, one simply to create the section, and one simply to map an existing section. The names of many of the new services end in _64, for example, $CRMPSC_PFN_64, but those without address arguments do not, for example, $CREATE_GPFN.

A process's ability to use the services described in this chapter may be limited by access mode, process quotas, limits, privileges, and SYSGEN parameters.

The level 3 page table entry (L3PTE) associated with each page of virtual address space contains an owner field (see Figure 2.12) that specifies which access mode owns the page. The memory management system service checks the owner field to determine whether the service's requestor has an access mode at least as privileged as the owner mode of the page and thus is able to manipulate the page in the desired fashion.

In general, a process is only permitted to affect P0, P1, and P2 address space, not system space, with these services. The only exception is when a process uses the buffer object services to double-map process-private address space into system space.

Almost all the memory management system services accept a desired virtual address range as one or more input arguments. Many of the services can partly succeed, that is, affect only a portion of the specified address range. A system service indicates partial success by returning an error status and the address range for which the operation completed.

3.1.1 Common Characteristics of the 32-Bit System Services

Many of the 32-bit memory management system services have similar structures and sequences and similar arguments. The input range for a 32-bit service is specified by the address of a two-longword array, the INADR argument. The first longword is the starting address, and the second, the address of the last byte to be created. The RETADR argument is the address of a two-longword array that receives the addresses of the starting and ending bytes actually created. The ACMODE argument specifies the owner of the address space, the least privileged mode that can access it.

Each 32-bit system service first executes code generated by a MACRO-32 macro that tests whether enough arguments have been supplied and, if not, returns the error status SS$_INSFARG to the requestor.

Each service validates its arguments. A typical service makes the following checks:

- It tests the accessibility of the INADR and RETADR arguments.
- It maximizes the ACMODE argument with the mode of the service requestor.
- It tests the starting and ending addresses and, if either is a system space address, returns the error status SS$_NOPRIV.

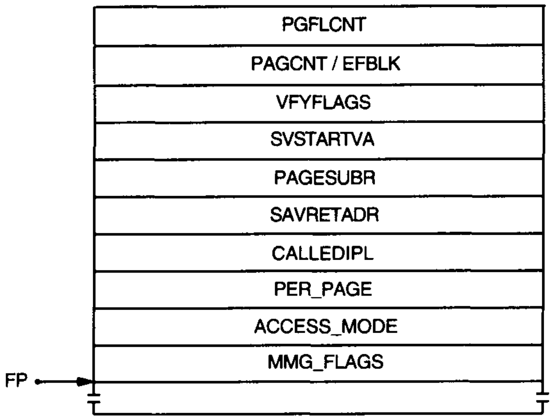
The service then explicitly creates scratch space on the stack to record information about the service request.

The macro $MMGDEF defines symbolic offsets into this scratch space, which is pointed to by the frame pointer (FP) register while the system service procedure is executing. Figure 3.1 shows the layout of the scratch space on the stack. Some fields are used by only a few system services; others are common to all.

MMG$L_MMG_FLAGS contains flag bits associated with the operation. Some of the 64-bit services use these same flags, passing them to inner mode routines as an argument.

- Bit MMG$V_CHGPAGFIL in this longword, when set, means page file quota should be charged for the operation.
- Bit MMG$V_NOWAIT_IPL0, when set, means that a memory management routine should return with an error status rather than waiting at interrupt priority level (IPL) 0 for I/O completion.
- Bit MMG$V_NO_OVERMAP, when set, means that address space to be created may not overlap existing address space.

Figure 3.1 Layout of Scratch Space on the Stack



- Bit MMG$V_PARTIAL_FIRST, when set, means that the first page to be mapped is only partially backed by section file (see Section 3.6.1.1, step 12).
- Bit MMG$V_PARTIAL_LAST, when set, means that the last page to be mapped is only partially backed by section file (see Section 3.6.1.1, step 12).
- Bit MMG$V_NO_IRP_DELETE, when set, means that an I/O request packet created by the $UPDSEC system service is currently in use and should not be deallocated to nonpaged pool.
- Bit MMG$V_DELPAG_NOP, when set, means that not all pages in the specified region could be deleted.
- Bit MMG$V_CLUSTER_DEL, when set, means that the per-page deletion routine (see Section 3.10.2) can delete a whole cluster of pages at once.
- Bit MMG$V_WINDOW, when set, means that the page is part of a memory-resident global section or a section mapped by page frame number (PFN).
- Bit MMG$V_SHARED_L3PTS, when set, means that the page is part of a memory-resident global section that is mapped with shared page tables.
- Bit MMG$V_RWAST_AT_IPL0, when set, means that the per-page deletion routine (see Section 3.10.2) should wait the kernel thread at IPL 0 rather than IPL 2. The bit is set by the $DELTVA and $DELTVA_64 system services. The IPL 0 wait enables Extended QIO Processor (XQP) kernel mode asynchronous system traps (ASTs) to be delivered, preventing a deadlock in certain circumstances.

MMG$L_ACCESS_MODE contains the access mode associated with the operation, the maximized ACMODE argument.

MMG$L_PER_PAGE is the per-page processing context area. It contains one defined flag, MMG$V_DELGBLDON. When set, the bit means that global pages in the range have already been deleted.

MMG$L_CALLEDIPL records the IPL from which the service was requested, typically 0.

MMG$L_SAVRETADR contains the value of the optional service RETADR argument.

MMG$L_PAGESUBR contains the procedure value of the executive routine that performs the requested service on a single page.

MMG$L_SVSTARTVA saves the starting virtual address specified by the service requestor.

MMG$L_VFYFLAGS contains the section flags passed as an argument to a service such as $CRMPSC and verified by the service.

MMG$L_PAGCNT and MMG$L_EFBLK are two names for the same field. MMG$L_PAGCNT, used by services related to buffer objects, contains the number of pages in a buffer object being created or deleted. MMG$L_EFBLK contains the number of the end-of-file block for a section file.

MMG$L_PGFLCNT contains the number of pages of page file quota that have been reserved against the job's quota for this request.

After creating and initializing the scratch space on the stack, a 32-bit memory management system service takes the following steps:

1. It performs argument validation.
2. It raises IPL to 2 to block the delivery of an AST. In addition to blocking process deletion, this prevents the execution of AST code that could cause unexpected changes to the page tables, working set list, region descriptor entries (RDEs), and other data structures.
3. If appropriate, it checks page ownership to ensure that a less privileged access mode is not attempting to alter the properties of pages owned by a more privileged access mode.
4. It calls the routine MMG$CREDEL, in module SYSCREDEL, passing it the procedure value of a per-page service-specific routine to accomplish the desired action of the system service. MMG$CREDEL performs general page processing and calls the per-page routine for each page in the desired range.
5. It reprobes write accessibility of any output arguments.
6. It returns the address range actually affected by MMG$CREDEL's actions in the optional RETADR argument.
7. It restores the entry IPL and returns to its requestor.

In some cases, step 4 in that sequence is replaced by calling a routine that affects all pages in the desired range.

MMG$CREDEL takes the following steps:

1. It tests the starting and ending addresses of the range and, if either is in system space, returns the error status SS$_NOPRIV.
2. It initializes MMG$L_PAGESUBR and MMG$L_SVSTARTVA in the scratch space and loads registers with information such as process control block (PCB) address, process header (PHD) address, page count, starting virtual address, and ending virtual address.
3. MMG$CREDEL calls the per-page routine. Unless the routine returns an error status, MMG$CREDEL continues to call it, once per page.
4. If the per-page routine returns the status SS$_REGISFULL, MMG$CREDEL converts it to SS$_VASFULL.
5. When an error occurs or there are no more pages, MMG$CREDEL returns to its caller with a status code and the address of the last affected page.

3.1.2 Common Characteristics of the 64-Bit System Services

The 64-bit system services have a common structure and sequence and similar arguments. They do not explicitly use scratch space on the stack.

A 64-bit service typically takes the following steps:

1. It performs argument validation, for example:
   - These services are written in C and must explicitly test whether too few or too many arguments have been supplied. Each checks the number of arguments and, if incorrect, returns either the error status SS$_INSFARG or SS$_TOO_MANY_ARGS.
   - It checks that output arguments are accessible and, if not, returns the error status SS$_ACCVIO.
   - If the service has a FLAGS argument, it checks that only valid flags were set in the argument and, if not, returns the error status SS$_IVSECFLG.
   - It maximizes the ACMODE argument.
2. It raises IPL to 2 to block AST delivery.
3. If appropriate, it checks page ownership.
4. It loops, calling a per-page service-specific routine, typically the same routine as its 32-bit counterpart.
5. It reprobes write accessibility of any output arguments.
6. It returns the address range actually affected in the RETURN_VA_64 and RETURN_LENGTH_64 arguments.
7. It restores the entry IPL and returns to its requestor.

3.2 Virtual Address Region Creation

The Create Virtual Region ($CREATE_REGION_64) system service is requested to create a region within process-private address space. Chapter 2 discusses regions and the RDEs that describe them.

Service arguments include the desired length, protection, and flags that specify whether the region is in P0, P1, or P2 space; whether its allocation is to be ascending or descending; whether address space within it should be created automatically in response to an access violation; whether it should be permanent; and whether its space is capable of being mapped with shared page tables. Only memory-resident global sections and Galaxywide global sections are mapped into such a region.

The service creates a region with the requested characteristics, assigns an ID to it, and returns its ID and address.

The $CREATE_REGION_64 system service procedure, EXE$CREATE_REGION_64 in module SYS_REGIONS, runs in kernel mode. EXE$CREATE_REGION_64 takes the following steps:

1. It calculates the number of PTEs in a page table page and the number of bytes mapped by an L3PT.
2. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:
   a. It checks that REGION_PROT is valid, returning SS$_IVPROTECT if not.
   b. It maximizes the create and owner access mode fields in the REGION_PROT argument with that of the requestor. It checks that the owner mode is less or equally privileged to the creator mode, returning SS$_IVREGPROT if not.
   c. It checks that the LENGTH_64 argument is nonzero and a multiple of the size of a page, returning the error status SS$_LEN_NOTPAGMULT if not.
   d. If VA$V_SHARED_PTS in the FLAGS argument is clear, EXE$CREATE_REGION_64 checks that the optional START_VA_64 argument, if supplied, is on a page-aligned boundary and returns the error status SS$_VA_NOTPAGALGN if not.
   e. If VA$V_SHARED_PTS in the FLAGS argument is set, indicating that the region can be mapped by shared page tables, EXE$CREATE_REGION_64 checks that the START_VA_64 argument is a multiple of the number of bytes mapped by one L3PT. On a system with an 8 KB page size, an L3PT maps 8 MB. It also rounds up the LENGTH_64 argument to such a multiple.
   f. It calculates the address of the process-permanent RDE corresponding to the specified address space.
   g. It checks that the LENGTH_64 argument can be expressed in the number of significant address bits for the system's page size and page table hierarchy, for example, 43 bits for a page size of 8 KB and a three-level page table. If not, it returns the error status SS$_VASFULL.

h. If the START_VA_64 argument was supplied, it checks that the sum of the START_VA_64 and LENGTH_64 arguments can be expressed in that number of bits, returning SS$_VASFULL if not. It also checks that the START_VA_64 and the sum of START_VA_64 and LENGTH_64 are within the process-permanent region specified in the FLAGS argument, returning SS$_VA_IN_USE if not.

3. It raises IPL to 2.

4. It allocates an RDE from the P1 allocation region and initializes it with information from the service arguments. It initializes RDE$Q_REGION_ID from the contents of PHD$Q_NEXT_REGION_ID and increments them.

5. If the argument START_VA_64 was not specified, it determines the alignment requirement for the starting address.

   — For a region without shared page tables, the starting address merely needs to be page-aligned.

   — For a region with shared page tables, EXE$CREATE_REGION_64 first attempts a 512-page alignment so that the shared pages can potentially be mapped as a 512-page granularity hint region.

   It scans the list of user-defined RDEs within the specified process-permanent region, looking for an unused piece of address space with at least the specified alignment and size. If it fails to find one and this is a shared page table region, it tries again, shrinking the desired alignment to the next smaller granularity hint region size, 64 pages, and then, if necessary, to eight pages, and finally to one page.

   If it fails to find an unused piece with single-page alignment that is large enough, it deallocates the RDE and returns SS$_VA_IN_USE to its requestor.

6. If START_VA_64 was specified, it scans the list of user-defined RDEs within the specified process-permanent region, which are ordered by virtual address. It looks for the place at which the new RDE should be inserted.

   — If the address range of the new RDE overlaps the range of an existing user-defined RDE, EXE$CREATE_REGION_64 deallocates the new RDE and returns the error status SS$_VA_IN_USE to its requestor.

   — If there is overlap with the process-permanent region, it adjusts that region so that it ends where the new one begins.

7. It inserts the RDE into the list and also at the front of the region ID list (see Figures 2.2 and 2.3).

8. It lowers IPL.

9. It records peak page file use and virtual size statistics, and stores return information about the newly created RDE in the RETURN_VA_64, RETURN_REGION_ID_64, and RETURN_LENGTH_64 arguments. It returns SS$_NORMAL to its requestor.

3.3 Process-Private Virtual Address Space Creation

Among the most basic memory management services are those that create process-private virtual address space: $CRETVA, $CRETVA_64, $EXPREG, $EXPREG_64, the various create and map section services, $MGBLSC, $MGBLSC_64, and $MGBLSC_GPFN_64. The image activator requests some of these services during image activation, as described in Chapter *Image Activation and Exit.* An image can request these services directly to alter process-private address space.

P0, P1, and part of P2 space are described by a single process-private level 2 page table (L2PT). Additional P2 space requires additional L2PTs. Each space is described by level 3 page tables (L3PTs), with an L3PTE for each page of address space.

Creating address space typically requires adding page table pages and modifying the RDE of the affected region as well as initializing L3PTEs to map the new address space. It may also require initializing one or more L2PTEs.

In the case of address space associated with a process-private section file, creating address space also involves allocating and initializing a process section table entry (PSTE). Chapter 2 describes page tables, PTEs, process sections, and PSTEs.

There are several limits on the amount of process-private virtual address space that can be created:

- A process's working set limit can constrain the size of that process's address space. When a process tries to expand its address space, the executive checks whether there is enough room in the dynamic working set list for the fluid working set (PHD$L_WSFLUID, initialized from the SYSGEN parameter MINWSCNT), plus the worst-case number of page table pages required to map it, to allow the process to perform useful work. If this check succeeds, the virtual address space creation can proceed. Otherwise, if the process's working set limit is smaller than its quota, the working set limit is increased. If the working set limit cannot be increased, the virtual address space creation fails with the error status SS$_INSFWSL. Chapter 5 describes working set limits, quotas, and expansion.

  Note that pages from memory-resident and Galaxywide global sections are not represented in a process's working set list. Shared L3PTs that map a memory-resident or Galaxywide global section are also not represented in the working set list.

- Another constraint on the total size of the process address space is page file quota. Each demand zero page, copy-on-reference (CRF) section page, and process-private page table page is charged against the job's page file quota, JIB$L_PGFLCNT. (Although the page file quota is externally represented as pagelets, the quota is internally maintained in pages.)

  Pages from PFN-mapped sections, memory-resident global sections, Galaxywide global sections, read-only sections, and writable non-CRF sections do not require page file quota. Shared L3PTs that map a memory-resident or Galaxywide global section also do not require page file quota.

- In versions prior to Version 7.3, creation of address space with page file backing store was limited because a process could page in only four files and the form of invalid PTE that describes a page file page was limited to a 20-bit number. OpenVMS recorded how many pages of pageable address space a process had created to ensure this limit was not reached.

  Such accounting is no longer necessary. As of Version 7.3, page file backing store is not assigned until modified pages are being written out. Assigned backing store is represented by an eight-bit page file index and a 24-bit page number. Thus, in each page file, a process could have a theoretical maximum of 2^{24} pages of pageable address space that requires page file backing store (for example, demand zero or copy-on-reference sections).

- In versions prior to Version 7.0, the SYSGEN parameter VIRTUALPAGECNT controlled the total number of L3PTEs mapping P0 and P1 space that any process could have. A limit was required because process page tables were mapped as part of a fixed-length PHD. With the removal of the page tables from system space, OpenVMS Version 7.0 and later releases place no such limits. The parameter is obsolete but still exists to provide compatibility to applications that determine their actions from the value of the parameter. It is set to the maximum value. System services no longer reference VIRTUALPAGECNT.

  In OpenVMS Version 7.0 and later, address space checks are made against PHD$Q_FREE_PTE_COUNT, which contains the number of unused process-private virtual pages. It is initialized from the contents of MMG$GQ_PROCESS_VA_PAGES, the number of pages of process-private address space between zero and the base of page table space, excluding the gap (see Chapter 1).

3.4 Demand Zero Virtual Address Space Creation

The simplest form of address space creation is the creation of a series of demand zero pages through the $CRETVA, $CRETVA_64, $EXPREG, or $EXPREG_64 system services. The services initialize PTEs, that is, create address space, to map the demand zero pages. A demand zero page is not itself created until the first time the process accesses it. These services do not create process sections, that is, they initialize PTEs with no corresponding PSTE.

For the $EXPREG and $EXPREG_64 system services, PTEs to map demand zero pages are initialized beginning at the first free address in the designated process-private address region.

For the $CRETVA and $CRETVA_64 system services, PTEs are initialized to map the specified address range. If any pages already exist in the requested range, they must be deleted first.

These system services can partly succeed, that is, a number of pages smaller than the number originally requested may be mapped. After several pages have already been successfully mapped, the service can run into one of the limits to address space creation.

3.4.1 $CRETVA System Service

The Create Virtual Address Space ($CRETVA) system service procedure, EXE$CRETVA in module SYSCREDEL, runs in kernel mode. It has an alternative entry point, MMG$CRETVA_K, called from code already in kernel mode, such as image activator routines and EXE$PROCSTRT in module PROCSTRT. The alternative entry point has additional arguments that enable the caller to specify the protection of the new address space, whether the new space may overlap existing space, and the contents of the copy characteristic and no-execute bits in each L3PTE.

EXE$CRETVA takes the following steps:

1. It creates and initializes the scratch space on the stack.
2. It validates its arguments (see Section 3.1.1).
3. It constructs template L3PTE contents for the new pages (see Figure 2.12).

   The template L3PTE indicates a demand zero page, with owner access mode the less privileged of the requesting access mode and the ACMODE argument. In the case of a normal system service request, the L3PTE has protection bits enabling read and write access to the owner mode. In the case of entry through MMG$CRETVA_K, the protection, copy characteristic, and no-execute bits are specified by the caller.
4. EXE$CRETVA raises IPL to 2 to block AST delivery while it is modifying the PHD.
5. It determines in which region of process-private address space the address range lies. It checks whether the access mode from which the service was requested is allowed to create pages in this region and, if not, returns the error status SS$_NOPRIV.
6. It rounds the starting and ending addresses down to an Alpha page boundary and calculates the desired page count based on the difference between them.
7. It checks that the address range is entirely within a region and that there is no overlap with already existing space. If either is false, EXE$CRETVA continues with step 10.
8. Typically the process is requesting the creation of address space within a region just beyond the end of what has already been defined. As an optimization for this common case, EXE$CRETVA calls MMG$TRY_ALL_64 (see Section 3.4.1.1) to test further whether the entire space can be created.

   If the entire address space cannot be created, EXE$CRETVA proceeds with step 10.
9. If none of the limits to growth of the process's virtual address space has been reached, EXE$CRETVA calls MMG$FAST_CREATE_64, in module SYSCREDEL.

MMG$FAST_CREATE_64 and its alternative entry point, MMG_STD$FAST_CREATE_64, determine the starting address in page table space of the first new PTE. The routine loops, initializing four L3PTEs in each iteration. Creating the address space in this manner is significantly faster than creating it one page at a time through MMG$CREPAG_64.

EXE$CRETVA continues with step 11.

10. If any of the limits to virtual address space growth described in Section 3.3 prevents creation of the entire space, EXE$CRETVA creates it one page at a time, stopping when the limit is reached. Page-by-page creation is also necessary if the specified address space overlaps already existing space, since the existing pages must first be deleted. In either of these cases, EXE$CRETVA calls MMG$CREDEL, specifying MMG$CREPAG_64 (see Section 3.4.1.2) as the per-page service-specific routine.
11. EXE$CRETVA returns any unused page file quota, records peak page file use and virtual size statistics, and stores return information in the optional RETADR argument.
12. If the process has any deleted sections to be cleaned up, it calls MMG$DALCSTXSCN, in module PHDUTL, to check whether any process section table entries can be deallocated (see Section 3.9.2).
13. It restores the IPL at entry and returns to its requestor.

3.4.1.1 MMG[_STD]$TRY_ALL_64 Routine

MMG$TRY_ALL_64 and its alternative entry point, MMG_STD$TRY_ALL_64, in module SYSCREDEL, test whether there is enough free space in the region, enough process-private address space (PHD$Q_FREE_PTE_COUNT), enough room in the dynamic working set list, and enough page file quota (see Section 3.3).

If all tests pass, it adjusts RDEQ_REGION_SIZE, RDEPQ_FIRST_FREE_VA, and PHD$Q_FREE_PTE_COUNT and charges against reserved page file quota. If necessary, it initializes L2PTEs to map new L3PT pages for the address space being created. It returns a status indicating its findings.

3.4.1.2 MMG[_STD]$CREPAG_64 Routine

MMG$CREPAG_64, with its alternative entry point, MMG_STD$CREPAG_64, in module SYSCREDEL, is the per-page service-specific routine for the $CRETVA, $CRETVA_64, $CRMPSC_FILE_64, $CRMPSC_GDZRO_64, $CRMPSC_GPFN_64, $CRMPSC_PFN_64, $EXPREG, $EXPREG_64, $MGBLSC_GPFN_64, and $MGBLSC_64 system services.

It is used when the entire page creation request cannot be performed as a single operation, possibly because the new pages would overlap existing address space that must be deleted first or because the process lacks enough quota and only part of the request can be satisfied.

MMG$CREPAG_64 is called with arguments that include the L3PTE contents for the new page, the address of the RDE for the region to contain the new space, and the total number of pages to create.

MMG$CREPAG_64 takes the following steps:

1. It tests whether the space required by the pages to be mapped is beyond the limit of the region's defined address space. If the pages are within the limit, MMG$CREPAG_64 continues with step 5.
2. Otherwise, it calls the local routine EXPANDCHK_64 (see Section 3.4.1.3) to check whether the region can accommodate the entire creation request and to create any necessary page table pages.
3. If the entire creation request can now be satisfied, MMG$CREPAG_64 continues with step 6.
4. Otherwise, it calls EXPANDCHK_64 to expand the region by a single page. MMG$CREPAG_64 returns any errors from EXPANDCHK_64 to its caller. If the expansion was successful, MMG$CREPAG_64 continues with step 6.
5. It calls MMG_STD$ADD_PTS, in module SYS_CREDEL_64, to create any page table pages necessary to map the page being created.
6. MMG$CREPAG_64 tests whether the page to be created already exists. If it does and the service requestor specified no address overmap, MMG$CREPAG_64 returns the status SS$_VA_IN_USE to its caller, which returns it as the system service status. (The image activator specifies the NO_OVERMAP flag when it requests the $CRETVA system service.)
7. If the page already exists but overmap is allowed, MMG$CREPAG_64 calls MMG$DELPAG_64 (see Section 3.10.2), to delete the virtual page. If the deletion is successful, MMG$CREPAG_64 continues at step 1. Otherwise, it continues at step 10.
8. If page file quota does not need to be charged, MMG$CREPAG_64 continues with step 9. Otherwise, it must charge the page against the process's reserved page file quota.

   If no more reserved page file quota is left, MMG$CREPAG_64 tries to reserve more quota from the process's job page file quota, JIB$L_PGFLCNT.

   If the charge cannot be made, MMG$CREPAG_64 adjusts RDE$PQ_FIRST_FREE_VA, RDE$Q_REGION_SIZE, and PHD$Q_FREE_PTE_COUNT to show expansion up to but not including the page that could not be mapped. It returns the error status SS$_EXQUOTA.
9. It stores the requested value into the L3PTE.
10. It returns to its caller.

3.4.1.3 EXPANDCHK_64 Routine

EXPANDCHK_64, in module SYSCREDEL, is called with arguments that include the RDE address, the number of bytes to expand, and the starting virtual address. It takes the following steps:

1. It tests whether there is enough free process-private address space (PHD$Q_FREE_PTE_COUNT) and, if not, returns the error status SS$_VASFULL.
2. Otherwise, it charges PHD$Q_FREE_PTE_COUNT to account for the pages to be created.
3. It tests whether there is enough free space left in the region. If not, it restores the charge against PHD$Q_FREE_PTE_COUNT and returns the error status SS$_REGISFULL.
4. Otherwise, it updates RDE$PQ_FIRST_FREE_VA to account for the expanded address space within the region.
5. It calls either MMG_STD$ADD_PTS or MMG_STD$ADD_L2PTS, in module SYS_CREDEL_64, to create any necessary page table pages.

   To create a page table page, each of these routines initializes a demand zero PTE and charges the process's page file quota. MMG_STD$ADD_PTS creates L3PTs and any necessary L2PTs as demand zero pages. If the process has insufficient resources for the charge, each routine returns error status SS$_EXPGFLQUOTA; in response, EXPANDCHK_64 returns both SS$_EXQUOTA and SS$_EXPGFLQUOTA to its caller.

   The MMG_STD$ADD_L2PTS routine is called to add any necessary L2PTs for memory-resident and Galaxywide global sections with shared L3PTs. Such sections are created through services such as $CRMPSC_GDZRO_64.
6. EXPANDCHK_64 then checks whether there is enough room in the dynamic working set list for the fluid working set (PHD$L_WSFLUID, initialized from the SYSGEN parameter MINWSCNT), plus the worst-case number of page table pages required to map it, to allow the process to perform useful work. If this check succeeds, the virtual address space creation can proceed. Otherwise, if the process's working set limit is smaller than its quota, the working set limit is increased.

   If the working set limit cannot be increased, it restores RDE$PQ_FIRST_FREE_VA and PHD$Q_FREE_PTE_COUNT to their previous values and returns the error status SS$_INSFWSL.
7. Otherwise, it returns SS$_NORMAL.

3.4.2 $CRETVA_64 System Service

The Create Virtual Address Space ($CRETVA_64) system service procedure, EXE$CRETVA_64 in module SYS_CREDEL_64, runs in kernel mode. It resembles the $CRETVA system service, but its arguments include a region ID, and all its address arguments are 64 bits. Thus it can be used to create address space in P0, P1, or P2 space, either in a process-private region or a user-created one.

EXE$CRETVA_64 takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:

   a. It confirms that the LENGTH_64 argument is nonzero and an integral number of pages and, if not, returns the error status SS$_LEN_NOTPAGMULT.

   b. It checks that the START_VA_64 argument, if specified, is on a page boundary and, if not, returns the error status SS$_VA_NOTPAGALGN.

2. It constructs template L3PTE contents for the new pages (see Figure 2.12).

   The template L3PTE indicates a demand zero page, with owner access mode the less privileged of the requesting access mode and the ACMODE argument. The L3PTE has protection bits enabling read and write access to the owner mode.

3. EXE$CRETVA_64 raises IPL to 2 to block AST delivery while it examines and possibly modifies RDEs.

4. It locates the RDE corresponding to the REGION_ID_64 argument. If there is none, it lowers IPL and returns the error status SS$_IVREGID.

5. If that region is intended for memory-resident and Galaxywide global sections, it returns the error status SS$_NOSHPTS. A process may not create process-private address space to be mapped in such a region. A page table shared by multiple processes only maps global pages that are shared by multiple processes.

   EXE$CRETVA_64 checks whether the access mode from which the service was requested is allowed to create pages in this region and, if not, lowers IPL and returns the error status SS$_IVACMODE.

6. It checks whether the starting address is within the space reserved for the region and, if not, lowers IPL and returns the error status SS$_PAGNOTINREG.

7. It calls MMG_STD$TRY_ALL_64 (see Section 3.4.1.1), to test whether the entire space can be created. If the entire address space cannot be created or if some of it overlaps existing address space, EXE$CRETVA_64 proceeds with step 9.

8. If none of the limits to growth of the process's virtual address space has been reached, EXE$CRETVA_64 calls MMG_STD$FAST_CREATE_64 (see Section 3.4.1), to create the entire address space. EXE$CRETVA_64 continues with step 10.

9. If any of the limits to virtual address space growth prevents creation of the entire space, EXE$CRETVA_64 creates it one page at a time, stopping when the limit is reached. Page-by-page creation is also necessary if the specified address space overlaps already existing space, since the existing pages must first be deleted. In either of these cases, EXE$CRETVA_64 loops, calling MMG_STD$CREPAG_64 (see Section 3.4.1.2) until the routine returns an error status or all pages are done.
10. EXE$CRETVA_64 returns any unused reserved page file quota, records peak page file use and virtual size statistics, and stores return information in the RETURN_VA_64 and RETURN_LENGTH_64 arguments. It lowers IPL and returns SS$_NORMAL to its requestor.

3.4.3 $EXPREG System Service

The Expand Program/Control Region ($EXPREG) system service is very similar to the $CRETVA system service. Its name is based on the early VAX/VMS use of the term region: an architecturally defined portion of virtual address space, such as the P0 or P1 space region. In a VAX/VMS region, address space could be created only densely. The Alpha service actually creates new demand zero address space at the next available address in the region rather than changing the boundaries of the region.

The $EXPREG system service procedure, EXE$EXPREG in module SYSCREDEL, runs in kernel mode. It has an alternative entry point, MMG$EXPREG, called from code already in kernel mode, such as EXE$ALOP1IMAG in module MEMORYALC. The alternative entry point enables the caller to specify the protection of the new space.

EXE$EXPREG selects the RDE corresponding to the region to be expanded and uses the contents of RDE$PQ_FIRST_FREE_VA as one end of the address range.

It converts its PAGCNT argument, the number of pagelets by which the region is to be expanded, to a number of physical pages, rounding up if necessary. It adds the number of bytes corresponding to that many physical pages to the end of the address range to form the new end of the address region.

It forms template L3PTE contents for the new page as EXE$CRETVA does (see Section 3.4.1).

As an optimization, EXE$EXPREG first checks whether the entire address space can be created. If so, EXE$EXPREG creates it all at once rather than page by page, calling the routine MMG$FAST_CREATE_64 (see Section 3.4.1). Otherwise, it calls the routine MMG$CREDEL, specifying MMG$CREPAG_64 (see Section 3.4.1.2) as the per-page service-specific routine.

3.4.4 $EXPREG_64 System Service

The Expand Virtual Address Space ($EXPREG_64) system service procedure, EXE$EXPREG_64 in module SYS_CREDEL_64, runs in kernel mode. It resembles the $EXPREG system service, but its arguments include a region ID, and all its address arguments are 64 bits. Thus it can be used to create new demand zero address space at the next available address in P0, P1, or P2 space, either in a process-permanent region or a user-created one. It is very similar to the $CRETVA_64 system service.

EXE$EXPREG_64 selects the RDE corresponding to the region to be expanded and uses the contents of RDE$PQ_FIRST_FREE_VA as one end of the address range.

It checks that its LENGTH_64 argument, the number of bytes by which the region is to be expanded, is an integral number of pages, returning the SS$_LEN_NOTPAGMULT error status if not.

It forms template L3PTE contents for the new page as EXE$CRETVA does (see Section 3.4.1).

As an optimization, EXE$EXPREG_64 checks whether the entire address space can be created. If so, EXE$EXPREG_64 creates it all at once rather than page by page, calling the routine MMG_STD$FAST_CREATE_64 (see Section 3.4.1). Otherwise, it loops, calling MMG_STD$CREPAG_64 (see Section 3.4.1.2) until the routine returns an error status or all pages are done.

3.4.5 Automatic Address Space Expansion

A special form of P1 space expansion occurs when a request for user stack space exceeds the remaining size of the user stack. OpenVMS can detect such a request made implicitly through an access violation.

In addition to the access violation exception service routine, several other executive software routines can also detect the need to expand the user stack:

- The AST delivery interrupt service routine (see Chapter *ASTs*), when it is unable to copy AST-related information from the kernel stack to the user stack
- The Adjust Outer Mode Stack Pointer ($ADJSTK) system service
- The exception dispatching routine, EXE$EXCEPTION in module EXCEPTION, when it is unable to copy the exception context area onto the user stack (see Chapter *Condition Handling*)

These routines call EXE$EXPANDSTK, in module SYSADJSTK, to try to expand the user stack. EXE$EXPANDSTK is also called by the access violation exception service routine, EXE$ACVIOLAT in module EXCEPTION, for an access violation that occurred in user mode. EXE$EXPANDSTK checks that

- An attempt to access an empty page occurred rather than a protection violation
- The inaccessible address is in P1 space and less than the high end of the user stack

If these conditions are true, EXE$EXPANDSTK requests the $CRETVA_64 system service to expand P1 space from its current low-address end to the specified inaccessible address. For the usual case, one in which a program requires more user stack space than requested at link time, the expansion typically occurs one page at a time.

Because this automatic expansion cannot be disabled on a process-specific or systemwide basis, a runaway program that uses stack space without returning it is not aborted immediately. Instead, the program runs until it reaches one of the limits to growth of virtual address space described in Section 3.3.

Another side effect of automatic expansion occurs when a program makes a possibly incorrect reference to an arbitrary P1 address lower than the top of the user stack. Rather than exiting with some error status, the program will probably continue to execute (after the creation of many demand zero pages).

If the stack expansion fails for any reason, the process is notified in a way that depends on the caller of EXE$EXPANDSTK:

- The $ADJSTK system service can fail with one of the error codes returned by the $CRETVA system service.
- An attempt to deliver an AST to a process with insufficient user stack space results in an AST delivery fault (SS$_ASTFLT) condition reported to the process.
- If the user stack cannot be expanded in response to a P1 space length violation, EXE$ACVIOLAT checks whether this is a multithreaded process. If so, and if the faulting virtual address was within a DECthreads guard page or a kernel thread's yellow stack zone, it reports an SS$_STKOVF exception to the process. Otherwise, an access violation fault is reported to the process.
- If there is not enough user stack to report an exception, EXE$EXCEPTION first tries to reset the user stack pointer to the high-address end of the stack. If that fails, EXE$EXCEPTION requests the $CRETVA system service in an attempt to recreate the address space. If that fails, EXE$EXCEPTION bypasses the normal condition handler search and reports the exception directly to the last chance handler. Typically, this handler aborts the currently executing image. Chapter *Condition Handling* contains more details.

In OpenVMS Version 7.0 and later versions, the concept of stack expansion has been broadened to include potential expansion of any process region created with the characteristic expand-on-access-violation. EXE$ACVIOLAT calls EXE$EXPANDSTK for any user mode access violation.

If the faulting virtual address is not in P1 space, EXE$EXPANDSTK checks whether the region flag RDE$V_EXPAND_ON_ACCVIO is set. If not, an access violation is reported to the process. If the region has the characteristic, EXE$EXPANDSTK requests the $CRETVA_64 system service to extend the region from its current end to the faulting virtual address.

3.5 Process and Global Sections

The system services that create and map sections are an alternative method of creating address space, one that enables a process to associate a portion of its address space with a specified portion of a file. The section may be specific to a process (called a process-private section, or simply, a process section) or it may be a global section, shared among several processes.

Table 3.1 summarizes the different types of section that can be created through these services, and the source of and backing for their pages. In addition to these types of section, a process can create demand zero pages backed by a page file through the system services $CRETVA, $CRETVA_64, $EXPREG, and $EXPREG_64.

The original Create and Map Section system service, $CRMPSC, enables a process to create many different types of section, both process and global. If the section does not exist, $CRMPSC creates it and then maps to it; if the section does exist and the process is allowed to map it, $CRMPSC maps the section. A new version of $CRMPSC was required to support 64-bit addresses. For simplicity, multiple 64-bit services were created: each 64-bit service is typically specific to a type of section. In the case of 64-bit services for global sections, there is a service to create a particular kind of global section, one to map that type of section, and another to both create and map it.

A global section is characterized by whether it is permanent or temporary. A temporary section is automatically deleted when no more processes are mapped to it. A permanent global section must be deleted explicitly through the $DGBLSC system service. Creating a permanent section requires PRMGBL privilege. The name space for global sections can be systemwide or specific to a UIC group. A security persona must have SYSGBL privilege to create a systemwide section.

In addition to section files, a process can create and map other types of section:

- A security persona with PFNMAP privilege can map virtual address space to a specific range of physical addresses. Typically, a process uses this capability to access a physical page in I/O space to communicate with a particular I/O device.
- A process can also create global page-file sections, demand zero global sections whose pages are backed by a page file.
- In OpenVMS Version 7.0 and later versions, a security persona holding the rights identifier VMS$MEM_RESIDENT_USER can create a memory-resident demand zero global section. Optionally, a memory-resident demand zero global section can be mapped with shared L3PTs.

Table 3.1 Section Types and Backing Store

Section Type/Attribute	Source of Contents	Backing Store	System Service
Process-Private Sections			
Demand zero	Demand zero page	Section file	$CRMPSC, $CRMPSC_FILE_64
Copy-on-reference	Section file	Page file	$CRMPSC, $CRMPSC_FILE_64
Read-only	Section file	Section file	$CRMPSC, $CRMPSC_FILE_64
Writable (and not copy-on-reference)	Section file	Section file	$CRMPSC, $CRMPSC_FILE_64
PFN-mapped	Physical page or I/O space	None	$CRMPSC, $CRMPSC_PFN_64
Global Sections			
Memory-resident demand zero	Demand zero page	None	$CREATE_GDZRO, $CRMPSC_GDZRO_64
Galaxywide demand zero	Demand zero page	None	$CREATE_GDZRO, $CRMPSC_GDZRO_64
Demand zero	Demand zero page	Section file	$CRMPSC, $CREATE_GFILE, $CRMPSC_GFILE_64
Page-file	Demand zero page	Page file	$CRMPSC, $CREATE_GPFILE, $CRMPSC_GPFILE_64
Copy-on-reference	Section file	Page file	$CRMPSC, $CREATE_GFILE, $CRMPSC_GFILE_64
Read-only	Section file	Section file	$CRMPSC, $CREATE_GFILE, $CRMPSC_GFILE_64
Writable (and not copy-on-reference)	Section file	Section file	$CRMPSC, $CREATE_GFILE, $CRMPSC_GFILE_64
PFN-mapped	Physical page or I/O space	None	$CRMPSC, $CREATE_GPFN, $CRMPSC_GPFN_64

- A security persona having SHMEM privilege and running in a Galaxy instance can create a memory-resident demand zero global section in shared memory, called a Galaxywide global section. A Galaxywide global section can be accessed by processes running on multiple OpenVMS instances and optionally be mapped with shared L3PTs.

The map global section system services are another way to create address space, one that enables a process to map a portion of its address space to an already existing global section.

The image activator (see Chapter *Image Activation and Exit*) requests the $CRMPSC and $MGBLSC system services to map portions of process address space to sections in image files and to previously installed global sections.

When the image activator opens a file, it does so specifying that all extents of the file should be mapped. However, an image may open a file itself and then itself request the $CRMPSC or $MGBLSC system service; in that case, the window control block (WCB) might not contain a complete description of the file.

The memory management subsystem cannot take a window turn (see Chapter *I/O System Services* for information on WCBs and window turns) on pages within a section. It therefore requires that the WCB describe all the extents of the mapped file. Such a WCB is called a cathedral window or a cathedral WCB.

Because the WCB occupies nonpaged pool, its extension is charged against the job's buffered I/O byte count quota, JIB$L_BYTCNT. Because the quota charge persists until the section is deleted, this charge is also made against the job's JIB$L_BYTLM, which limits the maximum charge against JIB$L_BYTCNT. When a job has insufficient JIB$L_BYTCNT for a request, the executive checks that the request is not larger than JIB$L_BYTLM before placing the kernel thread in resource wait. Charging the WCB extension against JIB$L_BYTLM prevents placing the kernel thread into what might otherwise be a never-ending resource wait.

3.6 Process-Private Sections

The $CRMPSC system service creates a process-private or global section and maps it into process-private address space. The particular actions it takes are determined by the options or flags with which the service is requested. The *OpenVMS System Services Reference Manual* describes the system service arguments and shows which flags can be used together. The 64-bit services $CRMPSC_FILE_64 and $CRMPSC_PFN_64 also create process-private sections and map them into process-private address space.

The sections that follow describe creation of a process-private section backed by section file or page file and creation of a PFN-mapped process-private section.

Section 3.7 and its subsections describe creation of global sections.

3.6.1 Creation of a Process-Private Section Backed by a File

To create a process-private section the system service must validate the arguments; allocate and initialize a PSTE to describe the section and connect it to its associated file, if any; determine starting and ending virtual addresses of the section; and initialize L3PTEs to describe a page in a section file.

3.6.1.1 $CRMPSC and Process-Private Section File Creation

The $CRMPSC system service procedure, EXE$CRMPSC in module SYSCRMPSC, runs in kernel mode. When requested to map a process-private section, it takes the following steps:

1. In addition to making the argument validation checks described in Section 3.1.1, EXE$CRMPSC checks the INADR argument: unless the SEC$V_EXPREG flag was specified in the FLAGS argument, it confirms that the starting address is on an Alpha page boundary and that the ending address is one byte less than a page boundary. (It takes into account the possibility that the addresses have been specified in reverse order.) If the addresses are not correct, it returns the error status SS$_INVARG.

2. It creates and initializes the scratch space on the stack.

3. It calls MMG$VFY_SECFLG, in module SYSDGBLSC, to test the compatibility of the flags in the FLAGS argument with each other. If the flags are incompatible, if the system service requestor specified the flag SEC$V_SHMGS, or if the argument is absent, it returns the error status SS$_IVSECFLG.

4. EXE$CRMPSC then confirms that the CHAN argument was supplied. (The requestor must have already opened the section file on the specified channel.) It confirms that the specified channel has been assigned; that its associated device is directory-structured, files-oriented, and random access; and that a file is open on the channel. In case of error, it returns the error status SS$_NOTFILEDEV or SS$_IVCHNLSEC.

5. If the WCB does not map the entire file, EXE$CRMPSC remaps the file with a cathedral WCB (see Section 3.5). It copies the end-of-file virtual block number from the file control block to MMG$L_EFBLK.

6. If the section to be mapped is a copy-on-reference section, EXE$CRMPSC sets bit MMG$V_CHGPAGFIL in MMG$L_MMG_FLAGS as a signal that the section must be charged against the job's page file quota.

7. It checks that the PAGCNT argument is positive and, if not, returns the error status SS$_ILLPAGCNT.

8. It raises IPL to 2 to block AST delivery.

9. Prior to allocating a PSTE, it calls MMG$DALCSTXSCN (see Section 3.9.2) to check whether any PSTEs can be deallocated. A section table entry cannot always be deallocated synchronously on request. For example, if direct I/O is in progress to pages in the section, those pages cannot be deleted and hence the section cannot

be. After the I/O completes, a subsequent call to MMG$DALCSTXSCN will result in deallocation of the section table entry.

10. Unless the section is copy-on-reference and demand zero (a section probably being mapped from an image file), EXE$CRMPSC allocates a PSTE (see Figure 2.7) and initializes it. (A copy-on-reference demand zero section does not need a PSTE; its page faults require no I/O from a section file.)

    When the process section is being created as a part of image activation (see Chapter *Image Activation and Exit*), the original source for much of the data stored in the PSTE is an image section descriptor in the image file.

    a. EXE$CRMPSC copies the SEC$V_WRT, SEC$V_DZRO, and SEC$V_CRF bits from the FLAGS to SEC$L_FLAGS.

    b. It stores in SEC$L_WINDOW the address of the WCB from the channel control block (CCB) or from the PSTE to which the CCB points. Recall that if multiple sections are mapped from the same file, there is one PSTE for each section but only one CCB and one WCB.

    c. It checks that the file has been opened in a manner consistent with the section flags: if the section is writable but not copy-on-reference, the file must have been opened for write access. If the file was opened for write access, then EXE$CRMPSC sets SEC$V_WRT in SEC$L_FLAGS. If the file was not opened for write access, but SEC$V_WRT is set, EXE$CRMPSC sets SEC$V_CRF so that the section will be created as copy-on-reference with backing store in a page file.

    d. It copies the VBN argument to SEC$L_VBN. If the VBN argument is 0, its default, EXE$CRMPSC replaces it with 1.

    e. It copies the PAGCNT argument, if present, to SEC$L_UNIT_CNT after checking that the file contains at least that many blocks between SEC$L_VBN and its end-of-file. If the argument is absent, it initializes SEC$L_UNIT_CNT to the difference between the end-of-file block and SEC$L_VBN.

    f. If this is the first section mapped on this file, it stores the section offset in CCB$L_WIND and the index in the PSTE forward and backward links. Otherwise, it inserts the PSTE into the chain of other PSTEs paging on that channel.

    g. It clears SEC$L_VPX, the virtual page index.

    h. It initializes SEC$L_REFCNT to 1 and sets the section table entry flag SEC$V_INPROG to ensure that the section is not inadvertently deleted before its PTEs are initialized. If the system service cannot complete, it may place the kernel thread into a wait state at IPL 0. If the process were deleted at that point, the Delete Process ($DELPRC) system service would be able to detect such a section by the set SEC$V_INPROG flag and decrement the biased reference count.

i. It converts the section pagelet fault cluster argument, PFC, to a page fault cluster value and stores the minimum of that and 127 in SEC$L_PFC.

11. EXE$CRMPSC forms a template L3PTE for the section's pages (see Figure 2.12). The L3PTE has both type bits set; the section table index in bits <47:32> (or zero for a copy-on-reference demand zero section); and the WRT, CRF, and DZRO bits copied from the section flags. It calculates the page owner mode and protection bits based on MMG$L_ACCESS_MODE, the writable flag in SEC$L_FLAGS, and the input section flags specifying the mode allowed to write the section pages.

12. If the SEC$V_EXPREG flag was specified in the FLAGS system service argument, it calculates the starting and ending addresses to map based on the pagelet count multiplied by 512 and the contents of RDE$PQ_FIRST_FREE_VA in the P0 or P1 space RDE, whichever is appropriate.

    If the SEC$V_EXPREG flag was not specified, it determines the address of the RDE corresponding to the INADR argument. If that region is intended for memory-resident and Galaxywide global sections, it returns the error status SS$_NOSHPTS. A process may not create process-private address space to be mapped with a shared page table. EXE$CRMPSC checks whether the access mode from which the service was requested is allowed to create pages in this region and, if not, returns the error status SS$_NOPRIV. It calculates the actual and useful address ranges to be mapped, based on the INADR and PAGCNT arguments and, depending on the section type, number of blocks in the section file.

    Regardless of the value of the SEC$V_EXPREG flag, an integral number of Alpha pages will be mapped. If the pagelet count does not represent an integral number of pages, the page at the high-address end of the section will be only partly occupied by the section. Its L3PTE will have the PTE$V_PARTIAL_SECTION bit set. Either MMG$V_PARTIAL_FIRST or MMG$V_PARTIAL_LAST in MMG$L_MMG_FLAGS is set, indicating that the first or last page to be mapped is partial. Which is partial depends on the order of mapping, which depends on how the address range was specified in the INADR argument.

13. EXE$CRMPSC determines whether the section must be mapped one page at a time:

    — If the new address space does not already exist, is entirely within a region, and can all be created without hitting any of the limits to growth described in Section 3.3, EXE$CRMPSC adjusts RDE$PQ_FIRST_FREE_VA and initializes the section's L3PTEs. It then increases SEC$L_REFCNT by the number of pages just mapped. If the section is not an integral number of physical pages, EXE$CRMPSC sets PTE$V_PARTIAL_SECTION in the L3PTE that maps the page with the highest address.

    — If the section to be mapped is copy-on-reference or demand zero, or if the space to be created overmaps existing space or cannot all be created, EXE$CRMPSC calls MMG$CREDEL, described in Section 3.1.1, specifying MAPSECPAG_RDE (see Section 3.6.1.2) as the per-page routine.

14. EXE$CRMPSC calculates the starting virtual page number of the section and stores it in SEC$L_VPX.
15. It decrements SEC$L_REFCNT to remove the extra reference, unnecessary now that the reference count reflects the mapped L3PTEs, and clears the SEC$V_INPROG flag.
16. If PHD$V_DALCSTX in PHD$L_FLAGS is set, indicating one or more sections to be deallocated, it calls MMG$DALCSTXSCN (see Section 3.9.2) to deallocate them.
17. EXE$CRMPSC returns any unused page file quota, records peak page file use and virtual size statistics, and stores return information in the optional RETADR argument.
18. It returns to its requestor.

3.6.1.2 MAPSECPAG_RDE Routine for a Process Section

MAPSECPAG_RDE is called with a number of arguments, including the L3PTE contents for the new page, number of pages in the section, number of pages to be mapped, addresses of the section table entry and RDE, and flags that control its actions.

For a process section, it takes the following steps:

1. Within initialization code, executed only once, MAPSECPAG_RDE sets the NO_OVERMAP flag in MMG$L_MMG_FLAGS if it is set in MMG$L_VFYFLAGS. It minimizes the requested number of pages to be mapped with the number of pages in the section. For a section file section being mapped in reverse order (from high address to low) whose highest address page is partial, it maps the first page with PTE$V_PARTIAL_SECTION set. It increments SEC$L_REFCNT.

   It replaces its own address in MMG$L_PAGESUBR so as to bypass the initialization code the next time it is entered.
2. MAPSECPAG_RDE increments the section table entry's reference count to reflect that one more L3PTE maps a page in that section.
3. It calls MMG$CREPAG_64, described in Section 3.4.1.2, which stores the template L3PTE contents into the next L3PTE and charges against job page file quota.
4. MAPSECPAG_RDE returns to its caller, MMG$CREDEL, which continues to call it until there are no more pages to be mapped or until one of the limits to growth is reached.

   For a section file section being mapped in forward order (from low address to high) whose highest address page is partial, MAPSECPAG_RDE maps the last page with PTE$V_PARTIAL_SECTION set.

3.6.1.3 $CRMPSC_FILE_64 System Service

The Create and Map Private Disk File Section ($CRMPSC_FILE_64) system service procedure, EXE$CRMPSC_FILE_64 in module SYS_CRMPSC_64, runs in kernel mode. It resembles the $CRMPSC system service requested to create a file section, but its arguments include a region ID, and all its address arguments are 64 bits. Thus it can be used to create a file section in P0, P1, or P2 space, either in a default region or a user-created one.

EXE$CRMPSC_FILE_64 takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:
   a. If the START_VA_64 argument was omitted and the flag SEC$V_EXPREG was clear, it returns the error status SS$_IVSECFLG.
   b. It rounds the FAULT_CLUSTER argument, if specified, up to a page boundary.
   c. It checks that the FILE_OFFSET_64 and LENGTH_64 arguments are multiples of the size of a disk block, returning the error status SS$_OFF_NOTBLKALGN or SS$_LEN_NOTBLKMULT if not.
   d. It checks that the START_VA_64 argument, if specified, is aligned on a page boundary, returning the error status SS$_VA_NOTPAGALGN if not.
   e. It confirms that the specified channel has been assigned; that its associated device is directory-structured, files-oriented, and random access; and that a file is open on the channel. In case of error, it returns the error status SS$_NOTFILEDEV or SS$_IVCHNLSEC.
   f. If the WCB does not map the entire file, EXE$CRMPSC_FILE_64 remaps the file with a cathedral WCB (see Section 3.5). It copies the end-of-file virtual block number from the file control block to MMG$L_EFBLK.
2. It raises IPL to 2 to block AST delivery.
3. It determines the address of the RDE corresponding to the REGION_ID_64 argument, returning the error status SS$_IVREGID if the ID is invalid. If that region is intended for memory-resident and Galaxywide global sections, it returns the error status SS$_NOSHPTS. It checks whether the access mode from which the service was requested is allowed to create pages in this region and, if not, returns the error status SS$_NOPRIV.
4. If the section is demand zero and copy-on-reference, EXE$CRMPSC_FILE_64 reduces the number of pagelets to be mapped to the number of pagelets between the FILE_OFFSET_64 argument and the end-of-file.
5. For a section that is not demand zero copy-on-reference, it allocates a PSTE (see Figure 2.7) and initializes it, as described in Section 3.6.1.1. (A copy-on-reference demand zero section does not need a PSTE; its page faults require no I/O from a section file.)

6. It forms a template L3PTE for the section's pages (see Figure 2.12). The L3PTE has both type bits set; the section table index in bits <47:32> (or zero for a demand zero copy-on-reference section); and the WRT, CRF, and DZRO bits copied from the section flags. It calculates the page owner mode and protection bits based on the maximized access mode and the writable flag in SEC$L_FLAGS. The $CRMPSC_FILE_64 service differs from the $CRMPSC service in that there is no input argument to specify access mode allowed to write the section: if the section is writable, the mode allowed to read determines the mode allowed to write.

7. If the SEC$V_EXPREG flag was specified in the FLAGS system service argument, EXE$CRMPSC_FILE_64 calculates the starting and ending addresses to map based on the LENGTH_64 argument and the contents of RDE$PQ_FIRST_FREE_VA in the RDE corresponding to the REGION_ID_64 argument. If that address range intersects with the gap (see Chapter 1), it moves the address range.

   If the SEC$V_EXPREG flag was not specified, it calculates the address based on the START_VA_64 and LENGTH_64 arguments. If the address range is not entirely within the specified region, it deallocates the PSTE and returns the error status SS$_PAGNOTINREG to its requestor.

   Regardless of the value of the SEC$V_EXPREG flag, an integral number of Alpha pages will be mapped. If the LENGTH_64 does not represent an integral number of pages, the page at the high-address end of the section will be only partly occupied by the section. Its L3PTE will have the PTE$V_PARTIAL_SECTION bit set.

8. EXE$CRMPSC_FILE_64 determines whether the section must be mapped one page at a time and maps it:

   — If the new address space does not already exist, is entirely within a region, and can all be created without hitting any of the limits to growth described in Section 3.3, it adjusts RDE$PQ_FIRST_FREE_VA and initializes the section's L3PTEs. It then increases SEC$L_REFCNT by the number of pages just mapped. If the section is not an integral number of physical pages, it sets PTE$V_PARTIAL_SECTION in the L3PTE that maps the page with the highest address.

   — If the section to be mapped cannot all be created at once, it loops, calling MMG_STD$CREPAG_64 (see Section 3.4.1.2) until the routine returns an error status or all pages are done. On each successful return, EXE$CRMPSC_FILE_64 increments SEC$L_REFCNT. When the section is completely mapped, it decrements SEC$L_REFCNT to remove the extra reference added at PSTE initialization.

     If MMG_STD$CREPAG_64 returns the error status SS$_ABORT, which means an overmapped page had to be deleted but a wait would have been required, EXE$CRMPSC_FILE_64 deletes the address space it created and repeats the loop, recreating the address space.

9. If PHD$V_DALCSTX in PHD$L_FLAGS is set, indicating the need to deallocate one or more overmapped and thus deleted sections, it calls MMG$DALCSTXSCN (see Section 3.9.2) to deallocate them.

10. It lowers IPL to 0.
11. It returns any unused reserved page file quota, records peak page file use and virtual size statistics, and stores return information in the optional RETURN_VA_64 argument.
12. If MMG_STD$CREPAG_64 returned an error status, EXE$CRMPSC_FILE_64 passes that status back to its requestor; otherwise, it returns SS$_CREATED.

3.6.2 PFN-Mapped Process Section Creation

To create a PFN-mapped section the system service must validate the arguments, determine starting and ending virtual addresses of the section, and initialize L3PTEs to describe the range of PFNs to be mapped. No PSTE is needed to describe the section.

The PFN fields in these L3PTEs contain the requested physical page numbers. The window bit is set in each L3PTE to indicate that the virtual page is PFN-mapped. The valid bit is set. These pages do not count against the process's working set. They cannot be paged, swapped, or locked into the process's working set. Moreover, no record is maintained in the PFN database that such pages are PFN-mapped.

3.6.2.1 $CRMPSC and PFN-Mapped Process Section Creation

The $CRMPSC system service enables a security persona with PFNMAP privilege to map a portion of its virtual address space to a specific range of physical addresses. Although the primary purpose of this feature is to map process-private address space to I/O addresses, it is also used to map specific physical memory pages. When such a section is larger than one page, it maps physically contiguous pages.

Requested to create a PFN-mapped section, EXE$CRMPSC takes the following steps:

1. In addition to making the argument validation checks described in Section 3.1.1, EXE$CRMPSC checks the INADR argument: unless the SEC$V_EXPREG flag was specified in the FLAGS argument, it confirms that the starting address is on an Alpha page boundary and that the ending address is one byte less than a page boundary. (It takes into account the possibility that the addresses have been specified in reverse order.) If the addresses are not correct, it returns the error status SS$_INVARG.
2. It creates and initializes the scratch space on the stack.
3. It calls MMG$VFY_SECFLG, in module SYSDGBLSC, to test the compatibility of the FLAGS arguments with each other. If the flags are incompatible, if the system service requestor specified the flag SEC$V_SHMGS, or if the argument is absent, it returns the error status SS$_IVSECFLG.
4. EXE$CRMPSC checks whether the CHAN argument is present, indicating an opened file, which would be incompatible with a PFN-mapped section. If so, it returns the error status SS$_IVSECFLG.

5. If SEC$V_WRT was specified in the FLAGS argument and if this system is a Galaxy instance, EXE$CRMPSC checks whether this is a request to map instance-private memory. If not, it returns the error status SS$_INVPFN; only private memory can be mapped writable.

6. It checks that the PAGCNT argument is positive and, if not, returns the error status SS$_ILLPAGCNT. (Note that for a PFN-mapped section, the PAGCNT argument specifies a number of pages, not pagelets.)

7. It raises IPL to 2 to block AST delivery.

8. It calls MMG_STD$SEC_PRIVCHK, in module SYSCRMPSC, to check whether the current security persona has the privileges necessary to create a PFN-mapped section and, if not, returns the error status SS$_NOPRIV.

9. It calls MMG$DALCSTXSCN (see Section 3.9.2), to deallocate any process section whose reference count has gone to zero.

10. EXE$CRMPSC forms a template L3PTE (see Figure 2.12) for pages in the section. The L3PTE has the valid and window bits set. EXE$CRMPSC calculates its page owner mode and protection bits based on MMG$L_ACCESS_MODE, the writable flag in SEC$L_FLAGS, and the bits in the FLAGS argument specifying the mode allowed to write the section pages.

11. If the SEC$V_EXPREG flag is clear, EXE$CRMPSC continues with step 15. If both the SEC$V_EXPREG and the undocumented SEC$V_GRANHINT flags were set in the FLAGS system service argument, it determines which, if any, granularity hint value is appropriate for the input PFN, input starting virtual address, section size, and state of the region in which the section is to be mapped. The input PFN is specified by the VBN argument, which is named for its more typical use.

    Chapter 1 describes how granularity hint regions improve translation buffer (TB) performance.

    It first tries a 512-page granularity hint region. If that cannot be made to work, possibly because the input PAGCNT argument is too much smaller than the granularity hint region size, it tries a 64-page region, and then an eight-page region. As part of testing for a granularity hint region, EXE$CRMPSC tries to expand the region in which the section is to be mapped, checking that the new address space is entirely within the virtual address region and can all be created without hitting any of the limits to growth described in Section 3.3. If the expansion is unsuccessful, EXE$CRMPSC continues with step 16.

    If the expansion is successful, EXE$CRMPSC adjusts RDE$PQ_FIRST_FREE_VA. Note that it may expand the virtual address region so as to align the starting virtual address on a granularity hint region boundary suitable for the input PFN and length, and it may require that additional PFNs be mapped to align the starting physical virtual address on a corresponding boundary.

    It incorporates as many pages as possible into granularity hint regions and maps them one region at a time, as described in step 14. It continues with step 17.

12. If the SEC$V_EXPREG flag was set but the SEC$V_GRANHINT flag was clear (or if an optimal granularity hint region could not be formed in the previous step), EXE$CRMPSC calculates the starting and ending section addresses based on the page count and contents of RDE$PQ_FIRST_FREE_VA in the P0 or P1 space RDE, depending on the INADR argument.

13. If the address space to be created overmaps existing space or cannot all be created, EXE$CRMPSC continues with step 16. If the address space to be created does not overmap existing address space and it can all be created, the routine checks whether the PFN-mapped section meets the requirements for a granularity hint region:

    — The page count must be 8, 64, or 512.

    — The starting virtual and physical addresses must be aligned multiples of the page count.

    It calculates the appropriate granularity hint value, making it zero if the section does not meet the requirements for a granularity hint region.

14. EXE$CRMPSC inserts the granularity hint value into the template L3PTE and then initializes all the section's L3PTEs. For each L3PT containing those L3PTEs, it takes the following steps:

    a. It tests whether the L3PT is still valid and, if not, faults it in.

    b. It acquires the MMG spinlock and confirms that the L3PT is still valid. If not, it releases the MMG spinlock, refaults the page, and reacquires the spinlock.

    c. If the L3PT did not previously map any window pages or locked pages, EXE$CRMPSC increments PHD$L_PTCNTLCK to indicate one more locked page table page.

    d. It sets PTE$V_WINDOW in the L2PTE that maps this L3PT and locks the L3PT into the process's working set list by setting WSLX$V_PFNLOCK in its working set list entry.

    e. It adds the number of PFN-mapped pages to the L3PT's PFN$W_PT_WIN_CNT.

    f. It releases the MMG spinlock.

    It continues with step 17.

15. If the SEC$V_EXPREG flag was clear, EXE$CRMPSC determines the address of the RDE corresponding to the INADR argument. If that region is intended for memory-resident and Galaxywide global sections, it returns the error status SS$_NOSHPTS. It checks whether the access mode from which the service was requested is allowed to create pages in this region and, if not, returns the error status SS$_NOPRIV. It calculates the actual and useful address ranges to be mapped, based on the INADR and PAGCNT arguments.

To varying extents, it incorporates pages into granularity hint regions:

- If the SEC$V_GRANHINT flag was clear, it continues with step 13.
- If the SEC$V_GRANHINT flag was set and the INADR range was specified in ascending order, it incorporates as many pages as possible into granularity hint regions and maps them one region at a time, as described in step 14, and continues with step 17.
- If the SEC$V_GRANHINT flag was set but the INADR range was specified in reverse order, it continues with step 13.

16. EXE$CRMPSC calls MMG$CREDEL (see Section 3.1.1), specifying MAPSECPAG_RDE (see Section 3.6.2.2) as the per-page routine.
17. If PHD$V_DALCSTX in PHD$L_FLAGS is set, indicating one or more sections to be deallocated, EXE$CRMPSC calls MMG$DALCSTXSCN (see Section 3.9.2) to deallocate them.
18. EXE$CRMPSC records peak virtual size statistics and stores return information in the optional RETADR argument.
19. It restores the IPL at entry and returns to its requestor.

3.6.2.2 MAPSECPAG_RDE Routine for a PFN-Mapped Section

MAPSECPAG_RDE is called with a number of arguments, including the L3PTE contents for the new page, number of pages in the section, number of pages to be mapped, address of the RDE, and flags that control its actions.

Called to create a PFN-mapped section page, MAPSECPAG_RDE takes the following steps:

1. Within initialization code, executed only once, MAPSECPAG_RDE sets the NO_OVERMAP flag in MMG$L_MMG_FLAGS if it is set in MMG$L_VFYFLAGS.
   It minimizes the number of pages requested in the PAGCNT argument with the number of pages in the address range specified by the INADR argument. It replaces its own address in MMG$L_PAGESUBR so as to bypass the initialization code the next time it is entered.
2. MAPSECPAG_RDE calls MMG$CREPAG_64 (see Section 3.4.1.2), which stores the template L3PTE contents into the next L3PTE.
3. MAPSECPAG_RDE calculates the contents of the next L3PTE by incrementing or decrementing the PFN value from the current PTE, depending on the order of mapping.
4. It returns to its caller, MMG$CREDEL, which continues to call it until there are no more pages to be mapped or until one of the limits to growth is reached.

3.6.2.3 $CRMPSC_PFN_64 System Service

The Create and Map Private Page Frame Section ($CRMPSC_PFN_64) system service is requested to create and map a PFN-mapped section. It resembles the $CRMPSC system service requested to create a PFN-mapped section, but its arguments include a region ID, and all its address arguments are 64 bits. Thus, it can be used to create a PFN-mapped section in P0, P1, or P2 space, either in a process-permanent region or a user-created one.

The requestor specifies section name, ident, protection, first PFN to be mapped, region ID, relative page number, number of pages in the section, access mode, number of pages to be mapped, and section flags. The requestor has not opened a section file to be mapped so does not specify channel number. The requestor does not specify page fault cluster; such a section incurs no page faults.

The $CRMPSC_PFN_64 system service procedure, EXE$CRMPSC_PFN_64 in module SYS_CRMPSC_64, runs in kernel mode.

It takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:
   a. If only seven arguments were passed, omitting START_VA_64, and if the flag SEC$V_EXPREG was clear, it returns the error status SS$_IVSECFLG.
   b. It sets the flag SEC$V_PFNMAP in the FLAGS argument in case it was clear.
   c. It checks that the PAGE_COUNT argument is nonzero and smaller than the maximum amount of physical memory, returning the error status SS$_ILLPAGCNT if not.
   d. It checks that the START_VA_64 argument, if specified, is aligned on a page boundary, returning the error status SS$_VA_NOTPAGALGN if not.
   e. If SEC$V_WRT was specified in the FLAGS argument and if this system is a Galaxy instance, EXE$CRMPSC_PFN_64 checks whether this is a request to map instance-private memory. If not, it returns the error status SS$_INVPFN; only private memory can be PFN-mapped writable.
   f. If the SEC$V_EXPREG flag was clear, EXE$CRMPSC_PFN_64 confirms that the START_VA_64 argument is within the specified region, returning the error status SS$_PAGNOTINREG if not.
2. It raises IPL to 2 to block AST delivery.
3. It determines the address of the RDE corresponding to the REGION_ID_64 argument, returning the error status SS$_IVREGID if the ID is invalid. If that region is intended for memory-resident and Galaxywide global sections, it returns the error status SS$_NOSHPTS. It checks whether the access mode from which the service was requested is allowed to create pages in this region and, if not, returns the error status SS$_NOPRIV.

4. It calls MMG_STD$SEC_PRIVCHK, in module SYSCRMPSC, to check whether the current security persona has the privileges necessary to create a PFN-mapped section and, if not, returns the error status SS$_NOPRIV.

5. EXE$CRMPSC_PFN_64 forms a template L3PTE for pages in the section. The L3PTE has the valid and window bits set and the starting PFN. EXE$CRMPSC_PFN_64 calculates its page owner mode and protection bits based on the maximized access mode. If the section is writable, the mode allowed to read determines the mode allowed to write.

6. If both the SEC$V_EXPREG and the undocumented SEC$V_GRANHINT flags were set in the FLAGS argument, EXE$CRMPSC_PFN_64 determines which, if any, granularity hint value is appropriate for the input PFN, input starting virtual address, section size, and state of the region in which the section is to be mapped.

   It first tries a 512-page granularity hint region. If that cannot be made to work, possibly because the input PAGE_COUNT argument is too much smaller than the granularity hint region size, it tries a 64-page region, and then an eight-page region. As part of testing for a granularity hint region, EXE$CRMPSC_PFN_64 tries to expand the region in which the section is to be mapped, checking that the new address space is entirely within the virtual address region and can all be created without hitting any of the limits to growth described in Section 3.3. If the expansion is unsuccessful, EXE$CRMPSC_PFN_64 continues with step 7.

   If the expansion is successful, EXE$CRMPSC_PFN_64 adjusts RDE$PQ_FIRST_FREE_VA. Note that it may expand the virtual address region so as to align the starting virtual address on a granularity hint region boundary suitable for the input PFN and length, and it may require that additional PFNs be mapped to align the starting physical virtual address on a corresponding boundary. It continues with step 9.

7. If the expansion in step 6 was unsuccessful or if SEC$V_EXPREG was clear, EXE$CRMPSC_PFN_64 rounds the starting and ending addresses down to an Alpha page boundary and calculates the desired page count based on the difference between them. It checks that the address range is entirely within the specified region and does not overlap with already existing space. If both are true, it continues with the next step. Otherwise, it continues with step 13.

8. EXE$CRMPSC_PFN_64 tries to expand the region in which the section is to be mapped, checking that the new address space can all be created without hitting any of the limits to growth described in Section 3.3. If expansion is unsuccessful, EXE$CRMPSC_PFN_64 continues with step 13.

   If the expansion is successful, EXE$CRMPSC adjusts RDE$PQ_FIRST_FREE_VA and continues with step 11.

9. If the SEC$V_GRANHINT flag was set, and the region grows toward ascending addresses, it incorporates as many pages as possible into granularity hint regions and maps them one granularity hint region at a time, as described in step 12. It continues with step 14.

10. If the SEC$V_GRANHINT flag was set but the region grows toward descending addresses, it continues with step 11.

11. EXE$CRMPSC_PFN_64 checks whether the PFN-mapped section meets the requirements for a granularity hint region:

    - The page count must be 8, 64, or 512.
    - The starting virtual and physical addresses must be aligned multiples of the page count.

    EXE$CRMPSC_PFN_64 calculates the appropriate granularity hint value, making it zero if the section does not meet the requirements to become a granularity hint region.

12. EXE$CRMPSC_PFN_64 inserts the granularity hint value into the template L3PTE and then initializes all the section's L3PTEs, incrementing the PFN for each new virtual page. For each L3PT containing those L3PTEs, it takes the following steps:

    a. It tests whether the L3PT is still valid and, if not, faults it in.

    b. It acquires the MMG spinlock and confirms that the L3PT is still valid. If not, it releases the MMG spinlock, refaults the page, and reacquires the spinlock.

    c. If the L3PT did not previously map any window pages or locked pages, EXE$CRMPSC_PFN_64 increments PHD$L_PTCNTLCK to indicate one more locked page table page.

    d. It sets PTE$V_WINDOW in the L2PTE that maps this L3PT and locks the L3PT into the process's working set list by setting WSLX$V_PFNLOCK in its working set list entry.

    e. It adds the number of PFN-mapped pages to the L3PT's PFN$W_PT_WIN_CNT.

    f. It releases the MMG spinlock.

    It continues with step 14.

13. When the section must be mapped one page at a time, EXE$CRMPSC_PFN_64 calls MMG_STD$CREPAG_64 (see Section 3.4.1.2), passing it the state of the SEC$V_NO_OVERMAP flag as well as MMG$V_NOWAIT_IPL0. EXE$CRMPSC_PFN_64 loops, calling MMG_STD$CREPAG_64 until the routine returns an error status or all pages are done.

    If MMG_STD$CREPAG_64 returns the error status SS$_ABORT, which means an overmapped page had to be deleted but a wait would have been required, EXE$CRMPSC_PFN_64 deletes the address space it created and repeats the loop, recreating the address space.

14. If PHD$V_DALCSTX in PHD$L_FLAGS is set, indicating the need to deallocate one or more overmapped and thus deleted sections, it calls MMG$DALCSTXSCN (see Section 3.9.2) to deallocate them.

15. It lowers IPL to 0.
16. It records peak page file use and virtual size statistics, and stores return information in the RETURN_VA_64 and RETURN_LENGTH_64 arguments.
17. If MMG_STD$CREPAG_64 returned an error status, EXE$CRMPSC_PFN_64 passes that status back to its requestor; otherwise, it returns SS$_CREATED.

3.7 Global Section Creation and Mapping

$CRMPSC and various other system services enable a process to create a global section or, if the section already exists, to map to it. The Install utility requests the $CRMPSC system service to create one or more global sections when an image is installed with the /SHARED qualifier.

The creation of a global section is similar to the creation of a process section except that additional data structures are involved. Chapter 2 shows the layouts of these data structures and describes them and their interrelations in more detail.

- A global section descriptor (GSD; see Figure 2.24), which enables subsequent map global section system service requests to determine whether the named section exists and to locate its global section table entry (GSTE).
- A GSTE (see Figure 2.7), analogous to the PSTE but part of the system header rather than of a PHD.
- Global page table entries (GPTEs), each of which describes the state of one global page in the section. GPTEs are used by the page fault handler when a process incurs a page fault for a global page. They are not used in address translation.

Each process has its own page table space and, in general, its own page tables to map process-private space. Typically, when a process maps to a global section, its L3PTEs that describe the specified address range are initialized with global page table indexes (GPTXs; see Figure 2.25).

Like a process-private section, a global section can consist of specific pages of memory or I/O address space. Creation of a global PFN-mapped section requires the PFNMAP privilege. The only data structure necessary to describe a global PFN-mapped section is a special form of GSD (see Figure 2.24). There are no GPTEs nor is there a GSTE. When a process maps to such a section, its L3PTEs are initialized with the valid and window bits set and PFNs based on GSD$L_BASEPFN.

Another type of global section is a demand zero section whose pages are backed in a page file. This type of section is called a global page-file section. Record Management Services (RMS) uses this type of section to implement global buffers on a file. The dynamic SYSGEN parameter GBLPAGFIL specifies the maximum number of page file pages that can be put to this use.

Another type of global section, new with OpenVMS Version 7.1, is a memory-resident global section. The pages of such a section do not page and are not backed up by a section file. Once initialized, the global pages are permanently valid. When a process maps to a memory-resident global section, any L3PTEs that map already valid pages of global section are initialized as valid. Once such pages are valid in a process's page table, they remain valid until deleted from the process's address space. They are not listed in the working set list and do not count against working set or page file quotas. A memory-resident global section is writable by definition.

Optionally, multiple processes can map memory-resident global sections with shared page tables, using the same L3PTs to map the global section. Shared L3PTs are permanently valid and not listed in the working set list. They do not count against working set quotas or page file quotas. Figure 2.28 shows shared page tables mapping a shared page. Use of shared page tables saves not only memory but also the time to map the section. For more flexibility, shared page tables can be created that give read-only access, enabling more reader processes to share page tables while fewer writer processes map with private page tables.

For optimum performance, the system manager registers a memory-resident global section in the Reserved Memory Registry (see Chapter 2) with the /ALLOCATE and /PAGE_TABLES qualifiers. These make it possible for granularity hint regions to be created for both the global section and its shared page tables.

On an OpenVMS Galaxy platform, a memory-resident global section can be created in memory shared among all the instances. Such a global section is referred to as a Galaxywide global section and as a shared memory global section. Galaxywide shared memory is reserved for various uses through the Galaxy Configuration utility. Through the utility the system manager can reserve a portion of shared memory for global sections but cannot apportion it to particular global sections.

Optionally, processes can map a Galaxywide global section with shared page tables and use the same L3PTs to map the section.

3.7.1 Creating Global Sections with $CRMPSC

Requested to create or map a global section, the $CRMPSC system service procedure, EXE$CRMPSC, takes the following steps:

1. As described in Section 3.6.1.1, it initializes scratch space on the stack, determines the actual and useful ranges to be mapped, and tests the compatibility of the flags in the FLAGS argument. It examines the FLAGS argument to determine what type of global section is to be created and what further checks are required:

   — If a global section is to be mapped and the requestor specified a value for the RELPAG argument, the RETADR argument must also have been specified.

   — If a PFN-mapped section or global page-file section is to be created, the CHAN argument should not be present.

- If a section file section is to be created, the CHAN argument must be present, the file must have been opened, and the WCB must map the entire file. If the section already exists, the CHAN argument need not be present.
- If the section is to be copy-on-reference, EXE$CRMPSC sets bit MMG$V_CHGPAGFIL in MMG$L_MMG_FLAGS.

2. It locks the GSD mutex for write access, raising IPL to 2. The GSD mutex synchronizes access to both the systemwide and group GSD lists.
3. EXE$CRMPSC calls MMG$DALCSTXSCN1 (see Section 3.9.2) to check the global (system) section table for any sections to be deleted.
4. It calls MMG_STD$GSDSCAN, in module SYSDGBLSC, to find the GSD, if any, that corresponds to the GSDNAM argument. MMG_STD$GSDSCAN attempts logical name translation of the GSDNAM argument, as described in the *OpenVMS Programming Concepts Manual.* If the translation fails, it uses the string specified by the service requestor as the global section name.

   MMG_STD$GSDSCAN scans the group or systemwide GSD list, depending on the type of section. In scanning the group list, it first compares the current security persona's UIC group code with the high word of GSD$L_PCBUIC. If they are equal, it then compares the global section names. Because a character string comparison is relatively lengthy, the routine first confirms that one is necessary by requiring that the hash values and the character string lengths be the same for the target section name and the one in the candidate GSD. If they are not the same, the global section names cannot be.

   If the names match, MMG_STD$GSDSCAN checks the match control information specified in the IDENT argument against GSD$L_IDENT. If there is a version incompatibility, MMG_STD$GSDSCAN continues to scan the list until it reaches the end or finds a match.

   Multiple versions of a global section with different version identifications and match control information can be installed. If a newer one were installed last and had match control specifying upward compatibility (match less or equal), it could be used with executables linked against it or earlier versions. If it had match control specifying no upward compatibility (match equal), an executable linked against an earlier version would not match; EXE$CRMPSC would continue to scan the list and find the earlier one.
5. If MMG_STD$GSDSCAN locates a matching GSD, EXE$CRMPSC is being requested to map to an existing section, and it transfers control to EXE$MGBLSC, at step 7 in the description in Section 3.8.1.
6. If no match is found, EXE$CRMPSC is being requested to create a new section. It calls MMG_STD$SEC_PRIVCHK, in module SYSCRMPSC, to check whether the current security persona has the privileges necessary to create the type of section specified by its FLAGS argument and, if not, unlocks the GSD mutex and returns the error status SS$_NOPRIV.

7. EXE$CRMPSC allocates paged pool for a GSD. The size of the GSD depends on whether the global section is PFN-mapped. If pool is unavailable, it unlocks the GSD mutex and returns the error status SS$_GSDFULL.

8. It begins to initialize the GSD, copying the section name to GSD$T_GSDNAM, storing the hash value in GSD$L_HASH, and clearing GSD$L_IPID.

9. If the section is PFN-mapped, EXE$CRMPSC clears GSD fields irrelevant to this type of section and copies the VBN argument to GSD$L_BASEPFN, the section name to GSD$T_PFNGSDNAM, and the contents of the PAGCNT argument to GSD$L_PAGES. (Note that for a PFN-mapped section, the PAGCNT argument specifies a number of pages, not pagelets.)

10. EXE$CRMPSC initializes GSD$L_FLAGS from the section flags and access mode.

11. EXE$CRMPSC calls MMG$INIT_ORB, in module SYSCRMPSC, which takes the following steps:

    a. It stores the current security persona's UIC in GSD$L_PCBUIC and clears GSD$L_FILUIC.

    b. It sets GSD$L_PROT to FFFF$_{16}$, a no-access protection mask. Protection information in the object rights block (ORB) is used instead of GSD$L_PROT.

    c. If the section is to map a file, MMG$INIT_ORB stores the address of the ORB associated with the open file in GSD$L_ORB. If there is no ORB associated with the file, MMG$INIT_ORB returns the error status SS$_ABORT. Otherwise, MMG$INIT_ORB copies ORB$L_OWNER into GSD$L_FILUIC and initializes the ACL mutex in the ORB.

       If the section is a PFN-mapped or global page-file section, MMG$INIT_ORB allocates an ORB from paged pool and initializes it, copying the current security persona's UIC to ORB$L_OWNER; bits from the PROT argument to ORBL_SYS_PROT, ORBL_GRP_PROT, ORB$L_OWN_PROT, and ORB$L_WOR_PROT; and the current security persona's class information, if any, to ORB$R_MIN_CLASS and ORB$R_MAX_CLASS. If pool for the ORB is unavailable, MMG$INIT_ORB returns the error status SS$_INSFMEM.

12. If MMG$INIT_ORB returns an error status, EXE$CRMPSC deallocates the GSD, unlocks the GSD mutex, and returns the error status to the system service requestor.

13. EXE$CRMPSC initializes GSD$L_IDENT from the IDENT argument.

14. If the section is PFN-mapped, EXE$CRMPSC continues with step 26.

15. Otherwise, it allocates a GSTE from the system header. If none is available, it deallocates the ORB and GSD, unlocks the mutex, and returns the error status SS$_SECTBLFUL.

16. EXE$CRMPSC takes most of the same steps to initialize a GSTE as for a PSTE (see steps 10a through 10i in Section 3.6.1.1). One additional step required for a global section is making the WCB a "shared" one if it is not already. This chiefly involves returning the byte count quota charged for it to the appropriate job, setting WCB$V_SHRWCB in WCB$B_ACCESS, and incrementing WCB$L_REFCNT to indicate one more reason the file should not be closed.

    Also, between clearing the virtual page index and setting the section reference count to 1, it executes a memory barrier instruction. The memory barrier ensures that another processor cannot see a nonzero reference count without also seeing a zero virtual page index. The zero virtual page index prevents the section table entry from being used to map a global section before it is fully initialized.

17. It stores the GSTE index in GSD$L_GSTX.

18. If the section is a section file section rather than a global page-file section, it copies the file owner UIC to GSD$L_FILUIC.

19. If the section is a global page-file section, it updates MMG$GL_GBLPAGFIL, the number of pages of page file that can be used for this purpose, to reflect any change in the dynamic SYSGEN parameter GBLPAGFIL. It subtracts the section's page count from MMG$GL_GBLPAGFIL.

    If mapping this section would exceed the allowed global page file count, EXE$CRMPSC deallocates the GSD, ORB, and GSTE; unlocks the mutex; and returns the error status SS$_EXGBLPAGFIL.

20. It converts the number of pagelets in the section to pages and allocates a set of contiguous GPTEs, one for each global page plus two additional GPTEs, one at the beginning of the set and one at the end. The two additional GPTEs are cleared to serve as stoppers, limits to modified page write clustering (see Chapter 4).

    If there are insufficient GPTEs, EXE$CRMPSC deallocates the data structures it built, restores the page file charge, unlocks the mutex, and returns the error status SS$_GPTFULL.

21. It zeros the beginning stopper GPTE and calculates the virtual page number of the second GPTE (skipping the stopper GPTE) and stores that in SEC$L_VPX.

22. It forms template PTE contents for the GPTEs (see Figure 2.26).

23. It then loops, initializing GPTEs with the template PTE contents.

24. If necessary, it sets PTE$V_PARTIAL_SECTION in the highest GPTE to indicate that the page will be only partly occupied by global section data.

25. It zeros the end stopper GPTE.

26. It inserts the GSD at the front of the group or systemwide list, enabling a more recently installed global section to supersede an earlier one (see step 4).

27. The global section has been created. It transfers control to EXE$MGBLSC (at step 11 in the description in Section 3.8.1) to map it into the process's virtual address space as an existing section.

3.7.2 $CREATE_GFILE System Service

The Create Permanent Global Disk File Section ($CREATE_GFILE) system service procedure, EXE$CREATE_GFILE in module SYS_GBLSEC_64, runs in kernel mode.

EXE$CREATE_GFILE takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:

   a. If the FAULT_CLUSTER argument was specified, it rounds it up to an integral number of pages.

   b. It checks the FLAGS argument to confirm that

      — SEC$V_DZRO and SEC$V_CRF are not both specified

      — If SEC$V_DZRO is set, SEC$V_WRT is as well

      If any condition is false, it returns the error status SS$_IVSECFLG. Otherwise, it sets the flags SEC$V_PERM and SEC$V_GBL in case they were clear. It stores the maximized access mode in SEC$V_AMOD. If the section is writable, the mode allowed to read determines the mode allowed to write.

2. It calls local routine $CREATE_GFILE_INT (see Section 3.7.3) to create the section. EXE$CREATE_GFILE returns any error status to its requestor.

3. If $CREATE_GFILE_INT returns SS$_NORMAL, the section already exists and was not created. EXE$CREATE_GFILE locks the GSD mutex, raising IPL to 2; decrements the section reference count, which had been incremented; and unlocks the mutex. It returns SS$_DUPLNAM to its requestor.

4. EXE$CREATE_GFILE calls MMG_STD$CHKPRO_AUDIT, in module SYSCRMPSC, to check whether the current security persona's creation of the global section needs to be audited.

5. EXE$CREATE_GFILE calls MMG_STD$DELGBLWCB (see Section 3.9.4) to close any open files associated with temporary global sections whose reference counts have gone to zero and to delete their WCBs.

6. If an audit is necessary, EXE$CREATE_GFILE lowers IPL to 0 and performs the actual audit of the global section creation.

7. It acquires the MMG spinlock, decrements the section reference count, which had been incremented, and releases the spinlock. It returns SS$_CREATED to its requestor if the section was created or an error status from MMG_STD$CHKPRO_AUDIT.

3.7.3 $CREATE_GFILE_INT Routine

$CREATE_GFILE_INT, in module SYS_GBLSEC_64, performs much of the work of the $CREATE_GFILE and $CRMPSC_GFILE_64 system services.

It takes the following steps:

1. It checks that the FILE_OFFSET_64 and LENGTH_64 arguments are multiples of the size of a disk block, returning the error status SS$_OFF_NOTBLKALGN or SS$_LEN_NOTBLKMULT if not.
2. It confirms that the specified channel has been assigned; that its associated device is directory-structured, files-oriented, and random access; and that a file is open on the channel. In case of error, it returns the error status SS$_NOTFILEDEV, SS$_IVCHNLSEC, or SS$_IVCHAN.
3. If the WCB does not map the entire file, $CREATE_GFILE_INT remaps the file with a cathedral WCB (see Section 3.5). It copies the end-of-file virtual block number from the file control block to MMG$L_EFBLK.
4. It locks the GSD mutex for write, raising IPL to 2.
5. It calls MMG_STD$DALCSTXSCN (see Section 3.9.2) to check the global (system) section table for any sections to be deleted.
6. It calls MMG_STD$GSDSCAN (see Section 3.7.1) to find the GSD, if any, that corresponds to the GSDNAM argument.
7. If MMG_STD$GSDSCAN returns an error status other than SS$_NOSUCHSEC, $CREATE_GFILE_INT unlocks the mutex, calls MMG_STD$DELGBLWCB (see Section 3.9.4) to close any open files associated with temporary global sections whose reference counts have gone to zero and to delete their WCBs, and returns the error status to its caller.
8. Otherwise, $CREATE_GFILE_INT checks that the global section ident is positive and, if not, unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_IVSECIDCTL to its caller.
9. If the section already exists, it checks whether the section is a PFN-mapped, page-file, or memory-resident section and, if so, unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_GBLSEC_MISMATCH.

   It checks that the requesting access mode is allowed to map the section and, if not, unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_NOPRIV.

   It determines the address of the GSTE, acquires the MMG spinlock, and increments SEC$L_REFCNT to prevent section deletion. It releases the MMG spinlock. With the section's deletion blocked, $CREATE_GFILE_INT can safely unlock the GSD mutex. It returns SS$_NORMAL to its caller.

10. If the section does not already exist, it calls MMG_STD$SEC_PRIVCHK, in module SYSCRMPSC, to check whether the current security persona has the privileges necessary to create a section file global section and, if not, unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_NOPRIV.

11. It allocates paged pool for a GSD. If pool is unavailable, it unlocks the GSD mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_GSDFULL.

12. It initializes the GSD, copying the section name to GSD$T_GSDNAM, storing the hash value in GSD$L_HASH, and clearing GSD$L_IPID. It initializes GSD$L_FLAGS from the section flags and access mode.

13. It calls MMG_STD$INIT_ORB (see Section 3.7.1) to allocate and initialize an ORB that describes the protection on the GSD. If it returns an error status, $CREATE_GFILE_INT deallocates the GSD, unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status to its caller.

14. It checks whether the IDENT_64 argument is greater than SEC$K_MATLEQ and, if so, deallocates the GSD and ORB, unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_IVSECIDCTL to its caller.

15. It allocates a GSTE from the system header. If none is available, it deallocates the GSD and ORB, unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_SECTBLFUL.

16. $CREATE_GFILE_INT takes most of the same steps to initialize a GSTE as for a PSTE for a process section (see steps 10a through 10i in Section 3.6.1.1). One additional step required for a global section is making the WCB a shared one if it is not already. This chiefly involves returning the byte count quota charged for it to the appropriate job, setting the bit WCB$V_SHRWCB in WCB$B_ACCESS, and incrementing WCB$L_REFCNT to indicate one more reason the file should not be closed.

17. It stores the GSTE index in GSD$L_GSTX.

18. It copies the section flags from the GSD to the GSTE.

19. It copies the file owner UIC to GSD$L_FILUIC.

20. It calculates the number of pages to be mapped based on the LENGTH_64 argument and actual number of blocks in the section file to be mapped.

21. It sanity checks that the number of pages can be represented as a positive 32-bit number and, if not, returns SS$_GPTFULL to its caller.

22. It allocates a set of contiguous GPTEs, one for each global page plus two additional GPTEs, one at the beginning of the set and one at the end. The two additional GPTEs are cleared and serve as stoppers, limits to modified page write clustering (see Chapter 4).

    If there are insufficient GPTEs, $CREATE_GFILE_INT unlocks the mutex, deallocates the data structures it built, calls MMG_STD$DELGBLWCB, and returns the error status SS$_GPTFULL.

23. It updates performance cells (PMS$GL_GBLPAGCNT and PMS$GL_GBLPAGMAX).
24. It zeros the beginning stopper GPTE and calculates the virtual page number of the second GPTE (skipping the stopper GPTE) and stores that in SEC$L_VPX.
25. It forms template PTE contents for the GPTEs (see Figure 2.26).
26. It then loops, initializing GPTEs with the template PTE.
27. If necessary, it sets PTE$V_PARTIAL_SECTION in the highest GPTE to indicate that the page will be only partly occupied by global section data.
28. It zeros the end stopper GPTE.
29. It inserts the GSD at the front of the group or systemwide list.
30. It unlocks the mutex and returns SS$_CREATED to its caller.

3.7.4 $CRMPSC_GFILE_64 System Service

The Create and Map Global Disk File Section ($CRMPSC_GFILE_64) system service procedure, EXE$CRMPSC_GFILE_64 in module SYS_GBLSEC_64, runs in kernel mode.

EXE$CRMPSC_GFILE_64 takes the following steps:

1. It validates the arguments with which it was requested, making the same checks as EXE$CREATE_GFILE (see Section 3.7.2), with the following additions:
   - If the START_VA_64 argument was omitted or contains zero, and the flag SEC$V_EXPREG was clear, it returns the error status SS$_IVSECFLG.
   - It checks that the START_VA_64 is aligned on a page boundary, returning the error status SS$_VA_NOTPAGALGN if not.
   - It checks that the SECTION_OFFSET_64, LENGTH_64, and MAP_LENGTH_64 arguments are multiples of the size of a disk block, returning the error status SS$_OFF_NOTBLKALGN or SS$_LEN_NOTBLKMULT if not.
2. It sets the flag SEC$V_GBL in case it was clear.
3. It calls local routine $CREATE_GFILE_INT (see Section 3.7.3) to create the section if it does not exist. EXE$CRMPSC_GFILE_64 returns any error status to its requestor.
4. It calls MMG_STD$CHKPRO_AUDIT to check access to the file.
5. If access is allowed, EXE$CRMPSC_GFILE_64 determines how much of the section to map: if the MAP_LENGTH_64 argument was omitted, it calculates the difference between the length of the created section and the SECTION_OFFSET_64 argument. It calls $MGBLSC_GFILE_INT (see Section 3.8.2.1) to map the section.
6. EXE$CRMPSC_GFILE_64 acquires the MMG spinlock, decrements the section reference count, which had been incremented, and releases the spinlock.

7. If the section was created, it records peak page file use and virtual size statistics, and stores return information in the RETURN_VA_64 argument.
8. It returns to its requestor the status from $CREATE_GFILE_INT or, if there was a mapping error, the error status from $MGBLSC_GFILE_INT.

3.7.5 $CREATE_GPFILE System Service

The Create Permanent Global Page File Section ($CREATE_GPFILE) system service is requested to create a demand zero global section whose backing store is the page file. The requestor specifies section name, ident, length, access mode, and section flags. Section protection is specified by the requestor rather than being derived from a section file. The requestor does not open a section file so does not specify channel number or offset into the file. The requestor does not specify page fault cluster; its value is determined by the SYSGEN parameter PFCDEFAULT.

The $CREATE_GPFILE system service procedure, EXE$CREATE_GPFILE in module SYS_GBLSEC_64, runs in kernel mode. It resembles EXE$CREATE_GFILE (see Section 3.7.2), with the following major differences:

- There are no output arguments to validate.
- There is no need to check the channel argument or remap the section file.
- The section flags set automatically by EXE$CREATE_GPFILE are SEC$V_PERM, SECV_GBL, SECV_DZRO, SEC$V_WRT, and SEC$V_PAGFIL.
- It calls $CREATE_GPFILE_INT (see Section 3.7.6).

3.7.6 $CREATE_GPFILE_INT Routine

$CREATE_GPFILE_INT, in module SYS_GBLSEC_64, performs much of the work of the $CREATE_GPFILE and $CRMPSC_GPFILE_64 system services.

It resembles $CREATE_GFILE_INT (see Section 3.7.3), with the following major differences:

- If a global section has already been created with matching name and ident, it must be a page file section for a successful match. If not, $CREATE_GPFILE_INT returns the error status SS$_GBLSEC_MISMATCH.
- The GSTE is initialized with a zero SEC$L_WINDOW.
- If the system manager has changed SYSGEN parameter GBLPAGFIL, $CREATE_GPFILE_INT records the new value in MMG$GL_LAST_GBLPAGFIL and updates MMG$GL_GBLPAGFIL, the number of available pages of global page file, by the difference between the old and new value.
- It checks that there are enough available pages of global page file for the new global section, returning SS$_EXGBLPAGFIL to its caller if not.
- It charges the section's pages against MMG$GL_GBLPAGFIL.

- The prototype GPTE has its type 0 and type 1 bits clear (see Figure 2.26).

3.7.7 $CRMPSC_GPFILE_64 System Service

The Create and Map Global Page File Section ($CRMPSC_GPFILE_64) system service is requested to create a demand zero global section whose backing store is the page file and map to it. The requestor specifies section name, ident, length, access mode, and section flags. Section protection is specified by the requestor rather than being derived from a section file. The requestor does not open a section file so does not specify channel number or offset into the file. The requestor does not specify page fault cluster; its value is determined by the SYSGEN parameter PFCDEFAULT.

The $CRMPSC_GPFILE_64 system service procedure, EXE$CRMPSC_GPFILE_64 in module SYS_GBLSEC_64, runs in kernel mode. It resembles EXE$CRMPSC_GFILE_64 (see Section 3.7.4), with the following major differences:

- There is no need to check the channel argument or remap the section file.
- The section flags set automatically by EXE$CRMPSC_GPFILE_64 are SEC$V_GBL, SECV_DZRO, SECV_WRT, and SEC$V_PAGFIL.
- EXE$CRMPSC_GPFILE_64 checks that the START_VA_64 and SECTION_OFFSET_64 arguments are page-aligned, returning the error status SS$_VA_NOTPAGALGN or SS$_OFF_NOTPAGALGN if not. It checks that the LENGTH_64, and MAP_LENGTH_64 arguments are page multiples, returning SS$_LEN_NOTPAGMULT if not.
- It calls $CREATE_GPFILE_INT and $MGBLSC_GPFILE_INT (see Sections 3.7.6 and 3.8.2.2).

3.7.8 $CREATE_GPFN System Service

The Create Permanent Global Page Frame Section ($CREATE_GPFN) system service is requested to create a PFN-mapped global section with no backing store. The requestor specifies section name, ident, protection, length, access mode, starting PFN, and section flags. The requestor has not opened a section file to be mapped so does not specify a channel argument or file offset. The requestor does not specify page fault cluster; such a section incurs no page faults.

The $CREATE_GPFN system service procedure, EXE$CREATE_GPFN in module SYS_GPFN_64, runs in kernel mode.

EXE$CREATE_GPFN takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:
   a. It sets the flags SECV_PFNMAP, SECV_PERM, and SEC$V_GBL in the FLAGS argument in case they were clear.

b. It checks that the PAGE_COUNT argument is nonzero and is smaller than the maximum amount of physical memory, and that the starting and ending PFNs can be expressed with 32 bits, returning the error status SS$_ILLPAGCNT if any test fails.

c. It checks that the protection code is valid, returning SS$_IVPROTECT if not.

d. If SEC$V_WRT was specified in the FLAGS argument and if this system is a Galaxy instance, it checks whether this is a request to map instance-private memory. If not, it returns the error status SS$_INVPFN; only private memory can be PFN-mapped writable.

2. It calls local routine $CREATE_GPFN_INT (see Section 3.7.9) to create the section. EXE$CREATE_GPFN returns any error status to its requestor.

3. If $CREATE_GPFN_INT returns SS$_NORMAL, the section already exists and was not created. EXE$CREATE_GPFN locks the GSD mutex, decrements GSD$L_REFCNT, which had been incremented, and unlocks the mutex. It returns SS$_DUPLNAM to its requestor.

4. EXE$CREATE_GPFN calls MMG_STD$CHKPRO_AUDIT to audit the section creation.

5. EXE$CREATE_GPFN locks the GSD mutex, decrements GSD$L_REFCNT, which had been incremented, and unlocks the mutex. It returns SS$_CREATED to its requestor.

3.7.9 $CREATE_GPFN_INT Routine

$CREATE_GPFN_INT, in module SYS_GPFN_64, performs much of the work of the $CREATE_GPFN and $CRMPSC_GPFN_64 system services.

It takes the following steps:

1. It locks the GSD mutex for write, raising IPL to 2.

2. It calls MMG_STD$DALCSTXSCN1 (see Section 3.9.2) to check the global (system) section table for any sections to be deleted.

3. It calls MMG_STD$GSDSCAN (see Section 3.7.1) to find the GSD, if any, that corresponds to the GSDNAM argument.

4. If MMG_STD$GSDSCAN returns an error status other than SS$_NOSUCHSEC, $CREATE_GPFN_INT unlocks the mutex and returns the error status to its caller.

5. Otherwise, $CREATE_GPFN_INT performs an additional sanity check on the global section ident: it confirms that the ident is positive and, if not, unlocks the mutex and returns the error status SS$_IVSECIDCTL to its caller.

6. If the section already exists, it checks whether the section is a PFN-mapped section and, if not, unlocks the mutex and returns the error status SS$_GBLSEC_MISMATCH.

It checks that the requesting access mode is allowed to map the section and, if not, unlocks the mutex and returns the error status SS$_NOPRIV.

Otherwise, it increments GSD$L_REFCNT to prevent section deletion. With the section's deletion blocked, $CREATE_GPFN_INT can safely unlock the GSD mutex. It returns SS$_NORMAL to its caller.

7. If the section does not already exist, $CREATE_GPFN_INT calls MMG_STD$SEC_PRIVCHK, in module SYSCRMPSC, to check whether the current security persona has the privileges necessary to create the type of section specified by its FLAGS argument and, if not, unlocks the mutex and returns the error status SS$_NOPRIV.
8. It allocates paged pool for an extended GSD. If pool is unavailable, it unlocks the GSD mutex and returns the error status SS$_GSDFULL.
9. It initializes the GSD, copying the section name to GSD$T_GSDNAM, storing the hash value in GSD$L_HASH, and clearing GSD$L_IPID. It clears GSD fields irrelevant to this type of section and copies the START_PFN argument to GSD$L_BASEPFN, the section name to GSD$T_PFNGSDNAM, and the contents of the PAGE_COUNT argument to GSD$L_PAGES. It initializes GSD$L_FLAGS from the section flags and access mode.
10. It calls MMG_STD$INIT_ORB (see Section 3.7.1) to allocate and initialize an ORB that describes the protection on the GSD. If it returns an error status, $CREATE_GPFN_INT deallocates the ORB and GSD, unlocks the mutex, and returns the error status to its caller.
11. It checks that the IDENT_64 argument is greater than SEC$K_MATLEQ and, if not, deallocates the GSD and ORB, unlocks the mutex, and returns the error status SS$_IVSECIDCTL to its caller.
12. Otherwise, it increments GSD$L_REFCNT to prevent section deletion, unlocks the mutex, and returns the success status SS$_CREATED to its caller.

3.7.10 $CRMPSC_GPFN_64 System Service

The Create and Map Global Page Frame Section ($CRMPSC_GPFN_64) system service is requested to create and map a PFN-mapped global section. The requestor specifies section name, ident, protection, first PFN to be mapped, region ID, relative page number, number of pages in the section, access mode, number of pages to be mapped, and section flags.

The $CRMPSC_GPFN_64 system service procedure, EXE$CRMPSC_GPFN_64 in module SYS_GPFN_64, runs in kernel mode.

EXE$CRMPSC_GPFN_64 takes the following steps:

1. It validates the arguments with which it was requested, making the same checks as EXE$CREATE_GPFN (see Section 3.7.8), with the following additions:

   — If the START_VA_64 argument was omitted and the flag SEC$V_EXPREG was clear, EXE$CRMPSC_GPFN_64 returns the error status SS$_IVSECFLG.

   — It tests the accessibility of the region ID argument, returning the error status SS$_ACCVIO if it is not accessible.

   — It checks that the START_VA_64 is aligned on a page boundary, returning the error status SS$_VA_NOTPAGALGN if not.

2. It calls $CREATE_GPFN_INT (see Section 3.7.9) to create the global section. If it returns an error status, EXE$CRMPSC_GPFN_64 returns the error status to its requestor.
3. Otherwise, it calls $MGBLSC_GPFN_INT (see Section 3.8.3.1) to perform the mapping.
4. EXE$CRMPSC_GPFN_64 locks the GSD mutex for write, decrements the extra section reference added by $CREATE_GFPN_INT, and unlocks the mutex.
5. If the mapping succeeded, it records peak virtual size statistics, and stores return information in the RETURN_VA_64 and RETURN_LENGTH_64 arguments. If the output arguments are inaccessible, it returns SS$_ACCVIO to its requestor.
6. Otherwise, it returns to its requestor the status from $MGBLSC_GPFN_INT.

3.7.11 $CREATE_GDZRO System Service

The Create Permanent Global Demand Zero Section ($CREATE_GDZRO) system service is requested to create a permanent, memory-resident, demand zero global section without backing store, either in local memory or in Galaxywide shared memory.

The requestor specifies section name, ident, length, access mode, and section flags. Section protection is specified by the requestor rather than being derived from a section file. Optionally, the requestor can specify in which resource affinity domain (RAD) the system service should create the global section.

The $CREATE_GDZRO system service procedure, EXE$CREATE_GDZRO in module SYS_GDZRO_64, runs in kernel mode. It resembles EXE$CREATE_GPFN (see Section 3.7.8), with the following major differences:

- If the requestor specified a RAD, EXE$CREATE_GDZRO checks that RAD support is enabled and returns error status SS$_BADRAD if not. If RAD support is enabled, it checks that the requestor specified a single RAD and one that actually exists, returning SS$_BADRAD if not.
- It sets the flags SECV_DZRO, SECV_PERM, SECV_WRT, SECV_GBL, and SEC$V_MRES in case they were clear.
- If the request is to create a Galaxywide section (if SEC$V_SHMGS is set), it checks that the service was requested from IPL 0, returning SS$_BADPARAM if not.

- If the request is to create a memory-resident section, it calls $CREATE_GDZRO_INT (see Section 3.7.12).
- If the request is to create a Galaxywide section, it calls $CREATE_SHMGS_INT (see Section 3.7.13).
- It returns the number of bytes that had been reserved for the global section in argument RESERVED_LENGTH_64, if present.

3.7.12 $CREATE_GDZRO_INT Routine

$CREATE_GDZRO_INT, in module SYS_GDZRO_64, performs much of the work of the $CREATE_GDZRO and $CRMPSC_GDZRO_64 system services when either is requested to create a memory-resident section.

It takes the following steps:

1. If the requestor specified in which RAD the global section should be allocated, $CREATE_GDZRO_INT checks that there is memory associated with that RAD, returning error status SS$_BADRAD if not.
2. It validates system service arguments, as described in Section 3.1.2.
3. It locks the GSD mutex for write, raising IPL to 2.
4. It calls MMG_STD$DALCSTXSCN (see Section 3.9.2) to check the global (system) section table for any sections to be deleted.
5. It calls MMG_STD$GSDSCAN (see Section 3.7.1) to find the GSD, if any, that corresponds to the GS_NAME_64 argument.
6. If MMG_STD$GSDSCAN returns an error status other than SS$_NOSUCHSEC, $CREATE_GDZRO_INT unlocks the mutex and returns the error status to its caller.
7. Otherwise, it checks that the IDENT_64 argument is positive and, if not, unlocks the mutex and returns the error status SS$_IVSECIDCTL to its caller.
8. If the section already exists, it checks whether the section is a demand zero memory-resident or Galaxywide global section and, if not, unlocks the mutex and returns the error status SS$_GBLSEC_MISMATCH.

   It checks that the requesting access mode is allowed to map the section and, if not, unlocks the mutex and returns the error status SS$_NOPRIV.

   Otherwise, it increments GSD$L_REFCNT to prevent section deletion. With the section's deletion blocked, it can safely unlock the GSD mutex. It returns SS$_NORMAL to its caller.
9. If the section does not already exist, $CREATE_GDZRO_INT calls MMG_STD$SEC_PRIVCHK, in module SYSCRMPSC, to check whether the current security persona has the privileges necessary to create the type of section specified by its FLAGS argument and, if not, unlocks the mutex and returns the error status SS$_NOPRIV.

10. If the section to be created is a memory-resident section, $CREATE_GDZRO_INT makes an additional check for the rights identifier VMS$MEM_RESIDENT_USER. If the persona does not hold this identifier, it unlocks the mutex and returns the error status SS$_NOMEMRESID.
11. It allocates paged pool for a GSD. If pool is unavailable, it unlocks the GSD mutex and returns the error status SS$_GSDFULL.
12. It initializes the GSD, copying the section name to GSD$T_GSDNAM, storing the hash value in GSD$L_HASH, and clearing GSD$L_IPID. It clears GSD fields irrelevant to this type of section and initializes GSD$L_FLAGS from the section flags and access mode.
13. It checks that the LENGTH_64 argument, converted to pagelets, fits within 32 bits and, if not, deallocates the GSD, unlocks the mutex, and returns the error status SS$_ILLPAGCNT.
14. It checks that the IDENT_64 argument is valid and, if not, deallocates the GSD, unlocks the mutex, and returns the error status SS$_IVSECIDCTL.
15. $CREATE_GDZRO_INT allocates a GSTE from the system header. If none is available, it deallocates the GSD, unlocks the mutex, and returns the error status SS$_SECTBLFUL.

    Otherwise, it initializes the GSTE:

    a. It copies the FLAGS argument to SEC$L_FLAGS and sets section flag SEC$V_WRT unless the section is a shared page table read-only section.
    b. It checks that the number of pagelets to be mapped is nonzero and, if not, deallocates the GSTE and GSD, unlocks the mutex, and returns SS$_LEN_NOTPAGMULT to its caller.
    c. It stores the number of pagelets to be mapped in SEC$L_UNIT_CNT.
    d. It clears SEC$L_WINDOW.
    e. It stores the section offset in the GSTE forward and backward links and clears SEC$L_VPX, the virtual page index.
    f. Before setting the section reference count to 1, $CREATE_GDZRO_INT executes a memory barrier instruction to ensure another processor cannot see a nonzero reference count without also seeing a zero virtual page index. The zero virtual page index prevents the section table entry from being used to map a global section while it is not fully initialized.
    g. It sets SEC$V_INPROG in SEC$L_FLAGS to indicate section initialization is in progress.
16. $CREATE_GDZRO_INT copies the IDENT_64 argument to GSD$L_IDENT and initializes GSD$L_FLAGS from the section flags and access mode.

17. It calls MMG_STD$INIT_ORB (see Section 3.7.1) to allocate and initialize an ORB that describes the protection on the GSD. If it returns an error status, $CREATE_GDZRO_INT deallocates the ORB, GSD, and GSTE; unlocks the mutex; and returns the error status to its caller.

18. It calls MMG_STD$USE_RES_MEM, in module MEM_ALLOC, iteratively to locate all the reserved memory descriptors (RMDs), if any, associated with this global section. A global section with memory reserved in multiple RADs has an RMD for each RAD. $CREATE_GDZRO_INT continues to call MMG_STD$USE_RES_MEM, accumulating reserved pages, until that routine returns an error status.

    MMG_STD$USE_RES_MEM takes the following steps:

    a. If a matching RMD exists, but its pages are already in use, MMG_STD$USE_RES_MEM returns the error status SS$_RESERVEDMEMUSED.

    b. If a matching RMD exists, its pages are not in use, and its pages were preallocated, it checks that the requested page count is less than or equal to the number reserved, returning the error status SS$_MRES_PFNSMALL if not. Otherwise, it zeros the pages if they were not already zeroed, stores the requested page count in RMD$L_IN_USE_COUNT, sets RMD$V_IN_USE in RMD$L_FLAGS, and returns SS$_NORMAL and the address of the RMD.

    c. If a matching RMD exists but its pages were not preallocated, MMG_STD$USE_RES_MEM compares the requested page count to the number reserved. If the request is larger, it checks whether the difference would reduce the system fluid page count (PFN$GL_PHYPGCNT) too much and returns the error status SS$_INSFLPGS if so. Otherwise, it reduces the fluid page count by the requested page count, stores the requested page count in RMD$L_IN_USE_COUNT, sets RMD$V_IN_USE in RMD$L_FLAGS, and returns SS$_NORMAL and the address of the RMD.

    d. If no matching RMD exists, the routine checks whether the requested page count would reduce the fluid page count too much and returns the error status SS$_INSFLPGS if so. Otherwise, it reduces the fluid page count by the requested page count and returns SS$_NORMAL.

    If MMG_STD$USE_RES_MEM returns SS$_RESERVEDMEMUSED but enough pages have been reserved for the global section, $CREATE_GDZRO_INT continues. Otherwise, $CREATE_GDZRO_INT returns any reserved memory, deallocates the GSD and GSTE, unlocks the mutex, and returns the error status to its caller.

    If MMG_STD$USE_RES_MEM returns any other error status, $CREATE_GDZRO_INT returns any reserved memory, deallocates the GSD and GSTE, unlocks the mutex, and returns the error status to its caller.

19. If the RMD pages were preallocated, $CREATE_GDZRO_INT sets SEC$V_MRES_ALLOC in both the GSD and GSTE flags.

20. It calls MMG_STD$USE_RES_MEM again, this time to locate an RMD for shared page tables for the global section. MMG_STD$USE_RES_MEM takes the steps previously described, with the exception that it returns SS$_NORESERVEDMEM if there is no RMD for a shared page table reservation.

21. If a shared page table RMD exists, $CREATE_GDZRO_INT allocates paged pool for a GSD to describe the shared page table global section. If the allocation fails, it calls MMG_STD$RETURN_RES_MEM, in module MEM_ALLOC, to indicate the memory is no longer being used, deallocates the GSD and GSTE, unlocks the mutex, and returns SS$_GSDFULL to its caller.

    Otherwise, $CREATE_GDZRO_INT initializes the shared page table global section GSD with information from the global section GSD. It sets SEC$V_SHARED_PTS and, if the system service FLAGS bit SEC$V_READ_ONLY_SHPT is clear, also sets SEC$V_WRT in the shared page table GSD$L_FLAGS. It copies the global section's SEC$V_RAD_HINT flag. It links the two GSDs together by storing the global section index of the global section in the shared page table GSD's GSD$L_RELATED_GSTX.

    It checks that the global section and the shared page table global section are both preallocated or both not. If they differ, it calls MMG_STD$RETURN_RES_MEM, deallocates the GSDs and GSTE, unlocks the mutex, and returns SS$_MRES_INCON to its caller.

    It allocates and initializes a GSTE for the shared page table global section and stores its index number in the global section's GSD$L_RELATED_GSTX. If the GSTE allocation fails, it calls MMG_STD$RETURN_RES_MEM to release any memory reserved for the global section or shared page table global section, deallocates the GSDs and GSTE, unlocks the mutex, and returns SS$_MRES_INCON to its caller.

22. $CREATE_GDZRO_INT allocates enough GPTEs to map the global section plus two stopper GPTEs. If it is unable to allocate them, it calls MMG_STD$RETURN_RES_MEM, deallocates the GSDs and GSTE, unlocks the mutex, and returns SS$_GPTFULL to its caller.

    Otherwise, it calculates the global page table index corresponding to the first page of the section and records it in GSTE$L_VPX. It updates global page table performance cells, PMS$GL_GBLPAGCNT and PMS$GL_GBLPAGMAX.

23. If there is an associated shared page table global section, it repeats the actions of the previous step in preparation for mapping that section.

    It allocates and initializes an S2 space L3PTE to map the shared page table pages one at a time so that it can initialize their L3PTEs.

24. It zeros the first and last GPTEs allocated for the global section and initializes the rest of them:

    — If this is a preallocated section, each GPTE is valid and contains the PFN associated with that page. If the reservation is described by multiple RMDs, $CREATE_GDZRO_INT stripes the allocation across the RMDs, allocating the largest possible chunk in each that is a granularity hint multiple (512, 64, 8, or 1). It returns any unused memory.

- If the pages of the section are allocated on demand, each GPTE has its valid bit clear, and its type 0, type 1, write, and demand zero bits set. The GPTE contains the GSTE index and a type of PFN$C_GBLWRT. Figure 2.26 shows the format of the section table index form of GPTE.

In the course of initializing GPTEs, $CREATE_GDZRO_INT increments PFN$L_SHRCNT to 1 for the PFN occupied by the GPT page. If the section's pages are preallocated, it confirms that the current page type of each PFN is PFN$C_UNKNOWN, generating the fatal bugcheck INCONMMGST if not. It also initializes the PFN database record of each PFN:

- PFN$L_SHRCNT is 1, and PFN$L_WSLX_QW is 0, that is, a memory-resident page is not part of any working set list.
- Page type is PFN$C_GBLWRT, and page state is PFN$C_ACTIVE.
- PFN$Q_PTE_INDEX and PFN$L_PT_PFN describe the GPTE that maps this page.
- PFN$W_REFCNT is 1.
- PFN$Q_BAK contains the index of the associated GSTE.

25. If there is an associated shared page table global section, it repeats most of the actions of the previous step to map that section, with the following major differences and additional actions:

    a. One difference is that if the shared page table pages were not preallocated, $CREATE_GDZRO_INT must allocate them and initialize their PFN database records. If the allocation fails, it places the kernel thread into a free page wait until a free page is available.

       It records the allocated PFNs in the GPTEs that map the shared page table section. Each GPTE has a set valid bit and a PFN; the rest of its fields are zero. The GPTEs simply record what memory has been allocated; the shared page tables will eventually be mapped through process-private L2PTEs (see Section 3.8.2.3).

    b. $CREATE_GDZRO_INT determines the page owner mode and protection bits to insert in the shared L3PTEs, based on the section's owner mode.

    c. It initializes PFN$W_PT_WIN_CNT in the PFN database record of each shared page table to 1 less than the number of shared L3PTEs in that page, the number of pages mapped by that shared page table page. A value of –1 in PFN$W_PT_WIN_CNT represents a count of zero.

    d. It maps each shared page table temporarily in S2 space and initializes each of the L3PTEs in it. Each L3PTE contains protection and owner mode determined in step 2. If the memory-resident global section's pages are preallocated, each L3PTE contains the corresponding PFN and a set valid bit. If MMG$M_NO_

MB is set in the MMG_CTLFLAGS SYSGEN parameter, it sets PTE$V_NO_MB in each L3PTE. In addition, each L3PTE contains the granularity hint bits corresponding to the physical and virtual memory alignment. If the memory-resident section's pages are not preallocated, each L3PTE contains a set type 0 bit and the global section page index.

It zeros any L3PTEs in the page table page that do not map global section pages.

e. It deallocates the S2 space L3PTE.

26. It inserts the global section GSD onto the group or system list.
27. It unlocks the GSD mutex.
28. It returns SS$_CREATED or SS$_CREATED_SHPT to its caller, depending on whether the section has an associated shared page table global section.

3.7.13 $CREATE_SHMGS_INT Routine

$CREATE_SHMGS_INT, in module SYS_GDZRO_64, creates a Galaxywide section. The $CREATE_GDZRO and $CRMPSC_GDZRO_64 system services call it when requested with flag SEC$V_SHMGS set.

In contrast to a memory-resident global section, a Galaxywide section is not described by RMDs. Because the instances of a Galaxy do not necessarily share mass storage, a common reserved memory registry cannot be relied on.

A Galaxywide global section must be created on each instance from which a process will map to the section. Creating the section requires creating a Galaxywide shared memory region, initializing data structures in shared memory to describe the region, and initializing global section data structures to describe the section. If the shared memory region is larger than 127 pages, a second shared memory region is created for shared page tables.

If the shared memory region has already been created by another instance, $CREATE_SHMGS_INT establishes this instance's connection to it and initializes the global section data structures.

$CREATE_SHMGS_INT takes the following steps:

1. It checks that the system is a Galaxy instance with shared memory support, returning error status SS$_INV_SHMEM if not.
2. It takes out an exclusive lock on a clusterwide resource whose name begins with the string "MMG$SEC_". If SEC$V_SYSGBL was not specified, the resource name includes the UIC group code of the current security persona.

   This prevents multiple processes on one or more clustered Galaxy instances from creating a system Galaxywide section at the same time or more than one process in the same UIC group from creating a group Galaxywide section at the same time.

   Subsequent error returns dequeue the lock.

3. $CREATE_SHMGS_INT takes the same steps as $CREATE_GDZRO_INT to validate service arguments and to create a GSD and GSTE (see steps 1 through 17 in Section 3.7.12).

4. It allocates enough GPTEs to map the section pages and two stopper pages. If there are insufficient free GPTEs and the global page table cannot be expanded, it releases all resources and returns the error status SS$_GPTFULL.

5. It faults in the global page table pages.

6. It calculates the GPTX corresponding to the first GPTE and records it in SEC$L_VPX.

7. It updates global page table performance cells, PMS$GL_GBLPAGCNT and PMS$GL_GBLPAGMAX.

8. It calls GLX$SHM_REG_CREATE, in module [GALAXY]GLX_SHM_REG, to create a shared memory region, preferably in the RAD, if any, specified on the call to $CREATE_SHMGS_INT. The region's name is the string "GLX$", concatenated with the global section name, $, and either the string "SYSGBL" or the hexadecimal representation of the UIC group code.

   If another Galaxywide shared memory region of the same name but different ident or different access mode already exists, $CREATE_SHMGS_INT releases all resources, and returns error status SS$_DUPLNAM, SS$_IDMISMATCH, or SS$_WRONGACMODE to its caller.

   If a region of the same name and characteristics but a different length exists, $CREATE_SHMGS_INT deallocates the GPTEs, reallocates GPTEs for the existing length, faults them into memory again, and recalls GLX$SHM_REG_CREATE.

   GLX$SHM_REG_CREATE initializes the GPTEs as writable invalid transition PTEs. Each physical page will be zeroed later when it is first referenced.

9. $CREATE_SHMGS_INT initializes the GSD and GSTE to reflect the access mode, ident, and flags of the actual Galaxywide section, which may have been created by another instance. It sets SEC$V_MRES_ALLOC in both the GSD and GSTE flags.

10. It temporarily maps a page of P1 space to the first physical page of the Galaxywide section. This will serve as a signal to image rundown that the process has begun to map this section. The signal is necessary because $CREATE_SHMGS_INT must drop IPL to 0, leaving the process vulnerable to deletion.

11. It unlocks the GSD mutex, lowers IPL to 0, and calls EXE$DISTRIBUTE_PROFILE, in module SYSOBJSUB, either to create an ORB for the section on this instance based on its already defined security profile or to propagate the security profile of a newly created section to the other Galaxy instances in the cluster.

12. It relocks the GSD mutex for write, raising IPL to 2, and unmaps the P1 space page.

13. If the section is larger than 127 pages, $CREATE_SHMGS_INT creates a shared page table section to map the Galaxywide global section:

    a. It allocates paged pool for a shared page table section GSD. If pool is not available, it continues with step 14: the Galaxywide section exists but must be mapped by private page tables on this instance.

    b. It initializes the GSD by copying information from the Galaxywide section GSD and from its arguments. It sets SEC$V_SHARED_PTS and, if the flags argument bit SEC$V_READ_ONLY_SHPT is clear, also sets SEC$V_WRT in the shared page table GSD$L_FLAGS. It stores the Galaxywide section's index in the shared page table GSD$L_RELATED_GSTX.

    c. It temporarily allocates an S2 L3PTE to map a page of shared page table during initialization.

    d. $CREATE_SHMGS_INT allocates and initializes a GSTE to describe the shared page table section. If allocation fails, it deallocates the shared page table GSD and continues with step 14.

    e. It allocates GPTEs to map the shared page table section plus two stoppers. If allocation fails, it deallocates the shared page table section's GSD and GSTE and continues with step 14. It updates global page table performance cells, PMS$GL_GBLPAGCNT and PMS$GL_GBLPAGMAX.

    f. It touches each global page table page that maps the shared page table section to fault it into memory.

    g. It calls GLX$SHM_REG_CREATE to create a shared memory region, preferably in the RAD, if any, specified on the call to $CREATE_SHMGS_INT, to contain the shared page table section. The region's name is the string "GLXSHPT$", concatenated with the global section name, $, and either the string "SYSGBL" or the group UIC code represented as a hexadecimal string. If it returns SS$_NOWAIT because the Galaxy lock that synchronizes access to the shared memory data structures is locked, $CREATE_SHMGS_INT repeats its attempt to create a Galaxywide region. If any other error occurs, it deallocates the shared page table section resources and continues with step 14.

    h. $CREATE_SHMGS_INT calculates the page owner mode and protection bits based on the region's access mode. If the section is writable, the mode allowed to read determines the mode allowed to write.

    i. It records the allocated PFNs in the GPTEs that map the shared page table section. Each GPTE has a set valid bit and a PFN; the rest of its fields are zero. The GPTEs simply record what memory has been allocated; the shared page tables will eventually be mapped through process-private L2PTEs (see Section 3.8.2.3).

j. It maps each shared page table temporarily in S2 space and initializes each of the L3PTEs in it. Each L3PTE contains protection and owner mode determined in step 2 and the corresponding PFN. If the L3PTE maps a page already in use on another instance, the routine sets the valid bit in the L3PTE; otherwise, it leaves it clear. When a process first accesses a page mapped by such an invalid L3PTE, it will page fault. The page fault exception service routine will recognize that this is an uninitialized page in a Galaxywide global section and zero it.

   It zeros any L3PTEs in the page table page that do not map global section pages.

k. It deallocates the S2 space L3PTE.

14. It inserts the Galaxywide section GSD into either the Galaxywide group or system global section list.
15. It unlocks the GSD mutex.
16. It dequeues the lock and returns the success status SS$_REMOTE if the section was created by another instance, SS$_CREATED_SHPT if the section and shared page table section were created, or SS$_CREATED if only the section was created.

3.7.14 $CRMPSC_GDZRO_64 System Service

The Create and Map to Global Demand Zero Section ($CRMPSC_GDZRO_64) service is requested to create and map either a memory-resident global section or a Galaxywide global section. The requestor specifies section name, ident, protection, section length, region ID, section offset, access mode, section flags, and optionally the map length. Optionally, the requestor can specify in which RAD the system service should create the global section.

The $CRMPSC_GDZRO_64 system service procedure, EXE$CRMPSC_GDZRO_64 in module SYS_GDZRO_64, runs in kernel mode.

It resembles EXE$CRMPSC_GPFN_64 (see Section 3.7.10), with the following major differences:

- If the requestor specified a RAD, EXE$CRMPSC_GDZRO_64 checks that RAD support is enabled and returns the error status SS$_BADRAD if not. If RAD support is enabled, it checks that the requestor specified a single RAD and one that actually exists, returning SS$_BADRAD if not.
- It sets the flags SECV_DZRO, SECV_PERM, SECV_WRT, SECV_GBL, and SEC$V_MRES in case they were clear.
- If the request is to create a Galaxywide section (if SEC$V_SHMGS is set), it checks that the service was requested from IPL 0, returning SS$_BADPARAM if not.

- If the request is to create a memory-resident section, it calls $CREATE_GDZRO_INT (see Section 3.7.12). If that routine returns an error status, EXE$CRMPSC_GDZRO_64 returns the error status to its requestor. Otherwise, it calls $MGBLSC_GDZRO_INT (see Section 3.8.2.3) to perform the mapping.
- If the request is to create a Galaxywide section, it calls $CREATE_SHMGS_INT (see Section 3.7.13). If that routine returns an error status, EXE$CRMPSC_GDZRO_64 returns the error status to its requestor. Otherwise, it calls $MGBLSC_GDZRO_INT (see Section 3.8.2.3) to perform the mapping.
- It returns the number of bytes that had been reserved for the global section in argument RESERVED_LENGTH_64, if present, and if the request is to create a memory-resident section.

3.8 Mapping a Global Section

The map global section system services can be considered a special case of the create and map section system services, one in which the global section already exists. Each of these services maps a range of process addresses to the named global section. Each usually has no effect on the global section database other than to include the latest mapping in various reference counts.

When a process maps to a global section backed by a file rather than a PFN-mapped section, each of its process L3PTEs in the designated range is initialized with a GPTX (see Figures 2.12 and 2.25). A GPTX is a pointer to the GPTE that records the current state of the global page.

3.8.1 $MGBLSC System Service

The Map Global Section ($MGBLSC) system service procedure, EXE$MGBLSC in module SYSCRMPSC, runs in kernel mode. It takes the following steps:

1. It creates and initializes scratch space on the stack. In addition to making the argument validation checks described in Section 3.1.1, EXE$MGBLSC checks the INADR argument: unless the SEC$V_EXPREG flag was specified in the FLAGS argument, it confirms that the starting address is on an Alpha page boundary and that the ending address is one byte less than a page boundary. (It takes into account the possibility that the addresses have been specified in reverse order.) If the addresses are not correct, it returns the error status SS$_INVARG.
2. It calls MMG$VFY_SECFLG, in module SYSDGBLSC, to test the compatibility of the section flags with each other. If the flags are incompatible or if the requestor specified SEC$V_SHMGS, it returns the error status SS$_IVSECFLG.
3. It locks the GSD mutex for write access to synchronize access to the GSD lists, raising IPL to 2.
4. It calls MMG$DALCSTXSCN1 (see Section 3.9.2) to check the global (system) section table for any sections to be deleted.

5. It calls MMG_STD$GSDSCAN (see Section 3.7.1) to scan the GSD list for the specified global section.

   If the section is not found, EXE$MGBLSC unlocks the GSD mutex and returns MMG_STD$GSDSCAN's error status to the system service requestor.

6. If the global section is mapped to a file, EXE$MGBLSC calculates the address of its GSTE from GSD$L_GSTX and the contents of PHD$L_PST_BASE_OFFSET in the system header.

7. If the section is memory-resident, EXE$MGBLSC unlocks the GSD mutex and returns SS$_GBLSEC_MISMATCH to its requestor.

8. If the section is copy-on-reference, it sets MMG$V_CHGPAGFIL in MMG$L_MMG_FLAGS so that the section pages will be charged against the process's page file quota.

9. It compares the section access mode with the mode bits in MMG$L_ACCESS_MODE to determine if the system service requestor is allowed to map the section. If not, EXE$MGBLSC unlocks the GSD mutex and returns the error status SS$_NOPRIV.

10. If the section is not PFN-mapped, it acquires the MMG spinlock, increments SEC$L_REFCNT so that the section cannot inadvertently be deleted before its pages are mapped into the process's address space, and releases the MMG spinlock.

    If the section is PFN-mapped, EXE$MGBLSC increments GSD$L_REFCNT to prevent section deletion. (Recall that a PFN-mapped global section has no associated GSTE.)

11. With the section's deletion blocked, EXE$MGBLSC can safely unlock the GSD mutex.

12. If the SEC$V_EXPREG flag was specified in the FLAGS system service argument, EXE$MGBLSC calculates the starting and ending section addresses based on the RELPAG argument, the section page count (GSD$L_PAGES for a PFN-mapped section or SEC$L_UNIT_CNT multiplied by pagelets per page for all others), and contents of RDE$PQ_FIRST_FREE_VA from either the P0 or P1 RDE. The INADR argument simply identifies in which process-private region the section is to be created.

    If the SEC$V_EXPREG flag was not specified, EXE$MGBLSC determines the address of the RDE in which the starting address falls. It checks whether the region is a shared page table region and, if so, continues with step 19, returning error status SS$_NOSHPTS. It checks whether the access mode from which the service was requested is allowed to create pages in this region and, if not, continues with step 19, returning error status SS$_NOPRIV.

    EXE$MGBLSC calculates the virtual address range to be mapped based on the RELPAG argument (in units of pages or pagelets, depending on section type), the section page count (GSD$L_PAGES for a PFN-mapped section or SEC$L_UNIT_CNT multiplied by pagelets per page for all others), and the INADR argument.

In either case, an integral number of Alpha pages will be mapped. If the pagelet count does not represent an integral number of pages, the highest address page of the section will be only partly occupied by the section. Its L3PTE will have the PTE$V_PARTIAL_SECTION bit set.

13. EXE$MGBLSC forms a template L3PTE for pages in the section.

    — If the section is PFN-mapped, the L3PTE has the valid and window bits set, and its PFN is based upon the contents of GSD$L_BASEPFN. (The L3PTE that maps the lowest address page of the section will have that PFN.)

    — If the section is backed by a section file, the L3PTE has the type 0 bit set and the type 1 bit clear to indicate a global page, and its GPTX is based upon the contents of SEC$L_VPX. (The L3PTE that maps the lowest address page of the section will have that GPTX.)

    EXE$MGBLSC calculates the L3PTE protection bits based on MMG$L_ACCESS_MODE, the writable flag in SEC$L_FLAGS, and the input section flags specifying the mode allowed to write the section pages.

14. It tests whether the current security persona has the necessary access (read, write, or execute) to the section based on the persona's access rights list and the ORB associated with the section.

    If the persona does not have the desired access, EXE$MGBLSC continues with step 19, returning the error status from the access check.

    If the persona does have access, EXE$MGBLSC also calls security auditing code, which checks whether a successful access should be audited, and if so, builds a message to be logged before the service exits.

15. EXE$MGBLSC determines whether the address space into which the section will be mapped overmaps existing space and whether the section is a PFN-mapped section.

    — If no space will be overmapped, if the number of pages in the section is equal to the number of pages to be mapped, if the section is not a PFN-mapped section, and if all pages can be created, EXE$MGBLSC acquires the MMG spinlock, increases the section's reference count by the number of pages to be mapped, and releases the MMG spinlock. It initializes each of the process's L3PTEs by inserting the appropriate GPTX along with the template L3PTE.

    — If the space to be created overmaps existing space or cannot all be created, or if the section is a PFN-mapped section, EXE$MGBLSC calls MMG$CREDEL, specifying MAPSECPAG_RDE (see Section 3.6.1.2) as the per-page routine.

16. If PHD$V_DALCSTX in PHD$L_FLAGS is set, indicating there are global sections to be deallocated, EXE$MGBLSC calls MMG$DALCSTXSCN (see Section 3.9.2).

17. EXE$MGBLSC returns any unused page file quota, records peak page file use and virtual size statistics, and stores return information in the optional RETADR argument.

18. It decrements the section reference count to remove the extra reference, unnecessary now that the reference count reflects the mapped PTEs.
19. It calls MMG$DELGBLWCB (see Section 3.9.4) to close open files associated with temporary global sections whose reference counts have gone to zero and to delete their WCBs.
20. It calls a security audit routine, which may log successful access to the section.
21. It returns to its requestor.

3.8.2 $MGBLSC_64 System Service

The Map to Global Section ($MGBLSC_64) system service procedure, EXE$MGBLSC_64 in module SYS_GBLSEC_64, runs in kernel mode.

EXE$MGBLSC_64 takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:
   a. If the START_VA_64 argument was omitted and the flag SEC$V_EXPREG was clear, it returns the error status SS$_IVSECFLG. If the START_VA_64 argument was nonzero and the flag SEC$V_EXPREG was set, it returns the error status SS$_IVSECFLG.
   b. It maximizes the ACMODE argument. There is no input argument to specify access mode allowed to write the section: if the section is writable, the mode allowed to read determines the mode allowed to write.
   c. It checks that the START_VA_64, SECTION_OFFSET_64, and LENGTH_64 arguments are multiples of the size of a page, returning the error status SS$_VA_NOTPAGALGN, SS$_OFF_NOTPAGALGN, or SS$_LEN_NOTPAGMULT if not.
2. It locks the GSD mutex, raising IPL to 2.
3. It calls MMG_STD$DALCSTXSCN (see Section 3.9.2) to check the global (system) section table for any sections to be deleted.
4. It calls MMG_STD$GSDSCAN (see Section 3.7.1) to find the GSD, if any, that corresponds to the GS_NAME_64 and IDENT_64 arguments.
5. If MMG_STD$GSDSCAN returns an error status, EXE$MGBLSC_64 unlocks the mutex, calls MMG_STD$DELGBLWCB (see Section 3.9.4), and returns the error status to its caller.
6. Otherwise, EXE$MGBLSC_64 performs an additional sanity check on the global section ident: it confirms that the ident is positive and, if not, unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_IVSECIDCTL to its caller.

7. Examining the GSD flags, EXE$MGBLSC_64 tests whether the section is a PFN-mapped one. If so, it unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_GBLSEC_MISMATCH to its requestor.
8. EXE$MGBLSC_64 compares the section access mode with the requestor's mode to determine if the system service requestor is allowed to map the section. If not, it unlocks the mutex, calls MMG_STD$DELGBLWCB, and returns the error status SS$_NOPRIV to its requestor.
9. It calculates the address of the GSTE. It acquires the MMG spinlock, increments the section's SEC$L_REFCNT to prevent its deletion, and releases the MMG spinlock. Having incremented the section reference count, it can unlock the GSD mutex.
10. It calls MMG_STD$CHKPRO_AUDIT to check access to the file. If access is not allowed, it decrements the reference count and returns an error status to its requestor.
11. EXE$MGBLSC_64 calls a routine to map the section, depending on the section type:
    - If it is a disk file section, it calls $MGBLSC_GFILE_INT (see Section 3.8.2.1).
    - If it is a page file section, it calls $MGBLSC_GPFILE_INT (see Section 3.8.2.2).
    - If it is a memory-resident section, it calls $MGBLSC_GDZRO_INT (see Section 3.8.2.3).
12. If any pages were mapped, EXE$MGBLSC_64 records peak page file use and virtual size statistics, and stores return information in the RETURN_VA_64, START_VA_64, and RETURN_LENGTH_64 arguments.
13. It calls a security audit routine, which may log successful access to the section.
14. If an output argument was inaccessible, it returns SS$_ACCVIO; otherwise, it returns to its requestor the status from the mapping routine.

3.8.2.1 $MGBLSC_GFILE_INT Routine

$MGBLSC_GFILE_INT, in module SYS_GBLSEC_64, performs much of the work of the $MGBLSC_64 and $CRMPSC_GFILE_64 system services. It is entered at IPL 2. It is called with information derived from the service arguments as well as pointers to the GSTE and GSD.

$MGBLSC_GFILE_INT takes the following steps:

1. It determines the address of the RDE corresponding to the REGION_ID_64 argument, returning the error status SS$_IVREGID if the ID is invalid. If that region is intended for memory-resident and Galaxywide global sections, it returns the error status SS$_NOSHPTS. It checks whether the access mode from which the service was requested is allowed to create pages in this region and, if not, returns the error status SS$_NOPRIV.
2. It checks that the SECTION_OFFSET_64 is within the global section, returning the error status SS$_OFFSET_TOO_BIG if not.

3. It calculates how many bytes of section there are between SECTION_OFFSET_64 and the section's end, minimizes that with the LENGTH_64 argument, and transforms that into a count of pages to be mapped. The count includes a partial page if SECTION_OFFSET_64 is not an integral number of pages.

4. If the SEC$V_EXPREG flag was specified in the FLAGS argument, $MGBLSC_GFILE_INT calculates the starting and ending addresses to map based on the LENGTH_64 argument and the contents of RDE$PQ_FIRST_FREE_VA in the RDE corresponding to the REGION_ID_64 argument. If that address range intersects with the gap (see Chapter 1), $MGBLSC_GFILE_INT moves the address range.

   If the SEC$V_EXPREG flag was not specified, $MGBLSC_GFILE_INT calculates the address based on the START_VA_64 and LENGTH_64 arguments. If the address range is not entirely within the specified region, it returns the error status SS$_PAGNOTINREG to its caller.

5. $MGBLSC_GFILE_INT forms a template L3PTE for the section's pages (see Figure 2.12). The L3PTE has the type 0 bit set, the global page table index in bits <47:32>, and the WRT, CRF, and DZRO bits copied from the section flags. It calculates the page owner mode and protection bits based on the access mode information passed from its caller, the writable flag in SEC$L_FLAGS, and the protection specified when the global section was created.

   If the caller is trying to write to the section but its protection prohibits write access, $MGBLSC_GFILE_INT returns SS$_NOPRIV to its caller.

6. If the new address space does not already exist, is entirely within a region, and can all be created without hitting any of the limits to growth described in Section 3.3, $MGBLSC_GFILE_INT adjusts RDE$PQ_FIRST_FREE_VA. It increases the section's reference count by the number of pages to be mapped. It initializes the section's L3PTEs.

   If the space to be created overmaps existing space or cannot all be created at once, $MGBLSC_GFILE_INT loops, calling MAPSECPAG_RDE (see Section 3.6.1.2) once per page until the routine returns an error status or all pages are done. On each successful return, $MGBLSC_GFILE_INT increments the section's reference count.

7. If PHD$V_DALCSTX in the process's PHD$L_FLAGS is set, indicating there are process sections to be deallocated, $MGBLSC_GFILE_INT calls MMG_STD$DALCSTXSCN (see Section 3.9.2).

8. It returns to its caller.

3.8.2.2 $MGBLSC_GPFILE_INT Routine

$MGBLSC_GPFILE_INT, in module SYS_GBLSEC_64, performs much of the work of the $MGBLSC_64 and $CRMPSC_GPFILE_64 system services. It is called with information derived from the service arguments as well as pointers to the GSTE and GSD.

It resembles $MGBLSC_GFILE_INT (see Section 3.8.2.1), with the following major differences:

- The section offset for a global section must be an integral number of pages.
- Mapping a global page file section does not take process page file quota.

3.8.2.3 $MGBLSC_GDZRO_INT Routine

$MGBLSC_GDZRO_INT, in module SYS_GDZRO_64, performs much of the work of the $MGBLSC_64 and $CRMPSC_GDZRO_64 system services. It is called with information derived from the service arguments as well as pointers to the GSTE and GSD. It maps an existing memory-resident global demand zero section or a Galaxywide global section.

It resembles $MGBLSC_GFILE_INT (see Section 3.8.2.1), with the following major differences:

- If the global section has an associated shared page table global section and the global section is being mapped into a shared page table region, the state of the SEC$V_WRT flag specified by the service requestor must match that state of the global section: if SEC$V_WRT is set, the GSTE flag SEC$V_READ_ONLY_SHPT must be clear, and vice versa. Otherwise, $MGBLSC_GDZRO_INT returns the error status SS$_IVSECFLG to its caller.
- If the global section is being mapped into a shared page table region, $MGBLSC_GDZRO_INT checks that the SECTION_OFFSET_64 and LENGTH_64 arguments are multiples of the number of bytes mapped by an L3PTE. If not, it returns the error status SS$_OFF_NOTPAGALGN or SS$_LEN_NOTPAGMULT.

  It also checks that if the START_VA_64 argument was supplied, its value is a multiple of the number of bytes mapped by an L3PTE. If not, it returns the error status SS$_VA_NOTPAGALGN.
- If the SEC$V_EXPREG flag was specified in the FLAGS argument, $MGBLSC_GDZRO_INT calculates the starting and ending addresses to map. It aligns the contents of RDE$PQ_FIRST_FREE_VA in the RDE corresponding to the REGION_ID_64 argument to a multiple of the number of pages mapped by an L3PTE to form the starting address. It adds the LENGTH_64 argument to form the ending address.
- In the case of a section mapped with shared page tables, the maximized ACMODE argument must match the global section's SEC$V_ACMODE bits. If not, $MGBLSC_GDZRO_INT returns the error status SS$_IVACMODE.
- $MGBLSC_GDZRO_INT forms a template PTE for the section's pages. In the case of a section with shared page tables, the template PTE maps a shared page table rather than a section page.

  It calculates the page owner mode and protection bits based on the access mode information passed from its caller, SEC$V_WRT, and the protection specified when the global section was created.

If the section is memory-resident in instance-private memory and has preallocated pages, $MGBLSC_GDZRO_INT determines the best possible granularity hint bits with which it can be mapped. (Granularity hint regions in shared memory are not currently supported.) If SEC$V_EXPREG is set, it may round down the starting virtual address to be able to form a larger granularity hint region. If all the section cannot be mapped at once or if it overmaps existing space, the routine clears the granularity hint bits and does not round down the starting address. It sets the valid bit. If MMG$M_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter and the current process has only one kernel thread, it sets PTE$V_NO_MB in the template PTE. The PFN in each PTE will be copied from the corresponding GPTE.

If the section is memory-resident but does not consist of preallocated pages, the routine leaves the valid bit clear and sets the type 0 bit. Each L3PTE will have a GPTX inserted.

In the case of a Galaxywide section, the PTE contents depend on whether the page being mapped has already been zeroed. If so, the PTE's valid and modify bits are set. If MMG$M_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter and the current process has only one kernel thread, PTE$V_NO_MB is also set. If the page has not been zeroed, the valid bit, type 0, and type 1 bits are cleared so that the page looks like an invalid transition page. The PFN in each PTE is copied from the corresponding GPTE.

If the section is mapped with shared page tables, it initializes the template PTE as any other L2PTE would be: kernel mode read and write enabled, executive mode read enabled, PTE$V_NOX set, and kernel mode as owner.

- In mapping a section with shared page tables, $MGBLSC_GDZRO_INT increments the shared page table section's reference count by the number of shared page table pages. It increments the PFN$W_PT_WIN_CNT for each associated L2PT by the number of shared page table pages it maps.
- In mapping a memory-resident section without shared page tables, it increments the PFN$W_PT_WIN_CNT for each associated L3PT by the number of section pages it maps.

3.8.3 $MGBLSC_GPFN_64 System Service

The Map Global Page Frame Section ($MGBLSC_GPFN_64) system service is requested to map an existing PFN-mapped global section. The requestor specifies section name, ident, region ID, relative page number, page count, access mode, and section flags. The requestor has not opened a section file to be mapped so does not specify channel number. The requestor does not specify page fault cluster; such a section incurs no page faults.

The $MGBLSC_GPFN_64 system service procedure, EXE$MGBLSC_GPFN_64 in module SYS_GPFN_64, runs in kernel mode. It resembles EXE$MGBLSC_64, with the following major differences:

- The section flags set automatically by EXE$MGBLSC_GPFN_64 are SEC$V_GBL and SEC$V_PFNMAP.
- It checks that the PAGE_COUNT and RELATIVE_PAGE arguments can be represented as a positive 32-bit number, returning error status SS$_ILLPAGCNT or SS$_ILLRELPAG if not.
- If a global section has already been created with matching name and ident, it must be a PFN-mapped section for a successful match. If not, EXE$MGBLSC_GPFN_64 returns the error status SS$_GBLSEC_MISMATCH.
- It calls $MGBLSC_GPFN_INT (see Section 3.8.3.1).

3.8.3.1 $MGBLSC_GPFN_INT Routine

$MGBLSC_GPFN_INT, in module SYS_GPFN_64, performs much of the work of the $MGBLSC_GPFN_64 and $CRMPSC_GPFN_64 system services. It is called with information derived from the service arguments as well as a pointer to the GSD. It resembles $MGBLSC_GFILE_INT (see Section 3.8.2.1), with the following major differences:

- $MGBLSC_GPFN_INT is not entered with the address of a GSTE because a PFN-mapped global section does not have an associated GSTE.
- It is entered with a relative page argument rather than with a section offset. It checks that the relative page number is within the section, returning the error status SS$_ILLRELPAG if not.
- It calculates how many pages there are between the relative page number and the end of the section to determine the maximum number of pages that can be mapped.
- The template L3PTE it forms has the valid and window bits set and a PFN derived from the GSD$L_BASEPFN field and the RELATIVE_PAGE argument.
- There is no code path to initialize all the process's L3PTEs at once. $MGBLSC_GPFN_INT loops, calling MMG_STD$CREPAG_64 (see Section 3.4.1.2) once per page until the routine returns an error status or all pages are done.

3.9 Global Section Deletion

Deleting a global section is more complex than creating one because the section must be reduced from one of many states to nonexistence. In addition, global writable pages must be written to their backing store before a global section can be fully deleted. To avoid stalling the kernel thread requesting the service until all associated I/O completes, the final steps in the deletion of a global section are often deferred to a time after the system service request and return.

The actual section deletion cannot occur until the reference count in the GSTE, the count of process-private PTEs mapped to the section, goes to zero. If the reference count is zero when the $DGBLSC service is requested, the global section is deleted. More commonly, however, global section deletion occurs later as a side effect of virtual address deletion, which itself might occur as a result of image exit or process deletion.

3.9.1 $DGBLSC System Service

The Delete Global Section ($DGBLSC) system service procedure, EXE$DGBLSC in module SYSDGBLSC, runs in kernel mode. It takes the following steps:

1. It creates and initializes scratch space on the stack.
2. It calls MMG$VFY_SECFLG to test the compatibility of the specified section flags.
3. It calls MMG_STD$GSDSCAN (see Section 3.7.1) to locate the GSD for the specified global section. MMG_STD$GSDSCAN returns at IPL 2 with the GSD mutex locked for write access. If the section does not exist, it unlocks the mutex and returns the error status SS$_NOSUCHSEC.
4. It confirms that the process's current security persona has PRMGBL privilege and, if the section to be deleted is a system global section, SYSGBL privilege. If the requestor specified SEC$V_SHMGS in the FLAGS, EXE$DGBLSC also checks for SHMEM privilege. If the security persona lacks a necessary privilege, EXE$DGBLSC returns the error status SS$_NOPRIV. Otherwise, it audits the use of privilege, as appropriate.
5. If the global section is a PFN-mapped section, EXE$DGBLSC confirms that the process's current security persona has PFNMAP privilege, unlocking the mutex and returning the error status SS$_NOPRIV if not. A PFN-mapped section is described solely by a GSD; there are no GSTE, GPTEs, or section reference count. The section can be deleted immediately. EXE$DGBLSC deallocates the ORB and GSD to paged pool. It continues with step 8.
6. If the global section is not a PFN-mapped section, EXE$DGBLSC checks whether it is a Galaxywide global section. If not, it removes the GSD from its current list and inserts it on the delete pending list, at global location EXE$GL_GSDDELFL. In either case, it clears the global section's permanent flag, SEC$V_PERM in GSD$L_FLAGS and, if there is an associated GSTE, in SEC$L_FLAGS as well. This step changes the section to a temporary global section that can be deleted when its reference count becomes zero.
7. If the section is a memory-resident section, EXE$DGBLSC checks whether the section has an associated shared page table section. If so, it clears SEC$V_PERM in the shared page table section's GSD$L_FLAGS and SEC$L_FLAGS.

   If the reference count in the GSTE is zero, the section can be deleted now; EXE$DGBLSC sets PHD$V_DALCSTX in the system header PHD$L_FLAGS as a signal for MMG$DALCSTXSCN.

8. It calls MMG$DALCSTXSCN (see Section 3.9.2) in case this section or any other can be deleted now.
9. It unlocks the GSD mutex.
10. It calls MMG$DELGBLWCB (see Section 3.9.4).
11. It restores the IPL at entry and returns to its requestor.

3.9.2 MMG[_STD]$DALCSTXSCN and MMG[_STD]$DALCSTXSCN1 Routines

MMG$DALCSTXSCN and its alternative entry point, MMG_STD$DALCSTXSCN, in module PHDUTL, are called to locate and deal with deletable section table entries, in both the global section and process section tables. Section deletion cannot occur until the section reference count goes to zero, generally as the result of virtual address space deletion or modified page writing.

A scan for deletable PSTEs or GSTEs is initiated from many of the services described in this chapter whenever virtual address space has been deleted, either explicitly or as a side effect of overmapping virtual address space (see Section 3.10.3).

MMG$DALCSTXSCN is entered at IPL 2 in kernel mode, with the address of a process header (PHD) whose section table should be scanned. In the case of deleted global sections, it is entered with the address of the system header and with the GSD mutex locked.

At the alternative entry point MMG[_STD]$DALCSTXSCN1, the routine first gets the address of the system header and then merges with MMG$DALCSTXSCN.

MMG$DALCSTXSCN takes the following steps:

1. It tests and clears PHD$V_DALCSTX, returning immediately if the bit was already clear.
2. It scans the list of section table entries, returning when it reaches the end of the list. It examines each entry's reference count, skipping to the next one if the count is nonzero.
3. If the reference count is zero, MMG$DALCSTXSCN tests whether the section is permanent and, if so, continues with step 2.
4. If the section has a zero reference count and is not permanent, it tests whether the section is a global section. If it is, MMG$DALCSTXSCN calls MMG$DELGBLSEC (see Section 3.9.3) to delete it and then continues with step 2.
5. If the section is a process-private section, MMG$DALCSTXSCN checks whether it is the only one still mapped from its section file.
   — If so, it restores the address of the WCB to CCB$L_WIND and inserts the section table entry into the free entry list.

- If there are other sections still mapped, it removes this one from the chain, inserts it into the free entry list, and, if necessary, adjusts CCB$L_WIND to point to a section table entry other than the one being deleted.

In either case, it continues with step 2.

3.9.3 MMG[_STD]$DELGBLSEC Routine

MMG$DELGBLSEC and its alternative entry point, MMG_STD$DELGBLSEC, in module SYSDGBLSC, are called by MMG$DALCSTXSCN to delete a temporary global section whose reference count has gone to zero, that is, one with no pages mapped by any process.

1. MMG$DELGBLSEC checks whether this is a memory-resident section. If so, and if the section has an associated shared page table section, it gets the address of the associated section's GSD, clears its GSD$L_RELATED_GSTX, and inserts it into the delete pending list so it can be cleaned up first. It continues with step 3.
2. If the section is not memory-resident or has no associated shared page table section, it removes the GSD from its current list, which could be the group or systemwide list or the delete pending list, and inserts it into the delete pending list so that no more processes can map to it.
3. Starting with SEC$L_UNIT_CNT, the number of pagelets in the section, it calculates the number of pages in the section.
4. It gets the starting GPTX from the GSTE.
5. If this section is a Galaxywide global section, MMG$DELGBLSEC reads the GPTE to get its PFN and then gets the shared memory region ID from the PFN database. It calls GLX$SHM_REG_DELETE, in module [GALAXY]GLX_SHM_REG, to delete the shared memory region.
   - If GLX$SHM_REG_DELETE returns an error status other than SS$_NOBREAK or SS$_NOWAIT, MMG$DELGBLSEC generates a fatal DELGBLSEC bugcheck.
   - If it returns the error status SS$_NOWAIT, indicating spinlock time out, MMG$DELGBLSEC calls it again.
   - If it returns SS$_NOBREAK or a success status, MMG$DELGBLSEC continues with step 12.
6. If this section is not a Galaxywide global section, MMG$DELGBLSEC acquires the MMG spinlock, raising IPL to IPL$_MMG.
7. It scans the section's GPTEs to determine the state of the global pages. Processing done in this scan eliminates references to these GPTEs from the PFN database records of both valid memory-resident global section pages and transition pages. It also reduces pointless modified page writing of pages from a page file global section

that is being deleted. If MMG$DELGBLSEC reaches the last GPTE rather than one of the end conditions in the following list, it continues with step 10.

- If it finds a valid page, it confirms that the page is a memory-resident section page, crashing the system with a DELGBLSEC bugcheck if not. It checks that the physical page's PFN$L_SHRCNT and PFN$W_REFCNT are both 1, crashing the system with a DELGBLSEC bugcheck if not. It then stores the contents of PFN$Q_BAK in the GPTE, resetting it to the invalid form, and reinitializes the page's PFN record fields.

  If the page is not from a preallocated memory-resident section, MMG$DELGBLSEC inserts the page at the head of the free page list.

  In either case, it calls MMG_STD$DECPTREF_GPT to decrement the reference count on the global page table page.
- If it finds a transition page on the free page list, it calls MMG$DEL_PFNLST, in module ALLOCPFN, to delete the page's virtual contents. The PFN is moved from its current position on the free page list to the head of the list, so that it can be reallocated before pages whose contents might still be useful. Its PFN record fields are reinitialized.
- If it finds a global page-file section page on the modified page list, it clears the saved modify bit in the physical page's PFN$L_PAGE_STATE field and calls MMG$DEL_PFNLST as described. It continues its scan of the section's GPTEs.
- If it finds a transition page on the modified page list that is not part of a global page-file section, the page must be written to its backing store before the section is deleted, and MMG$DELGBLSEC goes to step 8.
- If it finds a transition page that is not part of a global page-file section and that is not on the free or modified page list, the page is being written to its backing store. That I/O must complete before the section is deleted, and MMG$DELGBLSEC goes to step 9.

8. It requests the modified page writer to perform a selective purge of the modified page list to write this section's global pages to their backing store and release them (see Chapter 4).
9. It releases the MMG spinlock, restoring IPL to 2, stores the process ID of the current process in GSD$L_IPID as the target of an eventual cleanup AST. It sets PHD$V_DALCSTX in the system header so that MMG$DALCSTXSCN and thus MMG$DELGBLSEC will be called again some time later, possibly when modified page writing is complete. It returns.
10. If MMG$DELGBLSEC has scanned all the GPTEs for the section and found none for whose I/O it must wait, it scans the GPTEs again, this time to decrement the global page table page reference count and to release page file backing store.

    - If it finds a global page in a page file, it deallocates that page, decrements the global page table page reference count, and clears the GPTE.

- If it finds a demand zero global page, it simply decrements the global page table reference count and clears the GPTE. When an entire page of GPTEs is freed, the global page table page can be unlocked from the system working set.

11. It releases the MMG spinlock, setting IPL to 2.
12. It deallocates the GPTEs.
13. It tests whether the global section is a memory-resident section. If so, it resets this global section's RAD in the global section RAD array (RIH$PQ_GBLSEC_RADS) to –1. It calls MMG_STD$RETURN_RES_MEM, in module MEM_ALLOC, to return reserved memory pages and update the RMD. It continues with step 16.
14. It tests whether a file is open on the section. If not, this was a global page-file section, and MMG$DELGBLSEC adds its page count back to MMG$GL_GBLPAGFIL. It continues with step 16.
15. If there is a file open, and if this is a shared WCB, it decrements the reference count in the WCB. If the count is now zero, it inserts the WCB into a queue of delete pending WCBs.
16. It removes the GSD from the delete pending list and deallocates the GSD and ORB to paged pool, unless the ORB is still in use for an open section file.
17. It deallocates the GSTE.
18. If the section just deleted was a shared page table section, MMG$DELGBLSEC continues with step 1 to delete the associated memory-resident section.
19. It allocates nonpaged pool, forms it into an AST control block, queues a normal kernel mode AST to the current process, and returns to its caller. The specified AST procedure is GSD_CLEAN_AST.

GSD_CLEAN_AST, in module SYSDGBLSC, executes as a normal kernel mode AST procedure in the context of the process that requested the system service that triggered MMG$DELGBLSEC, possibly but not necessarily the process that requested deletion of the global section. Its enqueuing can be requested from MMG$DELGBLSEC or the modified page writer, and also by the routines that decrease section reference count, MMG$SUBSECREF and MMG$DECSECREF in module PHDUTL, when a temporary global section's reference count goes to zero. It takes the following steps:

1. GSD_CLEAN_AST tests whether the process is being deleted or already has this procedure active. If either is true, it returns.
2. It requests the Clear AST ($CLRAST) system service so that a subsequent kernel mode AST can be delivered.
3. If PHD$V_DALCSTX in the system header is set, it locks the GSD mutex, calls MMG$DALCSTXSCN (see Section 3.9.2) and unlocks the mutex.
4. It calls MMG$DELGBLWCB (see Section 3.9.4) to close the section file.
5. It returns.

3.9.4 MMG[_STD]$DELGBLWCB Routine

MMG$DELGBLWCB and its alternative entry point, MMG_STD$DELGBLWCB, in module SYSDGBLSC, are called to close an open file associated with a temporary global section whose reference count has gone to zero and to delete the WCB. It takes the following steps:

1. It makes several consistency checks, returning immediately if it is executing within a kernel thread that owns any mutexes, has kernel mode AST delivery disabled, has an active kernel mode AST, or if the file system impure area in this process is not yet initialized. Its subsequent processing requires delivery of a kernel mode AST, IPL 0 execution, and file system processing.
2. It removes a WCB from the delete pending list, returning if there is none.
3. It checks that the job has enough open file quota so that a deduction can be made from it. If not, MMG$DELGBLWCB inserts the WCB back on the delete pending list and returns.
4. It finds an available channel control block and stores in it the address of the unit control block on which the file represented by the WCB is open; the address of the WCB; and an indication that the channel has been assigned in kernel mode.
5. It locks the I/O database mutex for write; increments the unit's reference count, which will be decremented in the next step; and unlocks the I/O database mutex.
6. It lowers IPL to 0 and requests the Deassign Channel ($DASSGN) system service, the actions of which result in closing the file.
7. It raises IPL back to 2 and continues with step 2.

3.10 Virtual Address Space Deletion

Page deletion is generally more complicated than page creation. Creation involves taking the process from one known state (the address space does not yet exist) to another known state (for example, the process-private PTEs contain demand zero L3PTEs). Page deletion must deal with initial conditions that include all possible states of a virtual page.

Page creation may first require that the specified pages be deleted to put the process page tables into their known state. Thus, page deletion is often an integral part of page creation.

A process deletes part of its address space by requesting the $DELTVA, $DELTVA_64, or $CNTREG system service.

A page that has I/O in progress cannot be deleted until the I/O completes. A kernel thread trying to delete such a page may be placed in one of several wait states, depending on the page type and state, for example:

- A kernel thread trying to delete a page in the write-in-progress transition state is placed into a page fault wait state (with a request that a system event be reported when I/O completes) until the page write completes.
- A page in the read-in-progress transition state is faulted, with the result that the kernel thread is placed into page fault wait.
- Additional tests are required when a kernel thread deletes a global page with I/O in progress because there is no way to determine if the process deleting the page is also responsible for the I/O. Hence, if the process has any direct I/O in progress, the kernel thread may be placed into a resource wait for the resource RSN$_ASTWAIT until its direct I/O completes.

  Section 3.10.2 has further details.

3.10.1 $DELTVA System Service

A process requests the Delete Virtual Address Space ($DELTVA) system service to delete process-private address space. Service arguments are the range to be deleted and, optionally, the actual range deleted and the access mode associated with the request.

The $DELTVA system service procedure, EXE$DELTVA in module SYSCREDEL, runs in kernel mode. EXE$DELTVA takes the following steps:

1. It creates and initializes the scratch space on the stack.
2. It tests the accessibility of the INADR argument and maximizes the ACMODE argument with the mode of the service requestor.
3. It sets flag MMG$V_CLUSTER_DEL in MMG$L_FLAGS.
4. It raises IPL to 2.
5. It locates the RDE corresponding to the INADR argument's ending address. If there is none, it picks either the P0 or P1 RDE, depending on the value of the ending address. If that region is one with shared page tables, it returns the error status SS$_NOSHPTS. A process may not delete process-private address space that could be mapped by a shared page table.
6. EXE$DELTVA tests whether the service was requested from IPL 0 and whether the region is a process-permanent region. If both are true, it sets MMG$V_RWAST_AT_IPL0 to enable MMG$DELPAG_64, the per-page service-specific routine, to wait the kernel thread at IPL 0 if a wait is necessary. Section 3.10.2 has further details.
7. If the range to be deleted crosses the boundary between defined and undefined address space in the region, it adjusts the ending address to be within the defined address space.
8. It calls MMG$CREDEL (see Section 3.1.1), specifying MMG$DELPAG_64 as the per-page service-specific routine.

9. If contiguous address space was deleted, it calls CHECK_CONTRACT_64 (see Section 3.10.3) to contract the region and account for now empty page tables that will be deleted.

   Otherwise, if a page within the range could not be deleted, it calls CHECK_CONTRACT_64_1 (see Section 3.10.3) to contract the region page by page. CHECK_CONTRACT_64_1 simply alters the input arguments for CHECK_CONTRACT_64 to ensure that contraction is checked one page at a time.

10. It restores the IPL at entry.

11. EXE$DELTVA records peak page file use and virtual size statistics, and stores return information in the optional RETADR argument.

12. It executes an instruction memory barrier to flush any instructions that might have been prefetched from the deleted address space.

13. It returns to its requestor.

3.10.2 MMG[_STD]$DELPAG_64 Routine

When a virtual page is deleted, all process and system resources associated with the page must be returned. These can include the following:

- A physical page of memory for a valid or transition page
- A page file page for a page whose backing store address indicates already allocated blocks
- A working set list entry for a page in a process working set list
- Page file quota for a page with a page file type backing store

Deleting a process-private section page results in decrementing the reference count in the PSTE (see Figure 2.7). If the reference count goes to zero, the PSTE itself can be released. Deleting a global section page results in decrementing the reference count in the GSTE. If the reference count goes to zero, the GSTE itself can be released.

In addition, a valid or modified page with a section file backing store address rather than a page file backing store address must have its latest contents written back to the section file. (The contents of a page with a page file backing store address are unimportant after the virtual page is deleted and do not have to be saved before the physical page is reused.)

Deleting the contents of a physical page means that the PFN$Q_PTE_INDEX and PFN$L_PT_PFN fields in its PFN database record are cleared, destroying all ties between the physical page and any process-private virtual address. In addition, the page is placed at the head of the free page list, so that it can be reallocated before other pages whose contents might still be useful.

MMG[_STD]$DELPAG_64, in module SYSCREDEL, is the per-page service-specific routine for the $DELTVA, $DELTVA_64, and $CNTREG system services. Its arguments include the address to be deleted, the RDE address, the address of memory management flags (MMG$L_FLAGS in the case of entry from $DELTVA and $CNTREG), and the number of pages to be deleted.

It takes the following steps:

1. It gets the address of the L3PTE that maps the specified virtual address and, if necessary, faults the page table page into the process's working set list. It acquires the MMG spinlock, raising IPL.
2. It examines the L3PTE that maps the page to be deleted.
3. If the L3PTE contains zero, the page is a null page and has already been deleted. If the page is a demand zero L3PTE, it zeros the L3PTE and restores page file quota deducted for it.

   If MMG$V_CLUSTER_DEL is clear, indicating only one page is to be deleted at a time, MMG$DELPAG_64 returns to its caller after releasing the MMG spinlock and restoring the previous IPL.

   If MMG$V_CLUSTER_DEL is set, MMG$DELPAG_64 attempts to delete a cluster of similar pages, as many as the lesser of pages in the delete range and pages mapped by the same L3PT. It continues deleting, by clearing the L3PTE and incrementing JIB$L_PGFLCNT, until it has reached the maximum or found an L3PTE that is not zero or a demand zero PTE. It releases the MMG spinlock and returns.
4. If the page is neither a null nor a demand zero page, and its valid bit is clear, MMG$DELPAG_64 performs a sanity check that the NO_MB bit is clear in the PTE. If not, it generates the fatal bugcheck INCONMMGST.

   If the PTE valid bit is set or the NO_MB bit is clear, MMG$DELPAG_64 compares the requestor access mode with that of the page owner. If the access mode is insufficiently privileged, it releases the MMG spinlock and returns the error status SS$_PAGOWNVIO.
5. Otherwise, it determines the type of the virtual page, based on the valid and type bits in the L3PTE that maps it.
6. If the page is in a page file, MMG$DELPAG_64 deallocates the occupied page of page file, restores job page file quota, clears the L3PTE, releases the MMG spinlock, and returns.
7. If the page is from a demand zero process section, MMG$DELPAG_64 releases the MMG spinlock, lowers IPL, touches the page to fault it into the working set, and continues with step 1. Faulting it into the working set first ensures that an untouched demand zero page backed by a section file will be written back to it as all zeros. Handling it in this way minimizes the need for complex code to handle a relatively rare case.

8. If the page is an invalid page from a read-only or copy-on-reference process section, MMG$DELPAG_64 tests MMG$V_CLUSTER_DEL to see whether a cluster of pages can be deleted. If so, it attempts to delete a cluster of pages. The cluster size is the minimum of the number of pages in the delete range, pages mapped by the same L3PT, and the contents of MMG$GL_CLONE_CLUMP. (The intent is to constrain the maximum amount of time the memory management spinlock is held.) For each valid page it encounters, it removes the page from the working set list, releases the PFN, invalidates the TB, and clears the L3PTE. It continues, clearing L3PTEs, until it has reached the maximum count or a page that cannot be deleted in this cluster, for example, a page from another section. It adjusts the section reference count and, if the section is copy-on-reference, JIB$L_PGFLCNT as well, by the number of pages deleted. It releases the MMG spinlock and returns.

9. If the page is an invalid page from any other type of process section, or if page deletion clustering is not allowed, MMG$DELPAG_64 decrements the section reference count. If the page is copy-on-reference, MMG$DELPAG_64 also increments the job page file quota. It clears the L3PTE, releases the MMG spinlock, and returns.

10. If the page is any other type of transition page, MMG$DELPAG_64 examines the page's PFN$L_PAGE_STATE location bits to determine its actions:

    — Free page list. MMG$DELPAG_64 calls MMG$DEL_PFNLST, in module ALLOCPFN, to delete the page's virtual contents and modify the L3PTE. The PFN is moved from its current position on the free page list to the head of the list. Its PFN record is reinitialized. PFN$V_DELCON is set in the page's PFN$L_PAGE_STATE field. The PTE is reinitialized with its backing store contents. MMG$DELPAG_64 continues with step 2 to delete the virtual page in its new state.

    — Modified page list. If the page has page file backing store, MMG$DELPAG_64 clears the saved modify bit in the page's PFN$L_PAGE_STATE field so that the page, when deleted, will be inserted into the free page list, and calls MMG$DEL_PFNLST, as just described. The PTE is reinitialized with its backing store contents. MMG$DELPAG_64 continues with step 2 to delete the virtual page in its new state.

      If the page is a section page, MMG$DELPAG_64 releases the MMG spinlock, lowers IPL, touches the page to fault it into the working set, and continues with step 1. Handling the page this way simplifies MMG$DELPAG_64's subsequent steps to write the page to its section file.

    — Read-in-progress or release pending. MMG$DELPAG_64 releases the MMG spinlock, lowers IPL, touches the page to fault it into the working set, and continues with step 1.

    — Write-in-progress. The I/O must complete before the page can be deleted. MMG$DELPAG_64 releases the MMG spinlock and places the kernel thread into a page fault wait at the IPL of the caller. When the kernel thread is resumed, MMG$DELPAG_64 raises IPL to 2 and continues with step 1.

— Page read error. MMG$DELPAG_64 continues with the next step.

— Active. MMG$DELPAG_64 continues with the next step.

11. If the page is valid (or a transition page that is active or that incurred a page read I/O error), MMG$DELPAG_64 examines the page's PFN$L_PAGE_STATE field, the window bit in the PTE, and other memory management data structures as needed, to determine its actions:

— PFN-mapped section page. MMG$DELPAG_64 tests whether the process has direct I/O in progress. If not, it clears the valid, fault-on-read, fault-on-write, and NO_MB bits in the L3PTE. It decrements PFN$W_PT_WIN_CNT for the L3PT that maps the section page to indicate one less reason for that page table page to be locked into the working set list. If the count transitions to –1, it decrements PHD$L_PTCNTLCK, clears the L2PTE PTE$V_WINDOW bit, and clears WSL$V_PFNLOCK in the L3PT working set list entry. It invalidates any possible TB entry and clears the entire L3PTE. (Note that if the page being deleted is part of a PFN-mapped granularity hint region, the granularity hint bits are cleared in all other L3PTEs that map pages in the granularity hint region.) It releases the MMG spinlock and returns.

If the process has direct I/O in progress, its I/O must complete before this page can be deleted. When direct I/O is in progress to a typical process page, its PFN$W_REFCNT field is incremented. Thus a value larger than 1 indicates I/O in progress. A PFN-mapped page may have other processes mapped to it, some of which could be doing I/O to it, so its REFCNT value is not precise enough to determine whether the page is in use as an I/O buffer for this process. Furthermore, a page mapped by PFN may be one without any PFN database to examine.

If bit MMG$V_NOWAIT_IPL0 in MMG$L_MMG_FLAGS is set (as it would be if the page were being deleted as a side effect of creating a process section that overmapped the page), the kernel thread cannot wait at IPL 0 for the I/O to complete, and MMG$DELPAG_64 releases the MMG spinlock and returns the error status SS$_ABORT to its caller. Otherwise, it releases the MMG spinlock and places the kernel thread into a resource wait for resource RSN$_ASTWAIT (effectively, wait for an I/O completion) at IPL 0. When the kernel thread is placed back into execution, MMG$DELPAG_64 raises IPL to 2 and resumes at step 1.

— Galaxywide section page. MMG$DELPAG_64 determines the address of the shared memory section descriptor and, from that, the GSTX.

If the page is not mapped by a shared page table, it checks whether I/O is in progress to the section and whether the process has direct I/O outstanding. If so, the I/O must complete before the page can be deleted. It releases the MMG spinlock and places the kernel thread into a wait on RSN$_ASTWAIT. When the kernel thread is resumed, MMG$DELPAG_64 continues with step 1. If no I/O is outstanding, MMG$DELPAG_64 checks whether the page is part of a buffer object. If so, it releases the MMG spinlock and returns SS$_VA_IN_

USE. Otherwise, it decrements the global section reference count and continues as for a PFN-mapped page.

If the page is mapped by a shared page table, MMG$DELPAG_64 must check whether *all* the pages mapped by that shared L3PT can be deleted: it is not possible to alter an L3PTE in a shared page table. It checks whether any pages are part of a buffer object and, if so, unlocks the MMG spinlock and returns SS$_VA_IN_USE. It checks whether the process has any outstanding direct I/O in progress and, if so, places the kernel thread into a wait for it to complete. (Checking the PFN$W_REFCNT of all pages mapped by the shared L3PT is a lengthy operation and would not uniquely associate any pending I/O with this process in any case.)

Once all the I/O is complete, MMG$DELPAG_64 dissolves any granularity hint region that includes the L3PT. It decrements PFN$W_PT_WIN_CNT for the L2PT that maps the L3PT. It reduces the section reference count for both the Galaxywide section and the shared page table section. It clears the L2PTE that mapped the shared L3PT and invalidates any possible TB entry for the shared L3PT. If the process is multithreaded, MMG$DELPAG_64 invalidates all translation buffer entries on the system, because other threads of the process might be current on other processors. Otherwise, it merely invalidates all process-private TB entries on this processor. It releases the MMG spinlock and returns.

— Memory-resident section page. MMG$DELPAG_64 gets the GSTX from the PFN$Q_BAK information. It checks whether the page's reference count is elevated and whether the process has direct I/O outstanding. If both are true, I/O is presumed to be in progress to the page. The I/O must complete before the page can be deleted. It releases the MMG spinlock and places the kernel thread into a wait on RSN$_ASTWAIT. When the kernel thread is resumed, MMG$DELPAG_64 continues with step 1.

If the reference count is elevated but no direct I/O is outstanding, MMG$DELPAG_64 checks whether the page is part of a buffer object. If so, it releases the MMG spinlock and returns SS$_VA_IN_USE. If not, it checks whether the section is mapped with shared page tables. If so, it takes the actions described previously for deleting Galaxywide global section pages mapped by a shared page table.

If the section is not mapped with shared page tables and if only one page is being deleted, MMG$DELPAG_64 decrements the global section reference count and continues as for a PFN-mapped page. If more pages are being deleted and MMG$V_CLUSTER_DEL is set, it attempts to delete a cluster of pages. The cluster size is the minimum of the number of pages in the delete range, pages mapped by the same L3PT, and the contents of MMG$GL_CLONE_CLUMP. It continues, clearing L3PTEs and invalidating TB entries, until it has reached the maximum count or a page that is not part of this global section. It adjusts the section reference count and PFN$W_PT_WIN_CNT.

- Permanently locked in the working set. MMG$DELPAG_64 simply releases the MMG spinlock and returns a success code. Such a page cannot be deleted until the process is deleted or outswapped.
- Process-locked page. MMG$DELPAG_64 releases the MMG spinlock; calls MMG$LCKULKPAG, in module SYSLKWSET (described in Chapter 5) to unlock the page; and then resumes at step 1.
- I/O buffer page. If the PFN$W_REFCNT field for this page contains a value larger than 1, the page is in use as an I/O buffer. MMG$DELPAG_64 tests whether the page is part of a buffer object (see Section 3.12) and, if so, releases the MMG spinlock and returns the error status SS$_VA_IN_USE to its caller. If the page is not part of a buffer object, MMG$DELPAG tests against MMG$V_NOWAIT_IPL0 as previously described and either returns an error status or places the kernel thread into a wait until the I/O completes.
- Unmodified page and modified page with page file backing store. MMG$DELPAG_64 calls MMG$REL_PFN, in module ALLOCPFN, which sets the PFN$V_DELCON bit and clears the saved modify bit in the PFN$L_PAGE_STATE field so the page's contents will be deleted when it is inserted into the free page list. It clears the valid, modify, fault-on-execute, and fault-on-write bits in the L3PTE; invalidates any possible TB entry; and removes the page from the working set list. MMG$DELPAG_64 decrements its PFN$W_REFCNT field.

  It deallocates the associated physical page, as a result of which the L3PTE once again contains a backing store format, and then resumes with step 1, deleting the page as an invalid unmodified page-file section page.
- Modified page backed by a section file. MMG$DELPAG_64 calls MMG$WRT_PGS_BAK, in module SYSUPDSEC (see Chapter 4), to write the page to its backing store. The page's modify bit is cleared and its state is changed to write in progress. When the I/O completes, MMG$DELPAG_64 calls MMG$REL_PFN and decrements PFN$W_REFCNT. If the count transitions to zero, MMG$DELPAG_64 deallocates the associated physical page as previously described and resumes with step 1. If the count is still positive, it must place the kernel thread into a wait.
- Valid and unmodified page. MMG$DELPAG_64 decrements PFN$W_REFCNT, calls MMG$REL_PFN, and either waits the kernel thread for I/O to complete or deallocates the page. It resumes with step 1.

12. If the page is an invalid global page, MMG$DELPAG_64 examines the associated GPTE to determine the page type and validity of the master page.

    - If the global page is in transition and is a Galaxywide section page, one that has not yet been zeroed, MMG$DELPAG_64 treats the page as it does a valid L3PTE representing a Galaxywide section page.

- If the global page is an invalid memory-resident page, for example, one that has not yet been zeroed, MMG$DELPAG_64 treats the page as it does a valid L3PTE representing a memory-resident page.
- If the master page is a demand zero page or a page in a global page-file section, MMG$DELPAG_64 decrements the global section reference count and clears the process L3PTE. It releases the MMG spinlock and returns.
- If the global page is in transition being faulted from its backing store, MMG$DELPAG_64 tests and sets MMG$V_DELGBLDON in the memory management flags. If the bit was already set, it continues with the next step. Otherwise, MMG$DELPAG_64 must free the process's working set list entry associated with the global page. It calls a routine within the Purge Working Set ($PURGWS) system service to remove that page and any other global pages in the address range being deleted from the working set list and to change the PFN database accordingly. It resumes with step 1.
- If the global page is valid or in transition and has an elevated PFNW_REFCNT, MMGDELPAG_64 tests whether the page is part of a global buffer object. If the page is part of a system global buffer object and the reference count is 1, the page may be deleted. If the global buffer page is not part of a system global buffer object, MMG$DELPAG_64 releases the MMG spinlock and returns SS$_VA_IN_USE.

  If the page is not part of a global buffer object, it has I/O in progress. If the process has outstanding direct I/O, the direct I/O may be to the global page that the process is trying to delete. MMG$DELPAG_64 therefore places the kernel thread into a resource wait, as previously described, until the I/O completes. It resumes with step 1.

  If the process has no outstanding direct I/O, MMG$DELPAG_64 continues with the next step.
- If the global page is valid with no I/O in progress, or invalid and in a section file, or a transition page with no I/O in progress, MMG$DELPAG_64 examines its PFN$Q_BAK field to determine the type of section. If the section is demand zero, it continues with the next step. If the section is copy-on-reference, it first increments the job page file quota. For any type of section that is not demand zero, MMG$DELPAG_64 decrements the global section reference count, clears the process PTE, releases the MMG spinlock, and continues at step 1.
- If the global page is invalid and a page from a demand zero writable section, MMG$DELPAG_64 allocates a physical page and maps it temporarily to zero it. MMG$DELPAG_64 initializes the page's PFN database record, storing the address of the global table entry in PFN$Q_PTE_INDEX and setting PFN$L_PAGE_STATE global writable and active. It decrements the global section's reference count and calls MMG$INCPTREF, in module PAGEFAULT, to lock the global page table page. MMG$DELPAG_64 then inserts the page onto the modified page list, clears the process L3PTE, releases the MMG spinlock, and returns.

These steps ensure that an untouched demand zero page backed by a global section file will be written back to it as all zeros. This requirement is similar to that for a demand zero page in a writable process section. However, MMG$DELPAG_64 takes these steps rather than fault the page in first as it does a process-private page, for better performance in a more common case.

3.10.3 [MMG_STD$]CHECK_CONTRACT_64 and [MMG_STD$]CHECK_CONTRACT_64_1 Routines

CHECK_CONTRACT_64 or one of its alternative entry points, MMG_STD$CHECK_CONTRACT_64 and MMG_STD$CHECK_CONTRACT_64_1, in module SYSCREDEL, takes the following steps:

1. CHECK_CONTRACT_64 calls MMG$DALCSTXSCN (see Section 3.9.2) to see if any process sections can be deleted.
2. If this system has mapped any Galaxywide shared memory, CHECK_CONTRACT_64 locks the GSD mutex for write, calls MMG$DALCSTXSCN to see if any global sections can be deleted, and unlocks the mutex.
3. CHECK_CONTRACT_64 initializes PHD$PQ_PT_NO_DELETE1 and PHD$PQ_PT_NO_DELETE2 to prevent the asynchronous deletion of any L2PTs that map the deleted range so that it can scan them without holding the MMG spinlock the entire time.
4. It checks whether one end of the range to be deleted is the same address as the end of the defined address space in the region. If not, the range must be deleted one page at a time, and it continues with step 6.
5. It calls MMG_STD$DELETE_PTS, in module SYS_CREDEL_64, to delete now empty L2PTs and L3PTs that map the deleted address range, starting from the last defined space in the region. MMG_STD$DELETE_PTS examines each associated L1PTE:

   — It skips any L1PTE containing zero.

   — If the L1PTE contains the demand zero form of PTE, it clears it, decrements PHD$L_PTCNTMAX to indicate one less (L2) page table, and increments JIB$L_PGFLCNT to return charges against page file quota.

   — In any other case, it examines each PTE in the page of the L2PT mapped by that L1PTE. It deletes an empty L3PT by clearing the demand zero L2PTE and adjusting PHD$L_PTCNTMAX and JIB$L_PGFLCNT.

   CHECK_CONTRACT_64 adjusts PHD$Q_FREE_PTE_COUNT by the number of pages being deleted and RDE$PQ_FIRST_FREE_VA by the number of bytes of address space deleted.

6. Beginning with the page tables that map the last defined page in the region, based on the contents of RDE$PQ_FIRST_FREE_VA, it scans for a nonzero L3PTE. It begins with the L1PTE that maps that page, skipping it if it contains zero and going on to the next. When it finds a nonzero L1PTE, it scans the PTEs in that L2PT until it finds a nonzero one. It then scans the PTEs in that L3PT, until it finds a nonzero one. The nonzero L3PTE corresponds to what is now the last defined page in the region.

   It adjusts RDE$PQ_FIRST_FREE_VA to reflect the new end of the region.

7. It clears PHD$PQ_PT_NO_DELETE1 and PHD$PQ_PT_NO_DELETE2 to permit asynchronous page table deletion.
8. It calls MMG$EXTRADYNWS, in module SYSADJWSL, to recalculate the number of fluid working set list entries available to the process, given the number of page tables that have just been deleted.

3.10.4 $DELTVA_64 System Service

The Delete Virtual Address Space ($DELTVA_64) system service is requested to delete virtual address space. It resembles the $DELTVA system service, but its arguments include a region ID, and all its address arguments are 64 bits. Thus it can be used to delete virtual address space in P0, P1, or P2 space, either in a default region or a user-created one.

The $DELTVA_64 system service procedure, EXE$DELTVA_64 in module SYS_CREDEL_64, runs in kernel mode. It resembles EXE$DELTVA with the following major differences:

- It checks that the address range to be deleted lies within defined space in the specified region.
- It loops through the pages to be deleted, calling MMG_STD$DELPAG_64 (see Section 3.10.2) until it returns an error status or all pages are done.

3.10.5 $CNTREG System Service

The Contract Region ($CNTREG) system service procedure, EXE$CNTREG in module SYSCREDEL, runs in kernel mode. The $CNTREG system service is a special case of the $DELTVA system service. EXE$CNTREG simply converts the requested number of pagelets into a P0 or P1 page range and merges with EXE$DELTVA at step 7 in the description in Section 3.10.1.

Use of the $CNTREG system service is reserved to Hewlett-Packard Company. Any other use is unsupported.

3.11 Virtual Address Region Deletion

The Delete Virtual Region ($DELETE_REGION_64) system service is requested to delete a particular region. Its arguments include the ID of the region to be deleted and the access mode associated with the request.

The $DELETE_REGION_64 system service procedure, EXE$DELETE_REGION_64 in module SYS_REGIONS, runs in kernel mode. EXE$DELETE_REGION_64 takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it checks that the region to be deleted is a user-defined one, returning SS$_IVREGID if not.
2. It raises IPL to 2.
3. It locates the RDE corresponding to the REGION_ID_64 argument. If there is none, it returns the error status SS$_IVREGID.
4. It checks that the access mode from which the service is requested is at least as privileged as that of the owner of the region, returning the error status SS$_NOPRIV if not.
5. It calculates how many pages are in the region. If there are none, it removes the RDE from its two lists, deallocates it to P1 pool, and returns SS$_NORMAL to its requestor.
6. If there are pages in the region, it loops, calling MMG_STD$DELPAG_64 (see Section 3.10.2) to delete them. If the region grows in an ascending direction, it deletes the high-address pages first. If the region grows in a descending direction, it deletes the low-address pages first.

   It continues until MMG_STD$DELPAG_64 returns an error status or all pages are deleted. If no pages were deleted, EXE$DELETE_REGION_64 returns the error status from MMG_STD$DELPAG_64.
7. Otherwise, it calculates how many pages were deleted and the starting and ending addresses of the deleted pages. It executes an instruction memory barrier (see Chapter *Synchronization Techniques*) in case instructions were deleted that might have been prefetched.
8. It calls MMG_STD$CHECK_CONTRACT_64 (see Section 3.10.3) or MMG_STD$CHECK_CONTRACT_64_1, depending on whether a contiguous range of pages was deleted or whether there was at least one page that could not be deleted, to contract the region.
9. If MMG_STD$DELPAG_64 returned an error status, EXE$DELETE_REGION_64 returns that error status to its requestor along with the starting address and number of bytes deleted.
10. Otherwise, it removes the RDE from its two lists, deallocates it to P1 pool, and returns SS$_NORMAL to its requestor along with the starting address and number of bytes deleted.

3.12 Buffer Object Creation and Deletion

A buffer object is a special kind of I/O buffer. The pages that make up a buffer object are locked into physical memory and may be doubly mapped in system space as well as process-private space. Multiple I/O requests can be initiated to or from an existing buffer object with less overhead than with the standard I/O mechanisms, direct and buffered I/O.

The pages of a standard direct I/O buffer are probed and locked into memory when the I/O request is initiated. The PFN$W_REFCNT in the PFN database record of each page is incremented to lock the page. The page table pages that map the buffer pages are locked into the process's working set and into memory, and the process's header cannot be outswapped. When the I/O request completes, for each page of the buffer, PFN$W_REFCNT is decremented, and page table pages are unlocked.

Buffered I/O is initiated to or from a buffer allocated in nonpaged pool. On output, data is copied from the user's buffer to the pool buffer. On input, data is copied from the pool buffer to the user's buffer.

In contrast, the pages of a buffer object are probed only once and the pages are locked only once, at buffer creation. The L3PT pages that map the buffer object in process-private address space are locked into memory. Because the buffer object pages have a process-private mapping, there is no need to copy data between a process buffer and a system buffer. Tests for buffer object pages in the swapper make it possible to outswap a process body and header even though I/O may be in progress to its buffer object.

The pseudo terminal driver, [PTD]SYS$FTDRIVER, uses the buffer object mechanism. A process interacts with this driver through system services (in the privileged shareable image [PTD]PTD$SERVICES_SHR.EXE) that create and manage the process's buffer objects. The DECwindows terminal class driver, [DECW$XTERMINAL]DECW$XTDRIVER.EXE, also uses the buffer object mechanism.

OpenVMS Version 7.0 added several I/O system services that enable an application to use buffer objects for disk and tape I/O transfers: Set Up Fast I/O ($IO_SETUP), Perform Fast I/O ($IO_PERFORM), and Clean Up Fast I/O ($IO_CLEANUP). Typically, an application creates one or more buffer objects, requests $IO_SETUP to make its buffer objects known and to define other information for the I/O requests, and calls $IO_PERFORM repeatedly to do multiple transfers into or from the buffer objects. When all I/O is done, the application calls $IO_CLEANUP and then deletes the buffer objects. Consult the *OpenVMS System Services Reference Manual* for additional information on these services.

Use of the $CREATE_BUFOBJ system service is reserved to Hewlett-Packard Company. Any other use is unsupported. Use of the $CREATE_BUFOBJ_64 and $DELETE_BUFOBJ system services, however, is supported.

An image creates a buffer object by requesting the $CREATE_BUFOBJ or $CREATE_BUFOBJ_64 system service. (Actually, a user's image should not directly request $CREATE_BUFOBJ; it may, however, request a pseudo terminal system service that requests $CREATE_BUFOBJ.) The system service creates a data structure called a buffer object descriptor (BOD; see Chapter 2) that contains the process-private and system space addresses of the buffer.

The $IO_SETUP system service increments the reference count associated with the buffer object. The $IO_CLEANUP system service decrements the reference count.

When an image deletes a buffer object by requesting the $DELETE_BUFOBJ system service, if the reference count is nonzero, the buffer object is merely marked for deletion. Its actual deletion is deferred until the reference count is decremented to zero.

Chapter 4 provides additional information on the transitions of buffer object pages and their page tables.

3.12.1 $CREATE_BUFOBJ System Service

The Create Buffer Object ($CREATE_BUFOBJ) system service procedure, EXE$CREATE_BUFOBJ in module SYSLKWSET, runs in kernel mode. The service is requested with the following arguments:

- INADR, RETADR, and ACMODE—The standard memory management service arguments
- FLAGS—Flags, for inner mode requestors only, to specify that limit checking is to be bypassed or that the RETADR argument addresses should contain the system space addresses
- CREBUF_HANDLE—The address of a two-longword array to receive the buffer handle of the created buffer object

A buffer handle identifies the buffer object in subsequent I/O, $IO_SETUP, and $DELETE_BUFOBJ requests. The first longword of a buffer handle contains the address of the BOD. The second longword contains a sequence number, copied from BOD$L_SEQNUM, which is used to validate the buffer handle itself.

A buffer object created through this service must be in 32-bit process-private address space and is always doubly mapped in S0/S1 space.

EXE$CREATE_BUFOBJ takes the following steps:

1. It creates and initializes scratch space on the stack.
2. It tests the accessibility of the INADR argument and maximizes the ACMODE argument with the mode of the service caller.
3. If the service was requested from user mode, EXE$CREATE_BUFOBJ checks whether the current security persona has the rights identifier VMS$BUFFER_OBJECT_USER. If not, it returns the error status SS$_NOBUFOBJID.

If a user requestor specified either flag CBO$V_RETSVA or CBO$V_EXMAXLIM, EXE$CREATE_BUFOFJ returns the error status SS$_NOPRIV.

4. It raises IPL to 2 to block AST delivery.
5. It determines the number of pages the buffer object is to contain and adds that count to PMS$GL_BUFOBJ_PAGES_S0S1.
6. It tests that the CREBUF_HANDLE is writable and, if not, restores the previous value of PMS$GL_BUFOBJ_PAGES_S0S1 and returns the error status SS$_ACCVIO.
7. It allocates nonpaged pool for a BOD, charging the pool against the process's byte count quota and limit.
8. It initializes the BOD, copying the process ID to BOD$L_PID and the access mode to BOD$L_ACMODE, and links it to the tail of the PCB list at PCB$Q_BUFOBJ_LIST.
9. It allocates system page table entries (SPTEs) to map the buffer into system space and stores the address of the first SPTE in BOD$PQ_VA_PTE and the corresponding starting system virtual address in BOD$PQ_BASESVA.
10. It increments the master buffer object sequence number and stores that number in BOD$L_SEQNUM.
11. EXE$CREATE_BUFOBJ calls MMG$CREDEL (see Section 3.1.1), specifying LCKBUFOBJPAG (see Section 3.12.2) as the per-page service-specific routine.
12. When MMG$CREDEL returns, EXE$CREATE_BUFOBJ checks whether the buffer object contains all requested pages. If not, it reduces PMS$GL_BUFOBJ_PAGES_S0S1 and deallocates unused SPTEs.
13. If the requestor specified flag CBO$V_EXMAXLIM, EXE$CREATE_BUFOBJ sets BOD$V_NOQUOTA to indicate that system limits need not be checked.
14. If the requestor specified flag CBO$V_RETSVA, it calculates the ending system virtual address of the buffer to store in the RETADR argument.
15. It records peak page file use and virtual size statistics, and stores return information in the optional RETADR argument.
16. If the current value of PMS$GL_BUFOBJ_PAGES_S0S1 is a peak value, EXE$CREATE_BUFOBJ records it in PMS$GL_BUFOBJ_S0S1_PEAK. Similarly, if appropriate, it updates PMS$GL_BUFOBJ_PAGES_PEAK.
17. It increments BOD$L_REFCNT and stores the buffer handle in the requestor's CREBUF_HANDLE argument.
18. It restores the IPL at entry.
19. It returns to its requestor, passing any error status from LCKBUFOBJPAG.

3.12.2 [MMG_STD$]LCKBUFOBJPAG Routine

LCKBUFOBJPAG, in module SYSLKWSET, is the per-page service-specific routine for $CREATE_BUFOBJ. The $CREATE_BUFOBJ_64 system service uses its alternative entry point, MMG_STD$LCKBUFOBJPAG.

LCKBUFOBJPAG takes the following steps:

1. It tests that the process-private page is writable from the requesting access mode and, if not, returns the error status SS$_ACCVIO, which is returned to the service requestor.
2. It calculates the address of the L3PTE that maps the page and acquires the MMG spinlock, raising IPL to IPL$_MMG.
3. It tests whether the page is valid. If not, it releases the spinlock, touches the page to fault it, and resumes with step 2.
4. LCKBUFOBJPAG makes several consistency tests on the page, for example, checking that its owner mode is not more privileged than the maximized access mode, the page is a process page or a global writable page, and the page is not a PFN-mapped page. In the case of a system buffer object, the page must *not* be a process page. If any test fails, LCKBUFOBJPAG returns an appropriate error status, which is passed back to the service requestor.
5. If the page is part of a Galaxywide section, it calculates the address of the SHM_DESC structure that describes the section and increments its buffer object reference count. If the reference count transitions from 0 to 1, it also locks the section for I/O. It does not modify the PFN database record for the page because the PFN database for pages in Galaxywide shared memory is common to all Galaxy instances, and the page is not necessarily part of a buffer object on other instances. LCKBUFOBJPAG continues with step 16.
6. If the page is not part of a Galaxywide section, it tests and sets PFN$V_BUFOBJ in the physical page's PFN$L_PAGE_STATE. If the bit was already set, it increments PFN$W_BO_REFC for the buffer object page and continues with step 15.
7. Otherwise, it checks whether the page is a memory-resident global section page. If so, the page is already locked in physical memory, and it is therefore unnecessary to update physical memory use statistics. LCKBUFOBJPAG continues with step 11.
8. If the page was not already locked in memory, LCKBUFOBJPAG increments PMS$GL_BUFOBJ_PAGES.
9. It tests whether the page has already been locked into memory (a nonzero PFN$L_GBL_LCK_CNT for a global writable page or a set WSL$V_PFNLOCK for a process page) and, if so, continues with step 11.

10. It decrements PFN$GL_PHYPGCNT, the fluid page count, and checks whether the count is still high enough. If so, it continues with step 11. If not, it increments PFN$GL_PHYPGCNT, clears PFN$V_BUFOBJ, decrements PMS$GL_BUFOBJ_PAGES, and returns the error status SS$_INSFLPGS to its caller.
11. If this is the first lock of the page, it increments PFN$W_BO_REFC (to zero) and increments the page's PFN$W_REFCNT.
12. If this is a global page, it continues with step 15.
13. It modifies the PFN database record for the page table page that maps this buffer object page and increments PFN$W_BO_REFC. If this is the first buffer object page mapped by this page table page, it also sets the modify and buffer object bits in PFN$L_PAGE_STATE and increments PFN$W_REFCNT.
14. It stores an illegal address containing the process index in the buffer object page's PFN$L_PT_PFN as a troubleshooting aid. (Correct treatment of a buffer object page should never result in access of this field.)
15. It sets the modify bit in the buffer object page's PFN$L_PAGE_STATE.
16. Unless BOD$V_NOSVA is set, it initializes the SPTE that doubly maps the buffer object page with the PFN, kernel mode read and write enabled, kernel mode owner, valid address space match, modify, no-execute, fault-on-execute, and window bits set.
17. It increments BOD$L_PAGCNT to show that another page has been added to the buffer object.
18. It releases the MMG spinlock and returns.

3.12.3 $CREATE_BUFOBJ_64 System Service

The Create Buffer Object ($CREATE_BUFOBJ_64) system service procedure, EXE$CREATE_BUFOBJ_64 in module SYS_LKWSET_64, runs in kernel mode. It resembles the $CREATE_BUFOBJ system service, but all its address arguments are 64 bits. Thus it can be used to create a buffer object in P0, P1, or P2 space. By default the service double-maps the process buffer in S2 space.

Its FLAGS argument includes the bits

- CBO$V_SVA_32, to specify that the buffer be doubly mapped in S0/S1 space
- CBO$V_NOSVA, to specify that no system space mapping be created
- CBO$V_SYSBUFOBJ, to specify that the buffer be mapped only in system space

EXE$CREATE_BUFOBJ_64 is very similar to EXE$CREATE_BUFOBJ, with the following major differences:

- EXE$CREATE_BUFOBJ_64 rounds up the START_VA_64 argument to the next page boundary and rounds down LENGTH_64.

- If the requestor set the FLAGS argument CBO$V_SYSBUFOBJ, it checks that the service was requested from executive or kernel mode, returning SS$_BADPARAM if not.
- By default it adds the buffer object pages to PMS$GL_BUFOBJ_PAGES_S2 rather than PMS$GL_BUFOBJ_PAGES_S0S1. It checks the size of the buffer object against a limit set by SYSGEN parameter MAXBOBMEM, returning error status SS$_EXBUFOBJLIM if the buffer object is too large. It allocates L3PTEs that map S2 space to double-map the buffer and records their page table space address in BOD$PQ_VA_PTE. It sets BOD$V_S2_WINDOW in BOD$L_FLAGS.

  If the requestor set the FLAGS argument CBOV_SVA_32, EXECREATE_BUFOBJ_64 double-maps the buffer in S0/S1 space, as EXE$CREATE_BUFOBJ does.

  If the requestor set the FLAGS argument CBOV_NOSVA, EXECREATE_BUFOBJ_64 does not double-map the buffer into system space. It initializes BOD$PQ_VA_PTE and BOD$PQ_BASESVA to –1 and sets BOD$V_NOSVA.

  If the requestor set the FLAGS argument CBO$V_SYSBUFOBJ, it inserts the BOD at the tail of the system PCB.

3.12.4 $DELETE_BUFOBJ System Service

The Delete Buffer Object ($DELETE_BUFOBJ) system service procedure, EXE$DELETE_BUFOBJ in module SYSLKWSET, runs in kernel mode. The service is requested with the address of a buffer handle describing the buffer object to be deleted.

EXE$DELETE_BUFOBJ takes the following steps:

1. It probes accessibility of the buffer handle and in case of error returns the error status SS$_BADPARAM to its requestor.
2. It fetches the contents of the buffer handle, namely the BOD address and sequence number.
3. It acquires the MMG spinlock, raising IPL to IPL$_MMG.
4. It checks the following:
   - The BOD is actually linked into the process's BOD list at PCB$Q_BUFOBJ_LIST. In the case of a system buffer object, it checks that the BOD is linked into the system PCB BOD list.
   - The sequence number matches that in the BOD.
   - The BOD address is actually a nonpaged pool address.
   - BOD$B_TYPE contains DYN$C_BOD.
   - The requesting process's ID is the same as BOD$L_PID.

— The access mode from which the service was requested is at least as privileged as the creator of the buffer object.

If any consistency check fails, EXE$DELETE_BUFOBJ releases the MMG spinlock and returns either SS$_BADPARAM or SS$_NOPRIV.

5. It tests and sets BOD$V_DELPEN in BOD$L_FLAGS. Once BOD$V_DELPEN is set, no further I/O can be initiated to this buffer object. If the bit was already set, EXE$DELETE_BUFOBJ releases the spinlock and returns the error status SS$_BADPARAM.

6. It decrements BOD$L_REFCNT and, if the count is still positive, indicating outstanding I/O requests in progress, releases the MMG spinlock, restores the byte count quota and limit charged for the BOD, and returns.

7. If the reference count is zero, EXE$DELETE_BUFOBJ changes the state of the buffer object pages, one page at a time.

   For a buffer mapped in system space, the common case, it takes the following steps:

   a. If BOD$V_S2_WINDOW is set, indicating the buffer object is mapped in S2 space, it determines the address of the shared L3PTE that maps that buffer page. Otherwise, it indexes into the system page table window to get the address of the SPTE that maps that buffer page.

      Regardless of which system space the virtual page is in, if it is invalid, EXE$DELETE_BUFOBJ generates a fatal PAGNTRNVAL bugcheck.

   b. If the virtual page is valid, it clears the L3PTE and flushes any cached translation from the TB.

   c. It examines the physical page's PFN database record and tests whether the page is a Galaxy shared memory page. If not, it continues with step d. If so, it determines the address of the shared memory section descriptor corresponding to this page and decrements its buffer object reference count. If the count transitions to zero, it calls GLX$SHM_REG_DECREF to decrement the I/O reference count for that shared memory region.

   d. For a page that is not a Galaxy shared memory page, EXE$DELETE_BUFOBJ decrements the physical page's PFN$W_BO_REFC. If the count transitions to zero, it checks whether this a memory-resident section page. If so, it continues with step g.

      If not, it clears PFN$V_BUFOBJ in the page's PFN$L_PAGE_STATE field and decrements PMS$GL_BUFOBJ_PAGES to indicate one less physical buffer object page.

      If this is a global page, it continues with step f. If this is a process page, it calculates the address of the process-private L3PTE that maps the page and then the address of the L2PTE that maps that L3PT. It tests whether the page table page that maps the buffer object is valid and, if not, frees a working set list entry for it and makes it valid. It decrements the page table page's PFN$W_BO_REFC. If the reference count is now zero, EXE$DELETE_BUFOBJ clears PFN$V_BUFOBJ in PFN$L_PAGE_STATE for the page and decrements its PFN$W_REFCNT.

If the buffer object page is in a release pending state, EXE$DELETE_BUFOBJ calls MMG_STD$INCPTREF_64, in module PAGEFAULT, to increment the PFN$L_SHRCNT of the process page table page that mapped it. (To allow the page table page to be removed from the working set, its PFN$L_SHRCNT was decremented even though it continued to map a buffer object page in transition.) If the buffer object page is instead in an active state, EXE$DELETE_BUFOBJ tests whether the page is locked into the working set list. If so, it continues with step g; if not, with step f.

e. For a buffer object page that is a global page, EXE$DELETE_BUFOBJ tests whether the page is locked in memory (nonzero PFN$L_GBL_LCK_CNT). If so, it continues with step g.

f. It increments PFN$GL_PHYPGCNT, the system fluid page count, to show one more available page.

g. It decrements the buffer object page's PFN$W_REFCNT. If the count is now zero, the page is released to the modified page list. EXE$DELETE_BUFOBJ goes on to the next page.

When all buffer object pages have been processed, EXE$DELETE_BUFOBJ subtracts the number of buffer object pages from either PMS$GL_BUFOBJ_PAGES_S0S1 or PMS$GL_BUFOBJ_PAGES_S2 and deallocates the system space PTEs that doubly mapped the buffer object.

8. For a buffer object mapped only in process-private space, EXE$DELETE_BUFOBJ takes most of the same steps as for a buffer object mapped in system space, with the following exceptions:

   — It does not clear the L3PTE that mapped each buffer object page and therefore no TB invalidate is necessary.

   — It does not deallocate system space L3PTEs or update PMS$GL_BUFOBJ_PAGES_S*n*.

9. Regardless of whether the buffer object was mapped in system space, EXE$DELETE_BUFOBJ clears BOD$L_SEQNUM and BOD$B_TYPE to ensure invalidity of any subsequent reference to the deleted buffer object through its handle, removes the BOD from the PCB queue, and deallocates the BOD to nonpaged pool.

10. It releases the MMG spinlock, lowering IPL.

11. It restores the byte count quota and limit charged against the process for the BOD and buffer object pages.

12. It returns to its requestor.

3.13 Services That Return Information

OpenVMS provides several services that return memory-management-related information. They are described in the following sections.

3.13.1 $GET_REGION_INFO System Service

The Get Information About Specified Region ($GET_REGION_INFO) system service is requested to get information about a region within process-private address space, a region specified by ID or by virtual address. Its arguments include a function code that specifies whether a region ID or virtual address identifies the region, region identification, and a buffer in which information is returned.

The $GET_REGION_INFO system service procedure, EXE$GET_REGION_INFO in module SYS_REGIONS, runs in kernel mode. EXE$GET_REGION_INFO takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:
   a. If the FUNCTION argument is invalid, it returns the error status SS$_BADPARAM.
   b. If START_VA_64 was supplied, it checks that the argument can be expressed in the number of significant address bits for the system's page size and page table hierarchy, for example, 43 bits for a page size of 8 KB and a three-level page table. If not, it returns the error status SS$_PAGNOTINREG.
2. Depending on the function code, it locates the region of interest, based on its ID or starting virtual address. The function code can also specify that the target region is the one following the region at the starting virtual address. If the region was not found, EXE$GET_REGION_INFO returns the error status SS$_IVREGID or SS$_PAGNOTINREG, depending on the function code.
3. If the region was found, it copies information from the RDE to the requestor-specified buffer and stores the length of the information in the RETURN_LENGTH_64 argument.
4. It returns SS$_NORMAL to its requestor.

3.13.2 $GETSECI System Service

A process requests the Get Section Information ($GETSECI) system service to get information about a particular global page mapped into its address space. Its arguments include the address of a page in the global section of interest and an item list describing the information to be returned. Use of this undocumented system service is reserved to Hewlett-Packard Company.

Basically, the system service sanity checks the virtual address specified, determines the address of the GSTE that describes the corresponding global section, and returns the requested information from the GSTE to the service requestor.

The $GETSECI system service procedure, EXE$GETSECI in module SYSPARPRC, runs in kernel mode. EXE$GETSECI takes the following steps:

1. It raises IPL to 2.
2. It locates the RDE corresponding to the virtual address argument. If none is found, EXE$GETSECI returns the error status SS$_NONXPAG to its requestor.
3. If the address is within a region, but beyond the currently defined space in the region, EXE$GETSECI returns the error status SS$_LENVIO. If the address falls on a page of nonexistent address space within the defined space in the region, EXE$GETSECI returns the error status SS$_ACCVIO.
4. Otherwise, it determines the address of the L3PTE that maps the specified virtual address and faults the page table page into the process's working set list.
5. It compares the requestor's access mode to that of the page owner and, if it is less privileged, returns the error status SS$_PAGOWNVIO. The requestor can only get information about pages owned by the requesting mode and less privileged modes.
6. EXE$GETSECI determines the page type and rules out unsuitable pages:
   - If the page is a PFN-mapped page, EXE$GETSECI returns the error status SS$_PAGTYPVIO.
   - If the page is a process-private demand zero page, a process-private section page, or a page file page, it returns the error status SS$_NOTINSEC.
   - If the page is a transition or valid page, EXE$GETSECI determines the page type from the physical page's PFN database record. If the page is anything but PFNC_GLOBAL, PFNC_GBLWRT, or a Galaxywide section page, it returns the error status SS$_NOTINSEC.
7. For suitable page types, EXE$GETSECI uses the information in the PTE to get the corresponding GPTX and GSTX:
   a. For a valid or transition global page, it examines the physical page's PFN database record fields PFN$Q_PTE_INDEX, which contains the GPTX, and PFN$Q_BAK. If the page is a page file global page, EXE$GETSECI scans all the GSTEs, looking for one that includes that GPTX. If the page is not a page file global page, PFN$Q_BAK contains the global section index.
   b. For a valid Galaxywide section page, EXE$GETSECI gets the Galaxywide shared memory section ID from the physical page's PFN database record (SHM_ID$W_INDEX) and indexes the Galaxy shared memory descriptor table to get the GSTX.
   c. For an invalid global page that is not a transition page, EXE$GETSECI examines the GPTE. If the page is a page file global page, it scans all the GSTEs, looking for one that includes that GPTX. If the page is not a page file global page, the GPTE contains the GSTX.
8. EXE$GETSECI calculates the address of the GSTE.

9. If the page is a Galaxywide section page, it adds SEC$L_VPX to the GPTX value to transform it into a real index into the GPT.
10. It then processes the item list: checking access to each entry and each buffer an entry describes, validating item codes, and recording information in buffers. If an entry or buffer is inaccessible, EXE$GETSECI returns SS$_ACCVIO to its requestor. If an item code is invalid, it returns SS$_BADPARAM. Information that can be returned includes global section flags, access mode, ident, name, and relative page number of the input virtual address.

3.13.3 $FIND_GPAGE_64 System Service

A process requests the Find Mapped Global Page ($FIND_GPAGE_64) system service to determine the address at which a particular offset in a named global section is mapped in that process. Its input arguments include the offset in the global section offset, the global section name and ID, flags specifying the global section type, and access mode. Use of this undocumented system service is reserved to Hewlett-Packard Company.

Basically, the system service finds the specified global section, calculates the GPTX corresponding to the specified offset in that section, and then scans the process's page tables looking for the virtual page that corresponds to that GPTX. If it finds such a page, it calculates the corresponding virtual address and returns that to the requestor.

The $FIND_GPAGE_64 system service procedure, EXE$FIND_GPAGE_64 in module SYS_FIND_GPAGE_64, runs in kernel mode. EXE$FIND_GPAGE_64 takes the following steps:

1. In addition to making the checks described in Section 3.1.2, it validates its arguments as follows:
   a. It checks that the SECTION_OFFSET_64 argument and, if present, optional arguments START_VA_64 and LENGTH_64 arguments are multiples of the size of a page, returning the error status SS$_OFF_NOTPAGALGN, SS$_VA_NOTPAGALGN, or SS$_LEN_NOTPAGMULT if not.
   b. It checks that the START_VA_64 argument is within process-private space, returning the error status SS$_NOSUCHPAG if not.
   c. It minimizes the LENGTH_64 argument with the difference between START_VA_64 and the end of process-private space.
2. It locks the GSD mutex for read access, raising IPL to 2.
3. It calls MMG_STD$GSDSCAN, in module SYSDGBLSC, to find the GSD, if any, that corresponds to the GS_NAME_64 and IDENT_64 arguments. Section 3.7.1 describes MMG_STD$GSDSCAN. If none is found, it unlocks the mutex and returns the error status SS$_NOSUCHSEC.
4. Otherwise, EXE$FIND_GPAGE_64 performs an additional sanity check on the global section ident: it confirms that the ident is positive and, if not, unlocks the mutex and returns the error status SS$_IVSECIDCTL to its requestor.

5. Examining the GSD flags, EXE$FIND_GPAGE_64 tests whether the section is a PFN-mapped one. If so, it unlocks the mutex and returns the error status SS$_GBLSEC_MISMATCH.
6. It compares the section access mode with the requestor's mode to determine if the system service requestor is allowed to map the section. If not, it unlocks the mutex and returns the error status SS$_NOPRIV.
7. It calculates the address of the GSTE.
8. It calls MMG_STD$CHKPRO_AUDIT to check access to the file. If access is not allowed, it unlocks the mutex and returns an error status to its requestor.
9. It checks whether SECTION_OFFSET_64 is within the global section and, if not, unlocks the mutex and returns the error status SS$_OFFSET_TOO_BIG.
10. It calculates the GPTX corresponding to the sum of the section offset and the GPTX for the first page of the section.
11. It unlocks the mutex.
12. EXE$FIND_GPAGE_64 establishes an exception handler for any subsequent access violations and scans the process's page tables for a match. It loops through the L1PT and, for each nonzero L1PTE, scans the corresponding L2PT. For each nonzero L2PTE, it scans the corresponding L3PT looking for a match. It skips any L3PTE that is no-access, PFN-mapped, or a window page.

    If it finds an L3PTE whose valid and type 1 bits are clear, whose type 0 bit is set, and whose high 32 bits match the GPTX, it has found the matching page. If the page's owner access mode is more privileged than the maximized access mode, it returns error status SS$_NOPRIV. Otherwise, it calculates the virtual address that corresponds to that L3PTE and returns it and the status SS$_NORMAL to its requestor.

    For each valid or transition L3PTE it finds, it takes the following steps:

    a. It acquires the MMG spinlock, raising IPL to IPL$_MMG.
    b. It confirms that the L3PTE is still a valid or transition page. If not, it releases the spinlock, lowering IPL, and goes on to the next L3PTE.
    c. In the case of a valid or transition L3PTE, any GPTX information is in the PFN database rather than in the L3PTE. EXE$FIND_GPAGE_64 gets the GPTX from PFN$Q_PTE_INDEX in the PFN database record for the PFN mapped by that L3PTE.
    d. If the page is a Galaxywide shared memory page (PFN$V_SHARED is set in PFN$L_PAGE_STATE), the GPTX in the L3PTE is a section-relative index; EXE$FIND_GPAGE_64 adds the GPTX in the shared memory section descriptor to it.
    e. It releases the MMG spinlock, lowering IPL.

f. If the page is not a shared memory page, it checks whether the page type is PFN$C_GLOBAL or PFN$C_GBLWRT. If not, it goes on to the next L3PTE.

g. It compares the GPTX to the GPTX of interest. If that matches, it has found the matching page. If the page's owner access mode is more privileged than the maximized access mode, it returns the error status SS$_PAGOWNVIO. Otherwise, it calculates the virtual address that corresponds to that L3PTE and returns it and the status SS$_NORMAL to its requestor.

3.13.4 $GET_VA_RAD_INFOW System Service

The $GET_VA_RAD_INFOW system service is requested to get per-RAD page counts for a specified virtual address range. Its arguments include the starting virtual address, size of the range in bytes, and the address and size of a buffer in which the information is returned. A size of –1 means the virtual range ends at the end of process-private address space. Use of this system service is reserved to Hewlett-Packard Company. Any other use is unsupported.

The $GET_VA_RAD_INFOW system service procedure, EXE$GET_VA_RAD_INFOW in module PTECHECK, runs in kernel mode. It takes the following steps:

1. It validates its arguments, checking that the virtual address range is entirely within process-private address space, and returns the error status SS$_BADPARAM if not. It confirms that the return buffer is large enough for a longword array of page counts with an element for each RAD on the system, returning SS$_BADPARAM if not.
2. It checks that the requestor has write access to the buffer, returning SS$_ACCVIO if not. Otherwise, it clears the buffer.
3. It raises IPL to 2.
4. It determines the RDE associated with the beginning part of the range.
5. It determines the lesser of the last virtual page in that region and the ending virtual address.
6. It scans the L3PTEs that map that subrange. If the PTE maps a valid or transition page, it determines the RAD based on the PFN and increments that RAD's counter in the return buffer.
7. If the ending address of the subrange is less than the address of the entire virtual range, it determines the RDE associated with the next part of the range and continues with step 5.
8. Otherwise, it returns to its requestor.

3.14 $SETSWM System Service

A security persona with PSWAPM privilege can lock and unlock its process into the balance set by requesting the $SETSWM system service. A process locked into the balance set cannot be outswapped.

The Set Process Swap Mode ($SETSWM) system service procedure, EXE$SETSWM in module SYSSETMOD, runs in kernel mode. EXE$SETSWM checks that the security persona has privilege and simply sets (or clears) the PCB$V_PSWAPM bit in PCB$L_STS, the status longword in the software PCB. While setting or clearing the bit, EXE$SETSWM holds the SCHED spinlock.

When the swapper is searching for suitable outswap candidates, a process whose PCB$V_PSWAPM bit is set is passed over.

3.15 Set Page Protection System Services

A process can alter the protection of a set of pages in its address space by requesting either the $SETPRT or the $SETPRT_64 system service.

3.15.1 $SETPRT System Service

The Set Protection on Pages ($SETPRT) system service procedure, EXE$SETPRT in module SYSSETPRT, runs in kernel mode. It takes the following steps:

1. It transforms the contents of the PROT argument from a VAX protection encoding to the analogous Alpha protection bits. (VAX encoding was used for ease in porting VAX code to Alpha.)
2. It creates and initializes scratch space on the stack.
3. It tests the accessibility of the INADR argument and maximizes the ACMODE argument with the mode of the service requestor.
4. It raises IPL to 2 to block AST delivery.
5. It determines the address of the RDE corresponding to the INADR argument.
6. EXE$SETPRT calls MMG$CREDEL, specifying SETPRTPAG_64 (see Section 3.15.2) as the per-page service-specific routine.
7. If necessary, EXE$SETPRT transforms error statuses returned by SETPRTPAG_64 into more traditional ones (SS$_PAGTYPVIO into SS$_NOPRIV and SS$_NOSUCHPAG into SS$_ACCVIO).
8. If the PRVPRT argument was specified, EXE$SETPRT tests its accessibility and returns in it the most recent previous page protection returned from SETPRTPAG_64.
9. It restores the IPL at entry and returns to its requestor.

In general, the operation of this service is straightforward. However, its actions have one interesting side effect. If a section page for a read-only section has its protection set to writable, the copy-on-reference bit is set. This set bit forces the page to have its backing store address changed to the page file when the page is faulted, preventing a later attempt to write the modified section pages back to a file to which the process may be denied write access.

3.15.2 [MMG_STD$]SETPRTPAG_64 Routine

SETPRTPAG_64, with its alternative entry point MMG_STD$SETPRTPAG_64, in module SYSSETPRT, is the per-page service-specific routine for the $SETPRT and $SETPRT_64 system services.

It takes the following steps:

1. It calls MMG_STD$PTEREF_64, in module SVAPTE, to fault in the page table page containing the PTE that maps the page whose protection is to be changed. MMG_STD$PTEREF_64 takes the following steps:

    a. It confirms that the virtual address is within the current space of the region, returning SS$_LENVIO if not.

    b. It gets the address of the L3PTE that maps the specified virtual address.

    c. It acquires the MMG spinlock, raising IPL, and then checks whether all the PTEs involved in translating the specified virtual address are valid. If so, it returns the address of the PTE to its caller with the MMG spinlock held, ensuring that the state of the L3PT cannot be changed asynchronously.

    d. If some PTEs are invalid, the routine records the address of the L3PTE in PCB$Q_KEEP_IN_WS and the starting and ending addresses of the L2PT that maps it in PHD$PQ_PT_NO_DELETE1 and PHD$PQ_PT_NO_DELETE2. Recording these addresses enables the L2PT to be faulted into the working set list and temporarily locked there, temporarily preventing any of the L3PTs it maps from being deleted.

    e. MMG_STD$PTEREF_64 releases the MMG spinlock, lowering IPL to 2.

    f. It faults the L2PT, if necessary. If the L2PTE that maps the L3PT is null, MMG_STD$PTEREF_64 returns SS$_ACCVIO. Otherwise, it then faults the L3PT.

    g. It reacquires the MMG spinlock, clears PHD$PQ_PT_NO_DELETE1 and PHD$PQ_PT_NO_DELETE2, and stores a –1 in PCB$Q_KEEP_IN_WS. With the L3PT faulted into the working set list and the MMG spinlock held, the temporary measures are no longer necessary.

    h. It returns the address of the PTE to its caller, with the MMG spinlock still held.

2. If MMG_STD$PTEREF_64 returned an error status, SETPRTPAG_64 returns it to its caller. Otherwise, SETPRTPAG_64 tests whether the L3PTE is zero, indicating a null page, and, if so, releases the MMG spinlock and returns the error status SS$_NOSUCHPAG, which is transformed before being returned to a $SETPRT requestor.
3. SETPRTPAG_64 compares the requestor's access mode in MMG$L_ACCESS_MODE with that of the page owner. If the access mode is insufficiently privileged, the routine releases the MMG spinlock and returns the error status SS$_PAGOWNVIO.
4. Otherwise, it determines the type of the virtual page, based on the valid and type bits in the L3PTE that maps it.

   — If the page is a transition or demand zero page that is to become read-only, SETPRTPAG_64 releases the MMG spinlock, lowers IPL, touches the page to make it valid, and continues at step 1.

   — If the page is a demand zero page and will remain writable or is a page file page, SETPRTPAG_64 continues with step 5.

   — If the page is a process-private section page and the protection change would make a writable page read-only, SETPRTPAG_64 continues with step 5.

     If the page is already writable from some mode or is a copy-on-reference page, SETPRTPAG_64 continues with step 5.

     If the protection change would make a read-only page writable, SETPRTPAG_64 must change the page to be a copy-on-reference page: it charges the page against the process's job page file quota and changes the page's backing store to a page file. It continues with step 5, also setting the copy-on-reference bit in the L3PTE. An inability to charge the page against quota results in an error status return.

   — If the page is valid, SETPRTPAG_64 checks that it is not a PFN-mapped page and that it is a process page. If either is false, it returns the error status SS$_PAGTYPVIO.

     If the page is a valid process page and the protection change would make a writable page read-only, SETPRTPAG_64 continues with step 5, also clearing the fault-on-write bit if it was set.

     If the page is a valid process page and the protection change does not make a read-only page writable or if the page already has page file backing store, SETPRTPAG_64 continues with step 5.

     Otherwise, it changes the PFN$Q_BAK field for the physical page to a page file backing store form and decrements the section's reference count. It completes changing the page to a copy-on-reference page, taking the same steps as for a process-private section page.

— If the page is a global section page, SETPRTPAG_64 determines the page type from the global PTE. If it contains anything but a global section index for a copy-on-reference page, SETPRTPAG_64 returns the error status SS$_PAGTYPVIO. Otherwise, it continues.

5. It modifies the L3PTE to change the page's protection and, if the page is valid, invalidates any cached TB entry for the page.
6. It releases the MMG spinlock, restoring the previous IPL of 2, and returns to its caller.

3.15.3 $SETPRT_64 System Service

The Set Protection on Pages ($SETPRT_64) system service procedure, EXE$SETPRT_64 in module SYS_SETPRT_64, runs in kernel mode. It resembles the $SETPRT system service, but all its address arguments are 64 bits. Thus it can be used to change the protection of P0, P1, or P2 space pages.

It takes the following steps:

1. It validates its arguments.
2. It transforms the contents of the PROT argument from a VAX protection encoding to the analogous Alpha protection bits. (VAX encoding is used for consistency with the 32-bit service.)
3. It maximizes the ACMODE argument with the mode of the service requestor.
4. It rounds down the START_VA_64 argument to a page boundary and rounds up the LENGTH_64 to an integral number of pages that includes the user-specified starting and ending addresses.
5. It raises IPL to 2 to disable AST delivery.
6. It determines the address of the RDE corresponding to the START_VA_64 argument. If the address is not within a region, it returns the error status SS$_PAGNOTINREG or SS$_NOT_PROCESS_VA, whichever is appropriate.
7. It loops, calling MMG_STD$SETPRTPAG_64 (see Section 3.15.2) until the routine returns an error status or all pages are done.
8. If the protection on any page was changed successfully, EXE$SETPRT_64 converts the old protection into VAX encoding and records it in the RETURN_PROT_64 argument if it is accessible.
9. It returns the rounded-down starting address and rounded-up length to its requestor.

3.16 Set Fault System Services

A process can set the no-execute characteristic for each of a group of pages in its address space by requesting the undocumented $SETFLT or $SETFLT_64 system service. Use of these services is reserved to Hewlett-Packard Company. Any other use is unsupported.

3.16.1 $SETFLT System Service

The Set Fault Characteristic ($SETFLT) system service procedure, EXE$SETFLT in module SYSSETPRT, runs in kernel mode. It takes the following steps:

1. It creates and initializes scratch space on the stack.
2. It performs several consistency checks on the arguments, returning the error status SS$_BADPARAM if the FLAGS argument specifies anything other than no-execute or the error status SS$_ACCVIO if other arguments are inaccessible.
3. It maximizes the ACMODE argument with the mode of the service requestor.
4. It raises IPL to 2 to block AST delivery.
5. It determines the address of the RDE corresponding to the INADR argument.
6. EXE$SETFLT calls MMG$CREDEL (see Section 3.1.1), specifying SETFLTPAG_64 (see Section 3.16.2) as the per-page service-specific routine.
7. If necessary, EXE$SETFLT transforms error statuses returned by SETPRTPAG_64 into more traditional ones (SS$_PAGTYPVIO into SS$_NOPRIV and SS$_NOSUCHPAG into SS$_ACCVIO).
8. If MMG$CREDEL returns successfully, EXE$SETFLT executes an instruction memory barrier to flush any instructions that might have been prefetched from the pages whose fault-on-execute bit has just been set.
9. It restores the previous IPL and returns to its requestor.

3.16.2 [MMG_STD$]SETFLTPAG_64 Routine

SETFLTPAG_64, with its alternative entry point MMG_STD$SETFLTPAG_64, in module SYSSETPRT, is the per-page service-specific routine for the $SETFLT and $SETFLT_64 system services.

It takes the following steps:

1. It calls MMG_STD$PTEREF_64 (see Section 3.15.2) to fault in the page table page containing the PTE that maps the page whose no-execute characteristic is to be set. MMG_STD$PTEREF_64 returns the address of the PTE to SETFLTPAG_64 with the MMG spinlock held.

2. If MMG_STD$PTEREF_64 returned an error status, SETFLTPAG_64 returns it to its caller. Otherwise, SETFLTPAG_64 tests whether the L3PTE is zero, indicating a null page, and, if so, releases the MMG spinlock and returns the error status SS$_NOSUCHPAG, which is transformed before being returned to a $SETFLT requestor.
3. SETFLTPAG_64 compares the requestor's access mode with that of the page owner. If the access mode is insufficiently privileged, it releases the MMG spinlock and returns the error status SS$_PAGOWNVIO, which is passed back to the system service requestor.
4. Otherwise, it determines the type of the virtual page, based on the valid and type bits in the L3PTE that maps it.
   - If the page is a transition or demand zero page, SETFLTPAG_64 releases the MMG spinlock, lowers IPL, touches the page to make it valid, and continues at step 1.
   - If the page is valid, SETFLTPAG_64 checks that it is a process or global page and that it is not a PFN-mapped page, returning the error status SS$_PAGTYPVIO if either is false. If both are true, it sets the no-execute and fault-on-execute bits in the L3PTE.
   - If the page is a global page, a page file page, or a section page, SETFLTPAG_64 sets the no-execute bit in the L3PTE.
5. It invalidates any possible TB entry for the page; releases the MMG spinlock, lowering IPL; and returns.

3.16.3 $SETFLT_64 System Service

The Set Fault Characteristic ($SETFLT_64) system service procedure, EXE$SETFLT_64 in module SYS_SETPRT_64, runs in kernel mode. It resembles the $SETFLT system service, but all its address arguments are 64 bits. Thus it can be used to set the no-execute characteristic for P0, P1, or P2 space pages.

Its control flow resembles that of EXE$SETPRT_64 (see Section 3.15.3), with the exception that the per-page routine it calls is MMG_STD$SETFLTPAG_64 (see Section 3.16.2).

3.17 $COPY_FOR_PAGE System Service

A process can read data from a page whose fault-on-read bit is set by requesting the undocumented Copy Fault on Read Page ($COPY_FOR_PAGE) system service. The service is requested with three arguments: the number of bytes to be copied, the source virtual address, and the destination virtual address. As described in Chapter 1, the executive sets the fault-on-read bit in the SPTEs mapping the granularity hint region that contains executive and other installed resident images' code sections. The protection on these pages permits user access so that instructions in mode of caller system services and images installed resident can be executed by any access mode. Data fetches, in contrast, are blocked by the fault-on-read bit.

The System Dump Analyzer (SDA) utility and debuggers use this service when they are requested to display instructions in system space. This service provides the ability to fetch data from system pages set fault-on-read for the few instances in which it is required. Use of this service is reserved to Hewlett-Packard Company. Any other use is unsupported.

The $COPY_FOR_PAGE system service procedure, EXE$COPY_FOR_PAGE in module COPY_FOR_PAGE, runs in kernel mode. It takes the following steps:

1. It confirms that the data to be read is in system space, returning the error status SS$_BADPARAM if not.
2. It probes the protection on the system page to confirm that the access mode from which the service was requested is allowed to read the page and, if not, returns the error status SS$_ACCVIO.
3. It probes the output buffer page to confirm that the requesting access mode has write access and, if not, returns the error status SS$_ACCVIO.
4. EXE$COPY_FOR_PAGE examines the SPTE containing the start of the data.
   — If the page is invalid but the fault-on-read bit is set, the SPTE is inconsistent and EXE$COPY_FOR_PAGE generates the fatal bugcheck INCONMMGST.

   — If the page is valid and the fault-on-read bit is not set, EXE$COPY_FOR_PAGE simply copies the data to the requestor's output buffer.

   — If the page is valid and the fault-on-read bit is set, EXE$COPY_FOR_PAGE acquires the MMG spinlock, raising IPL, and temporarily double-maps the physical page or pages containing the data. The temporary mapping permits kernel mode read access and has the fault-on-read bit clear. The alternative to the double mapping is temporarily clearing the fault-on-read bit in the original SPTE. That alternative would not only make it possible for other threads of execution to fetch data from the page but would also require clearing and then resetting the bit in each SPTE that maps any page within the granularity hint region.

   EXE$COPY_FOR_PAGE releases the spinlock, lowering IPL. Using the temporary mapping of the physical page, it copies the data to the requestor's output buffer. It reacquires the spinlock to unmap the page or pages and releases the spinlock.
5. It returns to its requestor.

3.18 Relevant Source Modules

Source modules described in this chapter include

[LIB]MMGDEF.SDL
[LIB_H]MMG_FUNCTIONS.H
[LIB_H]MMG_ROUTINES.H
[SYS]COPY_FOR_PAGE.B32
[SYS]PHDUTL.MAR
[SYS]PTECHECK.C
[SYS]SVAPTE.MAR
[SYS]SYS_CREDEL_64.C
[SYS]SYS_CRMPSC_64.C
[SYS]SYS_GBLSEC_64.C
[SYS]SYS_GDZRO_64.C
[SYS]SYS_GPFN_64.C
[SYS]SYS_LKWSET_64.C
[SYS]SYS_REGIONS.C
[SYS]SYS_SETPRT_64.C
[SYS]SYSADJSTK.MAR
[SYS]SYSCREDEL.MAR
[SYS]SYSCRMPSC.MAR
[SYS]SYSDGBLSC.MAR
[SYS]SYSLKWSET.MAR
[SYS]SYSSETMOD.MAR
[SYS]SYSSETPRT.MAR

Chapter 4

Paging Dynamics

I consider that a man's brain originally is like a little empty attic, and you have to stock it with such furniture as you choose. . . . Now, the skillful workman is very careful indeed as to what he takes into his brain-attic. He will have nothing but the tools which may help him in doing his work, but of these he has a large assortment, and all in the most perfect order. It is a mistake to think that that little room has elastic walls and can distend to any extent. Depend upon it, there comes a time when for every addition of knowledge you forget something that you knew before. It is of highest importance, therefore, not to have useless facts elbowing out the useful ones.

Sir Arthur Conan Doyle, *A Study in Scarlet*

This chapter's subject is paging dynamics, the movement of pages of code and data between memory and mass storage. Specifically, it describes the transitions a page makes as it is faulted into and out of a working set list, and as it moves between its backing store and memory.

Section 4.9 describes the $FAULT_PAGE system service and its effect on page fault handling. The service enables an application to fault a set of pages prior to their use.

The chapter also discusses modified page writing, the allocation and use of page files, and the operation of the $UPDSEC and $UPDSEC_64 system services.

4.1 Overview

A typical virtual page begins life as a demand zero page or as a number of blocks in a section file on a mass storage medium. Commonly, a virtual page comes from an image. A process initiates execution of the image by requesting the Image Activate ($IMGACT) system service, better known as the image activator.

The image activator, described in detail in Chapter *Image Activation and Exit*, maps the entire image into the process's address space, using the memory management system services described in Chapter 3. It initializes data structures such as process section table entries (PSTEs) and page table entries (PTEs) to associate blocks of the image file with the pages they are to occupy. Chapter 2 discusses the memory management data structures.

Paging Dynamics

When an image begins to execute, few of its pages have been read into memory from the image file, and most of the level 3 page table entries (L3PTEs) that map the image have a clear valid bit. (The image activator did access some pages to relocate and fix up address references within the image.) When an image page whose valid bit is clear is referenced, a translation-not-valid exception results.

The processor changes access mode to kernel and switches to the kernel stack. It dispatches to the translation-not-valid exception service routine, more commonly known as the page fault handler.

The page fault handler examines the memory management data structures to determine what kind of virtual page this is and takes appropriate actions:

- For example, in the case of a demand zero page, it finds an available entry in the process's working set list, allocates a page of zeroed memory, and stores its page frame number (PFN) in the L3PTE with a set valid bit. It dismisses the exception. The process reexecutes the instruction whose attempted execution caused the page fault. This time, with the L3PTE valid bit set, the processor translates the virtual address to a physical address and execution continues.
- In the case of a virtual page in a mass storage file, the page fault handler determines which blocks contain the virtual page that triggered the fault, finds an available entry in the process's working set list, allocates a physical page of memory from the free page list, stores its PFN in the L3PTE with a clear valid bit, and requests an I/O operation to read those blocks into the allocated page. It places the kernel thread into a page fault wait state.

  When the I/O completes, I/O postprocessing code sets the valid bit in the L3PTE and makes the kernel thread computable. When the kernel thread is placed back into execution, it reexecutes the instruction whose attempted execution caused the page fault. This time, with the L3PTE valid bit set, the processor translates the virtual address to a physical address and execution continues.

Although many steps in page fault handling are common to most types of page, some depend on page type and state. Section 4.2 describes the common steps in page fault handling and serves as a framework for details of type- and state-specific processing described in subsequent sections.

Faulted in, a page remains valid and in the working set until removed. Reasons for removal include the following:

- Room is required for another page (see Chapter 5).
- The Purge Working Set ($PURGWS or $PURGE_WS) system service removes it (see Chapter 5).
- Swapper trimming removes it (see Chapter 6).
- Proactive memory reclamation removes it (see Chapter 5).
- Working set limit adjustment removes it (see Chapter 5).

Removed from the working set list, the page is inserted into the modified page list, if it has been modified; otherwise, it is inserted into the free page list. Sometime later, the swapper, in response to insufficient free pages or an excess of modified pages, writes modified pages to their backing store, typically a page file. It then inserts them into the free page list. Acting in this capacity, the swapper is called the modified page writer.

While the page is on the free or modified page list, it is essentially cached; the page fault handler can resolve a fault for it by simply updating the memory management data structures and placing the page back into the process's working set list. Such a page fault requires no I/O and is sometimes referred to as a soft page fault or a soft fault.

This chapter shows how the page fault handler manipulates the various memory management data structures in response to faults for different types of virtual page. It presents page fault handler action largely in terms of modifications to data structures and state transitions. It also describes the transitions that a virtual page makes when it is removed from a working set list.

Section 4.3 discusses the transitions of different kinds of process page. Section 4.5 covers the transitions of global pages. Sections 4.6 and 4.7 describe the transitions of system space pages and global page table pages.

4.2 Page Fault Handling

As described in Chapter 1, the translation buffer (TB) miss privileged architecture library (PALcode) routine generates a page fault exception when it detects an attempted reference to a virtual address whose L3PTE valid bit is clear. It also generates a page fault exception if either the level 1 page table entry (L1PTE) or the level 2 page table entry (L2PTE) involved in the translation is not valid but otherwise allows kernel mode read access.

The page fault handler is entered in response to a translation-not-valid fault, described in detail in Chapter 1. When it is entered, the stack contains the standard exception stack frame, pictured in Chapter *Interrupts, Exceptions, and Machine Checks*. The page fault is described by the contents of the following registers:

- R4—The fault virtual address
- R5—One of the following values:

  80000000 00000000_{16} for a write data fault
  00000000 00000000_{16} for a read data fault
  00000000 00000001_{16} for a read instruction fault

The page fault handler is implemented in a combination of MACRO-64 assembly language and MACRO-32:

- SCH$PAGEFAULT, in the MACRO-64 assembly language module SCHEDULER

- MMG$PAGEFAULT, in the MACRO-32 module PAGEFAULT

Assembly language is required to save all the scratch registers so they can be restored when the exception is dismissed. Forming the canonical kernel stack (see Chapter *Scheduling*) in case the kernel thread must be placed into a wait state also requires assembly language.

4.2.1 Common Steps in Page Fault Handling

Figure 4.1 summarizes the main steps in handling a typical fault for a page on a mass storage medium. The numbers in the figure are keyed to the explanations that follow:

❶ Entered first, SCH$PAGEFAULT saves the scratch registers on the stack and calls MMG$PAGEFAULT.

❷ MMG$PAGEFAULT checks the interrupt priority level (IPL) at which the page fault occurred. If the IPL is higher than 2, it generates the fatal PGFIPLHI bugcheck. Page faults above IPL 2 are not allowed for the following reasons:

— Code executes at an elevated IPL to perform a series of synchronized instructions. If a page fault occurs, the faulting kernel thread might be removed from execution, allowing another kernel thread to execute the same routine or access the same protected data structure. The alternative, looping in kernel thread context at elevated IPL until the page fault I/O completes, would reduce system performance and responsiveness.

Moreover, any loop at IPL 4 or above would block the I/O postprocessing necessary for page fault resolution. On a uniprocessor system, a loop above IPL 2 blocks swapper execution and would result in a deadlock if the free page list were empty and the page fault required allocation of a page of memory.

— When the system is executing at an IPL higher than 2, it may be running in system context. MMG$PAGEFAULT and related routines perform operations that require process context.

❸ MMG$PAGEFAULT acquires the MMG spinlock, raising IPL to IPL$_MMG. It makes an initial determination of what kind of working set list entry (WSLE) and page type this page will be, based on the address range in which the faulting virtual address falls, as shown in Figure 4.2. Later, it will distinguish among process pages that are process-private, global read-only, and global writable.

Note that global page table pages no longer page and thus do not appear in a working set list, but the page type bits are in the PFN database records for pages occupied by global page table pages. Historically, process-private page tables were part of the process header (PHD), and all the PHD pages had a page type of PFN$C_PPGTBL. Now that process-private page tables are mapped in page table space along with system space page tables, pages from the two address regions (the PHD pages from system space and the process-private page tables from page table space) have a type of PFN$C_PPGTBL.

Figure 4.1 Main Steps in Faulting a Page from a Mass Storage Medium

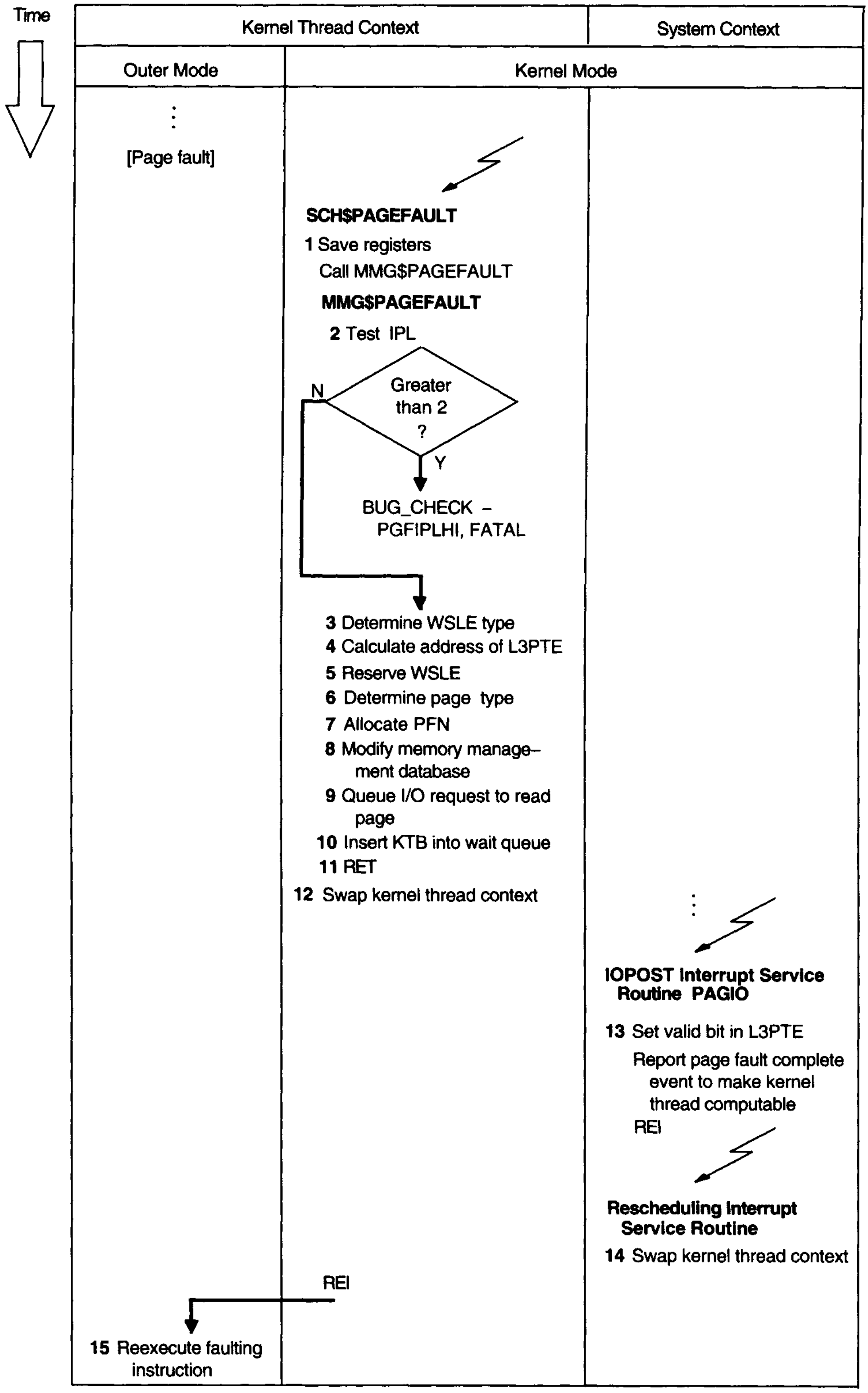
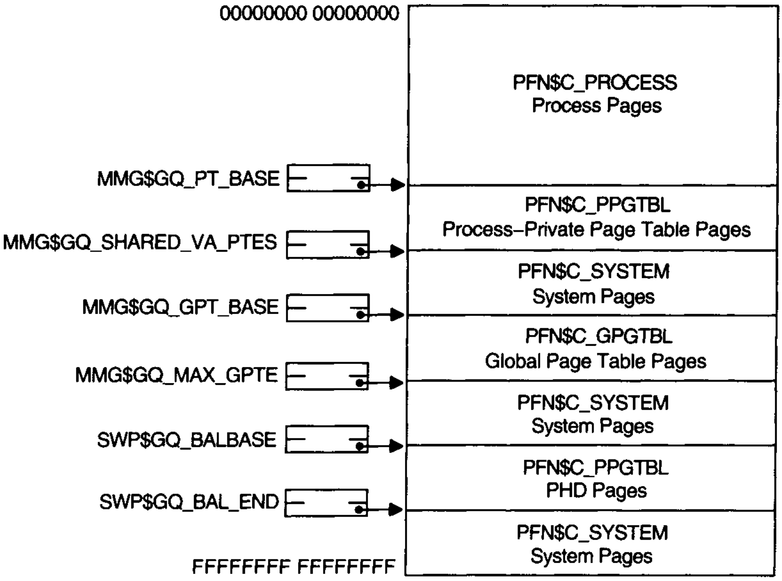

Time
Kernel Thread Context
System Context
Outer Mode
Kernel Mode
[Page fault]
SCH$PAGEFAULT
1 Save registers
Call MMG$PAGEFAULT
MMG$PAGEFAULT
2 Test IPL
Greater than 2 ?
N
Y
BUG_CHECK - PGFIPLHI, FATAL
3 Determine WSLE type
4 Calculate address of L3PTE
5 Reserve WSLE
6 Determine page type
7 Allocate PFN
8 Modify memory management database
9 Queue I/O request to read page
10 Insert KTB into wait queue
11 RET
12 Swap kernel thread context
IOPOST Interrupt Service Routine PAGIO
13 Set valid bit in L3PTE
Report page fault complete event to make kernel thread computable
REI
Rescheduling Interrupt Service Routine
14 Swap kernel thread context
REI
15 Reexecute faulting instruction

Figure 4.2 Page Types



❹ Indexing the process's page table space with the level fields from the faulting virtual address, MMG$PAGEFAULT calculates the addresses of the L1PTE, L2PTE, and L3PTE that map the page (see Figures 1.3 and 1.8).

Before examining the L3PTE, it determines whether the L1PTE and L2PTE that map the page table page containing the L3PTE are themselves valid. If the L1PTE is not valid, MMG$PAGEFAULT transforms the fault into one for the L2PT. If the L1PTE is valid but the L2PTE is not, it transforms the fault into one for the L3PT. These checks avoid the necessity of making the page fault handler recursive.

After the page table page has been faulted in, its PTE made valid, and the exception dismissed, the instruction that caused the original fault will reexecute and refault. If none of the PTEs are valid, there could be three page faults: one for the L2PT, one for the L3PT, and one for the data page.

❺ Depending on WSLE type, it calls MMG$FREWSLE, in module PAGEFAULT, to find room in the working set list for a new page, possibly by removing a page from it (see step 4 in Section 4.3.1). A typical process or process page table page is described by a WSLE in the process's working set list. Pages from memory-resident and Galaxywide global sections, however, are not described by WSLEs. A typical system pageable page is described by a WSLE in the system working set list. Global page table pages and page tables that map system space, however, are not described by WSLEs.

If MMG$FREWSLE returns an error status indicating that a free WSLE is not currently available, MMG$PAGEFAULT acquires the SCHED spinlock, releases the MMG spinlock, inserts the kernel thread's kernel thread block (KTB) into the appropriate resource wait queue, loads the status SS$_WAIT_CALLERS_MODE into R0, and continues with step 11.

If a free WSLE is available, MMG$PAGEFAULT retests the validity of the L1PTE and L2PTE mapping the page table page (one of whose L3PTEs maps the virtual address). This is done in case MMG$FREWSLE has removed the L2PT or L3PT from the working set. If either is no longer valid, MMG$PAGEFAULT transforms the fault into one for that page table page. After the page table page has been faulted in, its PTE made valid, and the exception dismissed, the instruction that caused the original fault will reexecute and refault, and the page fault handler will fault in the process page.

❻ It determines the type of page from the PTE contents. Its subsequent actions depend on the nature of the invalid page. Figure 2.12 shows the different forms of invalid L3PTE, and Chapter 3 describes how most of them are initialized in response to various system service requests.

❼ If necessary, MMG$PAGEFAULT allocates a physical page of memory. (If the virtual page is already in memory, for example, occupying a physical page on the free page list, this step is unnecessary.)

If a page of memory is not currently available, MMG$PAGEFAULT acquires the SCHED spinlock, releases the MMG spinlock, inserts the KTB into the free page wait queue, loads the status SS$_WAIT_CALLERS_MODE into R0, and continues with step 11.

❽ MMG$PAGEFAULT updates the memory management data structures.

❾ If the page does not need to be read, perhaps because it is a demand zero page or a page faulted from the free page list, MMG$PAGEFAULT releases the MMG spinlock, loads the status SS$_NORMAL into R0, and continues with step 11.

If the page must be read in from a mass storage device, MMG$PAGEFAULT builds an I/O request packet (see Section 4.14) that describes the read to be done, releases the MMG spinlock, and queues the request to the driver.

❿ MMG$PAGEFAULT acquires the SCHED spinlock. Before placing the kernel thread into a page fault wait state, it tests whether the faulted page is still invalid. On a symmetric multiprocessing (SMP) system, where MMG$PAGEFAULT is running on one processor, concurrent processing and completion of the I/O request on another may have already made the page valid. If the page is valid, MMG$PAGEFAULT releases the SCHED spinlock, loads the status SS$_NORMAL into R0, and continues with step 11.

If the page is still invalid, it checks whether the page was faulted by user mode code running in a multithreaded process. If so, it checks whether an upcall should be made to the thread manager and, if so, returns with additional status to

SCH$PAGEFAULT (see Chapter *Kernel Threads* for a description of upcalls and user mode thread management).

If the page is still invalid, but no upcall is necessary, it inserts the KTB into the page fault wait queue and loads the status SS$_WAIT_CALLERS_MODE into R0.

11 MMG$PAGEFAULT returns to SCH$PAGEFAULT.

12 SCH$PAGEFAULT's actions depend on the status from MMG$PAGEFAULT:

— If MMG$PAGEFAULT returned the status SS$_NORMAL, indicating that page fault handling is complete, SCH$PAGEFAULT restores the saved scratch registers and executes a CALL_PAL REI instruction to dismiss the page fault.

— If MMG$PAGEFAULT returned the status SS$_WAIT_CALLERS_MODE, indicating that the kernel thread must wait, SCH$PAGEFAULT takes the following actions:

a. It updates several systemwide data cells to reflect that this kernel thread is no longer current.

b. It selects a computable resident kernel thread with whose hardware context that of the waiting kernel thread can be swapped. If none is available, it will swap to the system hardware context.

c. It saves the nonscratch integer registers on the stack and, if the kernel thread is using floating-point arithmetic, the floating-point registers in the floating-point execution data block (FRED) in the PHD.

d. It swaps kernel thread context.

e. Running in the new kernel thread's context, it releases the SCHED spinlock, restores the new kernel thread's hardware context, and reenters it by executing the instruction CALL_PAL REI.

— If MMG$PAGEFAULT returned a status indicating that an upcall should be made, SCH$PAGEFAULT copies the page fault exception frame and some additional information about the page fault to the user mode stack. It modifies the saved program counter in the exception stack frame on the kernel stack and restores the saved scratch registers. It executes a CALL_PAL REI instruction to return to user mode and pass control to SCH$PAGEFAULT_UPCALL_JKT, in module SCHEDULER (see Chapter *Kernel Threads*).

13 Page read completion occurs as part of I/O postprocessing (see Chapter *I/O System Services*) and runs in system context. The I/O postprocessing routine PAGIO, in module IOCIOPOST, sets the valid bit in the L3PTE. If the kernel thread was placed into a page fault wait, it reports the scheduling event page fault completion for the kernel thread to make it computable. Otherwise, if an upcall to a user mode thread manager was made, it queues a user mode asynchronous system trap (AST) to the process to notify the thread manager of the page fault completion. PAGIO's actions are described in more detail in Section 4.11.

When the event is reported, if the process is resident and the kernel thread's priority is sufficiently high so that it should preempt, a rescheduling interrupt is requested. For simplicity, Figure 4.1 shows this step as occurring in system context, although it is more likely to occur in the context of whatever kernel thread is current. Section 4.17 describes the various wait states associated with page faults.

⓮ The rescheduling interrupt service routine selects the page faulting kernel thread for execution, swaps to its context, and then executes a CALL_PAL REI instruction.

⓯ The kernel thread reexecutes the instruction that caused the page fault, this time with the page valid.

4.2.2 Error Returns to SCH$PAGEFAULT

MMG$PAGEFAULT can also return several error status values to SCH$PAGEFAULT:

- SS$_ACCVIO
- SS$_PAGRDERR
- SS$_PAGRDERRXM

If the kernel thread has attempted access to another process's header, MMG$PAGEFAULT returns the error status SS$_ACCVIO, in response to which SCH$PAGEFAULT restores the scratch registers and transfers to EXE$ACVIOLAT, in module EXCEPTION. EXE$ACVIOLAT, described in Chapter *Condition Handling*, simulates an access violation exception to be reported to the access mode that incurred the page fault. If the fault occurred in an inner mode, the system may crash. Section 4.4.2 has further details.

If the system incurred a hardware error on a previous attempt to read the faulted page, MMG$PAGEFAULT determines the access mode in which this page fault occurred and the mode of the page owner. It returns the error status SS$_PAGRDERR when either of the following is true:

- The page fault occurred in user or supervisor mode.
- The page fault occurred in executive or kernel mode and the page is owned by executive or kernel mode.

If the page fault occurred in executive or kernel mode but the page is owned by user or supervisor mode, MMG$PAGEFAULT returns the error status SS$_PAGRDERRXM. This set of circumstances is called a cross-mode page read error.

In response to either status, SCH$PAGEFAULT restores the scratch registers and transfers to EXE$PAGRDERR, in module EXCEPTION. EXE$PAGRDERR, described in Chapter *Condition Handling*, generates the special condition SS$_PAGRDERR or SS$_PAGRDERRXM and reports it to the access mode that incurred the page fault.

If no other condition handler handles either condition, the condition is passed to the last chance condition handler for that mode. For executive mode, the last chance condition handler is EXE$EXCPTNE, in module EXCEPTION_ROUTINES; for kernel mode, the handler is EXE$EXCPTN, in the same module.

Each of these handlers checks whether the condition is SS$_PAGRDERRXM and, if so, requests the Exit ($EXIT) system service, specifying SS$_PAGRDERRXM as the reason for exit. Exiting the image from either executive or kernel mode will cause its process to be deleted. In the case of a cross-mode page read error, the process cannot continue execution, but the system is not affected.

For any other type of condition, in particular, SS$_PAGRDERR, the executive mode last chance condition handler generates the nonfatal bugcheck SSRVEXCEPT and requests the $EXIT system service, causing the process to be deleted. When such conditions occur in kernel mode, the kernel mode last chance condition handler generates the fatal bugcheck SSRVEXCEPT. In the case of a read error for a page owned by kernel mode, system operation may be affected and the executive crashes the system rather than risk system and file integrity.

4.3 Page Transitions for Process Pages

This section describes the transitions of different kinds of process page, which are of type PFN$C_PROCESS. Many of the transitions depend upon the initial location of the virtual page and the location of its backing store.

Initially, a process page is faulted in from a section file on a mass storage medium or created on demand as a page of all zeros, a demand zero page. (One other possibility is a page in a PFN-mapped section. Such a page remains valid throughout its life and is thus outside the scope of this chapter.) A page from a section file is further characterized by whether it is read-only or writable. All demand zero pages are writable.

When a read-only page is removed from the working set, there is no need to record its current contents; the page can be refaulted from its original location. When a writable modified page is removed from the working set, its current contents must be recorded to retain the modifications. The term *backing store* refers to the mass storage location of the modified page.

A writable section page can be characterized by whether it is copy-on-reference. When a process reads or writes a copy-on-reference page, it gets a private copy of the page. The backing store for a copy-on-reference page is a page file. The backing store for one that is not is its section file. A copy-on-reference page removed from the working set list is placed on the modified page list even if it has not been modified. When reducing the size of the modified page list, the modified page writer assigns a page file backing store location and writes the page to it. Subsequently faulted, the page is read from the backing store. This approach simplifies the management of the page at the cost of having to write the page to its backing store even when it has not been modified.

Most demand zero pages are created through the Create Virtual Address Space ($CRETVA and $CRETVA_64), the Expand Program/Control Region ($EXPREG), or the Expand Virtual Address Space ($EXPREG_64) system services. The backing store for such pages is a page file. It is also possible, however, for a process to create a section of demand zero pages backed by a section file. Chapter 3 describes the system services that create various kinds of virtual address space.

The sections that follow describe the transitions for several kinds of process page. Typically, the first transition occurs when the page is faulted in from a mass storage device. In subsequent transitions the page is removed from the working set. It may be placed into the free page list, or it may be placed into the modified page list and written to its backing store. During any of these transitions, the page may be faulted again.

Section 4.2.1 describes the page fault handling steps common to many types of page fault but omits the details of concomitant memory management data structure changes. The sections that follow describe the data structure changes.

Section 4.3.1 describes the initial fault and subsequent transitions of a process section page that is not copy-on-reference; and Section 4.3.2, of a process section page that is copy-on-reference. Section 4.3.3 describes the initial fault of a demand zero page. Its subsequent transitions depend on whether it is a demand zero section page backed in a section file or a simple demand zero page backed in a page file. Section 4.3.4 summarizes some additional kinds of page fault common to the page types already described.

Section 4.3.5 discusses the transitions of a process page that is part of a buffer object page.

4.3.1 Process Section Page That Is Not Copy-on-Reference

An L3PTE for a writable section page, one that is not copy-on-reference, initially contains a process section table index (PSTX) with the copy-on-reference bit (PTE$V_CRF) clear. The transitions that such a page can make are illustrated in Figure 4.3. The numbers in the figure are keyed to the explanations that follow. The column on the right shows how key PFN information changes as the page moves from one state to another.

For simplicity, clustered reads and writes are ignored here, but they are discussed in Sections 4.10 and 4.12.4.

❶ The first transition is faulting the page in from the file that contains it. As described in Section 4.2.1, MMG$PAGEFAULT locates the L3PTE that maps the faulting page and ensures the validity of the page table pages that map it. MMG$PAGEFAULT uses three other routines in module PAGEFAULT to perform key tasks and update memory management data structures accordingly:

- — MMG_STD$ININEWPFN_64, in module PAGEFAULT, allocates a page of memory and updates the PFN database that describes it (see Section 4.8.1).

- MMG_STD$INCPTREF_64, in module PAGEFAULT, increments the share count of the page table page that maps the virtual page in question (see Section 4.8.3).
- MMG_STD$MAKE_WSLE_64, in module PAGEFAULT, updates working set list information (see Section 4.8.2). In particular, it increments the physical page's PFN$W_REFCNT to indicate the page is in a working set list.

MMG$PAGEFAULT itself also increments the PFN$W_REFCNT field for the allocated physical page, bringing the count to 2, to indicate the I/O request about to be queued for this page.

It initializes the L3PTE:

- It inserts the PFN of the allocated page into the L3PTE.
- It leaves the protection, owner, and copy characteristics bits as they were.
- It initializes the type bits to indicate a transition page.
- If the page is writable but was faulted with read intent, it sets the fault-on-write bit.
- It sets the fault-on-execute bit either if the no-execute bit was set or if the page was faulted with read or write data intent rather than with execute intent.

Chapter 2 discusses the significance of the fault-on bits and the executive's use of them.

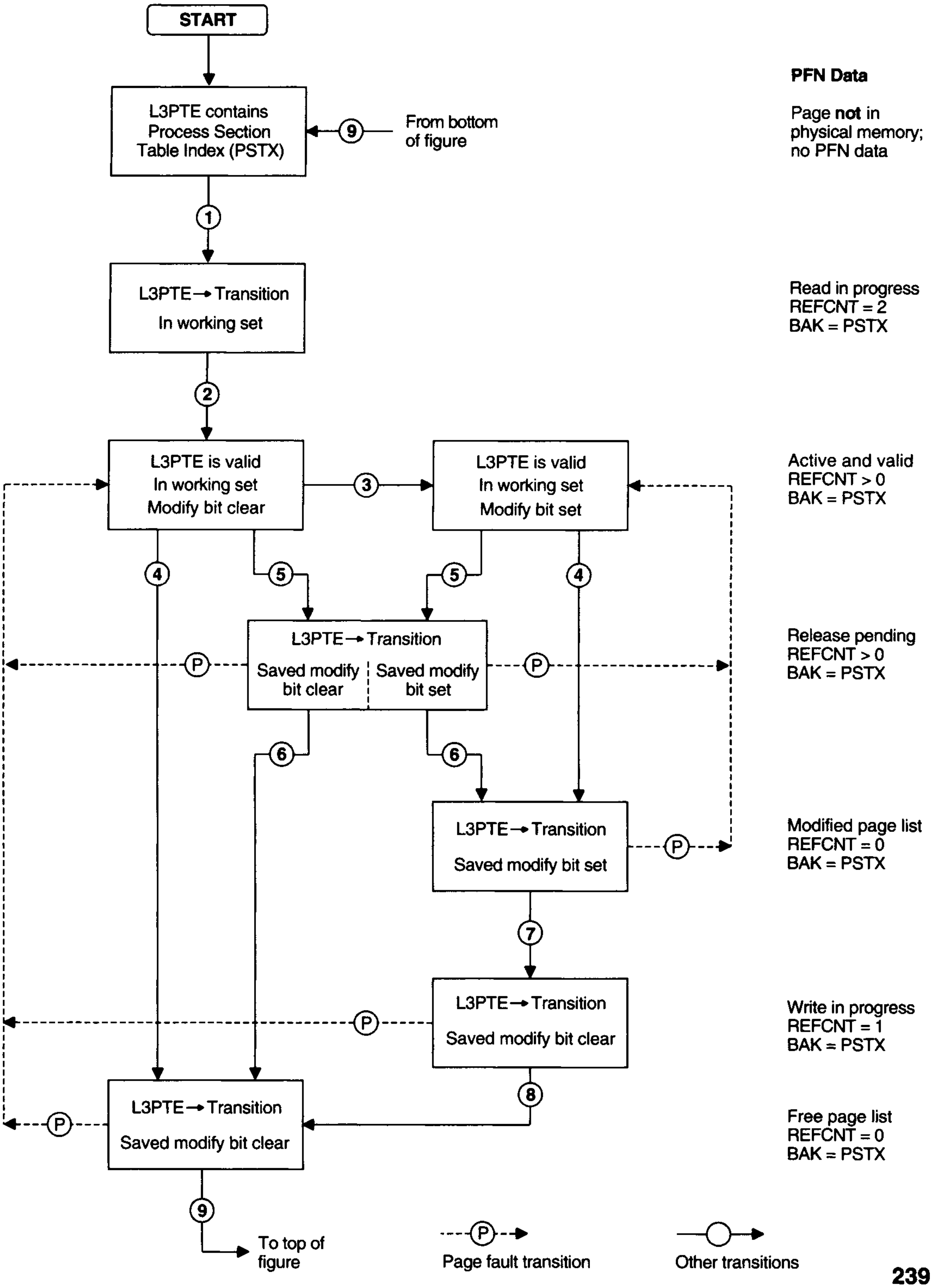
MMG$PAGEFAULT initializes the location bits in the page's PFN$L_PAGE_STATE field to read in progress. It initializes the page's PFN$Q_BAK field from the L3PTE's type and partial section bits and bits <63:32>.

It builds an I/O request packet (IRP; see Section 4.14) that describes the read to be done. From the PSTX in the original L3PTE contents, it locates the corresponding PSTE in the PHD. From information in the PSTE, it can calculate which virtual blocks in the file contain the virtual page.

If the last page in the section has the partial section bit set and is in the cluster to be read, MMG$PAGEFAULT must take extra steps. A partial section is one whose size in blocks is not an exact multiple of the number of blocks in a page. Thus, its last page is not entirely backed by a section file. For this kind of page fault, MMG$PAGEFAULT calculates the I/O request byte count such that the last page's contribution to the count includes only those pagelets that have backing store. It temporarily maps the PFN with a reserved system space L3PTE and clears the part of the partially backed page that has no backing store.

It queues the request to the driver for the device containing the page.

Figure 4.3 Page Transitions for a Process Section Page That Is Not Copy-on-Reference



❷ Because most of the work was done in response to the initial fault, there is little left to do when the page read completes. Holding the MMG spinlock, routine PAGIO, in module IOCIOPOST, decrements PFN$W_REFCNT. In the usual case, the reference count remains greater than zero. In that case, PAGIO changes the PFN$L_PAGE_STATE location bits to active and sets the valid bit in the process-private L3PTE.

If the page is writable, PAGIO tests the fault-on-write bit. If it is clear, indicating that the page was faulted with write intent, PAGIO sets the modify bit in the L3PTE. If the process is not multithreaded and if MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, PAGIO sets the no-TB-miss-memory-barrier-required bit (see Chapter 1) in the L3PTE.

It is, however, possible for PAGIO to decrement the reference count to zero. This can happen if the page was removed from the working set list, for example, through swapper trimming or automatic working set limit adjustment, before the page read completed. The page would have been put in the release pending state with a reference count of 1. If PAGIO decrements the reference count to zero, then instead of setting the valid bit, it inserts the page into the free page list.

❸ OpenVMS maintains the modify bit. When an attempt is made to write to a page that was originally faulted with read intent and one whose fault-on-write bit is set, the processor generates a fault-on-write exception. The exception service routine clears the fault-on-write bit in the L3PTE and sets the modify bit. The change is not noted at this time in the PFN database. When a page is faulted with write intent, the modify bit is set at the same time as the valid bit.

❹ A valid page becomes invalid when it is removed from the working set list as a result of any of the conditions listed in Section 4.1. Most of those result in calling MMG$FREWSLE or its alternative entry point, MMG_STD$FREWSLX_64, in module PAGEFAULT. Chapter 5 describes them in detail. Of most relevance to this chapter are the changes to memory management data structures when a page that is not copy-on-reference is removed from the process working set list:

a. The modify bit in the L3PTE is saved. The valid, modify, fault-on-write, fault-on-execute, and no-TB-miss-memory-barrier-required bits are cleared. Its PFN field is unchanged.

b. The translation buffer is invalidated to remove the cached but now obsolete contents of the L3PTE. If the page was never executed, only the data TB is invalidated. If the process is multithreaded and this is an SMP system, the invalidation is done on every CPU on which kernel threads of the process are active.

c. The saved modify bit from the L3PTE is inserted into the PFN$L_PAGE_STATE field, saving its value.

d. The page's PFN$W_REFCNT is decremented. If the reference count goes to zero, the page is put into the free or modified page list, according to the setting of the saved modify bit in PFN$L_PAGE_STATE. Since the PFN$L_BLINK field overlays the PFN$L_WSLX_QW field, inserting the page into the free or modified page list supplants the PFN$L_WSLX_QW field contents. The page's new location (free or modified page list) is inserted into the PFN$L_PAGE_STATE field.

e. Whether or not the reference count goes to zero, the WSLE is zeroed. PFN$W_PT_VAL_CNT for the page table page mapping this page is decremented. If the count makes the transition to –1, PHD$L_PTCNTVAL is also decremented (see Section 4.4.1). PCB$L_PPGCNT is decremented to indicate one less process page.

❺ If the reference count (decremented in step 4d) does not go to zero, there is outstanding direct I/O for this page. MMG_STD$FREWSLX_64 changes the page's PFN$L_PAGE_STATE location bits from active to release pending.

❻ When direct I/O for the page completes, the I/O postprocessing routine calls MMG_STD$IOUNLOCK_BUF, in module IOLOCK.

For each page in the I/O buffer, MMG_STD$IOUNLOCK_BUF decrements the page's PFN$W_REFCNT. If it goes to zero, MMG_STD$IOUNLOCK_BUF puts the page into either the free or the modified page list, based on the setting of the saved modify bit, and changes PFN$L_PAGE_STATE accordingly. In a typical direct I/O request, the process-private PTEs that map the buffer are copied to a nonpaged pool structure called a direct I/O buffer map (DIOBM) so that a driver can examine them. The L3PTs that contain the PTEs are not locked into memory. If the buffer, however, is too large to be efficiently described in this manner, the L3PTs are locked into memory and doubly mapped into system space; they must be unlocked at I/O completion.

If the L3PTs were locked into memory, MMG_STD$DECPTREF_PFNDB, in module PAGEFAULT, is called for each L3PT that maps I/O buffer pages. It decrements the PFN$L_SHRCNT field in the PFN database record for the L3PT (incremented when the I/O was initiated) to indicate one less reason for it to remain in existence (see Section 4.4.1).

If the page was placed into the free page list, the next stages in its processing are as described in step 9.

❼ If the page was placed into the modified page list, the modified page writer eventually removes the page and writes it to its backing store. A writable page that is not copy-on-reference is written back to the file where it originated. If the page is to be backed in a page file, the modified page writer assigns it to a page file, allocates space for it, and writes it.

The modified page writer sets the PFN$L_PAGE_STATE location bits for the page to write in progress and clears the saved modify bit. The reference count of 1 reflects the outstanding I/O operation.

Note that a section containing writable process pages that are not copy-on-reference cannot be produced by the linker. Such a section must be created with the Create and Map Section ($CRMPSC) or Create and Map Private Disk File Section ($CRMPSC_FILE_64) system service.

❽ When the modified page write completes, the page's reference count is decremented to zero. Because the saved modify bit is clear, the page is put into the free page list.

❾ A page placed on the free page list normally remains attached to the process for some time; that is, the L3PTE contains its PFN, and the PFN$L_PT_PFN and PFN$Q_PTE_INDEX fields in the PFN database record for that page jointly contain the address of the process-private L3PTE.

When the physical page is allocated for another purpose, several steps must be taken to break the ties between the process virtual page and the physical page that is about to be reused. The routine MMG$DEL_CONTENTS_PFN, in module ALLOCPFN, performs those steps:

a. It locates the L3PTE from the contents of the PFN$L_PT_PFN and PFN$Q_PTE_INDEX fields.

b. The L3PTE must be altered to reflect the backing store address of the page. For a page that is not copy-on-reference, the routine restores some of the L3PTE's contents before the initial page fault, namely, the PSTX from the page's PFN$Q_BAK field. It leaves the protection, owner, copy characteristics, and no-execute bits as they were.

c. It calls MMG_STD$DECPTREF_PFNDB, which decrements the PFN$L_SHRCNT field in the PFN database record for the L3PT to indicate one less reason for it to remain valid (see Section 4.4.1).

d. MMG$DEL_CONTENTS_PFN reinitializes the PFN database record for the physical page before reallocating it. In particular, it clears PFN$L_PT_PFN and PFN$Q_PTE_INDEX, the connection from the PFN database to its former process page table. It clears PFN$Q_BAK, the connection to the former contents of the page.

A subsequent fault for the virtual page requires rereading the page from the section file.

4.3.2 Process Section Page That Is Copy-on-Reference

The more common type of writable process page is a copy-on-reference page. The initial value in the L3PTE (START 1 in Figure 4.4) is a PSTX; the copy-on-reference bit (PTE$V_CRF) is set. The writable bit (PTE$V_WRT) is usually set.

Figure 4.4 illustrates the transitions that such a page makes from its initial page fault until it is written to page file backing store. The numbers in the figure are keyed to the explanations that follow. The column on the right shows how key PFN information changes as the page moves from one state to another.

Many of the transitions that occur here resemble the case just described. This section notes each transition but elaborates only those areas that are different.

❶ When a page fault occurs, MMG$PAGEFAULT performs the actions described in step 1 of Section 4.3.1. It also takes several additional steps:

a. It initializes PFN$Q_BAK to indicate that the page will have page file backing store but none has been assigned yet. (Section 4.16 provides further details on page file allocation.) At this time, all ties to the original section file have been broken. When the modified page writer first writes this page to its backing store (as it eventually will because the saved modify bit will be set), it will assign a page file and allocate blocks in it.

b. It updates the PFN$L_PAGE_STATE field location bits to the value read in progress, with the saved modify bit set. The page's backing store will be a page file, not a section file; the copy of the page in the section file must not be modified, yet each of the potentially many copies of the page may be modified. Setting the saved modify bit guarantees that an initial copy of the page will be written to the page file when it is first paged out, whether or not it has been modified.

c. If the last page in the section has the partial section bit set in its L3PTE and is in the cluster to be read, MMG$PAGEFAULT calculates the I/O request byte count accordingly and clears the part of the page without backing store, as described in step 1 of Section 4.3.1. In addition, it clears the partial section flag in PFN$Q_BAK, because once the page is faulted in, it is no longer partially backed; its backing store will be a whole page in a page file.

❷ After the read completes, PAGIO decrements the reference count of each page in the page fault cluster. If the reference count is greater than zero, it updates the PFN$L_PAGE_STATE location bits to active and sets the L3PTE valid bit. If the reference count is decremented to zero because the page has been removed from the working set list, it places the page on the modified page list and changes its PFN$L_PAGE_STATE location bits accordingly.

PAGIO also subtracts the number of pages read from the PSTE's reference count to show that many fewer L3PTEs mapping pages from that section file.

❸ This transition is described in Section 4.3.3.

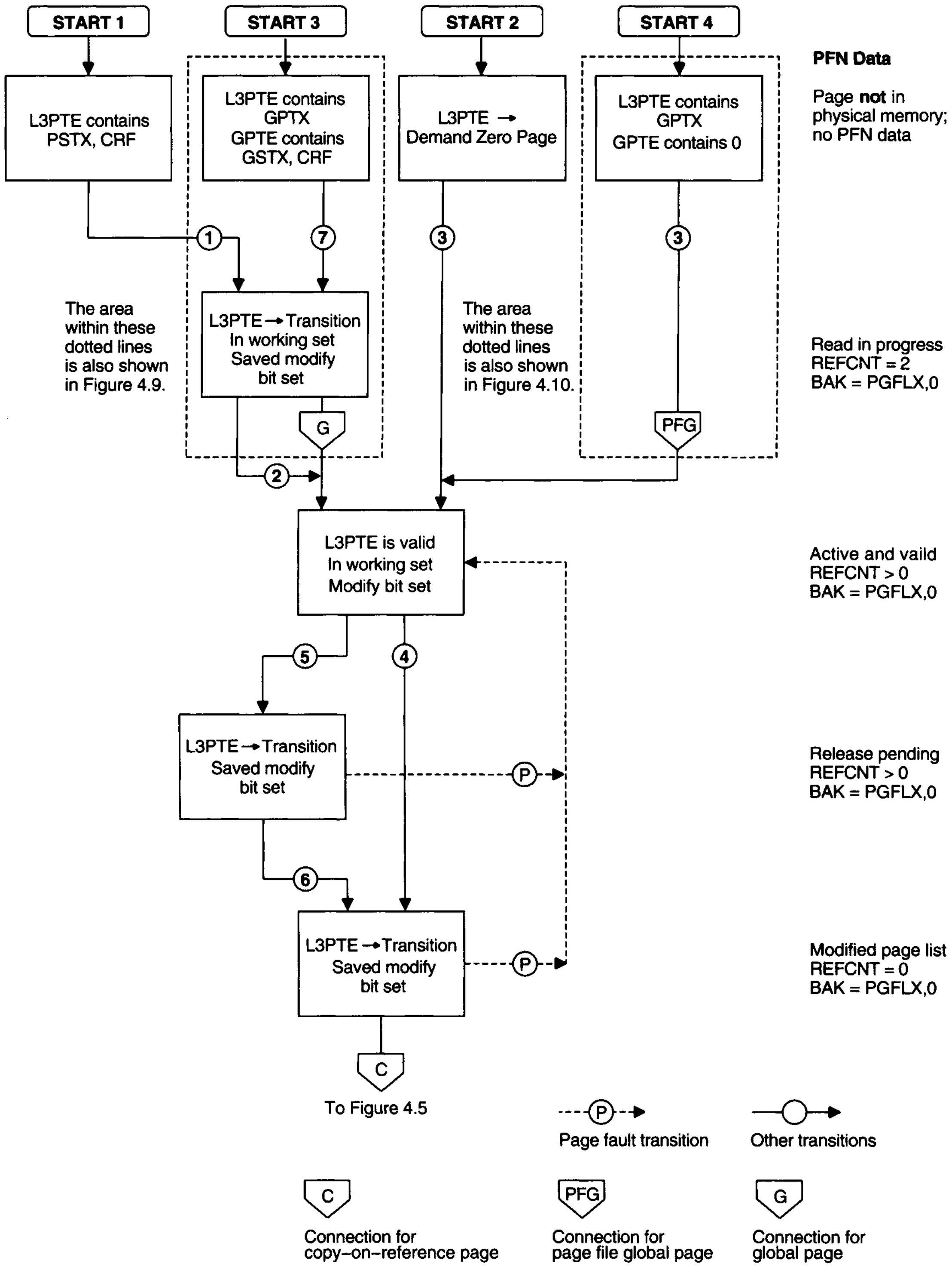
❹ When the copy-on-reference page is removed from the working set and its reference count goes to zero, the page is placed into the modified page list.

If the page has been modified, its assigned page file backing store, if any, contains an obsolete copy. That storage is deallocated, and the page number and page file number in PFN$Q_BAK are cleared.

❺ If the reference count did not go to zero when the page was removed from the process working set, the physical page is placed into the release pending state until the I/O completes.

❻ At that time, the page is put into the modified page list.

Figure 4.4 Page Transitions for Process and Global Copy-on-Reference Pages and for Demand Zero Pages


START 1
START 3
START 2
START 4
PFN Data
L3PTE contains PSTX, CRF
L3PTE contains GPTX GPTE contains GSTX, CRF
L3PTE → Demand Zero Page
L3PTE contains GPTX GPTE contains 0
Page not in physical memory; no PFN data
1
7
3
3
The area within these dotted lines is also shown in Figure 4.9.
L3PTE → Transition In working set Saved modify bit set
The area within these dotted lines is also shown in Figure 4.10.
Read in progress REFCNT = 2 BAK = PGFLX,0
G
PFG
2
L3PTE is valid In working set Modify bit set
Active and vaild REFCNT > 0 BAK = PGFLX,0
5
4
L3PTE → Transition Saved modify bit set
P
Release pending REFCNT > 0 BAK = PGFLX,0
6
L3PTE → Transition Saved modify bit set
P
Modified page list REFCNT = 0 BAK = PGFLX,0
C
To Figure 4.5
P
Page fault transition
Other transitions
C
Connection for copy–on–reference page
PFG
Connection for page file global page
G
Connection for global page

❼ This transition is described as transition 3 in Section 4.5.3.

When the modified page writer writes the page to its backing store in a page file, the page makes a transition from the modified page list.

Figure 4.5, the diagram for faults from a page file, shows this transition. The column on the right shows how key PFN information changes as the page moves from one state to another. The connection between Figure 4.4 and Figure 4.5 is indicated by path C in the two figures. A subsequent fault for the page is resolved from a page file.

The transitions for a page faulted from a page file (see Figure 4.5) resemble those described for a page that is not copy-on-reference (see Figure 4.3). The only difference in the PFN data between the two figures is that the PFN$Q_BAK field value in Figure 4.5 indicates that the page belongs in a page file, whereas the PFN$Q_BAK field value in Figure 4.3 contains a PSTX.

The other difference between the two figures is the entry point into the transition diagram. A page can start out in a section file (the L3PTE contains a PSTX) but a page can never start out in a page file. The entry into Figure 4.5 is from path C in Figure 4.4, from one of several initial states that eventually result in the physical page contents' being written to the page file.

4.3.3 Demand Zero Page

An L3PTE to map a typical demand zero page is initialized by the $CRETVA, $CRETVA_64, $EXPREG, or $EXPREG_64 system service. These services can be requested explicitly by an image or implicitly by the system on behalf of the process, for example, as part of image activation. Also, a process can request the $CRMPSC or $CRMPSC_FILE_64 system service to create a demand zero section backed by a section file. An L3PTE to map such a section has a PSTX with the PTE$V_CRF bit clear and the PTE$V_DZRO bit set. Either type of demand zero page is created the first time it is faulted.

Figures 4.4 (START 2) and 4.5 illustrate the transitions of a typical demand zero page, one backed in a page file.

The transitions of a demand zero section page resemble those in Figure 4.3 except for the steps to get to the active and valid state.

The following description corresponds to step 3 in Figure 4.4 for a simple demand zero page and to the entry into Figure 4.3 for a demand zero section page.

❸ When MMG$PAGEFAULT detects a page fault for a demand zero page, it calls MMG_STD$ININEWPFN_DZRO_64, in module PAGEFAULT, to allocate a free zeroed page (see Section 4.8.1). It calls MMG_STD$INCPTREF_64, in module PAGEFAULT, to increment the share count of the page table page that maps the virtual page in question (see Section 4.8.3). It calls MMG_STD$MAKE_WSLE_64 to update working set list information (see Section 4.8.2). In particular, it increments the physical page's PFN$W_REFCNT to indicate the page is in a working set list.

MMG$PAGEFAULT makes additional updates to memory management data structures:

a. It changes the PFN$L_PAGE_STATE location bits to active.

b. It initializes PFN$Q_BAK to indicate that the page will have page file backing store but none has been assigned yet. Assignment of a page file and allocation of actual blocks in that file are done later by the modified page writer.

   If the page is a demand zero section page, its backing store is the section file. MMG$PAGEFAULT clears the PTE$V_DZRO bit. Once the page has been created, it becomes a non-CRF section page.

c. It inserts the PFN into the L3PTE associated with the fault, setting the valid and modify bits, and leaving the protection, owner, copy characteristics, and no-execute bits as they were. If the no-execute bit is set, MMG$PAGEFAULT also sets fault-on-execute. If the process is not multithreaded and if MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, it also sets the no-TB-miss-memory-barrier-required bit (see Chapter 1) in the L3PTE.

Subsequent transitions for a demand zero page are shown in Figure 4.4 and described throughout Sections 4.3.1, 4.3.2, and 4.3.4.

4.3.4 Page Faults out of Transition States

Figures 4.3, 4.4, and 4.5 show some of the transitions that can occur when a virtual page is faulted while the associated physical page is in the transition state. Because these types of page fault require no I/O, they are referred to as soft page faults.

While these changes back to the valid state are straightforward, certain details about each fault should be mentioned. Most of the following transitions are represented in the figures by a P within a circle.

MMG$PAGEFAULT resolves a page fault from the free page list in the following way:

1. It first removes the page from the list.

2. If appropriate, MMG$PAGEFAULT copies the contents of the page to a page of physical memory allocated from the process's home resource affinity domain (RAD). All the following conditions must be true:

   — The page is not a global page.

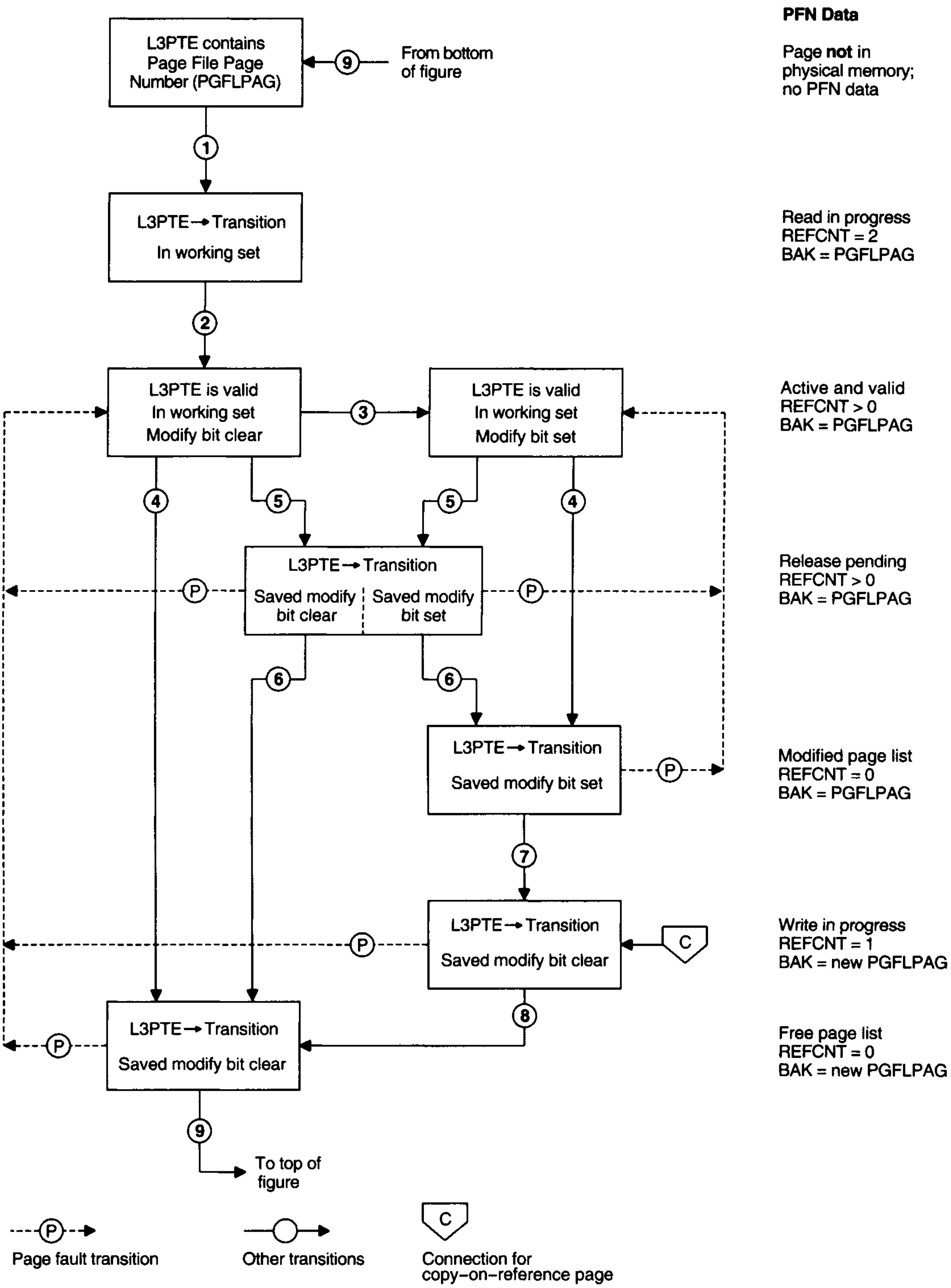
   — Its PFN$W_REFCNT is zero.

   — This is a nonuniform memory access (NUMA) platform with general RAD support, RAD-specific process allocation, and soft fault copy enabled.

   — The RAD associated with the page is not the same as the process's home RAD.

   After copying the contents of the page, MMG$PAGEFAULT breaks the page's connection with the page table that mapped it and reinserts the page at the front of the free page list.

Figure 4.5 Page Transitions for a Page Located in a Page File



Typically, however, MMG$PAGEFAULT instead uses the page just removed from the free page list.

3. It calls MMG_STD$MAKE_WSLE_64 (see Section 4.8.2) to update the memory management data structures to reflect the fact that the page is in the working set list. In particular, it increments the physical page's PFN$W_REFCNT.
4. MMG$PAGEFAULT changes the page's PFN$L_PAGE_STATE location bits to active.
5. MMG$PAGEFAULT initializes the L3PTE:
   - If the page is writable but has not been modified, it sets the fault-on-write bit in the L3PTE.
   - If the page is writable and was faulted with write intent, it sets the modify bit in the L3PTE.
   - If the process is not multithreaded and if MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, it also sets the no-TB-miss-memory-barrier-required bit in the L3PTE.
   - It sets the fault-on-execute bit in the L3PTE either if the no-execute bit was set or if the page was faulted with read or write data intent.
   - It sets the valid bit in the L3PTE. (Recall that a transition PTE retains the PFN of the physical page in which the virtual page resides.)

A page fault from the modified page list is resolved in exactly the same way. Figures 4.3 to 4.5 show that the page was previously modified but never written to its backing store by returning the page to its modified state. That is, the saved modify bit in its PFN$L_PAGE_STATE field remains set, causing the page to be put into the modified page list when it is removed from the working set again.

A page fault from the release pending state is similar to the previous two except that the page does not have to be removed from a page list and copy on soft page fault is not an option. Artistic license is taken in the figures to differentiate physical pages that were modified from pages that were not.

A transition deserving special comment is a page fault that occurs while the modified page writer is writing the page to its backing store. The saved modify bit is cleared before the write begins so that the page will be placed into the free page list when the write completes. Although the page has not yet been completely backed up, it is assumed that the write will complete successfully. A page fault for the page can thus put it into the active but unmodified state. The only difficulty occurs in the event of a write error. The modified page writer's I/O completion routine, WRITEDONE in module WRTMFYPAG, detects this and resets the saved modify bit.

In the case of a single-threaded process, a page fault for a process page (type PFN$C_PROCESS) being read in response to a previous page fault results in placing the kernel thread back into a page fault wait state. This can occur if a kernel thread in page fault wait is made computable to execute an AST. When the AST procedure completes, control returns to the instruction that caused the fault. If the page is still invalid, the kernel thread is placed back into page fault wait.

In the case of a multithreaded process, the kernel thread is placed in collided page wait.

4.3.5 Process-Private Buffer Object Page

A buffer object is a special kind of I/O buffer. The pages that make up a buffer object are locked into physical memory and may be doubly mapped in system space as well as process-private space. Because the pages are already locked into memory, there is no need for a device driver to lock them when initiating an I/O request and no need for the I/O postprocessing routine to unlock them. The implementation of buffer objects enables the body and process header of a process with I/O in progress to a buffer object to be swapped.

Chapter 3 details the system services that create and delete buffer objects, and Chapter 2 discusses the buffer object descriptor data structure associated with each buffer object.

A process-private buffer object page begins life as a process page, perhaps a demand zero page. Its initial transitions therefore are no different from those of that page type. The transitions particular to a buffer object page are illustrated in Figure 4.6. The column on the right shows how key PFN information changes as the page moves from one state to another. The numbers in the figure are keyed to the explanations that follow.

Figure 4.6 begins with the page already valid, in the process's working set. The Create Buffer Object ($CREATE_BUFOBJ or $CREATE_BUFOBJ_64) system service faults it into the working set if it is not already valid.

❶ The system service locks this page (and any other in the buffer object) into memory by incrementing the page's reference count, PFN$W_REFCNT; sets the buffer object and saved modify bits in PFN$L_PAGE_STATE; and increments the page's PFN$W_BO_REFC to zero.

Similarly, for the process-private page table page that maps the buffer object, it increments PFN$W_BO_REFC. If this is the first buffer object page mapped by this page table, it increments PFN$W_REFCNT and sets the buffer object and saved modify bits in PFN$L_PAGE_STATE.

Optionally, the system service initializes a system space L3PTE to double-map the buffer object page.

❷ When the buffer object page is removed from the working set, for example, as a result of replacement paging, the valid and modify bits in the process-private L3PTE that map it are cleared. The page's reference count is decremented to 1, and the location bits in PFN$L_PAGE_STATE are set to release pending. The share count for the process-private page table page that maps it is decremented.

❸ When the buffer object page is faulted back into the working set, its PFN$L_PAGE_STATE location bits are changed to active and its reference count is incremented. The share count for the process-private page table page that maps it is incremented.

❹ When the buffer object is deleted, the Delete Buffer Object ($DELETE_BUFOBJ) system service clears the system space L3PTE that double-maps the page, if any, and invalidates any cached entry from the TB. It decrements the buffer object page's PFN$W_BO_REFC. Typically, the page is part of only one buffer object, and PFN$W_BO_REFC is now zero. In that case, the service clears the buffer object bit in its PFN$L_PAGE_STATE field and decrements PFN$W_BO_REFC for the process-private page table page that maps the buffer object. If that goes to zero, it clears the buffer object bit in its PFN$L_PAGE_STATE field and decrements the page's reference count. Since the former buffer object page is in a release pending state, the service increments the page table page's share count. It decrements the former buffer object page's reference count.

❺ If the reference count is now zero, the page is released to the modified page list.

4.4 Page Transitions for Process-Private Page Table and PHD Pages

This section describes the transitions of two kinds of pages of type PFN$C_PPGTBL: page table pages and PHD pages.

4.4.1 Process-Private Page Table Page

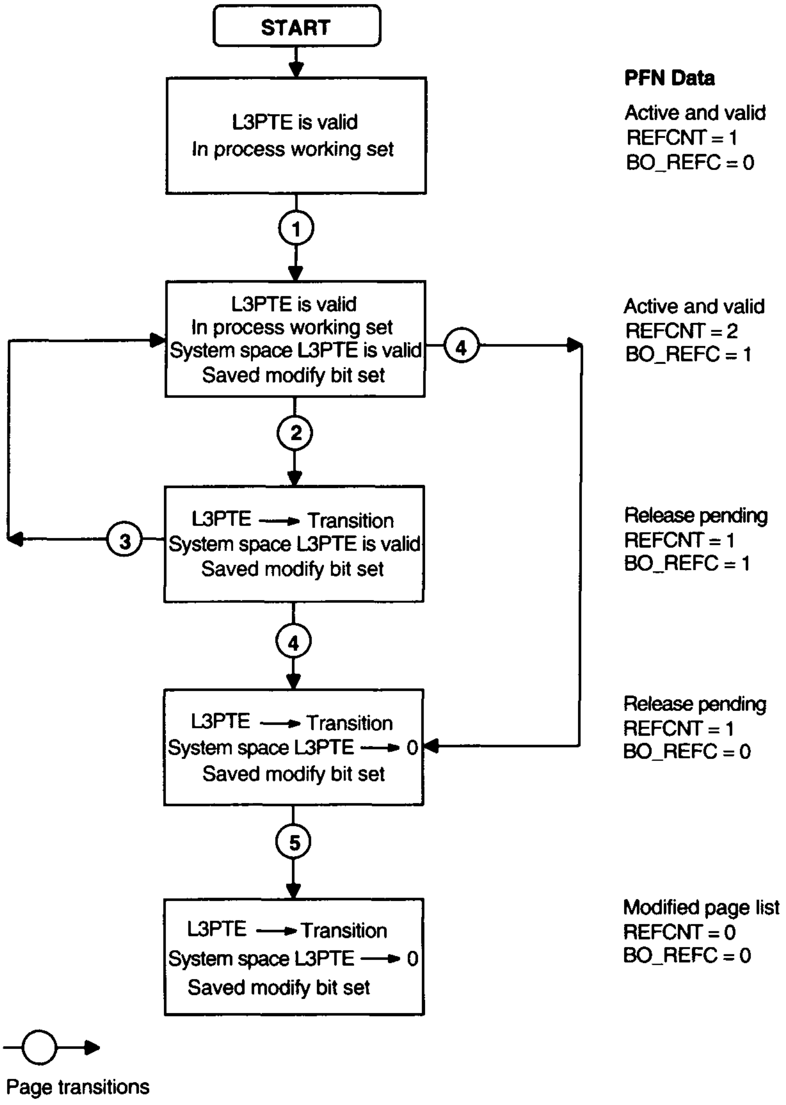
As described in Chapter 2, the L1PT, L2PTs and L3PTs that map a process's P0, P1, and P2 space are mapped in its page table space. They are only accessible from process context, except for circumstances in which the executive double-maps a page table page into system space to access it when that process context is not current.

The L2PT and L3PTs that map permanent P1 space are created when the process is created. That L2PT is sufficient to map all of P0, P1, and some P2 space. Until an image is activated in the process and additional address space created, most of the L2PTEs in the permanent L2PT are zero. When the process requests a system service to create address space, the system service initializes the L3PTEs that map that space and, if necessary, the L2PTEs.

Many of the transitions of a process page table page resemble those of other demand zero pages, described in Section 4.3.3. Some aspects of page table page transitions are unique, however.

Some of the transitions that such a page can make are illustrated in Figure 4.7. The numbers in the figure are keyed to the following explanations. The column on the right shows how key PFN information changes as the page moves from one state to another.

Figure 4.6 Page Transitions for a Buffer Object Page



For simplicity, some of the transitions shown in Figures 4.4 and 4.5 are omitted here, and this section is confined to transitions of process-private L3PTs.

❶ When a process faults a page in a region that expands automatically, such as a stack page and the page's L3PTE is mapped by a zero L2PTE, the page fault is transformed into a fault for the L3PT.

When MMG$PAGEFAULT detects a page fault for a process page table page that has not yet been created, it takes the following steps:

a. MMG$PAGEFAULT uses other routines in module PAGEFAULT to perform some of the related updates to memory management data structures:

   — MMG_STD$ININEWPFN_DZRO_64 allocates a free zeroed page (see Section 4.8.1).

   — MMG_STD$INCPTREF_64 increments the share count of the L2PT that maps the L3PT (see Section 4.8.3).

   — MMG_STD$MAKE_WSLE_64 updates working set list information (see Section 4.8.2). In particular, it increments the physical page's PFN$W_REFCNT to indicate the page is in a working set list.

b. MMG$PAGEFAULT updates the PFN$L_PAGE_STATE location bits to active.

c. It inititializes PFN$Q_BAK to indicate that the page will have page file backing store but none has been assigned yet. Assignment to a page file and allocation of space in it are done later by the modified page writer.

d. It inserts the PFN into the L2PTE associated with the fault, setting the valid, modify, and fault-on-execute bits, and leaving the protection, owner, copy characteristics, and no-execute bits as they were. If the process is not multithreaded and if MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, it also sets the no-TB-miss-memory-barrier-required bit in the L2PTE.

e. Finally, MMG$PAGEFAULT returns the status SS$_NORMAL to SCH$PAGEFAULT.

Control returns to the system service, which initializes L3PTEs, for example, to map a section. When done, the system service returns.

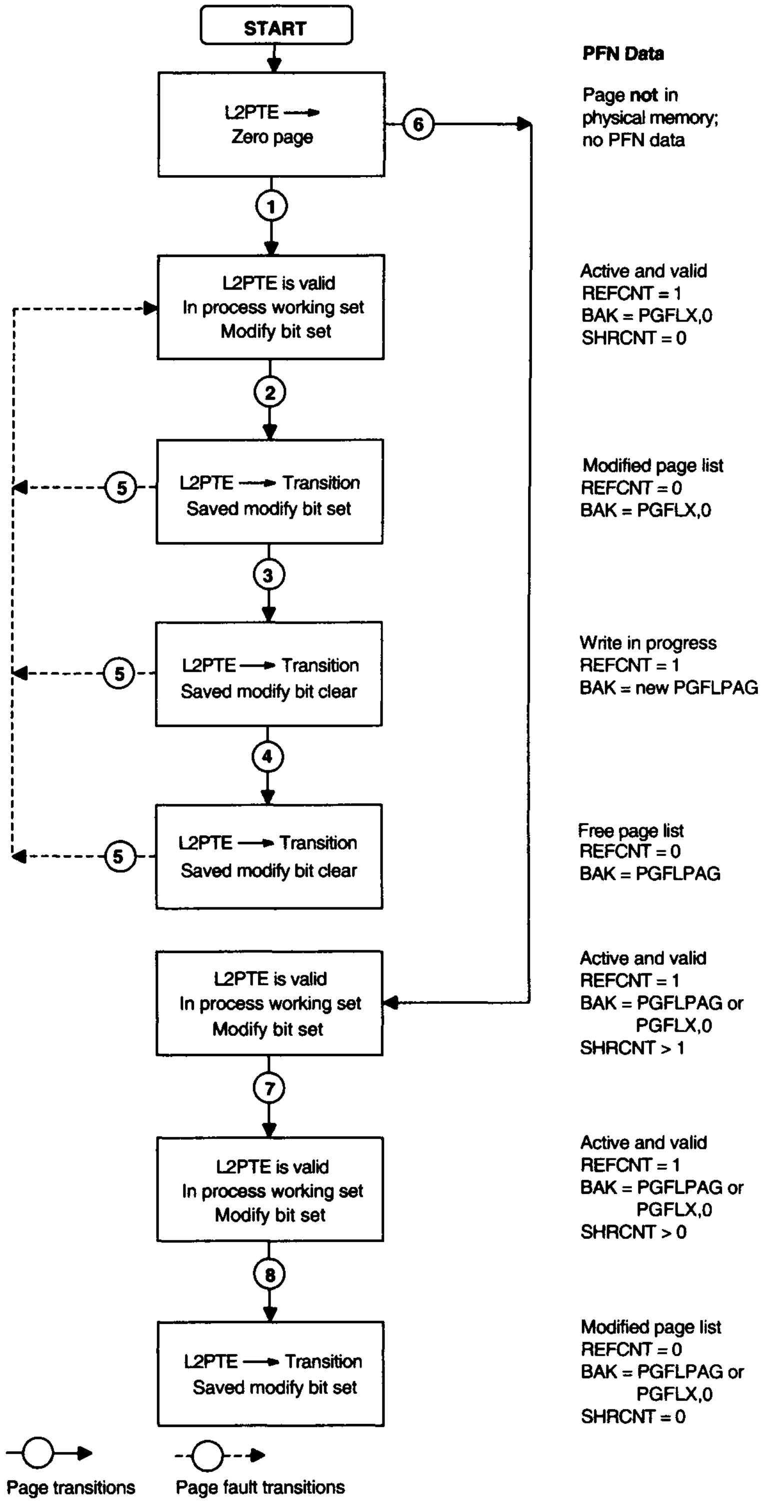
❷ If none of the process pages mapped by the L3PT is made valid, the process page table page can be removed from the working set as a result of replacement paging. MMG$FREWSLE increments the PHD's entry in the array at PHV$GL_REFCBAS_LW, the number of reasons the PHD should remain in memory, to account for the page table page as a transition page. Decrementing the page's reference count to zero, it inserts the page into the modified page list. It also decrements PCB$L_PPGCNT and clears the WSLE that was associated with the page.

It decrements the L2PT's PFN$W_PT_VAL_CNT and, if the L2PT maps no more valid WSLEs, it decrements PHD$L_PTCNTVAL.

❸ The modified page writer eventually removes the page from the modified page list, assigns it a page file, and writes it to allocated space in the page file.

❹ When the write completes, the page is placed into the free page list.

Figure 4.7 Page Transitions for Process Page Table Pages



❺ When the process later tries to access a page mapped by this L3PT, it incurs a page fault. MMG$PAGEFAULT calculates the virtual address of the L3PTE mapping the target address and discovers that the L3PT is not valid. It transforms the fault for the target address into one for the L3PT.

In Figure 4.7, the fault is shown as happening before the physical page containing the L3PT is reallocated for another use. MMG$PAGEFAULT faults the page from the free page list, updates the data structures that describe the page, and returns the status SS$_NORMAL to SCH$PAGEFAULT. When SCH$PAGEFAULT dismisses the exception, the instruction that attempted access to a page mapped by this L3PT is reexecuted.

❻ When MMG$PAGEFAULT processes the first page fault for a page mapped by this L3PT, it and its associated routines take the following actions:

a. MMG_STD$INCPTREF_64 increments the share count for the L3PT to indicate that it maps one more valid page. If this is the first valid page mapped by the page table page (that is, if the share count makes the transition from 0 to 1), it locks the WSLE for the page table page into the process's working set list by setting the WSL$V_WSLOCK bit and also increments PHD$L_PTCNTACT, the number of active page table pages for the process, and the PHD's entry in the array at PHV$GL_REFCBAS_LW.

b. When updating the data structures related to the working set list, such as the WSLE for the faulted page, MMG_STD$MAKE_WSLE_64 also increments PFN$W_PT_VAL_CNT for the page table page to indicate one more valid entry in the process's working set list mapped by that page table page. If the count makes the transition from –1 to 0, it also increments PHD$L_PTCNTVAL, the number of page table pages that map valid WSLEs.

Whenever the process faults another page mapped by this L3PT, the L3PT's share count and PFN$W_PT_VAL_CNT are incremented.

❼ Whenever one of the pages mapped by this L3PT is removed from the working set, MMG$FREWSLE decrements PFN$W_PT_VAL_CNT to indicate the L3PT maps one less valid page. When the count makes the transition to –1, the page table page is dead, and MMG$FREWSLE also decrements PHD$L_PTCNTVAL.

Once the page table page is dead, its WSLE is a candidate for reuse by a page being newly faulted into the working set. While the page table page describes transition pages, however, the WSLE cannot be reused. To free the WSLE, MMG$FREWSLE severs all ties between the transition pages on the free page list and the page table page, moves those pages to the head of the free page list, and requests a selective purge of the modified page list (see Section 4.12). Chapter 5 contains further information on how a dead page table page is removed from the working set.

❽ As the contents of each page are deleted, MMG_STD$DECPTREF_PFNDB is called to update the data structures describing the L3PT. It decrements the share count for the L3PT to indicate one less reason for it to remain valid.

When the share count makes the transition from 1 to 0, MMG_STD$DECPTREF_PFNDB takes the following additional steps:

a. It decrements the PHD's entry in the array at PHV$GL_REFCBAS_LW, the number of reasons the PHD should remain in memory. If that count goes to zero, MMG_STD$DECPTREF_PFNDB wakes the swapper process to outswap the PHD.

b. It locates the WSLE for the page table and clears its WSL$V_WSLOCK bit to unlock it from the process's working set list.

c. It decrements PHD$L_PTCNTACT, the number of active page table pages for the process.

The L3PT page is removed from the working set list and placed on the modified page list. Eventually, it is written to its backing store and placed on the free page list. When the L3PT page is allocated for another use, its connections to the L2PT must be severed: the backing store location of the L3PT is stored in the L2PTE, and the L2PT's share count is decremented. If the share count is now zero, the L2PT's WSLE is unlocked from the working set list, PHD$L_PTCNTACT is decremented to indicate one less active page table page, and the PHD reference count is decremented.

4.4.2 Process Header Page

Historically, a process's page tables were a pageable part of its PHD. Unlike other system space pages, PHD pages belonged to the associated process and were listed in its working set list. A process was therefore not allowed to fault a page in another process's PHD.

A specific check for this circumstance was added to MMG$PAGEFAULT. When it determined that a page fault for a system space page was within the balance set slots, if one process was trying to fault a page in another process's PHD, it transformed the page fault into an access violation. It used the page fault exception parameters as access violation parameters (see Section 4.2.2).

To eliminate the possibility that the process had been outswapped after faulting the page table in its PHD's previous balance set slot and that it was now trying to access it in the new balance set slot, MMG$PAGEFAULT also tested and cleared bit PHD$V_NOACCVIO in PHD$L_FLAGS, which had been set by the swapper at inswap. If the bit was set, MMG$PAGEFAULT dismissed the page fault so that the faulting instruction could reexecute, recalculating the page table address.

With the page tables removed from the PHD as of OpenVMS Version 7.0, all the pages of the header are nonpageable, and MMG$PAGEFAULT's check is largely superfluous, except for one particular case: A FRED page or expansion PHD page is created by first storing the demand zero format in its PTE and then touching it to materialize the page and lock it into the working set. Between those two steps, if the kernel thread of another process were to fault the newly created page, it would incur an access violation.

4.5 Page Transitions for Global Pages

The transitions of global pages, which are of types PFN$C_GLOBAL and PFN$C_GBLWRT, resemble those of process pages. A major difference, however, is the presence of both a global page table entry (GPTE) and potentially multiple process-private L3PTEs that refer to the same page.

Page transitions for memory-resident global section pages are described in Section 4.5.5.

4.5.1 Global Read-Only Page

This section assumes much of the detail shown earlier in Figure 4.3 and focuses on an example in which two processes map the same global page. Figure 4.8 illustrates the transitions that occur for a global read-only page in an already created section that is mapped by two processes. The column on the right shows how key PFN information changes as the page moves from one state to another. In the figure, the term VA_PTE represents the combination of fields PFN$L_PT_PFN and PFN$Q_PTE_INDEX. The numbers in the figure are keyed to the explanations that follow.

When the global section is initially created, as described in Chapter 3, the data structures described in Chapter 2 are initialized. The GPTE for the page represented in Figure 4.8 contains a global section table index (GSTX), which locates the global section table entry (GSTE) containing information about the global section file.

❶ When process A maps the section, each L3PTE representing a page in the section is initialized with a global page table index (GPTX), effectively a pointer to the associated GPTE.

❷ When process B maps the section, its L3PTEs contain exactly the same GPTXs as those in process A's L3PTEs.

❸ Process B happens to fault the global page first. After reserving an entry in process B's working set list, MMG$PAGEFAULT takes the following steps, many of which are the same as those taken for a process section page (see step 1 in Section 4.3.1):

a. Because process B's L3PTE contains a GPTX, MMG$PAGEFAULT indexes the global page table with it to get the GPTE. The GPTE contains a GSTX, indicating that the global page resides on mass storage.

b. It calls MMG_STD$ININEWPFN_64 to allocate a physical page (see Section 4.8.1).

c. MMG_STD$ININEWPFN_64 calls MMG_STD$INCPTREF_64 to update the data structures describing the global page table page that maps the faulted page (see Section 4.8.3).

d. MMG_STD$ININEWPFN_64 calls MMG_STD$MAKE_WSLE_64 to update working set list information (see Section 4.8.2). In particular, it increments the physical page's PFN$W_REFCNT to indicate the page is in a working set list.

e. MMG$PAGEFAULT inserts the PFN of the allocated page into the GPTE, leaving the protection, owner, and copy characteristics bits as they were. It initializes the type bits to indicate a transition page.

f. It sets the PFN$L_PAGE_STATE location bits to read in progress.

g. It stores the GSTX in the PFN$Q_BAK field.

h. It sets the fault-on-execute bit in B's L3PTE either if the no-execute bit was set or if the page was faulted with read data intent.

MMG$PAGEFAULT initiates a read of the faulted page from its section file. While the read is in progress, the GPTE contains a transition PTE but process B's L3PTE still contains the GPTX. The reference count for the page indicates two references: one for the read in progress and one because the page is in process B's working set (the share count field is nonzero).

❹ After the read completes, the I/O postprocessing routine PAGIO locates the process-private L3PTE through the PTE index and page table PFN stored in the I/O request packet. It temporarily maps the process's L3PT into system space. It takes the following steps for each page in the page fault cluster:

a. It decrements the page's reference count. The reference and share counts are both 1 at this point.

b. It changes the PFN$L_PAGE_STATE location bits to active.

c. It sets the valid bit in the GPTE to record the fact that this page is physically resident and in a process working set.

d. It inserts the PFN into the process's L3PTE, setting the valid bit.

PAGIO reports the scheduling event page fault completion for process B's kernel thread so that it becomes computable.

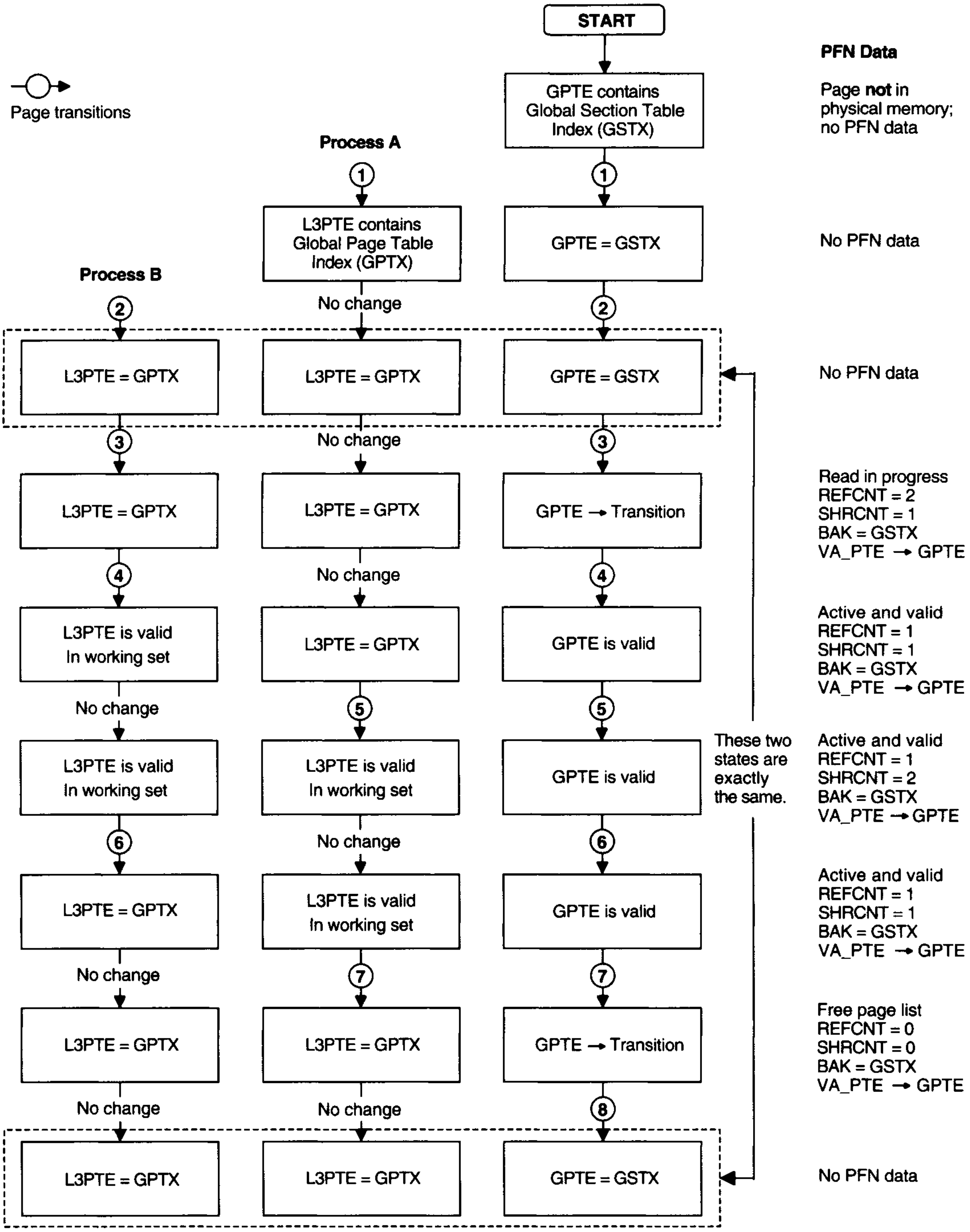
Section 4.11 contains further details.

❺ When process A faults the same global page, MMG$PAGEFAULT's initial action is the same as it was in step 3, because the L3PTE contains a GPTX. Now, however, MMG$PAGEFAULT finds a valid GPTE. Resolution of this page fault is simple and requires no I/O. Such a fault is known as a soft page fault.

Through MMG_STD$MAKE_WSLE_64 and MMG_STD$INCPTREF_64, whose actions are described in Sections 4.8.2 and 4.8.3, MMG$PAGEFAULT initializes the WSLE for process A, increments its PCB$L_GPGCNT, and increments the share count for the global page to 2.

MMG$PAGEFAULT inserts the PFN from the GPTE into process A's L3PTE, leaving the protection, owner, copy characteristics, and no-execute bits as they were and setting the valid bit. If the process is not multithreaded and if MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, it also sets the no-TB-miss-memory-barrier-required bit in the L2PTE.

Figure 4.8 Page Transitions for a Global Read-Only Page Mapped by Two Processes



❻ When MMG$FREWSLE removes the global page from process B's working set, it invalidates any cached TB entry for that virtual page and restores process B's L3PTE to its previous state (rather than some transition form). MMG$FREWSLE retrieves the GPTX from the physical page's PFN$Q_PTE_INDEX field and inserts it in process B's L3PTE as a GPTX.

It decrements the share count for the L3PT to indicate that it maps one less page. It decrements the share count for the global page itself. The share count is still positive, and thus the GPTE remains valid. It updates the data structures related to process B's working set list, for example, clearing the WSLE. It decrements process B's PCB$L_GPGCNT.

❼ When MMG$FREWSLE removes the global page from process A's working set, it restores the process L3PTE as described in step 6.

It decrements the share count, this time to zero. It therefore clears the valid, fault-on-read, fault-on-write, modify, and NO_MB bits in the GPTE to turn it into a transition PTE and decrements the page's reference count. A global read-only page with a reference count of zero, such as this one, is placed into the free page list and its PFN$L_PAGE_STATE location bits are updated accordingly. The other PFN database record fields are unchanged.

❽ When the physical page is reused, the ties must be broken between the physical page and, in this case, the GPTE. None of the processes mapped to this page are affected in any way by this step.

The contents of the PFN$Q_BAK field, a GSTX, are inserted into the GPTE located by the contents of PFN$Q_PTE_INDEX. MMG_STD$DECPTREF_PFNDB, described in Section 4.4.1, is called to update the data structures describing the global page table page that contains the GPTE. The PFN$Q_PTE_INDEX and PFN$L_PT_PFN fields are then cleared, breaking the connection between the physical page and the global page table.

These steps return the process and global page tables to the state following step 2, although it is pictured here as a different state to simplify the figure.

4.5.2 Global Writable Page

The transitions that occur for a global writable page, which is of type PFN$C_GBLWRT, are the same as those for a process page that is not copy-on-reference. The only difference between such transitions and those illustrated in Figure 4.3 is that the GPTE, not the process-private L3PTE, is affected by the transitions of the physical page.

The process-private L3PTE for a global page contains a GPTX up to the time that the page is made valid. Only then is a PFN inserted into the process L3PTE. As soon as the page is removed from the process working set, the GPTX is restored to the process L3PTE. All ties to the PFN database are made through the GPTE, which retains the PFN while the physical page is in the various transition states.

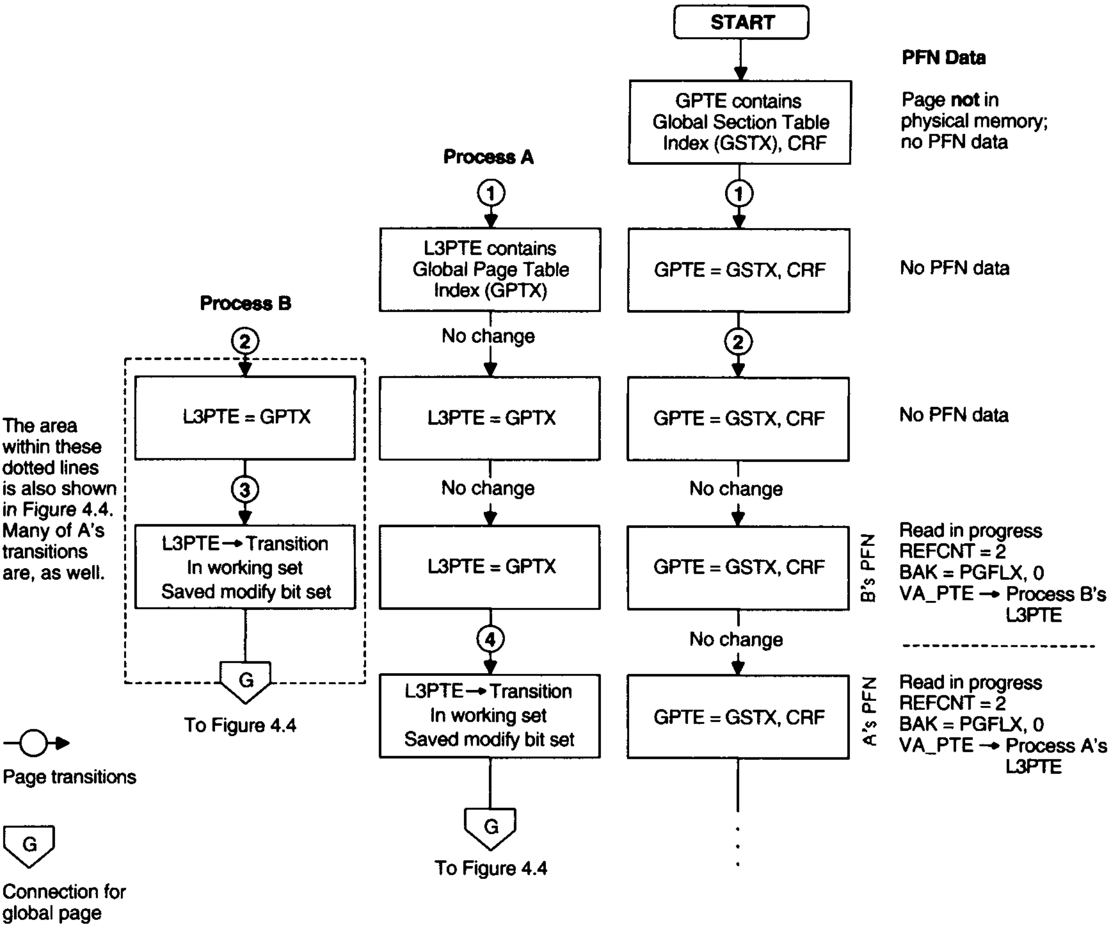
4.5.3 Global Copy-on-Reference Page

A global copy-on-reference page is shared only in its initial state. As soon as the fault occurs, the page is treated exactly like a process page.

Figure 4.9 illustrates the transitions that occur for a global copy-on-reference page. The column on the right shows how key PFN information changes as the page moves from one state to another. In the figure, the term VA_PTE represents the combination of fields PFN$L_PT_PFN and PFN$Q_PTE_INDEX. The numbers in the figure are keyed to the explanations that follow.

The initial conditions are identical to those in Figure 4.8. After the section is created, each of its GPTEs contains a GSTX. In this case, the copy-on-reference bit is set in each GPTE.

Figure 4.9 Page Transitions for a Global Copy-on-Reference Page


START
PFN Data
GPTE contains Global Section Table Index (GSTX), CRF
Page not in physical memory; no PFN data
Process A
1
L3PTE contains Global Page Table Index (GPTX)
1
GPTE = GSTX, CRF
No PFN data
Process B
2
No change
2
L3PTE = GPTX
L3PTE = GPTX
GPTE = GSTX, CRF
No PFN data
The area within these dotted lines is also shown in Figure 4.4. Many of A's transitions are, as well.
3
No change
No change
L3PTE→Transition In working set Saved modify bit set
L3PTE = GPTX
GPTE = GSTX, CRF
B's PFN
Read in progress
REFCNT = 2
BAK = PGFLX, 0
VA_PTE → Process B's L3PTE
G
4
No change
To Figure 4.4
L3PTE→Transition In working set Saved modify bit set
GPTE = GSTX, CRF
A's PFN
Read in progress
REFCNT = 2
BAK = PGFLX, 0
VA_PTE → Process A's L3PTE
Page transitions
G
To Figure 4.4
G
Connection for global page

❶ Process A maps the page; the GPTX is stored in its L3PTE.

❷ Process B maps the page; the same GPTX is stored in its L3PTE. Up to this point, nothing is different from Figure 4.8.

❸ When process B faults the page, MMG$PAGEFAULT locates the GPTE from the GPTX and notes that the page is located in a global section file and is copy-on-reference. MMG$PAGEFAULT, in concert with MMG_STD$ININEWPFN_DZRO_64, MMG_STD$INCPTREF_64, and MMG_STD$MAKE_WSLE_64 (see Sections 4.8.1 to 4.8.3), allocates a page from the free page list and updates the pertinent memory management data structures as follows:

a. The GPTE is not altered and retains its GSTX contents.

b. The PFN$Q_PTE_INDEX and PFN$L_PT_PFN fields get the location of process B's L3PTE.

c. The share count for the page table page containing process B's L3PTE is incremented. Section 4.4 details other changes to data structures related to the page table page.

d. The PFN$L_PAGE_STATE type bits for the physical page are set to process page.

e. An entry in process B's working set list is initialized to describe the faulted page.

f. The PFN$L_WSLX_QW field is set to the index of the WSLE.

g. PCB$L_PPGCNT is incremented.

h. The reference count is incremented twice, once for the page's membership in the working set and once for the I/O in progress.

i. Process B's L3PTE is changed to a transition PTE with the PFN of the allocated page. The protection, owner, and copy characteristics bits are left as they were. If the page is writable but was faulted with read intent, MMG$PAGEFAULT sets the fault-on-write bit. (Code common to several page types sets the fault-on-write bit, although it is unnecessary in this case.) It sets fault-on-execute either if the no-execute bit was set or if the page was faulted with read or write data intent.

j. The physical page's PFN$Q_BAK is initialized to indicate that the page will have page file backing store but none has been assigned yet.

k. PFN$Q_BAK is also initialized from the GPTE's type and partial section bits, <63:32>.

l. The PFN$L_PAGE_STATE location bits are set to read in progress with the saved modify bit set.

Note that all ties between process B and the global section are broken. The page is now treated like a process copy-on-reference page. The two boxes for process B within the dotted lines in Figure 4.9 are also pictured within dotted lines in Figure 4.4.

MMG$PAGEFAULT initiates a read of the faulted page.

❹ When process A faults the same page, the same steps are taken, this time with a different physical page.

Thus, both process A and process B get the same initial copy of the global page from the global section file, but from that point on, each process has its own private copy of the page to modify.

4.5.4 Global Page-File Section Page

A global page-file section provides a way for processes to share global pages without a backing store file. A global page-file section page is initially faulted as a demand zero page and from then on is indistinguishable from other global writable pages except that its backing store is in a page file.

Figure 4.10 illustrates the transitions of a global page-file section page. The column on the right shows how key PFN information changes as the page moves from one state to another. The numbers in the figure are keyed to the explanations that follow.

The initial conditions are identical to those in Figure 4.8. The section is created; each of its GPTEs contains a zero in the PFN field.

❶ Process A maps the page; the GPTX is stored in its L3PTE.

❷ Process B maps the page; the same GPTX is stored in its L3PTE.

❸ When process B faults this page, MMG$PAGEFAULT locates the GPTE from the GPTX and notes that the page is demand zero. MMG$PAGEFAULT calls MMG_STD$ININEWPFN_DZRO_64 to allocate a free page from the zeroed page list (see Section 4.8.1).

MMG$PAGEFAULT, in concert with MMG_STD$ININEWPFN_DZRO_64, MMG_STD$INCPTREF_64, and MMG_STD$MAKE_WSLE_64 (see Sections 4.8.1 to 4.8.3) makes the following modifications to the pertinent memory management data structures:

a. The PFN$Q_PTE_INDEX and PFN$L_PT_PFN fields for the allocated page locate the GPTE.

b. The PFN$L_PAGE_STATE type bits for the allocated global page are set to global writable.

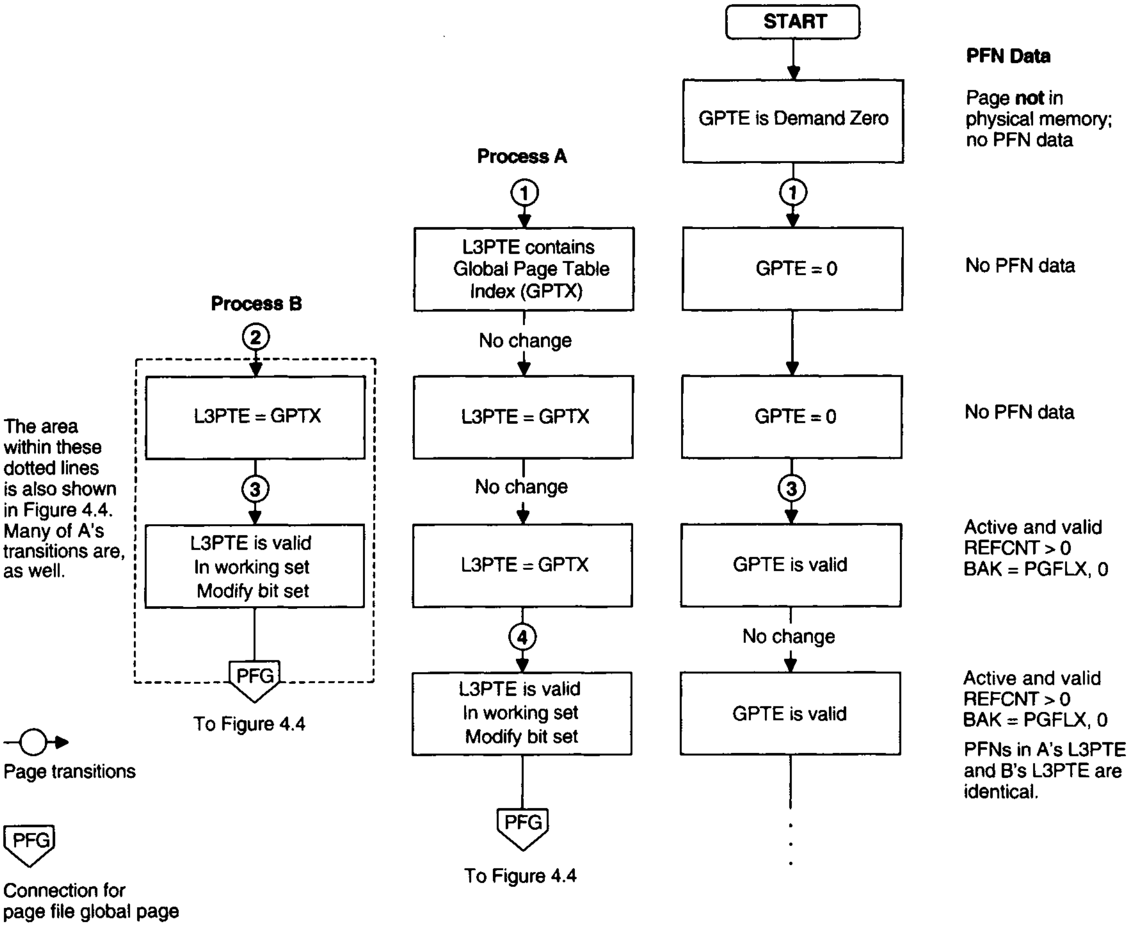
c. An entry in process B's working set list is initialized to describe the faulted page.

d. The share count for the page table page containing process B's L3PTE is incremented. Section 4.4 details other changes to data structures related to the page table page.

e. PCB$L_GPGCNT is incremented.

f. The share and reference counts for the allocated page are incremented.

Figure 4.10 Page Transitions for a Global Page-File Section Page



g. Its PFN$L_PAGE_STATE location bits are set to active.

h. MMG$PAGEFAULT inititializes PFN$Q_BAK to indicate that the page will have page file backing store but none has been assigned yet. Assignment to a page file and allocation of space in it are done later by the modified page writer.

i. It inserts the PFN into the process-private L3PTE, setting the valid and modify bits, and leaving the protection, owner, copy characteristics, and no-execute bits as they were. If the no-execute bit is set, MMG$PAGEFAULT also sets fault-on-execute. If the process is not multithreaded and if MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, it also sets the no-TB-miss-memory-barrier-required bit.

j. It inserts the PFN into the GPTE, setting the valid and modify bits and leaving the protection, owner, copy characteristics, and no-execute bits as they were.

❹ When process A faults the same page, MMG$PAGEFAULT locates the GPTE from the GPTX and finds that the GPTE is valid. It inserts the PFN, valid, and modify bits from the valid GPTE into process A's L3PTE, leaving the protection, owner, copy characteristics, and no-execute bits as they were. If the no-execute bit is set, MMG$PAGEFAULT also sets fault-on-execute. If the process is not multithreaded and if MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, it also sets the no-TB-miss-memory-barrier-required bit.

Transitions for a global page-file section page resemble those of a page located in a page file (see Figure 4.5). However, for a global page-file section page, the GPTE, not the process L3PTE, is affected by the transitions of the physical page. Once the global page is removed from a process's working set, the process L3PTE reverts to the GPTX form.

4.5.5 Memory-Resident Global Demand Zero Section Page

A memory-resident global section is created by the $CREATE_GDZRO or $CRMPSC_GDZRO_64 system service. If the system manager reserved preallocated physical pages for the section, its pages are zeroed during system initialization or section creation, and valid GPTEs are initialized with the PFNs of the allocated pages.

Figure 4.11 illustrates the transitions that occur for a memory-resident global section page that is mapped with shared page tables and that does not occupy preallocated pages. The example is for an 8 MB global section, which can be mapped with a single L3PT.

In the figure, the term VA_PTE represents the combination of fields PFN$L_PT_PFN and PFN$Q_PTE_INDEX. The term ShL3PTE refers to a PTE in a shared L3PT, and the term ShPT, to a shared page table. The numbers in the figure are keyed to the explanations that follow. The column on the right shows how key PFN information changes as the page moves from one state to another.

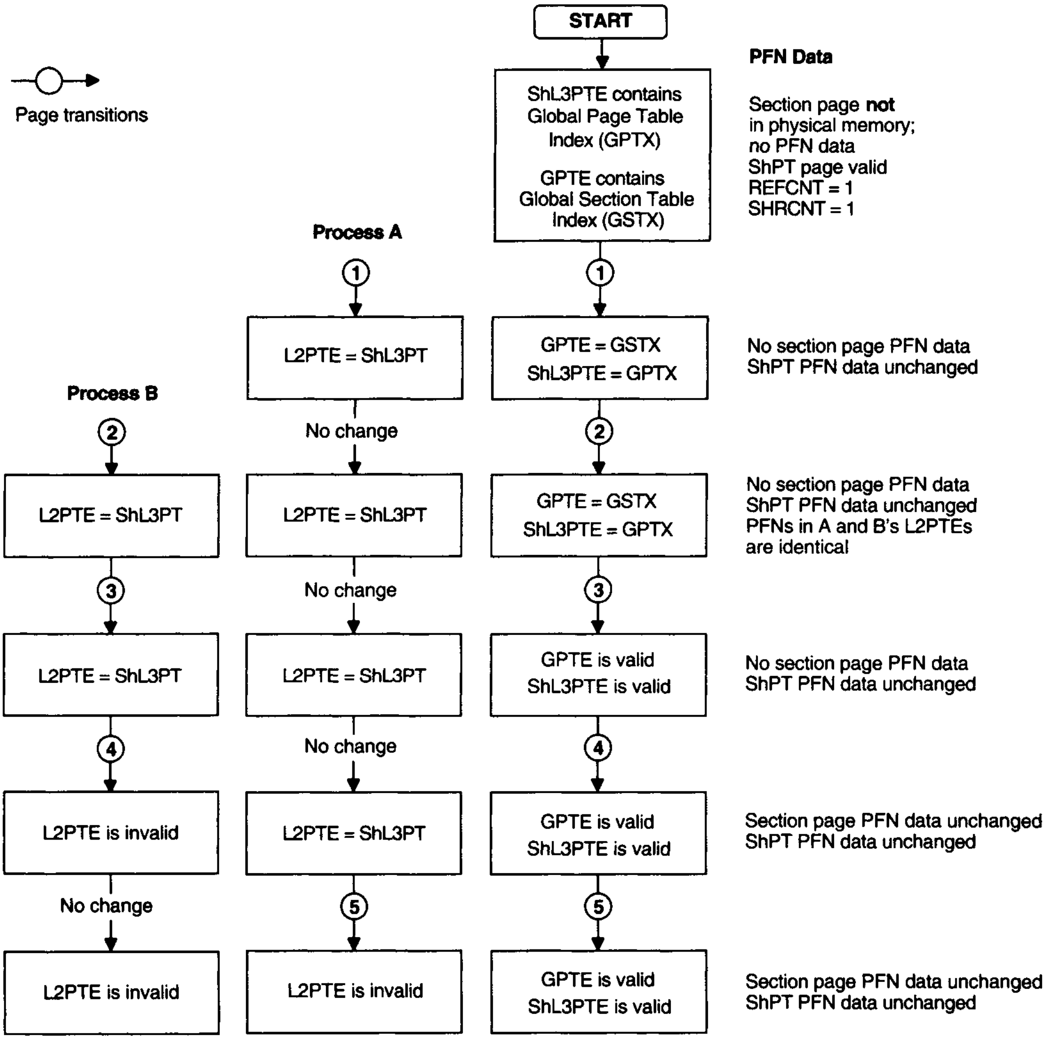
When the section is created, pages are allocated for the shared page tables. Each shared L3PTE is invalid and contains the GSTX of the global section. Each GPTE that maps the memory-resident global section is initialized with a valid bit clear; type 0, type 1, writable, and demand zero bits set; and the GSTX (see Figure 2.26).

❶ Process A maps the global section; its L2PTEs are initialized to map the shared page tables.

❷ Process B maps the global section; its L2PTEs are initialized to the same values.

❸ Process B faults a page in the memory-resident global section. At the first fault of a page in such a section, MMG$PAGEFAULT calls MMG_STD$ININEWPFN_DZRO_64 (see Section 4.8.1). Because no WSLE is needed, the process's L3PT share count is not incremented, nor is its PFN$W_PT_VAL_CNT or PCB$L_GPGCNT.

Figure 4.11 Page Transitions for a Memory-Resident Global Section Page Mapped with Shared Page Tables



When process A accesses the same page, the page mapped by the shared L3PT is already valid. Because the global section page is mapped by a shared page table, the page's share count remains unchanged: the page is mapped by exactly one L3PTE.

❹ When process B deletes the virtual address space occupied by the global section, its L2PTE is cleared, severing the ties to the shared L3PT.

❺ When process A deletes the virtual address space occupied by the global section, its L2PTE is cleared.

The global section and shared page table pages remain valid until the global section is deleted.

4.6 Page Transitions for System Pages

This section describes page faults for pageable system space pages, which are of type PFN$C_SYSTEM:

- Read-only pages from pageable image sections in executive images
- Writable pages from pageable image sections in executive images
- Paged pool pages

The only pageable image sections in system space are from executive images. Although most executive images are nonpageable, some have pageable image sections. In theory, the base images, SYS$BASE_IMAGE.EXE and SYS$PUBLIC_VECTORS.EXE, can contain pageable code and data. In OpenVMS Alpha Version 7.3, however, they have no pageable sections.

By default, when an executive image is mapped, a section table entry in the system section table (which also serves as the global section table) is initialized to describe each pageable section in the image. Each system space L3PTE that maps a page in a pageable section has both type bits set to indicate the process section index form of invalid L3PTE and contains the index of the section's entry in the system section table. Note that it is possible to disable any paging of executive images by setting the SYSGEN parameter S0_PAGING to a nonzero value.

If the section is writable, each of its L3PTEs also has the copy-on-reference and writable bits set. Chapter *The Modular Executive* describes the mapping of executive images in detail.

4.6.1 System Page That Is Not Copy-on-Reference

The transitions for a read-only system section page resemble those described in Section 4.3.1. This section mainly notes the details that differ from those for a process section page that is not copy-on-reference. The numbers that follow correspond to those in Figure 4.3.

❶ MMG$PAGEFAULT locates an entry in the system working set list for the faulted page. It allocates a page from the free page list. There is no need to update data structures describing the page table page that contains the system space L3PTE. System space L3PTEs do not page. PFN$W_PT_VAL_CNT is not maintained for system space page table pages. The page type stored in the PFN$L_PAGE_STATE type bits is system page.

MMG$PAGEFAULT initializes the page's PFN$Q_BAK field from the L3PTE's type and partial section bits and bits <63:32>. It locates the system section table entry just as it would a PSTX and calculates which virtual blocks contain the faulted page.

❷ After the I/O completes, PAGIO, the I/O postprocessing routine, reports a page fault completion event for the kernel thread that faulted the page. PAGIO sets the address space match bit in the L3PTE when setting the valid bit.

❹ The system working set is not subject to purging, swapper trimming, or working set limit adjustment. A page is removed from the system working set list only when space is required for another page. Also, unloading an executive image may result in deletion of pages.

On an SMP system, when a page is removed from the system working set list, the cached L3PTE contents must be flushed from the TBs of all members of the system. Chapter *Symmetric Multiprocessing* describes how the processors cooperate to perform the invalidation.

4.6.2 System Page That Is Copy-on-Reference

The transitions for a copy-on-reference system section page resemble those described in Section 4.3.2 and shown in Figure 4.4.

The page type stored in the PFN$L_PAGE_STATE type bits is system page.

4.6.3 Demand Zero System Page

The transitions for a demand zero system page resemble those described in Section 4.3.3 and shown in the path labeled START 2 in Figure 4.4.

The page type stored in the PFN$L_PAGE_STATE type bits is system page.

4.7 Page Transitions for Global Page Table Pages

In versions of OpenVMS Alpha prior to Version 7.0, global page table pages, which are of type PFN$C_GPGTBL, were pageable. They no longer page in the system working set list. Once faulted into memory, they are locked in memory, typically for the duration of the system boot.

The L3PTEs that map the global page table initially have the demand zero page form of invalid PTE. A nonresident page of global page table may be faulted into memory when one of its GPTEs is allocated for a global section being created. The page table page is locked in memory and is not represented by a WSLE.

The page type stored in the PFN$L_PAGE_STATE type bits is global page table page. The share count for the global page table page is incremented once, for the first GPTE that maps a global section page.

A page table page can be deleted when it no longer maps any global pages, in the unlikely event that it is at the high address end of the global page table and the system manager has dynamically decreased the GBLPAGES SYSGEN parameter from its value at system boot.

4.8 Page Fault Support Routines

Several support routines are used in most types of page faults. These are described in the sections that follow.

4.8.1 MMG_STD$ININEWPFN_64 and MMG_STD$ININEWPFN_DZRO_64

MMG_STD$ININEWPFN_64 and MMG_STD$ININEWPFN_DZRO_64, in module PAGEFAULT, allocate a page of available physical memory. The latter ensures that the page is zeroed first. Their arguments include the address of the virtual page that was faulted, the address of the PTE that maps it, and the address of the working set list in which the virtual page is entered.

As described in Chapter 2, there can be multiple free page lists. On a non-NUMA platform, there are free and zeroed page lists for each page color. Instead of allocating the next available page from the free page list, a page whose color matches the faulting virtual address is allocated.

On a NUMA platform with RAD support enabled, there are free and zeroed page lists for each RAD. Pages can be allocated by several different methods:

- A page from the next RAD in the round-robin sequence (RIH$C_RANDOM_RAD)
- A page from the same RAD as the CPU on which the allocation code is executing (RIH$C_CURRENT_RAD)
- A page from the same RAD as the process for which it is being allocated (RIH$C_HOME_RAD)
- A page from the base RAD, the one on which OpenVMS booted (RIH$C_BASE_RAD)

A method can be specified for process pages, system pages, global pages, and pages allocated by the swapper for inswap. SYSGEN parameter RAD_SUPPORT, whose fields are defined by macro $RIHDEF, has fields to specify an allocation method for each of these categories. If bit RIH$V_SPECIAL in it is clear, default allocation methods are used:

- RIH$C_HOME_RAD for process pages
- RIH$C_CURRENT_RAD for system pages
- RIH$C_RANDOM_RAD for global pages
- RIH$C_RANDOM_RAD for swapper allocation

In addition, when a global section is created by the $CRMPSC_GDZRO_64 or $CREATE_GDZRO system service, it can be associated with a specific RAD to override the allocation method in use for other global sections.

For any method that specifies a particular RAD, if no free memory on that RAD is available at the time of allocation, the page is allocated from another RAD.

MMG_STD$ININEWPFN_64 allocates a page of physical memory from the free page list specified by the allocation method for the page type, the free page list of the appropriate color, or the front of the only free page list. Alternative entry point MMG_STD$ININEWPFN_DZRO_64 allocates from the appropriate zeroed page list; if a zeroed page of the right color or RAD is unavailable, it zeros a free page.

If this is a process page, it stores the location of the L3PTE in the PFN$L_PT_PFN and PFN$Q_PTE_INDEX fields of that page's PFN database record and a type code of process page in PFN$L_PAGE_STATE.

If this is a process page table page, it stores the location of the process-private L2PTE that maps the L3PT in PFN$L_PT_PFN and PFN$Q_PTE_INDEX and initializes the L3PT page's PFN$L_PAGE_STATE type bits to process page table.

If this is a global page, it stores in PFN$L_PT_PFN and PFN$Q_PTE_INDEX the location of the GPTE, rather than that of a process-private L3PTE, and a type code of global page in PFN$L_PAGE_STATE.

If the page is a memory-resident global section page or a global page table page, neither of which is listed in a working set list, it initializes PFN$W_REFCNT and PFN$L_SHRCNT to 1. Otherwise, it calls MMG_STD$MAKE_WSLE_64.

4.8.2 MMG_STD$MAKE_WSLE_64

MMG_STD$MAKE_WSLE_64, in module PAGEFAULT, updates data structures related to the working set list. Its arguments include the address of the virtual page that was faulted, the address of the PTE that maps it, and the address of the working set list in which the virtual page is entered. The WSLE to be used already has the type bits that describe the page.

It initializes the WSLE with the virtual address of the page being faulted and sets its valid bit.

If this is a process page, it stores the index of the WSLE in the PFN$L_WSLX_QW field of the PFN database record for the physical page and increments its PFN$W_REFCNT field to indicate that the contained virtual page is in a working set list. It increments the L3PT's PFN$W_PT_VAL_CNT to indicate another valid WSLE mapped by this page. If this is the first, it increments PHD$L_PTCNTVAL (see Section 4.4). It increments the field PCB$L_PPGCNT to indicate one more process page in the working set.

If this is a process page table page, MMG_STD$MAKE_WSLE stores the index of the WSLE in the PFN$L_WSLX_QW field of the PFN database record for the page table page and increments its reference count to indicate that the page table page is in a working set list.

It increments PFN$W_PT_VAL_CNT of the page table page that maps the page being faulted. If that page table page previously mapped no valid WSLEs, it increments PHD$L_PTCNTVAL to indicate the process has one more page table that maps valid WSLEs. It increments PCB$L_PPGCNT.

If this is a global page, MMG_STD$MAKE_WSLE_64 increments the share count for the page and, if the count makes the transition from 0 to 1, its reference count as well. Working set list index (WSLX) information is not kept for a global page. It increments the process's L3PT's PFN$W_PT_VAL_CNT to indicate another valid WSLE mapped by this page. If this is the first, it increments PHD$L_PTCNTVAL (see Section 4.4.1). It increments the process's PCB$L_GPGCNT to indicate one more global page in the working set. It calls MMG_STD$INCPTREF_64 to lock the process page table that maps the global page into the working set list.

If this is a system page, MMG_STD$MAKE_WSLE_64 stores the index of the WSLE in the PFN$L_WSLX_QW field of the PFN database record for the page. If this is a global page table page, it clears PFN$L_WSLX_QW, because global page table pages are not entered in the system working set list. In either case, it increments its reference count to indicate that the page is in a working set list and PCB$L_PPGCNT in the system PCB.

4.8.3 MMG_STD$INCPTREF_64

MMG_STD$INCPTREF_64 is called with the address of the PTE that maps the virtual page that was faulted to lock the associated page table page into memory.

It first determines whether the page table is process-private, system, or global.

For a process-private or global page table page, it identifies the PFN that the page table page occupies and increments PFN$L_SHRCNT. If the share count had previously been zero, it locks the page table page's WSLE into the working set list by setting its WSL$V_WSLOCK bit, increments PHD$L_PTCNTACT to indicate another active page table page, and increments the PHD reference count (the PHD's entry at the PHV$GL_REFCBAS_LW array).

In the case of a system page, it simply returns: system page table pages do not page.

4.9 $FAULT_PAGE System Service

The Fault Page ($FAULT_PAGE) system service enables an application to initiate page faults prior to actual use of the pages. Because the application is not placed into a wait state, it continues to execute during the I/O to fault the pages into memory.

Its arguments are the starting virtual address to be faulted, the number of bytes to be faulted, and the desired page fault cluster size in bytes.

The system service procedure, EXE$FAULT_PAGE in module SYSSETPRT, runs in the mode of the service requestor. It takes the following steps:

1. It establishes EXE$SIGTORET as a condition handler so that any error signaled by MMG$PAGEFAULT is returned as an error status to the service's requestor (see Chapter *Condition Handling*).

2. It adjusts its input arguments, if necessary, and copies them to the low general registers that are preserved as part of an exception stack frame. In particular, it passes the desired cluster size.
3. At a known location, it tries to access the first page in the first cluster.
   - If the page is invalid, a page fault occurs. MMG$PAGEFAULT allocates physical memory and working set list entries for the desired cluster's worth of pages. It initiates the I/O but does not place the process into page fault wait during the I/O. Instead, it modifies the exception stack frame to record the number of bytes in the I/O request and to advance the program counter so that control will return to the instruction following the page fault. EXE$FAULT_PAGE begins the next cluster at the page following the I/O buffer.
   - If the page is valid, no page fault occurs, and the next cluster begins at the next page.
4. EXE$FAULT_PAGE loops, accessing the first page in the next cluster until there are no more clusters.
5. It returns to its requestor.

4.10 Page Read Clustering

To make reading and writing as efficient as possible, MMG$PAGEFAULT implements a feature called clustering. It checks whether pages adjacent to the virtual page being faulted are located in the same file in adjacent virtual blocks. If so, it requests a multiple-page read so that a cluster of pages will be brought into the working set at one time. One *N*-page request has less CPU and I/O overhead than *N* one-page requests. The special SYSGEN parameter NOCLUSTER determines whether page fault clustering is enabled. Its default value of zero enables clustering.

This section discusses clustering in page read I/O.

The modified page writer and the Update Section File on Disk system services also cluster their write operations, both to make their writes as efficient as possible and to allow subsequent clustered reads for the pages that are being written. Section 4.12.4 summarizes clustering by the modified page writer, and Section 4.13, by the Update Section File on Disk system services.

Table 4.1 indicates the limit to which the object of each type of memory management I/O request is clustered and what determines that limit. For example, when reading a page from a page file, MMG$PAGEFAULT tries to read SYSGEN parameter PFCDEFAULT pages.

When MMG$PAGEFAULT determines that a read is required to satisfy a page fault, it attempts to identify a cluster of pages to be read at once. The manner in which this cluster is formed depends on the initial state of the faulting PTE, as described in the next sections.

4.10.1 Terminating Conditions for Clustered Reads

Beginning with the PTE of the faulting page, MMG$PAGEFAULT scans adjacent PTEs in the direction of higher virtual addresses, checking for adjacent virtual pages that have the same backing store location. It continues until it reaches the desired cluster size or until it reaches one of the following other terminating conditions:

- It encounters a type of PTE different from that of the original faulting PTE (see Section 4.10.2).
- The page table pages that map the next PTE are themselves not valid.
- Another WSLE is not available. (Each page in the cluster must be added to the working set.)
- No physical page is available.

The scan is initially made toward higher virtual addresses because programs typically execute sequentially toward higher virtual addresses and these pages are more likely to be needed soon. If that scan does not form a cluster of at least two pages, MMG$PAGEFAULT scans for pages at lower virtual addresses on the assumption that pages at lower virtual addresses but near the faulting page are likely to be needed soon.

4.10.2 Matching Conditions During the Page Table Scan

The match criterion for adjacent PTEs depends on the form of the initial PTE:

- If the original PTE contains a PSTX, successive PTEs must contain exactly the same PSTX.
- If the original PTE contains a page file page number, successive PTEs must contain PTEs with the same page file index and successively increasing (or decreasing) page numbers.
- If the original PTE contains a GPTX, successive PTEs must contain successively increasing (or decreasing) indexes. In addition, the GPTEs must all contain exactly the same GSTX.

Table 4.1 Cluster Factor in I/O Requests Issued by Memory Management

Type of I/O Request	Cluster Factor
	Process Page Read
Page in section file	*pfc*/PFCDEFAULT[1]

[1] The cluster factor for a private or global section can be specified at link time or when the cluster is mapped by explicitly declaring a cluster factor (*pfc*). If it is unspecified, the SYSGEN parameter PFCDEFAULT is used.

Table 4.1 *(continued)* Cluster Factor in I/O Requests Issued by Memory Management

Type of I/O Request	Cluster Factor
Process Page Read	
Page in page file	PFCDEFAULT[2]
Page table page	PAGTBLPFC[2]
$FAULT_PAGE–induced fault	Cluster argument
System Page Read	
System section page[3]	SYSPFC[2]
Paged pool page	PFCDEFAULT[2]
Global Page Read	
Global page	*pfc*/PFCDEFAULT[1]
Global copy-on-reference page	*pfc*/PFCDEFAULT[1]
$FAULT_PAGE–induced fault	Cluster argument
Modified Page Write	
To page file	MPW_WRTCLUSTER[2]
To private section file	MPW_WRTCLUSTER[2]
To global section file	MPW_WRTCLUSTER[2]
To swap file (set bit PFN$V_SWPPAG)	1
Update Section File on Disk System Service Write	
Private section	MPW_WRTCLUSTER[2]
Global section	MPW_WRTCLUSTER[2]
Swapper I/O	
Swapper I/O	n/a

[1] The cluster factor for a private or global section can be specified at link time or when the cluster is mapped by explicitly declaring a cluster factor (*pfc*). If it is unspecified, the SYSGEN parameter PFCDEFAULT is used.

[2] This is a SYSGEN parameter.

[3] Pageable executive routines originate in executive image sections, described by section table entries in the system header.

4.10.3 Maximum Cluster Size for Page Read

The maximum number of pages that can make up a cluster is a function of the type of page being read:

- The cluster factor for process page table pages is taken from PHD$L_PGTBPFC. The default value of this field is the special SYSGEN parameter PAGTBLPFC, whose default value is 16 pagelets, resulting in a cluster factor of one page. Increasing this value is likely to have a negligible effect on most systems.
- The cluster factor for pages read from a page file is taken from the PFL$L_PFC field of the page file control block (see Figure 2.31). The usual contents of this field are the value of SYSGEN parameter MPW_WRTCLUSTER.
- The cluster factor for pages read from a process or global section file is taken from the SEC$L_PFC field of the process or global section table entry (see Figure 2.7). This field usually contains zero, in which case the default page fault cluster is used. Just as for clustered reads from the page file, this default is taken from PHD$L_DFPFC.

  There are two methods by which the cluster factor of a process or global section can be controlled. At link time, the page fault cluster factor in an image section descriptor can be set to nonzero through the linker cluster option and its PFC argument:

  ```
  CLUSTER = cluster-name,[base-address],pfc,file-spec[,...]
  ```

  Second, the page fault cluster factor for a section mapped through the $CRMPSC system service can be specified through the optional PFC argument. The page fault cluster factor for a section created through the $CREATE_GFILE, $CRMPSC_FILE_64, or $CRMPSC_GFILE_64 system service can be specified through the optional FAULT_CLUSTER argument

4.11 Page Read Completion

The I/O postprocessing routine IOC$IOPOST, in module IOCIOPOST, detects page read completion when the flags IRP$V_PAGIO and IRP$V_FUNC in IRP$L_STS are both set.

Page read completion is not reported to the faulting kernel thread in the normal fashion with a special kernel mode AST because none of the postprocessing has to be performed in the context of the faulting kernel thread. Holding the MMG spinlock, the I/O postprocessing routine PAGIO performs the postprocessing needed. It performs the following steps for each page in the page fault cluster that was successfully read:

1. PAGIO decrements the reference count in the page's PFN database record, indicating that the read in progress has completed.
2. If the reference count is now zero, it puts the page into the free or modified page list, depending on the value of the saved modify bit, and continues with the next page.

3. If the reference count is nonzero, it sets the location bits in PFN$L_PAGE_STATE to active.
4. It sets the valid bit in the L3PTE. If the page is writable and was faulted with write intent, it sets the modify bit in the L3PTE to avoid the need for a modify fault. If the process is not multithreaded, if MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, and if this is a process-private address, it also sets the no-TB-miss-memory-barrier-required bit in the L3PTE. For an S0/S1 space page, it also sets the address space match bit.
5. If the page is a global page that is not copy-on-reference, the valid bit set in step 4 was actually in the GPTE. In this case, the process (slave) L3PTE must also be altered: PAGIO inserts the PFN, partial section, type 0, global, global write, and valid bits from the GPTE into the slave L3PTE. If appropriate, it sets the modify bit in the slave L3PTE.
6. If the page is a process page table, PAGIO decrements the PHD reference count to indicate that the I/O is complete (PHV$GL_REFCBAS_LW array element).

After processing the pages that were read successfully, PAGIO tests whether an I/O error occurred. If so, it takes the following steps for the page that incurred the error:

1. PAGIO decrements the reference count in the page's PFN database record, indicating that the read in progress has completed.
2. It changes the page's PFN$L_PAGE_STATE location bits to read error, setting the delete-contents bit and clearing the saved modify bit.
3. It records the I/O error status in PFN$W_SWPPAG. When the process later refaults this page, MMG$PAGEFAULT will return SS$_PAGRDERR to SCH$PAGEFAULT, which will generate a condition to report the actual I/O error to the access mode that incurred the page fault (see Section 4.2.2).
4. If the virtual page is a copy-on-reference page, PAGIO restores its backing store location to the physical page's PFN$Q_BAK field. If the error occurred on the last page of the transfer, and that page was partially backed, it clears the partial section flag in PFN$Q_BAK.
5. If the page's reference count is now zero and the process is memory-resident, PAGIO releases the page to the free page list. If the process is outswapped, PAGIO inserts the page into the bad page list instead. When the process is inswapped, the page will be removed from the bad page list.

After tending to the individual pages, PAGIO determines whether the pages are from a copy-on-reference section. If so, it subtracts the number of pages read from the section's reference count.

PAGIO tests whether an upcall to a user mode thread manager was made when the kernel thread faulted this page. If so, it transforms the IRP into a user mode AST control block (ACB) for the thread manager and queues an AST to it (see Chapter *Kernel Threads* for a description of upcalls and user mode thread management). Otherwise, it reports the scheduling event page fault completion for the page faulting kernel thread so that it is made computable. The priority increment value is zero; that is, there is no boost to the kernel thread's scheduling priority. If any of the pages just read were collided pages, it also makes kernel threads in the collided page wait state computable. Collided pages are discussed in Section 4.17.3.

If an error occurred and more of the transfer remains to be done, PAGIO updates the IRP to describe the rest of the transfer (excluding any pages already done and the page that incurred the error) and requeues the IRP to the device driver.

4.12 Modified Page Writing

Once a second as well as in response to particular events, the executive checks whether any of the swapper's tasks need to be performed and wakes it if necessary; one such task is writing pages from the modified page list to mass storage.

The modified page writer, MMG$WRTMFYPAG, in module WRTMFYPAG, is a subroutine of the swapper process. Within its main loop, the swapper calls MMG$WRTMFYPAG to form a cluster of modified pages that have the same backing store and request a write I/O operation. Writing multiple modified pages together makes more efficient both the write to backing store and any subsequent refault into memory.

At completion of the write I/O operation, the modified page writer's special kernel mode AST routine is entered to place the pages into the free page list and, if appropriate, to initiate the writing of more modified pages.

4.12.1 Requesting the Modified Page Writer

During system operation other executive routines request the writing of modified pages by invoking the routine MMG$PURGE_MPL, in module WRTMFYPAG, with arguments identifying the requested operation and its scope. The possible operations are

- Writing pages within a virtual address range (an SVAPTE request)
- Writing pages mapped by a particular page table page (a PAGE_TABLE request)
- Writing pages to shrink the modified list to a target size (called a MAINTAIN request)
- Writing all pages backed by section files (an OPCCRASH request)

Modified page writing is requested in a number of circumstances:

- When the modified page list has exceeded its high limit, defined by the SYSGEN parameter MPW_HILIMIT (MAINTAIN)
- When the free page list is below its low limit, defined by the SYSGEN parameter FREELIM, and can be replenished by writing modified pages (MAINTAIN) in preference to outswapping a resident process

- When particular modified pages must be written to their backing store (SVAPTE and PAGE_TABLE)
- When the OPCCRASH image, running during system shutdown, must write all pages in the list that are backed by section files to their backing store (OPC-CRASH)

Originally, the modified page list was sometimes emptied, or flushed, during normal operations. In VAX VMS Version 5, the flushing was replaced by selective purging, that is, writing all modified pages whose PTEs fall within a specified system virtual address range (the SVAPTE request).

For selective purging of process-private space modified pages, in OpenVMS Alpha, the PTEs of interest are L2PTEs or L3PTEs. In OpenVMS Alpha Version 7.0 and later releases, each process's page tables are mapped in the same process-private virtual address range; an SVAPTE request must therefore include the process's page table base register (PR$_PTBR) contents to identify which process's pages are to be written. For global and other shared PTEs, the PR$_PTBR recorded is the primary processor's hardware privileged context block (HWPCB) PR$_PTBR. For selective purging of writable global pages, the PTEs of interest are GPTEs.

Selective purging is requested under the following circumstances:

- When a process body has been outswapped but its PHD, whose slot is needed, cannot be outswapped because some of its L2 or L3PTEs map transition pages on the modified page list (an SVAPTE request; see Chapter 6)
- When a writable global section with transition pages still on the modified page list is deleted (an SVAPTE request; see Chapter 3)
- When a process needs to reuse a WSLE that describes a dead page table page, one that is now inactive but still maps transition pages on the modified page list (an SVAPTE or PAGE_TABLE request; see Chapter 5)

The modified page writer may be requested multiple times before it is actually called by the swapper. MMG$PURGE_MPL therefore stores the requested command with the highest rank in MPW$GL_STATE; from low to high rank, the ordering is MAINT_STATE (from the MAINTAIN command), SELECTIVE (from the SVAPTE and PAGE_TABLE commands), and CRASH_STATE (from the OPCCRASH command). It records information about each request:

- For an SVAPTE request, MMG$PURGE_MPL increments MPW$GL_REQCNT, the number of outstanding SVAPTE requests. It checks whether there is already a request outstanding to purge this page table and, if not, stores the low and high SVAPTE addresses and corresponding PR$_PTBR contents in a 32-entry table beginning at local symbol MPW$GQ_SVAPTE. It summarizes the page table base registers for which SVAPTE requests have been made in MPW$GQ_PTBR_MASK by setting the bit corresponding to the low eight bits of the address. Use of this summary mask is described in Section 4.12.2.

Unless this request is the result of shrinking a process's working set list (MPW$V_NO_MPL_FLUSH is set in MPW$GL_FREWFLGS), it clears SCH$GL_MFYLOLIM and SCH$GL_MFYLIM.

- For a PAGE_TABLE request, MMG$PURGE_MPL increments MPW$GL_PT_REQCNT, the number of outstanding PAGE_TABLE requests. It checks whether there is already a request outstanding to purge this page table and, if not, stores the PFN of the page table in a 32-longword array beginning at local symbol MPW$GQ_PT. Unless this request is the result of shrinking a process's working set list (MPW$V_NO_MPL_FLUSH is set in MPW$GL_FREWFLGS), it clears SCH$GL_MFYLOLIM and SCH$GL_MFYLIM.
- In response to a MAINTAIN request, MMG$PURGE_MPL records the target size in SCH$GL_MFYLOLIM and SCH$GL_MFYLIM. (If a previous MAINTAIN request has been made, MMG$PURGE_MPL uses the lesser of its target size and the current target size.) It clears MPWGL_REQCNT, MPWGL_PT_REQCNT, and MPW$GQ_PTBR_MASK.
- For an OPCCRASH request, MMG$PURGE_MPL clears MPW$GL_REQCNT, SCH$GL_MFYLOLIM, and SCH$GL_MFYLIM so that all pages on the modified page list will be flushed.

Once modified page writing to shrink the list (MAINTAIN) is initiated, the modified page writer continues writing modified pages until the size of the list is at or below the contents of SCH$GL_MFYLOLIM. The modified page writer typically compares the target modified page list size with the value of the SYSGEN parameter MPW_LOLIMIT and uses the larger as a target size. Chapter 6 describes the calculation of the target modified page list size for the different circumstances in which the swapper initiates modified page writing.

When an SVAPTE, PAGE_TABLE, or OPCCRASH request initiates modified page writing to purge or flush the list, both the lower and upper limits for the modified page list are set to zero. For an SVAPTE request, the modified page writer scans the entire list and writes all pages whose PTE addresses fall within the specified range. For an OPCCRASH request, the modified page writer scans the entire list and writes all pages not backed by a page file. For a PAGE_TABLE request, the modified page writer scans the target page table for PTEs indicating modified pages and writes those pages to their backing store.

Before the modified page writer exits, it restores its two limits to the values contained in the SYSGEN parameters MPW_HILIMIT and MPW_LOLIMIT.

4.12.2 Operation of the Modified Page Writer

Called by the swapper, the modified page writer initiates the writing of modified pages. The modified page writer forms a cluster and queues an I/O request. When the I/O request completes, the modified page writer's special kernel mode AST routine is entered. After performing necessary processing on the pages that have been written, it checks whether more modified pages must be written and, if so, forms another cluster.

At the completion of that request, the special kernel mode AST routine may queue yet another request.

The modified page writer can initiate up to SYSGEN parameter MPW_IOLIMIT concurrent I/O requests. The default value of MPW_IOLIMIT is 4. As described in Chapter 2, during system initialization MPW_IOLIMIT nonpaged pool data structures are allocated. Each contains an IRP and two arrays that describe the pages in the cluster. These structures are queued to a listhead at MPW$GL_IRPFL and MPW$GL_IRPBL. Figure 4.12 shows the layout of this data structure, known as a modified page writer I/O request packet (MPW IRP).

MMG$WRTMFYPAG proceeds in the following fashion:

1. It compares the number of pages on the modified page list to SCH$GL_MFYLIM. If there are fewer pages on the list, it simply exits.
2. It acquires the MMG spinlock, raising IPL to IPL$_MMG.
3. It sets bit SCH$V_MPW in SCH$GL_SIP to indicate that modified page writing is in progress. If the bit was already set, MMG$WRTMFYPAG releases the MMG spinlock and exits.
4. Otherwise, it tests whether this is a request to do selective purging that includes SVAPTE requests and, if so, whether 1 second has elapsed since the previous one. If not, it clears SCH$V_MPW in SCH$GL_SIP, releases the MMG spinlock, and exits. This test helps limit the time the modified page writer spends scanning the modified page list. If the selective purging is limited to PAGE_TABLE requests, the modified page writer does not delay them and continues with the next step.
5. MMG$WRTMFYPAG calls MMG$PURGE_MPL, specifying the default command of MAINTAIN to shrink the list to MPW_LOWAITLIMIT pages.

   — If a previous SVAPTE or PAGE_TABLE request has been made and not yet satisfied, MMG$PURGE_MPL returns immediately.

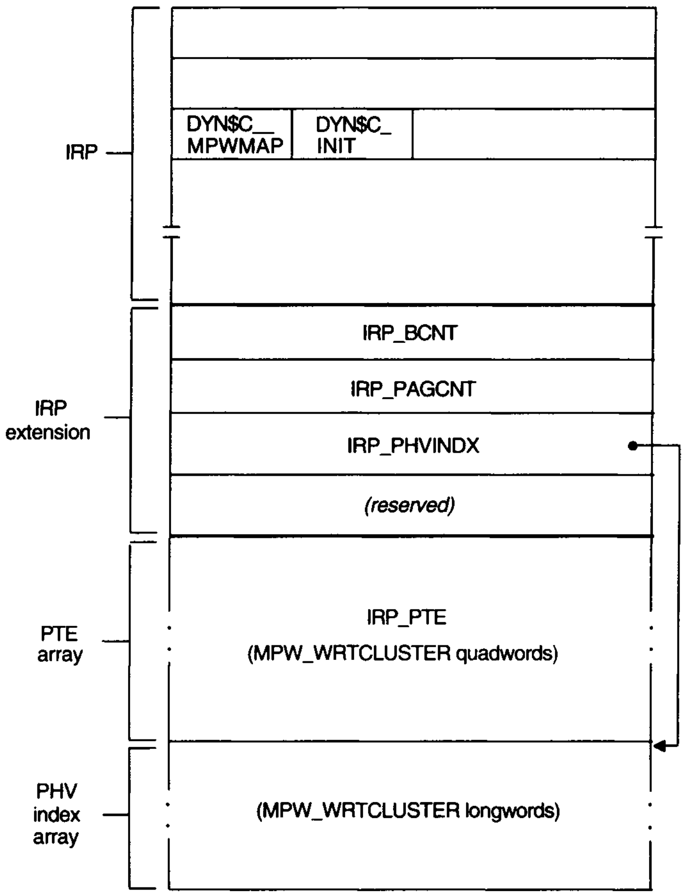
   — If no previous SVAPTE, PAGE_TABLE, or other MAINTAIN requests have been made, it changes MPW$GL_STATE to MAINTAIN and stores the larger of MPW_LOWAITLIMIT and SCH$GL_MFYLOSV in SCH$GL_MFYLIM and SCH$GL_MFYLOLIM.

   — If a previous MAINTAIN request has been made, it stores the lesser of the previous and current requested limits in SCH$GL_MFYLIM and SCH$GL_MFYLOLIM.

   This step establishes the default for modified page writing if no unsatisfied requests have been made.
6. If there are pending PAGE_TABLE requests, MMG$WRTMFYPAG acts on each of them by taking the following steps:

   a. It confirms that the page is still a process page table, that it maps no valid pages, but that it does map transition pages. If any one of the tests fails, it ignores that page and proceeds with the next PAGE_TABLE request. (After

Figure 4.12 Layout of a Modified Page Writer IRP (MPW IRP)



the PAGE_TABLE request was made, the process could have executed and faulted in pages mapped by this page table.)

b. It maps the page table page into system space and scans its PTEs, looking for non-null entries. If it finds a PTE that is not a transition page, this page table is not a candidate to be written to a page file, and MMG$WRTMFYPAG proceeds with the next page.

c. If it finds a transition PTE describing a page on the modified page list, it allocates an IRP and processes the page and possibly a cluster of other pages mapped by this page table page, following steps 10 through 19.

d. It reacquires the MMG spinlock and confirms that the page is still a page table page mapping no valid entries. The page's state could have changed while MMG was unlocked. If the state has changed, it goes on to the next PAGE_

TABLE request. If the state is unchanged, it continues scanning the page table page.

When all the PAGE_TABLE requests have been processed, if there are pending SVAPTE requests, MMG$WRTMFYPAG continues with the next step. Otherwise, it continues with step 21.

7. MMG$WRTMFYPAG removes an MPW IRP from the list. If none is available, it continues with step 21.

8. It scans the modified page list to find a page with which to begin a cluster. Entered the first time, it begins with the first page on the list. Subsequently, it typically resumes with the page at which the last scan stopped. If that page is no longer on the modified page list, MMG$WRTMFYPAG tries the pages that preceded and followed it on the list. If neither of them is still on the list, it selects the first page on the list.

   From the page's PFN database, it determines the page type (for example, process, system, or global), the virtual address of the PTE that maps the page, and the physical address of the corresponding L1PT. In the case of a process page, it maps into system space the page table page containing that PTE.

   Its processing of the modified page depends on the type of request it is performing (the value of MPW$GL_STATE):

   — If performing a MAINTAIN request, it accepts the page.

   — If performing an SVAPTE request, it tests MPW$GQ_PTBR_MASK to see whether the corresponding L1PT address matches any of the requested ranges and, if so, whether the address of the page's PTE falls within that range. If not, it goes on to the next page in the list.

   — If performing an OPCCRASH request, it accepts the page.

9. MMG$WRTMFYPAG determines the type of the first page in the cluster from its PFN database record PFN$L_PAGE_STATE type bits.

10. Based on the page type, it gets the address of the relevant PHD, either that of a process or of the system.

11. It examines the PFN$Q_BAK field to determine the type of backing store: page file, section file, or swap file page (see Section 4.12.5).

12. If the backing store is in a page file, MMG$WRTMFYPAG first checks whether the system is shutting down. If so, writing the page to a page file is pointless, and it skips the page. It continues with the next page in the list.

    If the system is not shutting down, it then tests whether this page is a process page table page containing all zeros that can be deleted now. If so, it deallocates the page's page file backing store, decrements the share count of the next level page table page, restores page file quota to the process, removes the page from the modified page list, severs its connection to the process's next level page table, and inserts it on the zeroed page list. If MMG$WRTMFYPAG is processing PAGE_

TABLE requests, it continues at step 6d with the next PTE. Otherwise, it continues at step 9 with the next page on the modified page list.

If the page is not a process page table page that can be deleted now, MMG$WRTMFYPAG tests whether its last attempt to allocate space in a page file failed. If so, it rejects this page as a starting point and goes on to the next page in the modified page list, continuing with step 9. The allocation failure information is cleared each time MMG$WRTMFYPAG is called. If the last attempt to allocate space in a page file was successful, MMG$WRTMFYPAG allocates a cluster of page file pages (see Section 4.12.6).

13. Unless the backing store is a swap file page, MMG$WRTMFYPAG tries to form a cluster of pages, as described in Section 4.12.5. It scans adjacent PTEs looking for transition PTEs that map pages on the modified page list, until either the desired cluster size is reached or one of the other terminating conditions described in Section 4.12.4 is reached.

    Except for PAGE_TABLE requests, it scans first toward lower virtual addresses and then toward higher virtual addresses. This scan begins first toward smaller virtual addresses for the same reason that the page read cluster routine begins toward larger addresses. Given that the program is more likely to reference higher addresses, it would be inefficient to initiate a write operation only to have the page immediately faulted and likely modified again. The modified page writer writes first those pages with a smaller likelihood of being referenced in the near future.

    In handling a PAGE_TABLE request, it only scans forward: the goal is to write all the modified pages mapped by that page table.

14. When it can no longer cluster, it records the PTEs and their associated PHD vector indexes in the MPW IRP.

15. If the cluster is one of page file pages, MMG$WRTMFYPAG updates the PFN$Q_BAK field for each page to show the actual page file page allocated.

16. It removes each page from the modified page list, decrementing SCH$GL_MFYCNT to show one less modified page.

17. It changes the PFN$L_PAGE_STATE location bits for each page to write in progress and also clears the saved modify bit. It increments the reference count for each page to reflect the I/O in progress. If the page is a page table page, MMG$WRTMFYPAG also increments the PHV$GL_REFCBAS_LW array element corresponding to the PHD.

18. It releases the MMG spinlock, fills in the MPW IRP, and queues it to the backing store driver.

19. MMG$WRTMFYPAG reacquires the MMG spinlock and, if SCH$GL_MFYCNT is less than SCH$GL_MFYLOLIM, goes to step 8 to try to form another cluster of pages to write.

20. In local routine MMG$_MPW_END, the modified page writer performs end processing. Depending on the operation performed, the modified page writer may declare as available the resource RSN$_MPWBUSY or the resource RSN$_MPLEMPTY. If no modified page write I/O requests are outstanding, it clears SCH$V_MPW in SCH$GL_SIP.
21. The modified page writer releases the MMG spinlock and processes any global section descriptors (GSDs) on the delete pending list, possibly queuing a kernel mode AST to the creator of each global section. (Writing modified pages to backing store may enable the deletion of such global sections.) Chapter 3 describes this processing in detail.

Whenever a modified page write request completes, MMG$WRTMFYPAG's special kernel mode AST routine is entered. Section 4.12.3 describes this routine.

4.12.3 Modified Page Write Completion

The modified page writer's special kernel mode AST routine, WRITEDONE in module WRTMFYPAG, takes the following steps:

1. It acquires the MMG spinlock, raising IPL to IPL$_MMG.
2. It deallocates the MPW IRP to its own lookaside list. (Holding the MMG spinlock blocks any possible allocation from the list, so it is safe for WRITEDONE to continue to access IRP fields after deallocating it.)
3. It examines the characteristics of each page in the cluster:
    a. If the page is a page table page, it decrements the PHV$GL_REFCBAS_LW array element corresponding to that PHD.
    b. If the page's backing store was a swap file page, WRITEDONE clears PFN$V_SWPPAG_VALID in the PFN$L_PAGE_STATE field to indicate that the contents of PFN$W_SWPPAG are no longer valid.
    c. It decrements the reference count for the page. If the count goes to zero, it places the page into the free page list.
    d. If the RPTEVT bit in the PFN$L_PAGE_STATE field is set, WRITEDONE reports a page fault completion scheduling event for the kernel thread that owns the page. This bit is set when deletion of the page has been stalled while it is being written to its backing store.
4. If there are pending PAGE_TABLE requests, WRITEDONE processes them, rejoining the flow described in Section 4.12.2 at step 7.
5. If SCH$GL_MFYCNT is greater than SCH$GL_MFYLOLIM, WRITEDONE attempts to form another MPW cluster, rejoining the flow described in Section 4.12.2 at step 8.

4.12.4 Modified Page Write Clustering

The modified page writer scans the page table that contains the modified page being processed, attempting to form a cluster of adjacent virtual pages with the same backing store. It scans in both directions from the page being processed, except for PAGE_TABLE requests, for which the scan is from the beginning of the page table page to its end. The terminating conditions for the scan include the following:

- The PTE does not indicate a transition page.
- The PTE indicates a page in transition, but the physical page is not on the modified page list.
- Bit PFN$V_SWPPAG_VALID in the page's PFN$L_PAGE_STATE field is set. Such a page is treated in a special way by the modified page writer.
- The PFN$Q_BAK field for the first page in the cluster and the page in question indicate that their backing store location is a process or global section file, but the section indexes are not the same.
- In the case of a PAGE_TABLE request, the end of the page table page is reached.
- The next page table page is not valid, implying that there are no transition pages in that page table page.

In OpenVMS versions prior to Version 7.3, a page that would need page file backing store was assigned to a page file when the virtual page was first faulted into memory. Clustering of pages to be written to a page file terminated when a page was reached that was assigned to a different page file than that of the first page in the cluster. In OpenVMS Version 7.3, page file assignment is deferred until a modified page is about to be written.

The maximum size of a modified page write cluster depends on its type (see Section 4.12.5).

4.12.5 Backing Store for Modified Pages

The modified page writer attempts to cluster when writing modified pages to their backing store addresses. It encounters three different clustering situations for the three possible backing store locations.

The set bit PFN$V_SWPPAG_VALID in PFN$L_PAGE_STATE indicates that the process has been outswapped and this page remained behind, probably as the result of an outstanding read request. The modified page writer writes a single page to the swap file page whose number is in PFN$W_SWPPAG. It does not attempt to cluster because virtually contiguous pages in an I/O buffer are unlikely to be adjacent in the outswapped process body. The process body is outswapped with pages ordered as they appear in the working set list, not in virtual address order. A description of how the PFN$W_SWPPAG field is loaded is found in Chapter 6, where the entire outswap operation is discussed.

If the backing store address is in a section file, the modified page writer creates a cluster up to the value of the SYSGEN parameter MPW_WRTCLUSTER. Any of the terminating conditions listed in the previous section can limit the size of the cluster. If the last page in the section is in the cluster and the partial section bit is set in its L3PTE, the modified page writer calculates the I/O request byte count such that the last page's contribution to the count includes only those pagelets that have backing store.

If the backing store address is in a page file, adjacent pages with page file backing store are written at the same time. The modified page writer attempts to allocate MPW_WRTCLUSTER pages in the most appropriate page file. The desired cluster factor is reduced to the number of pages actually allocated. Section 4.12.6 describes allocation of space within a page file.

The cluster created for a write to a page file consists of several smaller clusters, each representing a series of virtually contiguous pages (see Figure 4.13):

1. The modified page writer creates a cluster of virtually contiguous pages.
2. If the desired cluster size has not yet been reached, the modified page list is searched until another page with page file backing store is found.
3. Pages virtually contiguous to this page form the second minicluster that is added to the eventual cluster to be written to the page file.
4. The modified page writer continues in this manner, building a large cluster that consists of a series of smaller clusters, until either the cluster size is reached or no more pages on the modified page list have page file backing store. Each smaller cluster can terminate on any of the conditions listed in the previous section, or on the two terminating conditions for the large cluster.

4.12.6 Page File Space Allocation

Before the modified page writer searches for more pages to form a cluster bound for a page file, it must determine the maximum size of the write cluster, namely the number of contiguous page file pages, up to a maximum of MPW_WRTCLUSTER, that can be allocated.

The modified page writer calls MMG_STD$ALLOC_PAGSWP_PAGES, in module PAGE_FILE, to allocate a cluster of pages in the most appropriate page file. Each page file is described by a nonpaged pool data structure called a page file control block (PFL). Chapter 2 shows the PFL (see Figure 2.31) and describes its fields.

MMG_STD$ALLOC_PAGSWP_PAGES takes the following steps:

1. If there are no PFLs on any of the four lists of page files at MMG$GA_PAGE_FILES (see Chapter 2), it issues the following message on the console terminal and returns:

   ```
   %SYSTEM-W-NOPAGEFILE, no page file installed; system trying
   to continue
   ```

2. If there are no PFLs on the first three lists, it issues the following message and returns:

   ```
   %SYSTEM-W-PAGEFILEFULL, all page or swap files are full;
   system trying to continue
   ```

3. Beginning with the first nonzero pointer in the first two lists, it selects the first PFL in that list. It moves the pointer to the next PFL in the list to rotate use of page files.

4. If it had to select a PFL from the second list because there were no page files with at least one cluster's worth of adjacent pages or if fewer than one-fourth of the total installed page file pages are free, it issues the following message:

   ```
   %SYSTEM-W-PAGEFRAG, page file filling up; please create more space
   ```

5. If it had to select a PFL from the third list because there were no page files with set bits in the directory bitmap or if fewer than one-sixteenth of the total installed page file pages are free, it issues the following message:

   ```
   %SYSTEM-W-PAGECRIT, page file nearly full; system trying to continue
   ```

   Each of the preceding four messages is issued only once between the time the system is booted and the time it is shut down.

6. It tries to allocate the number of pages represented by the file's PFL$L_CUR_ALLOC_EXPO. The field is initialized to represent the minimum of 1,024 and SYSGEN parameter MPW_WRTCLUSTER. Its value is expressed as an index into the directory quadword counter array at PFL$L_DIR_CLUSTER (see Chapter 2).

   Beginning at the quadword specified by PFL$L_STARTBYTE, MMG_STD$ALLOC_PAGSWP_PAGES scans the directory bitmap for a quadword with that many adjacent bits set. If the starting set bit is bit 0 in a quadword, and the adjacent high bits in the preceding quadword are also set, it adjusts the starting directory bit position to begin at the high bits in the preceding quadword.

7. Starting at the bit position in the storage bitmap corresponding to the starting bit position in the directory bitmap, it checks for adjacent high bits set in the preceding cluster. If there are any, it adjusts the final start bit for the allocation.

8. It then tries to allocate as many pages as were requested and clears the corresponding bits in the storage and directory bitmaps.

9. It updates PFL$L_DIR_CLUSTER, PFL$L_STARTBYTE, and the counters that describe the state of the file as well as those that summarize all files.

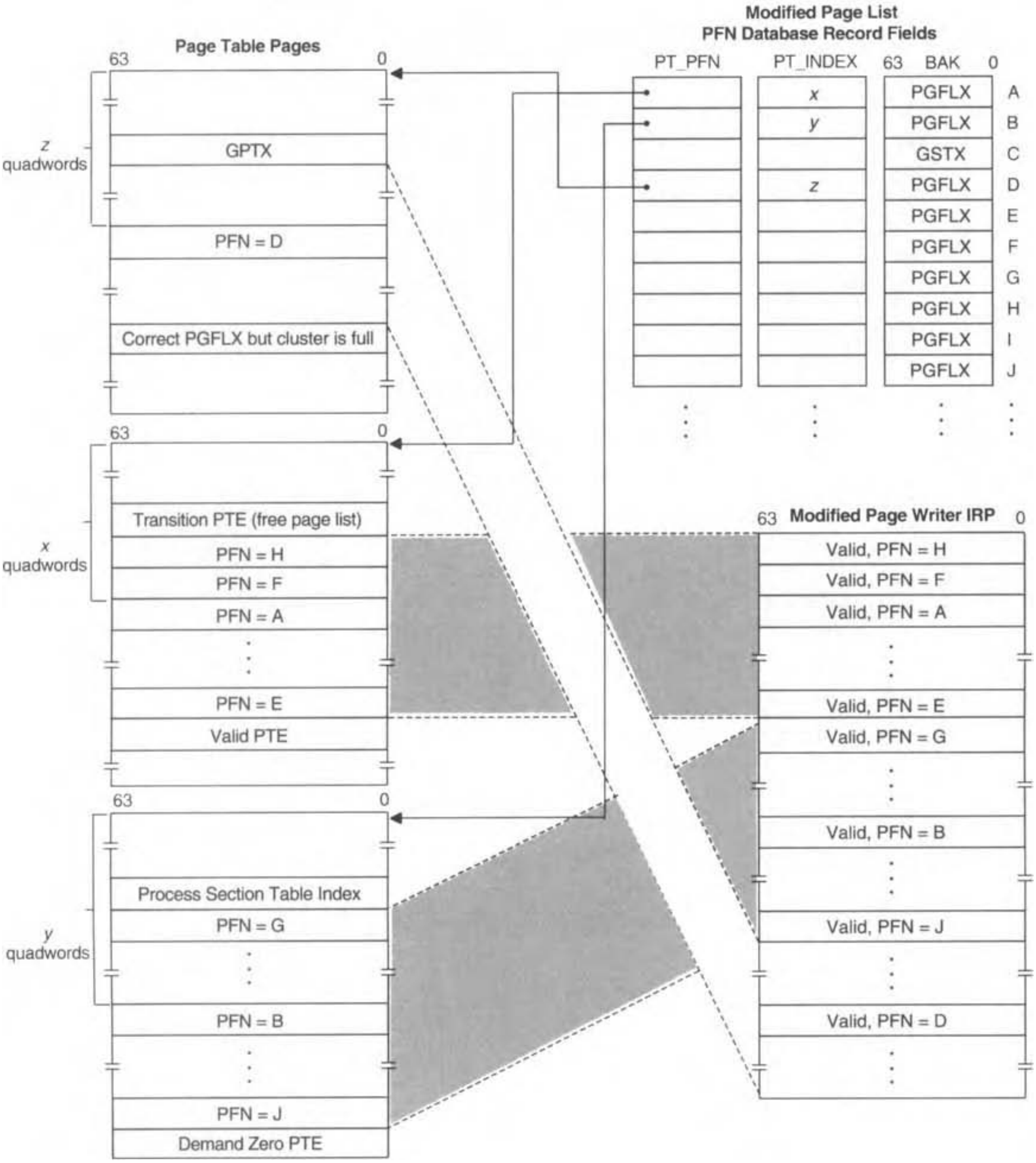
10. It returns the number of pages allocated to its caller.

4.12.7 Example of Modified Page Write to a Page File

Figure 4.13 illustrates a sample cluster for writing to a page file. The modified page list and fields in the PFN database, pictured in the upper right-hand corner of the figure, are shown as sequential arrays to simplify the figure.

1. The first page on the modified page list is PFN A. The modified page writer maps into system space the page table page that maps PFN A (A's PFN$L_PT_PFN contents) and starts with the PTE that maps it (PFN$Q_PTE_INDEX). By scanning backward through the page table page, the modified page writer locates first PFN F and then PFN H. The L3PTE preceding the one that contains PFN H is also a transition PTE, but the page is on the free page list. This page terminates the backward search.
2. The modified page writer IRP begins with PFN H, PFN F, and PFN A. The search now goes in the forward direction, with each page that has page file backing store added to the map up to and including PFN E. Because the next L3PTE is valid, the first minicluster is terminated.

Figure 4.13 Clustered Write to a Page File



3. The next page on the modified page list, PFN B, leads to mapping a different page table page and adding a second cluster to the map. This cluster begins with PFN G and ends with PFN J. The backward search was terminated with an L3PTE containing a section table index. The forward search terminated with a demand zero PTE.

   Note that this second cluster consists of pages belonging to a different process than that of the first cluster. The difference is reflected in the process header vector index array, which contains a longword element for each L3PTE in the map (see Figure 4.12).

4. The next page on the modified page list is PFN C. This page belongs in a global section file and is skipped during the current scan.

5. PFN D leads to a third cluster that is terminated in the backward direction by an L3PTE that contains a GPTX. The search in the forward direction terminates when the desired cluster size is reached, even though the next PTE was bound to the same page file. The cluster size is either MPW_WRTCLUSTER or the number of adjacent pages available in the page file, whichever is smaller. In any case, this cluster will be written with a single write request.

6. Note that reaching the desired cluster size resulted in leaving some pages on the modified page list bound for the same page file, such as PFN I.

4.13 Update Section File on Disk System Services

The Update Section File on Disk ($UPDSEC[W] and $UPDSEC_64[W]) system services enable a process to write a specified range of pages in a process or global section to their backing store in a controlled fashion, without waiting for the modified page writer to do the backup. They are especially useful for frequently accessed pages that may never be written by the modified page writer because they are always being faulted from the modified page list back into the working set before they are backed up.

These system services are a cross between modified page writing and a normal write request. In particular, they resemble modified page writing in that the services write a cluster of adjacent virtual pages to backing store to enable the cluster to be faulted in at a later time. In other words, both modified and unmodified pages can be written. By default all pages in the range are eligible to be written.

In the case of global pages, determining which pages have been modified is not feasible. The system service runs in the context of one process and can scan its PTEs for set modify bits. However, to determine whether a particular global page has been modified requires looking at the PFN database and the PTEs of all processes mapped to this page. (The state of the saved modify bit in the GPTE does not necessarily reflect the state of the page.) Because there are no back pointers for valid global pages, this information is unavailable. Therefore, all pages in a global section are eligible to be written to their backing store location, regardless of whether the pages have been modified.

The default value of 0 in the UPDFLG argument specifies that all read/write global section pages are to be written to backing store, whether or not they have been modified. A value of 1 specifies that the requestor is the only or the last process having the global section mapped for write access and that, of the global section pages, only the modified ones should be written to the section file on disk.

As for any I/O request, the requestor can request completion notification with an event flag and I/O status block or an AST.

The cluster factor is the minimum of MPW_WRTCLUSTER and the number of pages in the input range.

4.13.1 $UPDSEC System Service

The pages eligible to be written are specified by the INADR argument. The direction of search for pages is determined by the order in which the INADR address range is specified.

The system service procedure EXE$UPDSEC, in module SYSUPDSEC, runs in kernel mode. It checks the validity of the input address range, clears the event flag associated with the I/O request, charges the process direct I/O quota, and allocates nonpaged pool to serve as an extended I/O packet. The pool is used to queue one or more modified page write I/O requests and to keep track of how much of the section the service has processed.

EXE$UPDSEC locates the region descriptor entry (RDE) corresponding to the INADR argument's starting address.

EXE$UPDSEC then calls MMG$CREDEL, in module SYSCREDEL, specifying UPDSECPAG_RDE, in module SYSUPDSEC, as the per-page service-specific routine. Chapter 3 describes the actions of MMG$CREDEL and its use of per-page service-specific routines. Routines UPDSECQWT_64, PTEPFNMFY, MMG$WRT_PGS_BAK, and MMG$UPDSECAST, all in module SYSUPDSEC, are also part of this system service.

UPDSECPAG_RDE calls UPDSECQWT_64 to form the first cluster and to initialize and queue the IRP to the driver for the backing store device.

UPDSECQWT_64 takes the following steps:

1. It touches the first page table page that maps pages in the specified range to fault it into the working set list.
2. It acquires the MMG spinlock, raising IPL to IPL$_MMG.
3. It scans in the specified direction of the range for the first candidate page to meet all the following conditions:
   - Its owner access mode is not more privileged than that of the service requestor.
   - It is not part of a memory-resident section, Galaxy global section, or PFN-mapped section.

 — It is a valid or transition process or global page.
 — It is writable but not copy-on-reference.
 — It has been modified.

4. Having found one candidate page, it calls MMG$WRT_PGS_BAK.
5. MMG$WRT_PGS_BAK scans the process's page table in the specified direction for adjacent pages that have similar characteristics; in particular, the backing store for the pages must be the same. The adjacent pages do not necessarily have to have been modified but they do all have to be valid or transition, that is, resident.

   It tries to form a cluster of up to MPW_WRTCLUSTER pages. In the case of process pages, the cluster begins with the first modified page in the range. In the case of global pages, if the UPDFLG is clear, the cluster begins with the first global page. By setting the low bit of the UPDFLG parameter, the requestor can indicate that it is the only process whose modified pages should be written. In that case, the process's L3PTEs and the PFN database are used to select candidate pages for backing up. Only pages modified by this process can be the beginning pages of a cluster.

   The cluster is terminated by failure to meet the constraints previously listed for selecting the first page. In addition, any of the following conditions terminate the cluster:

    — A page with different backing store
    — More than one adjacent unmodified page

      This constraint, new with OpenVMS Alpha Version 7.2, improves performance by minimizing the number of unmodified pages written to backing store. It is particularly helpful for large sections with scattered modified pages.

6. Having formed a cluster, MMG$WRT_PGS_BAK modifies the PFN database records for the pages in it. It increments the PFN$W_REFCNT field for each page. If the page is on the free or modified page list, it removes it from the list and changes its PFN$L_PAGE_STATE location bits to write in progress and clears the saved modify bit. If the page was valid, it also clears the modify bit in the PTE. If the page is writable from any access mode, it sets the fault-on-write bit in the PTE so that a subsequent write can trigger resetting the modify bit.
7. If the last page in the section is in the cluster and has the PTE$V_PARTIAL_SECTION bit set, MMG$WRT_PGS_BAK calculates the I/O request byte count such that the last page's contribution to the count includes only those pagelets that have backing store.
8. It initializes an IRP, releases the MMG spinlock, and queues the I/O request to the backing store driver.

When the write completes, I/O postprocessing code decrements the PFN$W_REFCNT for each page and queues a special kernel mode AST to the kernel thread that requested the $UPDSEC system service. The AST routine MMG$UPDSECAST first checks whether all the pages requested by the system service call have been written or whether another write is required. To perform the check, it calls UPDSECQWT_64, which forms another cluster and queues another write request if necessary. If all requested pages have been written, MMG$UPDSECAST enters the normal I/O completion path involving event flags, I/O status blocks, and user-requested ASTs, thus notifying the kernel thread.

4.13.2 $UPDSEC_64 System Service

The $UPDSEC_64 system service is requested to write pages in a section to their backing store. It resembles the $UPDSEC system service, but all its address arguments are 64 bits. Thus it can be used to update a section in P0, P1, or P2 space.

The number of pages written is specified by the LENGTH_64 argument. The direction of search for modified pages is determined by whether the starting virtual address's region is ascending or descending.

The $UPDSEC_64 system service procedure, EXE$UPDSEC_64 in module SYS_UPDSEC_64, runs in kernel mode. It resembles EXE$UPDSEC, using the same routines: MMG_STD$UPDSECQWT_64, an alternative entry point to UPDSECQWT_64; PTEPFNMFY, MMG$WRT_PGS_BAK, and MMG$UPDSECAST, all in module SYSUPDSEC.

4.14 Input and Output That Support Paging

There is little special-purpose code in the I/O subsystem to support page and swap I/O. MMG$PAGEFAULT and the swapper each build their own IRPs but queue these packets to a device driver in the normal fashion. These are the only differences:

- Special Queue I/O Request ($QIO) entry points for page and swap I/O (in module SYSQIOREQ) bypass many of the usual $QIO checks to minimize overhead.
- An IRP describing a page or swap request is distinguished from other IRPs by a flag in IRP$L_STS. These flags are detected by the I/O postprocessing routine, which dispatches to special completion paths for page read and other types of memory management I/O.

Tables 4.2 to 4.4 summarize the I/O requests issued by memory management components. The first table lists the type of paging or swapping I/O, the priority of each such request, the relevant process identification, and information about the priority boost the process receives at I/O completion. For more information on priority classes and boosts, see Chapter *Scheduling*.

Table 4.2 Summary of I/O Requests Issued by Memory Management, Part I

Type of I/O Request	Priority IRP$B_PRI	Process ID IRP$L_PID	Priority Boost
Process page read Global copy-on-reference page read Process page table read	Base priority of faulting kernel thread	PID of faulting kernel thread	0
System page read Global non-copy-on-reference page read	Base priority from system PCB—16	PID of faulting kernel thread	0
Modified page write	MPW_PRIO[1]	PID of swapper[2]	None[3]
Update section write	Base priority of requestor	PID of requestor	2
Swapper I/O	SWP_PRIO[1]	PID of swapper	None[3]

[1] This is a SYSGEN parameter.

[2] The modified page writer is a subroutine of the swapper process.

[3] The swapper is a real-time process and is therefore not subject to priority boosts.

Tables 4.3 and 4.4 list more information about each type of I/O request, summarizing the uses to which the memory management components put several fields in the IRP. Some of these fields overlay standard fields that are not required for their more typical uses. Thus the space can be used for storing other information needed by these components. The column WCB Source specifies from which memory management data structure the address of the window control block (WCB) is obtained. This address is stored in the field IRP$L_WIND.

In Table 4.3 the columns PARAM_0, PARAM_1, and PARAM_3 describe the contents of the IRP fields IRPQ_PARAM_0, IRPQ_PARAM_1, and IRP$Q_PARAM_3 for each type of read operation requested by the memory management subsystem. The PTE column identifies the type of PTE that maps the pages to be read.

In addition to those fields, the contents of IRP$Q_PARAM_2 record whether MMG$PAGEFAULT informed a user mode thread manager of a page fault. A value of –1 indicates no upcall was made. Any other value is the virtual address of the faulted page and indicates that PAGIO (see Section 4.11) should inform the thread manager of page fault completion.

Table 4.3 **Summary of I/O Requests Issued by Memory Management, Part II (Read Requests)**

Page Type	PTE	PARAM_0	PARAM_1	PARAM_3	WCB Source
		Process Page Read			
Section file	L3PTE	–1	0/PSTX[1]	0	PSTE
Page file	L3PTE	–1	0	0	PFL
Page table	L2PTE or L1PTE	–1	0	0	PFL[2]
		System Page Read			
System section[3]	System space L3PTE	–1	0	0	SSTE
Paged pool	System space L3PTE	–1	0	0	PFL
		Global Page Read			
Global	GPTE	Slave PTE address	0	Page table PFN	GSTE
Global copy-on-reference	L3PTE	Slave L3PTE index	GPTX	0	GSTE

[1] If the page is copy-on-reference, IRP$Q_PARAM_1 contains the PSTX.

[2] Process page tables originate as demand zero pages whose backing store is a page file.

[3] Pageable executive code originates in executive images, described by section table entries in the system header, abbreviated here as SSTE.

Table 4.4 lists write requests. For Update Section File on Disk I/O, IRP$PQ_ACB64_AST and IRP$Q_ACB64_ASTPRM contain the AST address and parameter specified by the system service requestor. For modified page writer and swapper I/O, IRP$L_IIRP_P1 contains the address of the special kernel mode AST routine. The PTE column identifies the type of PTE involved in the I/O request. In most cases it is an L3PTE in a process's P0, P1, or P2 page table. In other cases, the PTE is contained in an array within an MPW IRP (see Figure 4.12).

The AST column identifies the procedure that runs at I/O completion. WRITEDONE, the modified page writer's special kernel AST, is described in Section 4.12.3. IODONE, the swapper's special kernel AST routine, is described in Chapter 6.

Table 4.4 Summary of I/O Requests Issued by Memory Management, Part III (Write Requests)

Type of I/O Write Request	PTE	AST	ASTPRM	WCB Source
		Modified Page Write		
To page file	MPW IRP	WRITEDONE	0	PFL
To private section file	MPW IRP	WRITEDONE	0	PSTE
To global section file	MPW IRP	WRITEDONE	0	GSTE
To swap file (nonzero SWPPAG)	MPW IRP	WRITEDONE	0	PFL
		Update Section Write		
Private section	L3PTE	AST address	AST argument	PSTE
Global section	GPTE	AST address	AST argument	GSTE
		Swapper I/O		
Swapper I/O	Swapper map	IODONE	0	PFL

4.15 Reference Counts

Much of the memory management subsystem's activity is asynchronous, initiated in response to process actions but completed in other contexts. The memory management database keeps track of the current state of various structures and resources through reference counts. Certain count transitions trigger additional memory management subsystem activity. This section summarizes those reference counts and the activities triggered by their transitions. These counts are mentioned throughout this and the other memory management chapters.

Table 4.5 lists these reference counts with a brief description of each. The sections that follow describe each count in more detail.

Table 4.5 Memory Management Reference Counts

Reference Count Location	Meaning
PFN$L_REFCNT	Number of reasons the current contents of physical page should stay in memory
PFN$L_SHRCNT for process page table page	Number of valid and transition pages it maps

Table 4.5 *(continued)* **Memory Management Reference Counts**

Reference Count Location	Meaning
PFN$L_SHRCNT for global page table page	Number of GPTEs that map global section pages
PFN$L_SHRCNT for global page	Number of L3PTEs that are mapped to it
PHD$L_PTWSLELCK array element	Number of locked WSLEs and window PTEs mapped by this process page table page
PHD$L_PTWSLEVAL array element	Number of valid WSLEs mapped by this process page table page
PHD$L_PTCNTACT	Number of active page table pages with nonzero PTEs
PHD$L_PTCNTLCK	Number of page table pages with non-negative PHD$L_PTWSLELCK counts
PHD$L_PTCNTVAL	Number of page table pages with non-negative PHD$L_PTWSLEVAL counts
PHV$GL_REFCBAS array element	Number of reasons the PHD should remain resident

4.15.1 PFN$W_REFCNT

PFN$W_REFCNT counts the number of reasons a physical page should retain its current contents. A value of zero means the associated page is on the free, modified, zeroed, or bad page list.

The count is incremented for the following reasons:

- The associated process-private virtual page is added to a working set list.
- PFN$L_SHRCNT of a global page makes the transition from 0 to 1, when it is placed in a process working set list.
- The associated virtual page is being faulted in.
- A page is locked as part of a direct I/O buffer with I/O in progress.
- A section page is being written to its backing store by the Update Section File on Disk system services or the modified page writer.
- The associated virtual page is part of a buffer object.
- The associated virtual page is a process page table page whose PFN$W_BO_REFC is greater than –1.
- A page is being outswapped as part of a process header or body.
- A page has just been inswapped as part of a process header or body.

PFN$W_REFCNT is decremented for the following events:

- The associated virtual page is removed from a working set list.
- PFN$L_SHRCNT of a global page makes the transition from 1 to 0, when it is no longer part of any process working set list.
- Page fault I/O completes.
- Any other type of direct I/O completes when the buffer pages are unlocked.
- Update section I/O completes.
- Modified page write I/O completes.
- A buffer object is deleted.
- PFN$W_BO_REFC transitions to –1, indicating that a process page table page no longer maps any buffer objects.
- Outswap completes.

When a page's reference count transitions from 1 to 0, the page is inserted into the free or modified page list, depending on the state of its saved modify bit.

4.15.2 PFN$L_SHRCNT and PHD$L_PTCNTACT

As shown in Table 4.5, the meaning of PFN$L_SHRCNT depends on the type of virtual page occupying the physical page.

4.15.2.1 Process Page Table Pages

For an L3PT that maps process-private pages PFN$L_SHRCNT is the number of valid and transition process-private pages it maps (excluding buffer object pages in the release pending state). For an L1PT or L2PT that maps process-private page table pages, PFN$L_SHRCNT is the number of valid and transition process-private page table pages it maps. A count of zero means the page table page maps no valid and no transition pages.

The count is incremented for the following events:

- A physical page is allocated for a virtual page being faulted that is mapped by this page table, whether the page table is an L1PT, L2PT, or L3PT.
- A WSLE is created to map a global page.
- An L3PT that maps a direct I/O buffer too large to be described by a DIOBM is locked into memory when a system space window is created to double-map the L3PTEs that describe the buffer.
- A buffer object page in release pending state is faulted and made valid.
- A buffer object page in release pending state is made valid and locked as part of locking down a direct I/O buffer.
- A buffer object page mapped by that page table page is being deleted (but the virtual address space that it occupied still exists).

- An unmodified global demand zero page is materialized rather than faulted in to expedite deletion of the page.

When the share count for a page table page transitions from 0 to 1, the page table is considered active. The executive locks it into the process working set by setting its WSL$V_WSLOCK bit and increments PHD$L_PTCNTACT to indicate one more process page table page mapping valid or transition pages and also increments the appropriate PHV$GL_REFCBAS_LW array element.

The share count is decremented for the following events:

- A global or buffer object page mapped by that page table page is removed from the process working set list.
- Contents of a virtual page mapped by that page table page are deleted and the associated physical page is deallocated.
- An L3PT that maps a direct I/O buffer too large to be described by a DIOBM is unlocked from memory at I/O completion.
- When the process body has been outswapped, each process page is deleted.
- An empty page table is moved from the modified page list to the zeroed page list.

When the count transitions from 1 to 0, the executive decrements PHD$L_PTCNTACT and the appropriate PHV$GL_REFCBAS_LW array element and clears the WSL$V_WSLOCK bit.

4.15.2.2 Global Pages and Global Page Table Pages

For a global page table page, PFN$L_SHRCNT is 1 if the global page table page maps any global section pages; otherwise, it is 0. Because global page table pages are no longer pageable, there is no reason to maintain an accurate count of the number of global pages each global page table page maps.

For a global page, PFN$L_SHRCNT is the number of L3PTEs that map to the global page. The count is incremented for the following events:

- A WSLE is created to map a global page.
- At inswap, a process is reconnected to a valid global page.

The count is decremented for the following events:

- A global page is removed from a process's working set.
- After outswap, a process is disconnected from a valid global page.
- Prior to outswap, a process is disconnected from a valid writable global page.

When the share count for a global page transitions from 0 to 1, the executive increments PFN$W_REFCNT to indicate one more reason for the page to remain resident. When the count transitions from 1 to 0, the executive decrements PFN$W_REFCNT.

4.15.2.3 System Pages

One other use is made of PFN$L_SHRCNT for system pages. The count records the number of times a particular page has been locked into the system working set through the routine MMG$LOCK_SYSTEM_PAGES, in module LOCK_SYSTEM_PAGES. When the count transitions from 0 to 1, the routine sets the WSLE's WSL$V_WSLOCK bit and increments the system header's PHD$L_PTCNTACT.

Such a page is unlocked through routine MMG$UNLOCK_SYSTEM_PAGES, in the same module. When the share count transitions from 1 to 0, the routine clears the entry's WSL$V_WSLOCK to unlock it from the system working set and decrements PHD$L_PTCNTACT. Chapter 5 describes these routines.

4.15.3 PFN$W_PT_VAL_CNT and PHD$L_PTCNTVAL

For a page that maps process-private pages, PFN$W_PT_VAL_CNT contains the number of valid pages mapped by that page table page, excluding pages in memory-resident global sections. A value of –1 means the page is not a page table page or that it maps no such pages. This count is only kept for process-private page tables. In the case of the L1PT, its count includes itself.

The count is incremented for the following events:

- A page mapped by that page table page is faulted into the working set list.
- A transition page mapped by that page table page is added to the working set list so that it can be locked as part of a direct I/O buffer.

When the count transitions from –1 to 0, the executive increments PHD$L_PTCNTVAL to indicate one more process page table page mapping valid pages.

The count is decremented for the following events:

- A page mapped by that page table page is removed from the working set list.
- A valid, unmodified process page mapped by that page table page is being deleted.

When the count transitions from 0 to –1, the page table page is considered dead; the executive decrements PHD$L_PTCNTVAL.

4.15.4 PFN$W_PT_LCK_CNT and PHD$L_PTCNTLCK

For a page that maps process-private pages, PFN$W_PT_LCK_CNT contains the number of locked pages mapped by that page table page. A value of –1 means the page table page maps no such pages (or is not currently in use as an L3PT).

The count is incremented when a page mapped by that page table page is locked into the working set list or into memory.

The count is decremented when a page mapped by that page table page is unlocked from the working set list or from memory.

When either PFN$W_PT_LCK_CNT or PFN$W_PT_WIN_CNT transitions from –1 to 0, the executive increments PHD$L_PTCNTLCK to indicate one more process page table page mapping locked or window pages. When either count transitions from 0 to –1, the executive decrements PHD$L_PTCNTLCK.

4.15.5 PFN$W_PT_WIN_CNT

For a page that maps process-private pages, PFN$W_PT_WIN_CNT contains the number of window and memory-resident global section pages mapped by that page table page. A window page is a virtual page that is a double mapping of a physical page. For example, a virtual page in a process or global section mapped by PFN is a window page.

For a shared L3PT, PFN$W_PT_WIN_CNT contains the number of pages mapped by the shared L3PT. A shared L3PT is counted as a window page for the process-private L2PT that maps it.

A value of –1 in PFN$W_PT_WIN_CNT means the page table page maps no such pages (or is not currently in use as a page table).

The count is incremented for the following events:

- A virtual page is created that is a window page and that is mapped by that page table page.
- A PFN-mapped section is created and mapped with a granularity hint region
- A page of shared L3PT is mapped by that L2PT.
- A memory-resident section mapped by that shared L3PT is created.

The count is decremented for the following events:

- A window page or PFN-mapped page mapped by that page table page is deleted.
- A page of shared L3PT mapped by that L2PT is deleted.

When either PFN$W_PT_LCK_CNT or PFN$W_PT_WIN_CNT transitions from –1 to 0, the executive increments PHD$L_PTCNTLCK to indicate one more process page table page mapping locked or window pages. When either count transitions from 0 to –1, the executive decrements PHD$L_PTCNTLCK.

4.15.6 PHV$GL_REFCBAS_LW Array Element

PHV$GL_REFCBAS_LW contains the address of an array with one element for each balance set slot (see Chapter 2). Each element counts the number of reasons the current PHD must continue to occupy that balance set slot, that is, the number of process page table pages tightly connected to that PHD slot. A value of –1 for an element means the corresponding balance set slot does not contain any PHD. A value of 0 means that the slot has been assigned to a process.

A PHV$GL_REFCBAS_LW element is incremented for the following reasons:

- PHD$L_IOREFC, the number of process-private buffers locked down for I/O, transitions from 0 to 1.
- A process page table page is being faulted in from a page file.
- A process page table page maps valid or transition pages.
- A process page table is being written to a page file by the modified page writer.

Typically, either MMG$DECPHDREF or MMG$DECPHDREF1, both in module PAGEFAULT, is called to decrement a PHV$GL_REFCBAS_LW element. A PHV$GL_REFCBAS_LW element is decremented for the following events:

- PHD$L_IOREFC, the number of process-private buffers locked down for I/O, transitions from 1 to 0.
- Page fault I/O for a process page table page completes.
- A process page table page's PFN$L_SHRCNT transitions to 0, indicating the page table maps no valid or transition pages.
- Modified page write I/O for a process page table page completes.

When the count transitions to 0, the swapper is awakened to clean up the slot so that it is available for another process.

4.16 Use of Page Files

During system initialization and operation, one or more page files are placed into use. In OpenVMS versions prior to Version 7.3, when a process was created, it was assigned to a page file, and space in that page file was reserved for it. When the process faulted a copy-on-reference or demand zero page, the page was charged against the reserved space. Allocation of particular blocks in the page file was deferred until the modified page writer actually prepared to write the page. A process could be assigned concurrently to as many as four page files during its lifetime.

As of OpenVMS Version 7.3, a process is not assigned to particular page files; instead, it can page in any installed page file. A virtual page is not associated with a page file until the modified page writer allocates space to hold the page. After the page is faulted back in, if modified, it can be written to a different page file.

A PFL (see Chapter 2) describes each page file in use. Space in a page file is managed in units the size of an Alpha page. Section 4.12.6 describes the allocation of actual pages in the page file.

When process pages backed by a page file are deleted, MMG_STD$DEALC_PAGSWP_PAGES, in module PAGE_FILE, is called to deallocate the associated page file pages, if any. It updates PFLL_REFCNT, PFLL_FREPAGCNT, and MMG$GQ_PAGEFILE_ALLOCS to reflect fewer pages in use. It sets the corresponding bits in the storage bitmap and, if an entire cluster of pages is newly available, the corresponding bit in the directory bitmap. If the directory bitmap is updated, it updates the PFL$L_DIR_CLUSTER counts accordingly.

If PFL$L_REFCNT is decreased to zero and a deinstall request is pending, it calls MMG_STD$DINSPAGSWPFIL, in module PAGEFILE, to deinstall the page file.

4.17 Paging and Scheduling

Page fault handling can influence the scheduling state of kernel threads in several ways. If a read is required to satisfy a page fault, the faulting kernel thread is placed into a page fault wait state or a collided page wait. If a resource such as physical memory is not available, the kernel thread is placed into an appropriate wait state. The kernel thread waits with its program counter (PC), processor status (PS), and other registers reflecting its state at the time it executed the instruction that generated the page fault.

Chapter *Scheduling* describes scheduling, wait states, priority increment classes, resource waits, and the reporting of scheduler events.

4.17.1 Page Fault Wait State

A kernel thread is placed into page fault wait when a read is required to resolve a page fault. The I/O postprocessing routine PAGIO detects that a page read has completed and reports the scheduling event page fault completion for the kernel thread. As a result, the kernel thread is removed from the page fault wait state and made computable. No priority boost is associated with page fault read completion.

4.17.2 Free Page Wait State

If not enough physical memory is available to satisfy a page fault, the faulting kernel thread is placed into a free page wait state. Whenever a page is deallocated and the free page list was formerly empty, routine MMG$DALLOC_PFN, in module ALLOCPFN, checks for kernel threads in this state. It reports the scheduling event free page available so that each kernel thread in the free page wait state is made computable.

MMG$DALLOC_PFN makes no scheduling decision about which kernel thread will get the page. There is no first-in/first-out approach to the free page wait state; rather, all kernel threads waiting for the page are made computable. The next kernel thread to execute will be the highest priority resident computable kernel thread.

4.17.3 Collided Page Wait State

It is possible for a page fault to occur for a page that is already being read from its backing store. If the page is anything but a process page, or if it is a process page of a multithreaded process, the page is referred to as a collided page. The collided bit is set in the PFN$L_PAGE_STATE field, and the kernel thread is placed into the collided page (COLPG) wait state.

When the page fault I/O is complete, the page read completion code in PAGIO checks if the collided bit was set for any page in the cluster just read. If so, and if the pages are not process pages, it reports the scheduling event collided page available for each kernel thread in that wait state. It does not check whether a kernel thread is waiting for the collided page that was faulted in. If the pages are process pages, PAGIO reports the scheduling event only for kernel threads of the same process.

The lack of checking has two advantages:

- No special code determines which kernel thread executes first. All kernel threads are made computable, and the normal scheduling algorithm selects the kernel thread that executes next.
- The probability of a collided page is small. The probability of two different collided pages is even smaller. If a kernel thread waiting for another collided page is selected for execution, that kernel thread will incur a page fault and be placed back into the collided page wait state. Nothing unusual occurs, and the operating system avoids a lot of special-case code to handle a situation that rarely, if ever, occurs.

4.17.4 Resource Wait States

Several types of resource wait are associated with memory management. A kernel thread waiting for one of these resources is placed into the miscellaneous wait state (see Chapter *Scheduling*) until the resource is available.

Early versions of the VAX/VMS operating system also could place a process into a wait for resource RSN$_SWPFILE (RWSWP). When a process was unable to increase its swap file allocation to accommodate a larger working set, it was placed into this resource wait until space became available in the swap file. The timing and form of swap file allocation changed, and this resource wait is not used by the OpenVMS Alpha executive.

Versions of OpenVMS prior to Version 7.3 could also place a kernel thread into a wait for resource RSN$_PGFILE (RWPFF) when it faulted a modified page with page file backing store out of its working set and the page file had not been initialized yet. Page file assignment is now made when the page is being written from the modified page list, and this resource wait is no longer used by the OpenVMS Alpha executive.

4.17.4.1 Resource Wait for RSN$_ASTWAIT (RWAST)

A kernel thread that faults a page is placed into this wait when the kernel thread has no direct I/O quota left against which the page fault I/O request can be charged.

4.17.4.2 Resource Wait for RSN$_NPDYNMEM (RWNPP)

A kernel thread that faults a page is placed into this wait when MMG$PAGEFAULT is unable to allocate nonpaged pool for an IRP for the page fault I/O.

4.17.4.3 Resource Wait for RSN$_MPWBUSY (RWMPB)

A kernel thread that faults a modified page out of its working set may be placed into this wait when any of the following is true:

- The modified page list contains more pages than the SYSGEN parameter MPW_WAITLIMIT.
- The modified page list contains more pages than the SYSGEN parameter MPW_LOWAITLIMIT and the modified page writer is active, writing modified pages.
- A page table page that maps no valid pages is being removed from the working set list, and modified page writing is required to sever the connections between the modified page list and transition pages mapped by the page table page (see Chapter 5 for more details on dead page table pages).

The kernel thread is not placed into this wait unless all the following conditions are also true:

- The process holds no mutexes.
- The process is not the swapper process.
- Bit MMG$V_NOWAIT in MMG$GL_FREWFLGS is clear.
- One or more page files have been installed.

The modified page writer declares the availability of the resource RSN$_MPWBUSY in processing a MAINTAIN request when it has written enough modified pages so that the list is left with MPW_LOWAITLIMIT or fewer pages. Also, if the modified page list size drops below the current high limit for the list when a page is faulted from it, and if the modified page writer is not currently active, resource RSN$_MPWBUSY is declared available. Otherwise, a process could be hung waiting for that resource until there is enough activity to increase the list size above the limit again.

4.17.4.4 Resource Wait for RSN$_MPLEMPTY (RWMPE)

A kernel thread in RWMPE is waiting for the modified page writer to signal that it has flushed the modified page list. The only kernel thread currently placed into this wait is one executing the OPCCRASH image, which forces a flush of the modified page list prior to stopping the system.

4.18 Relevant Source Modules

Source modules described in this chapter include

[LIB]RIHDEF.SDL
[SYS]ALLOCPFN.MAR
[SYS]EXCEPTION.M64
[SYS]EXCEPTION_ROUTINES.MAR
[SYS]IOCIOPOST.MAR
[SYS]IOLOCK.MAR
[SYS]PAGE_FILE.C
[SYS]PAGEFAULT.MAR
[SYS]PAGEFILE.MAR
[SYS]SCHEDULER.M64
[SYS]SYS_UPDSEC_64.C
[SYS]SYSLKWSET.MAR
[SYS]SYSUPDSEC.MAR
[SYS]SYSVA_ALLOC.C
[SYS]WRTMFYPAG.MAR

This Page Intentionally Left Blank

Chapter 5
Working Set List Dynamics

"Then you keep moving round, I suppose?" said Alice.
"Exactly so," said the Hatter, "as the things get used up."
"But what happens when you come to the beginning again?"
Alice ventured to ask.
"Suppose we change the subject," the March Hare interrupted,
yawning. "I'm getting tired of this. I vote the young lady
tell us a story."

Lewis Carroll, *Alice's Adventures in Wonderland*

The pages of physical memory in use by a process are called its working set. A data structure called the working set list describes just those pages in a compact form.

This chapter describes the composition of the working set list, the ways in which it shrinks and expands to describe a varying number of pages, and the system services by which a process affects its working set and working set list.

5.1 Overview

The term *working set* refers to the virtual pages of a process that are currently valid and in physical memory. A valid page is one whose page table entry (PTE) valid bit is set.

As an image is executed in a process, code, data, and page table pages are faulted into the process's working set. Chapter 4 describes the page fault mechanism in detail. Execution of asynchronous system trap (AST) procedures, condition handlers, and system services that touch pageable process-private space can cause additional faults into the working set. The working set continues to grow as code running in the process context faults pages until the process occupies as much physical memory as it requires or is allowed. Each subsequent page fault requires that a page be removed from the working set to make room for the new page.

The executive maintains a list of working set pages for each process, called the working set list.

The working set list facilitates

- Selecting a page to remove from the working set when a page needs to be faulted in but the process already occupies all the physical memory it is currently allowed, or when the process's working set is being shrunk
- Determining which pages to write when a process is outswapped
- Determining which pages to read when a process is inswapped

Section 5.2 describes the structure and makeup of the working set list. Section 5.3 gives a detailed description of replacement paging, that is, removing one virtual page from the working set to make room for another.

The size of the working set list and the number of its entries constrain a process's use of physical memory. The working set list size varies over the process's lifetime. It can be affected by the authorization file entry for an interactive user, SYSGEN parameters, availability of physical memory, and the recent paging history of the process. Section 5.4 describes these effects, and Section 5.2.3 discusses the capacity of the working set list.

By requesting the following system services, code executing in a process can affect the process's working set and working set list:

- Adjust Working Set Limit ($ADJWSL)
- Lock Pages in Working Set ($LKWSET and $LKWSET_64)
- Lock Pages in Memory ($LCKPAG and $LCKPAG_64)
- Unlock Pages from Working Set ($ULWSET and $ULWSET_64)
- Unlock Pages from Memory ($ULKPAG and $ULKPAG_64)
- Purge Working Set ($PURGWS and $PURGE_WS)

These services are described in later sections of this chapter.

Section 5.10 explains the means by which a process can prevent the removal of a particular page from its working set.

This chapter is primarily concerned with the process working set list, although some of it is equally applicable to the system working set list (see Chapter 2).

5.2 The Working Set List

A process's working set includes the process's P0, P1, P2, and page table pages as well as the system space pages that contain its process header (PHD). Each of these pages is described by a working set list entry (WSLE). The working set list is self-describing, containing WSLEs that describe the working set list itself as well as the other PHD pages.

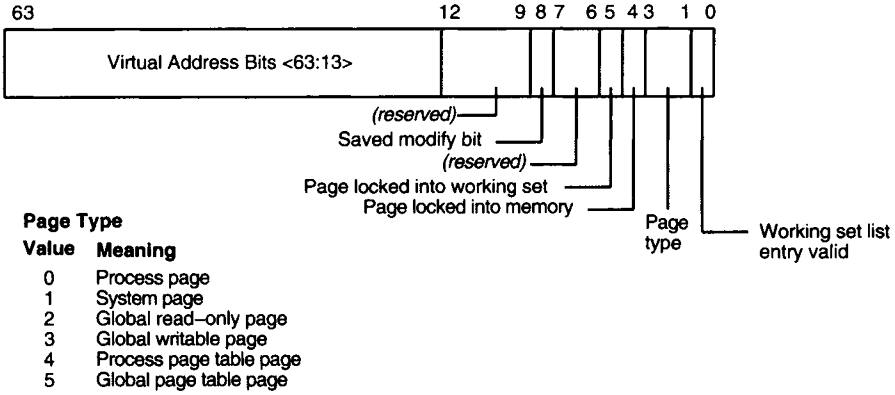
Pages that are part of a section mapped by page frame number (PFN) are valid for the entire time the process maps such pages, and they do not appear in the working set list.

The working set also includes global pages in use by the process, with the exception of pages of memory-resident and Galaxywide global sections. These pages are valid for the entire time the process maps such pages, and they do not appear in the working set list. If a memory-resident or Galaxywide global section is mapped with shared page tables, they are valid for the entire time the section is mapped and do not appear in the working set list.

5.2.1 The WSLE

The format of a valid WSLE is shown in Figure 5.1. Note that the upper bits are the same as the upper bits of a virtual address. This allows the WSLE to be passed as a virtual address to several utility routines that ignore the byte offset bits (WSLE control bits).

Figure 5.1 Format of a WSLE



Page Type

Value	Meaning
0	Process page
1	System page
2	Global read–only page
3	Global writable page
4	Process page table page
5	Global page table page

As OpenVMS Alpha Version 7.0 expanded the size of a meaningful virtual address to 64 bits, it expanded the size of a WSLE from a longword to a quadword to accommodate the larger virtual address.

Although the working set list currently remains in the PHD, it may move in a future release. For that reason, a process's working set list is located through the pointer CTL$GQ_WSL, which currently points to the working set list within the P1 space mapping of the PHD.

Table 5.1 shows the meanings of the WSLE control bits. The MACRO-32 macro $WSLDEF defines their symbolic values, which begin with the string WSL$V_.

Table 5.1 WSLE Control Bits

Field Name	Meaning
VALID	When set, this bit indicates that the WSLE is in use.
PAGTYP	This field, a duplicate of the contents of the PFN$L_PAGE_STATE type bits, identifies the page type and specifies the action required when the page is removed from the working set.
PFNLOCK	When set, this bit indicates one of the following types of page locked into the working set: · Page locked into physical memory with the $LCKPAG[_64] system service · Process-private page table that maps pages locked into physical memory, window pages, memory-resident global section pages, or Galaxywide global section pages
WSLOCK	When set, this bit indicates one of the following types of page locked into the working set: · Permanently locked page · Page locked with the $LKWSET[_64] system service · Process-private page table page that maps one or more valid or transition pages
MODIFY	This bit, used when the process is outswapped, records the logical OR of the modify bit in the PTE and the saved modify bit in the page's PFN$L_PAGE_STATE field.

5.2.2 Regions of the Working Set List

The working set list is divided into three regions: one containing entries for pages that are permanently locked; one containing entries for pages locked after process creation, chiefly by user request; and one containing dynamic entries. These regions are described in more detail later in this section.

Figure 5.2 shows the fields in the fixed portion of the PHD that describe the working set list. Many of them locate the different regions of the working set list through a quadword index from the beginning of the working set list to a particular WSLE. (In OpenVMS versions prior to Version 7.0, a WSLE was identified through a longword index from the beginning of the PHD. The index base was changed to allow for the possibility of removing the working set list from the PHD.) For example, the following steps compute the address of the end of the working set list from the quadword index in PHD$L_WSLAST:

1. Multiply the contents of PHD$L_WSLAST by 8.
2. Add the result to the address of the beginning of the working set list.

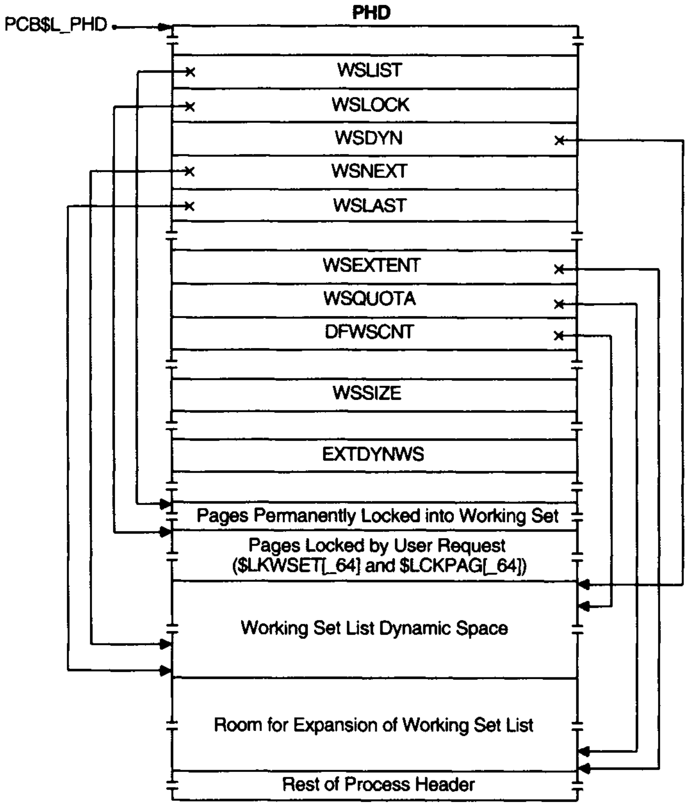
Three of the fields shown, PHD$L_DFWSCNT, PHD$L_WSQUOTA, and PHD$L_WSEXTENT, do not locate region boundaries but instead represent a number of WSLEs. These fields nonetheless contain quadword indexes, providing easier comparison with fields that do locate boundaries. The following steps convert such a field into the number of WSLEs it represents:

1. Subtract the contents of PHD$L_WSLIST from it.
2. Add 1 to the result.

This chapter refers to the converted contents of a quadword index field using its field name without the PHD$L_ prefix, for example, WSQUOTA. Note that names used in this way represent a number of WSLEs, or pages.

Two of the fields shown, PHD$L_WSSIZE and PHD$L_EXTDYNWS, each contain an actual number of WSLEs. This chapter refers to their contents as WSSIZE and EXTDYNWS.

Figure 5.2 Working Set List


PCB$L_PHD
PHD
WSLIST
WSLOCK
WSDYN
WSNEXT
WSLAST
WSEXTENT
WSQUOTA
DFWSCNT
WSSIZE
EXTDYNWS
Pages Permanently Locked into Working Set
Pages Locked by User Request ($LKWSET[_64] and $LCKPAG[_64])
Working Set List Dynamic Space
Room for Expansion of Working Set List
Rest of Process Header

Working Set List Dynamics

The permanently locked region of the working set list describes pages that are forever a part of the process working set. Pages whose WSLEs are in this region cannot be unlocked and are not candidates for working set replacement. They include the following:

- Kernel stack page or pages
- Page containing the P1 pointer area
- PHD pages—the fixed portion, the PHD page arrays, the maximum process section table, and enough pages for a working set list of as many entries as the SYSGEN parameter PQL_DWSDEFAULT, converted from pagelets to pages
- Process-private level 3 page table (L3PT) that maps the kernel stack, P1 pointer area, and P1 window to the PHD (see Chapter 2)
- Process-private level 2 page table (L2PT) that maps P0 and P1 L3PTs
- Process-private level 1 page table (L1PT)

The value in PHD$L_WSLIST is a quadword index to the first WSLE in this region. Its value is 1. The WSLE with index value 0 is reserved so that the executive need not distinguish a null working set list index from an index of 0.

The second region contains WSLEs for pages that are locked by user request, specifically through the $LKWSET, $LKWSET_64, $LCKPAG, and $LCKPAG_64 system services. Pages whose WSLEs are in this region are not candidates for working set replacement. PHD expansion pages resulting from working set list growth or creation of floating-point register and execution data structure (FRED; see Chapter *Kernel Threads*) pages are locked into this region of the working set list.

PHD$L_WSLOCK contains the quadword index to the first WSLE in the locked region. PHD$L_WSDYN points to the WSLE immediately following the last WSLE in this region. To lock a page into the working set list, the executive swaps its WSLE with that pointed to by PHD$L_WSDYN and increments PHD$L_WSDYN. Consequently, the user-locked region is increased by one WSLE and the dynamic region is decreased by one.

The two locked regions of the working set list are completely filled with valid WSLEs. Rather than keep a count of locked pages, the executive can simply calculate the difference between the contents of PHD$L_WSDYN and PHD$L_WSLIST.

The dynamic region begins at the entry identified by the contents of PHD$L_WSDYN. PHD$L_WSLAST contains the quadword index for the last WSLE; its contents identify the end of the dynamic region. The dynamic region is not necessarily dense; there may be empty entries between those specified by PHD$L_WSDYN and PHD$L_WSLAST. The dynamic region of the working set list describes process-private and global pages that have not been locked into the working set list and process-private page table pages. These pages are candidates for working set replacement.

The dynamic region is treated as a ring buffer for page replacement. The entry most recently inserted into the working set list is pointed to by PHD$L_WSNEXT. The entry following it is the point in the ring buffer at which page replacement typically occurs. The page replacement algorithm, explained in Section 5.3, is a modified first-in/first-out (FIFO) scheme.

A process-private L2PT or L3PT page that maps valid, transition, or PFN-mapped pages is locked into the dynamic region of the working set list through the WSLOCK bit in the WSLE and is not a candidate for working set replacement while locked. Page table pages locked in this manner remain in the dynamic region, although locked, for a number of reasons. They are considered dynamic because they are unlocked when all the valid, transition, and window pages they map are removed from the working set. Leaving them in the dynamic region results in less CPU overhead than switching them into and out of the locked region. Note that a page table page that maps only buffer object pages is not locked into the working set list. Chapter 3 provides further information.

The dynamic region can also contain entries temporarily locked into it by kernel mode code. The virtual pages locked in this way fall within the address range specified by fields PCB$Q_KEEP_IN_WS and PCB$Q_KEEP_IN_WS2. Section 5.10 contains further information.

The executive guarantees a minimum size for the dynamic region. Although most Alpha instructions generate few memory references, the executive must ensure that an instruction that references memory can execute. All the pages referenced in an instruction must be valid for the instruction to complete execution. If the dynamic region of the working set is too small, an infinite page fault loop could occur during the attempted execution of one instruction. An instruction could begin to execute, incur a page fault, restart, incur a different page fault, replace the first faulted page in the working set list, restart, reincur the first page fault, and so on, unable to complete execution. More realistically, the dynamic region of the working set should be large enough to allow a typical image to make reasonable progress without continual page faults.

5.2.3 Working Set List Parameters

Three critical parameters govern working set list dynamics: size, limit, and capacity (see Figure 5.3).

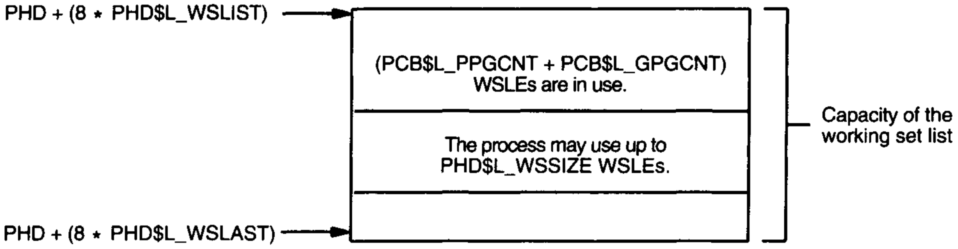
The process's working set size is the number of WSLEs currently in use. No single field contains this value; instead, it is the sum of two separately maintained counts, PCB$L_PPGCNT and PCB$L_GPGCNT.

The maximum number of WSLEs the process is allowed to use is known as its working set limit. It is maintained in a field that is somewhat confusingly called PHD$L_WSSIZE. Despite its name, it contains the working set limit, not the size (which is the sum of the two fields listed in the previous paragraph).

Working Set List Dynamics

The amount of memory allocated to hold the working set list data structure varies during the life of a process. The amount of memory currently allocated for the working set list (PHD$L_WSLAST minus PHD$L_WSLIST, plus 1 entries) is referred to in this chapter as the working set list capacity. When the capacity increases, the working set list data structure itself may grow and consume more physical memory.

Figure 5.3 Working Set List Parameters



When the working set limit is reduced, the working set list capacity is not necessarily altered. The working set list simply becomes more sparsely populated with valid WSLEs and more heavily populated with invalid WSLEs.

Table 5.2 shows process-specific and systemwide working set list parameters, quotas, and limits. Note that for compatibility with OpenVMS VAX, user authorization file (UAF) quotas and SYSGEN parameters related to the working set list are typically specified externally in units of pagelets and converted to pages for internal use by the executive.

Table 5.2 Working Set Lists: Limits and Quotas

Description	Location or Name	Comments
Working set size in pages	PCB$L_PPGCNT + PCB$L_GPGCNT	Cannot grow above working set limit; Updated each time a page is added to or removed from the working set; Reduced by proactive memory reclamation, swapper trimming, and $ADJWSL

Table 5.2 ***(continued)*** **Working Set Lists: Limits and Quotas**

Description	Location or Name	Comments
Working set limit	PHD$L_WSSIZE	Cannot grow above working set capacity; Implicitly set by LOGINOUT[1]; Altered by $ADJWSL, by automatic working set limit adjustment, image exit, and swapper trimming; Altered by locking pages, creating address space, and requesting direct I/O
Default working set limit	PHD$L_DFWSCNT	Set by LOGINOUT[1]; Altered by DCL command SET WORKING_SET/LIMIT
Normal maximum working set limit (index)	PHD$L_WSQUOTA	Set by LOGINOUT[1]; Altered by DCL command SET WORKING_SET/QUOTA
Extended maximum working set limit (index); upper value for automatic working set limit adjustment	PHD$L_WSEXTENT	Set by LOGINOUT[1]; Altered by DCL command SET WORKING_SET/EXTENT
Upper limit to normal maximum working set limit (index)	PHD$L_WSAUTH	Set by LOGINOUT[1]; Cannot be altered
Upper limit to extended maximum working set limit (index)	PHD$L_WSAUTHEXT	Set by LOGINOUT[1]; Cannot be altered
Index of first WSLE	PHD$L_WSLIST	Always 1
Index of first locked WSLE	PHD$L_WSLOCK	The same for all processes in a given system

Index of first dynamic WSLE	PHD$L_WSDYN	Initialized by SHELL;
		Altered by $LKWSET[_64], $LCKPAG[_64], $ULWSET[_64], and $ULKPAG[_64];
		Altered by PHD expansion and contraction

[1] The manner in which a process is created determines how a value for this is defined. It may be defined several times during different steps of process creation (see Chapters *Process Creation* and *Process Dynamics*).

Table 5.2 ***(continued)*** **Working Set Lists: Limits and Quotas**

Description	Location or Name	Comments

Index of most recently inserted WSLE	PHD$L_WSNEXT	Initialized by SHELL;
		Altered by $ADJWSL;
		Updated each time an entry is added to or released from the working set;
		May be altered if capacity decreased or locked region increased

Index of last WSLE (determines capacity of list)	PHD$L_WSLAST	Initialized by SHELL; May be altered by $ADJWSL, page fault handler, image exit, automatic working set limit adjustment, working set page replacement
Sufficient number of dynamic WSLEs for a process to execute without continuous faults	PHD$L_WSFLUID	Set by SHELL to the value of MINWSCNT
Number of dynamic WSLEs excluding both PHD$L_WSFLUID pages and a reasonable number of page table pages	PHD$L_EXTDYNWS	Updated each time size of dynamic working set region is changed
Authorized default working set limit in pagelets	UAF$L_DFWSCNT	Converted to pages and copied to PHD$L_DFWSCNT
Authorized normal maximum working set limit in pagelets	UAF$L_WSQUOTA	Converted to pages and copied to PHD$L_WSAUTH and PHD$L_WSQUOTA
Authorized extended maximum working set limit in pagelets	UAF$L_WSEXTENT	Converted to pages and copied to PHD$L_WSEXTENT and PHD$L_WSAUTHEXT
Systemwide minimum number of fluid working set pages	MINWSCNT	SYSGEN parameter
Number of pagelets to which the swapper attempts to shrink a working set before outswapping it	SWPOUTPGCNT	SYSGEN parameter
Systemwide maximum working set limit in pagelets	WSMAX	SYSGEN parameter
System working set limit in pagelets	SYSMWCNT	SYSGEN parameter

Table 5.2 *(continued)* Working Set Lists: Limits and Quotas

Description	Location or Name	Comments
Default value for working set limit default in pagelets (used by $CREPRC)	PQL_DWSDEFAULT	SYSGEN parameter
Minimum value for working set limit default in pagelets (used by $CREPRC)	PQL_MWSDEFAULT	SYSGEN parameter
Default value for normal maximum working set limit in pagelets (used by $CREPRC)	PQL_DWSQUOTA	SYSGEN parameter
Minimum value for normal maximum working set limit in pagelets (used by $CREPRC)	PQL_MWSQUOTA	SYSGEN parameter
Default value for extended maximum working set limit in pagelets (used by $CREPRC)	PQL_DWSEXTENT	SYSGEN parameter
Minimum value for extended maximum working set limit in pagelets (used by $CREPRC)	PQL_MWSEXTENT	SYSGEN parameter

During system initialization, enough virtual address space is reserved in each PHD for the maximum-size working set list, one with as many entries as the number of pages represented by the SYSGEN parameter WSMAX.

Each process is created with its initial working set limit and working set list capacity set to the same value, the process's default working set limit, DFWSCNT (assuming that DFWSCNT is less than or equal to WSMAX converted to pages). For a typical interactive process, DFWSCNT is specified by the UAF entry. The executive thus initially allocates physical memory for only a relatively small working set list.

When a process runs an image, it begins faulting pages; the working set size increases, growing toward the working set limit. Once it reaches the limit, subsequent page faults require the removal of pages from the working set. With the working set limit, the executive governs the amount of physical memory a process may use.

5.2.3.1 Working Set Limit

During system initialization, the SYSGEN parameters that affect minimum working set limits are adjusted to allow for at least a minimum dynamic working set list region. That is, SYSBOOT ensures that the values of PQL_MWSDEFAULT and PQL_DWSDEFAULT represent a number of pages large enough to accommodate the sum of the following:

- The SYSGEN parameter MINWSCNT, the minimum number of fluid pages in the working set
- The worst-case number of L3PT pages to map MINWSCNT pages, namely, MINWSCNT
- The maximum PHD
- The kernel stack page or pages
- The minimum number of page tables to map the P1 space defined by the SHELL*xx* module

The manner in which a process is created determines how values for WSQUOTA, WSEXTENT, and several other working set list paramters are defined. They are defined and potentially redefined several times during different steps of process creation. In the case of the typical interactive process, the values come from its authorization file record and are minimized with WSMAX. Chapters *Process Creation* and *Process Dynamics* supply further information.

The process can increase its default working set limit through the Digital command language (DCL) command SET WORKING_SET. A running image can increase the process's current working set limit by requesting the $ADJWSL system service. The executive can increase a process's working set limit through automatic working set limit adjustment. These mechanisms are discussed in Section 5.4.

A programmer with a good understanding of an image's paging behavior can voluntarily reduce the process's working set limit by requesting the $ADJWSL system service. There are several other less direct mechanisms by which the working set limit is decreased:

- Automatic working set limit adjustment can reduce the limit (see Section 5.4.3).
- The swapper process can initiate a reduction of the working set limit with a mechanism known as swapper trimming or working set shrinking. In an effort to acquire needed physical memory, the swapper reduces the working sets and working set limits of processes in the balance set before actually removing processes from the balance set. Process selection is performed by a table-driven, prioritized scheme. Chapter 6 describes the conditions that trigger this mechanism and the criteria by which processes are selected.
- The process's working set limit is also reset at image exit to its default value, DFWSCNT (see Chapter *Image Activation and Exit*).

Whenever the executive adjusts the working set limit or locks pages into the working set list, it checks that the dynamic region of the working set list has enough space. For a typical process and address space, the executive checks that the number of dynamic WSLEs is at least twice MINWSCNT. In this check, it ignores any working set list extension above WSQUOTA, since any extension above quota is subject to swapper trimming. To facilitate the check, the executive maintains the field PHD$L_EXTDYNWS, which effectively contains the number of WSLEs in the dynamic region of the working set list beyond the sum of the minimum number required and the number in use for page tables.

The executive actually calculates the number of entries to be reserved as the sum of MINWSCNT, PHD$L_PTCNTLCK (number of page table pages mapping locked or window pages), and the smaller of MINWSCNT and the number of unlocked page table pages.

For example, when a process tries to lock a page into its working set list, the executive checks that PHD$L_EXTDYNWS has a value of at least 2, one entry for the page and another for its page table page.

5.2.3.2 Working Set Size

A process's working set size increases as it executes code and faults pages into its working set.

The process's working set size decreases as the result of its deleting virtual address space (explicitly or, for example, at image exit) or requesting the $PURGWS or $PURGE_WS system service. With a mechanism known as proactive memory reclamation, the executive may reduce the working set size of a long-waiting process or a periodically waking process to reclaim memory for a depleted free page list. Proactive memory reclamation differs from swapper trimming in that the former reduces the working set size but not the limit. This mechanism and the conditions that trigger it are described in Section 5.5.

A process's working set size also decreases as an effect of having its working set limit decreased below its working set size.

5.2.3.3 Working Set Capacity

Whenever the working set limit is increased to a value that would exceed the working set list capacity, the capacity must grow as well to accommodate the new limit. As described in Chapter 2, the working set list capacity is dynamic; it grows toward the process section table (PST). When the working set list must expand into the area already occupied by the PST, the PST is moved to higher addresses. However, there is not always room in the PHD for the expanded working set list. The total space available for both the working set list and the PST is determined by the two SYSGEN parameters WSMAX and PROCSECTCNT.

Because a process is allowed to create more than PROCSECTCNT sections, the PST can grow into space that would have been available for the working set list. In that case, the working set list capacity can grow no further, and the process must make do with its current capacity and a limit no larger than that capacity.

Furthermore, because the working set list contains WSLEs for all the PHD pages in physical memory, its size and the size of the PHD are interrelated to a small extent. As the working set grows, the working set list in the PHD grows, and more WSLEs are required to describe the PHD pages in memory. The size of the PHD is constrained to be no larger in pages than half of the process's working set quota.

This constraint preserves a reasonable number of WSLEs for non-PHD pages. A process with a large value for working set extent and a relatively small value for working set quota could have the expansion of its working set limited by this constraint.

At image exit, in addition to reducing the process's working set limit, the executive may reduce the working set list capacity; if possible, the executive resets PHD$L_WSLAST by moving it toward lower addresses past any empty WSLEs. It continues until it reaches a valid WSLE or until the working set list capacity is just equal to the working set limit. Additionally, when the executive is scanning the working set list to find an entry for a page being faulted, it may move PHD$L_WSLAST in the same way, compressing invalid entries at the high-address end of the working set list. The executive must strike a balance between spending too much overhead compressing empty entries so that PHD$L_WSLAST is precise and spending too much overhead searching for a valid replacement WSLE when the working set list is sparse (see Section 5.3.1).

5.3 Working Set Replacement

When code executing in a process references an invalid virtual page, the page fault handler must take steps to make the page valid. It must also create a WSLE for the page. If there is no room in the working set list for another entry, one must be removed. The page fault handler uses the dynamic region of the working set list to determine which virtual page to discard.

The dynamic region of the working set list can contain unused WSLEs. When the working set limit is reduced, the working set list capacity is usually left intact, resulting in a sparse working set list. This makes adding a page to the working set slightly more complex. That a WSLE is empty does not necessarily mean the process can make use of it; the size of the working set must be less than the working set limit. If the process is already at its limit, a nonempty WSLE must be found whose virtual page can be removed from the working set to make room for the new page.

The executive uses a modified FIFO scheme for its working set list replacement algorithm. The entry most likely to have been in the working set list for the longest time, the one following that pointed to by PHD$L_WSNEXT, is the one first considered for replacement.

5.3.1 Scan of the Working Set List

When the page fault handler needs an empty WSLE, it calls routine MMG$FREWSLE, in module PAGEFAULT. The following steps summarize its flow. Subsequent sections describe more details of particular aspects of its flow.

MMG$FREWSLE scans the dynamic region of the working set list. It begins by checking whether the WSLE whose index is in PHD$L_WSNEXT is empty. If not, it starts with the next WSLE.

1. If the WSLE is empty (contents are zero), MMG$FREWSLE checks whether the entry can be used (see Section 5.3.2). If it can be used, it is selected.
2. If the WSLE is not empty (contents are nonzero) but is an active page table page (one that maps valid or window pages), the WSLE cannot be used.
3. If the WSLE is not empty but is a process-private page table page that maps no pages at all (its WSLE$V_WSLOCK bit is clear), MMG$FREWSLE makes additional checks (see Section 5.3.4) to see whether the WSLE is suitable for reuse. If not, the WSLE is skipped.

   If the page table maps no valid or window pages, it may be usable. MMG$FREWSLE takes the steps described in Section 5.3.3 to determine whether the page table page can be released and its WSLE reused.
4. If the WSLE is not empty, MMG$FREWSLE makes additional checks (see Section 5.3.4) to see whether the WSLE is suitable for reuse. If not, the WSLE is skipped.
5. If the WSLE is selected for reuse and is not empty, MMG$FREWSLE takes the actions described in Section 5.3.5.
6. If the WSLE is not selected, the index is incremented, and the steps in this list are repeated until a usable WSLE is found. If the index exceeds the end of the list, it is reset to the beginning of the dynamic working set list.

Once a WSLE is selected for reuse, PHD$L_WSNEXT is updated to contain its quadword index.

5.3.2 Using an Empty Entry in the Working Set List

When an empty WSLE is found, MMG$FREWSLE checks whether a page can be added to the working set. If there are fewer pages in the working set than WSQUOTA, a new physical page may be added to the working set. It may also be possible to add physical pages to the working set above WSQUOTA (up to WSEXTENT), depending on the size of the free page list.

The following checks are required for an empty WSLE to be usable:

1. If the working set size (PCB$L_PPGCNT plus PCB$L_GPGCNT) equals the working set limit, the empty WSLE may not be used, and a page in the working set must be replaced.

2. If the working set size has not reached its limit, the size is compared to WSQUOTA. If the size is less than WSQUOTA and there are more than FREELIM pages on the free page list, a new page is allowed into the working set. The empty WSLE is used.
3. If the working set has WSQUOTA or more pages, the number of pages on the free page ıst is compared to the SYSGEN parameter GROWLIM. If there are more than GROWLIM pages on the free page list, a new page is allowed into the working set. The empty WSLE is used.

   Notc that to extend the working set size above WSQUOTA, the working set limit must have been extended above WSQUOTA. For the working set limit to have been extended above WSQUOTA, the free page list must have contained more than the SYSGEN parameter BORROWLIM pages. For more information on working set limits, BORROWLIM, and automatic working set limit adjustment, see Section 5.4.
4. Even if the free page list is below the limit at which another page could be added to the process's working set, a new page can be added if the working set contains fewer fluid pages than PHD$L_WSFLUID: if the size of the working set minus locked pages and minus PHD$L_PTCNTMAX is less than or equal to SYSGEN parameter MINWSCNT, the empty WSLE is used.

If an empty but unusable WSLE is found at the end of a working set list that has reached its limit, the working set list capacity is reduced; PHD$L_WSLAST is reset to point to the last unavailable (nonzero) WSLE in the working set list.

5.3.3 Releasing a Dead Page Table Page

MMG$FREWSLE determines whether a WSLE describing a page table page can be reused to describe a page being faulted into the working set list. It first checks, however, that the executive is not looping endlessly, trying to remove a WSLE from a working set list that contains only nonremovable entries: if it has already tested as many page table pages as there are dynamic pages in the working set, that means there are no fluid pages in the working set list. It tests whether it may increase the working set limit based on the number of free pages and the relation between the working set limit and WSQUOTA.

- If there are at least BORROWLIM free pages and the limit is less than WSEXTENT, the limit can be increased up to WSEXTENT.
- If there are fewer than BORROWLIM free pages and the limit is less than WSQUOTA, the limit can be increased up to WSQUOTA.

If the limit cannot be increased, MMG$FREWSLE simply returns without having freed a WSLE.

If the limit can be increased, MMG$FREWSLE increases it without going over the maximum previously determined. If possible, it increases the limit by MINWSCNT pages. If that is not possible, it increases the limit by 2 pages. If that is not possible, it increases the limit by only 1 page.

Having increased the limit, MMG$FREWSLE scans the working set list again for a free entry or a WSLE that describes a page table page that could be removed from the working set list. After retesting as many page table pages as there are dynamic pages, it simply returns without having freed a WSLE.

If it has not already tested too many WSLEs that describe page tables, it calls SCANDEADPT to examine the page table page. There are several possible outcomes:

- The WSLE describes a page table page that maps valid pages and is therefore not reusable.
- The WSLE describes a page table page that maps only free and modified pages and can be released from its current use for reuse after the ties between those pages and the page table page are severed, that is, after no virtual pages mapped by the page table page are cached in the free or modified page list.
- The WSLE describes a page table page that maps only free and modified pages pages but the working set list contains enough dynamic entries that this one need not be released now. An attempt is made to leave a page table page in the working set list to keep its virtual pages cached on page lists, in case the process refaults them.

SCANDEADPT first determines whether the process has any dead page table pages. A dead page table page is one that maps no valid or window pages. It checks by comparing PHD$L_PTCNTVAL, the number of page table pages with valid WSLEs, to PHD$L_PTCNTACT, the number of active page table pages. If PHD$L_PTCNTACT is larger than PHD$L_PTCNTVAL, the difference between them is the number of dead page table pages. If there are none, it returns immediately. MMG$FREWSLE skips this WSLE and continues its scan of the working set list.

If there are any dead page table pages, SCANDEADPT checks how full the working set list is. It checks whether the dynamic region of the working set list has at least twice MINWSCNT entries, not counting those that describe dead page table pages or page table pages that map pages locked into memory or into the working set list. If so, it has sufficient dynamic entries; the dead page table page scan is postponed, and SCANDEADPT returns. MMG$FREWSLE skips this WSLE and continues its scan of the working set list.

In making this check, SCANDEADPT uses the process's working set limit if one of the following is true:

- The working set size is less than MINWSCNT.
- The limit is less than or equal to WSQUOTA and the free page list has at least FREELIM pages.
- The limit is greater than WSQUOTA and the free page list has at least GROWLIM pages.

If memory is relatively scarce, the process will not be allowed to expand its working set, so SCANDEADPT restricts its test to the dynamic entries in a working set of WSQUOTA pages.

If there are not sufficient dynamic WSLEs to leave a potentially dead page table page in the working set list, SCANDEADPT checks whether this page is a dead page table page by examining the PFN database record of the physical page occupied by the page table page. If PFN$W_PT_VAL_CNT is non-negative, the page table page maps pages in the working set list and cannot be released. SCANDEADPT returns, and MMG$FREWSLE goes on to the next WSLE.

If PFN$W_PT_VAL_CNT is negative, the page table page is dead, and SCANDEADPT increments PMS$GL_DPTSCN to indicate one more dead page table scan. It stores the working set list index of the dead page table in PHD$L_NEXT. It must scan each PTE within the page table page to determine whether it is a transition PTE. If the page table page contains transition PTEs for pages on the free page list, SCANDEADPT must modify the PFN database for those pages before the WSLE can be reused. It moves each such page to the front of the free page list and sets the delete contents bit in the page's PFN$L_PAGE_STATE field.

If the page table page contains transition PTEs for pages on the modified page list, those pages must be written to their backing store before the page table page can be released from the working set list. SCANDEADPT sets the delete contents bit in each page's PFN$L_PAGE_STATE field and requests a selective purge of the modified page list so that those pages will be written. If the process's working set is being shrunk, SCANDEADPT requests that pages in a particular address range be purged (an SVAPTE request); otherwise, it requests that pages mapped by this page table page be purged (a PAGE_TABLE request). SCANDEADPT checks that at least one page file has been installed and, if not, simply returns. Otherwise, it returns to MMG$FREWSLE with a status indicating it should return to its caller to wait. The kernel thread is placed into a resource wait for RSN$_MPWBUSY until the modified page list is selectively purged. Chapter 4 describes the selective purge mechanism and the resource waits.

If the page table contained transition PTEs only for free page list pages, those pages have been released, and SCANDEADPT returns to MMG$FREWSLE with a status indicating that it should reuse this WSLE.

5.3.4 Skipping WSLEs

The operating system uses both process-specific criteria and frequency-of-use information maintained by the hardware to modify its strict FIFO page replacement algorithm. The working set replacement routine can skip a limited number of WSLEs with particular characteristics. The number is specified by the special SYSGEN parameter TBSKIPWSL. In addition, it skips entries temporarily locked into the working set by kernel mode code.

The architecture defines a processor register related to translation buffer (TB) use called TB check (TBCHK). Kernel mode code can execute the instruction CALL_PAL MFPR, specifying the TBCHK register and a virtual address to determine whether the translation for a particular virtual page is cached. The presence of a TB entry for a page indicates the page has been referenced recently and may therefore be a poor candidate to remove from the working set.

Additionally, a process can declare a procedure to be notified of every pending removal from its working set list. The procedure can return a status indicating that this page is a poor choice.

Kernel mode software running in process context calls the routine MMG$DECLARE_WSL_PAGER, in module SYSLKWSET, with two arguments. One is the procedure value of a routine to be called when a page is about to be removed from that process's working set. The other is a parameter to be passed to that procedure. Because the working set list removal procedure may be called from outside the context of that process, the procedure must be within code loaded into system space. MMG$DECLARE_WSL_PAGER stores the procedure value in PCB$A_FREWSLE_CALLOUT and the parameter in PCB$L_FREWSLE_PARAM.

The working set list replacement algorithm works in the following manner. Before a valid WSLE is reused, a check is first made to see if a translation for the virtual page described by that WSLE is in the TB. If the translation for that page is cached in the TB, and fewer than TBSKIPWSL entries have been skipped during this scan of the working set list, MMG$FREWSLE skips that WSLE and resumes the search for an available WSLE with the next one. Note that the translation of an address referenced by another kernel thread of this process may be cached in the TB of one or more other processors. Because the overhead to check the other processors is greater than the possible benefit of keeping an active page in the working set list, only the TB of this processor is checked.

If the translation is not cached in the TB, MMG$FREWSLE compares the address of the virtual page decribed by the WSLE to the starting and ending addresses in PCB$Q_KEEP_IN_WS and PCB$Q_KEEP_IN_WS2. If the address falls within that range, MMG$FREWSLE skips that WSLE and resumes the search for an available WSLE with the next one.

If the translation is not cached in the TB, the virtual page is not within that range, and PCB$A_FREWSLE_CALLOUT is nonzero, MMG$FREWSLE calls the working set removal procedure with the specified parameter, the virtual address, a flag, and the addresses of the PCB and PHD. Initially the value of the flag is zero. If the procedure returns the status SS$_RETRY, MMG$FREWSLE skips that WSLE and resumes the search for an available WSLE with the next one.

After TBSKIPWSL WSLEs have been skipped in this manner, the checks for whether the translation is cached in the TB and whether the removal procedure will approve are abandoned and the next valid WSLE that is not within the PCB$Q_KEEP_IN_WS range is simply reused. First, however, if there is a working set removal procedure, MMG$FREWSLE calls it with a flag value indicating that the selected page will definitely be removed from the working set.

If the value of TBSKIPWSL is set to zero, the skipping of WSLEs whose translations are in the TB is disabled; although the working set removal procedure is still called, it is called with the flag value indicating that the selected page will definitely be removed from the working set. The default value of TBSKIPWSL is 8.

Use of the working set removal procedure is limited to kernel mode applications and is currently intended for support of a graphics subsystem. Accessing its own copy of the process's page table, the graphics hardware determines how to treat a particular page based on its valid bit. The device driver for the graphics hardware requests notification of working set list removals so it can maintain the copy of the page table. If a page selected for removal is currently in use by the graphics hardware, the driver's notification routine would indicate that the page is a poor choice. Use of this mechanism is reserved to Hewlett-Packard Company; any other use is unsupported.

5.3.5 Reusing WSLEs

The virtual page that the WSLE represents must be removed before the WSLE can be reused. Typically, the virtual page is valid and must be made invalid. This section is confined to a description of WSLEs representing valid pages.

For such a page, MMG$FREWSLE takes the following steps:

1. It tests whether the page has been modified. If not, it continues with step 2. If the page has been modified, MMG$FREWSLE tests whether its backing store is a page file and, if so, how full the modified page list is.

   If the modified page list has fewer pages than the SYSGEN parameter MPW_WAITLIMIT, or if modified page writing is in progress and the list has fewer pages than the SYSGEN parameter MPW_LOWAITLIMIT, MMG$FREWSLE proceeds with step 2.

   Otherwise, to avoid deadlocks, MMG$FREWSLE checks that the kernel thread does not hold any mutexes, that it is not the swapper, that bit MMG$V_NOWAIT in MMG$GL_FREWFLGS is clear, and that at least one page file has been installed. If any condition is false, MMG$FREWSLE proceeds with step 2.

   If all are true, it returns a status to the page fault handler indicating that the kernel thread should be placed into a resource wait. The kernel thread is placed into the resource wait RSN$_MPWBUSY until the modified page list has dropped below MPW_LOWAITLIMIT pages.

2. At alternative entry point MMG$FREWSLX_64, the routine clears the valid, modify, no-TB-miss-memory-barrier-required, fault-on-execute, fault-on-write, and address space match bits in the PTE. If the fault-on-execute bit was set, it invalidates any cached copy of the PTE from the data stream translation buffer (DTB). If the bit was clear, it invalidates any cached copy from both the instruction stream translation buffer (ITB) or the DTB (see Section 5.3.6).

3. If the page was modified, it sets the saved modify bit in the page's PFN$L_PAGE_STATE field.

4. If the page is a global page, whether read-only or writable, MMG$FREWSLE changes the PTE to the global page table index form.

It updates the data structures describing the process-private page table page that maps the page being removed. It decrements the share count for the page table page to indicate that it maps one less valid or transition page. If this was the last valid or transition page mapped by the page table page (that is, if the share count makes the transition from 1 to 0), it locates the WSLE for the page table page and clears its WSL$V_WSLOCK bit. It also decrements

— PHD$L_PTCNTACT, the number of active page table pages for the process

— The PHD reference count, the number of reasons the PHD should remain in memory, which is kept in the PHD's entry in the array at PHV$GL_REFCBAS_LW (see Chapter 2)

MMG$FREWSLE decrements the share count for the global page to indicate one less process is mapping it. If the count is still nonzero, it proceeds with step 6. If the count goes to zero, it clears the valid, modify, fault-on-write, fault-on-execute, and address space match bits in the global page table entry (GPTE).

5. For a page that is a process page, a global page with a zero share count, or a process page table, it decrements the reference count for the page to indicate one less reference to it.

   — If the reference count goes to zero, MMG$FREWSLE calls MMG$REL_PFN, in module ALLOCPFN, to insert the page at the end of the free or modified page list, depending on the state of its saved modify bit. If the page has been modified and has an assigned page file backing store, MMG$REL_PFN releases its backing store, which has a now-obsolete copy of the page. The PFN$Q_BAK field is reset to show unallocated page file backing store.

   — If the reference count is nonzero, indicating possible direct or paging I/O in progress, MMG$FREWSLE examines the PFN$L_PAGE_STATE field and, if the page is active, changes its state to release pending.

   For a process page, it also updates the data structures describing the page's page table page. It decrements the share count for the page table page to indicate that it maps one less valid or transition page. If this was the last valid or transition page mapped by the page table page (that is, if the share count makes the transition from 1 to 0), it locates the WSLE for the page table page and clears its WSL$V_WSLOCK bit. It also decrements PHD$L_PTCNTACT, the number of active page table pages for the process, and the PHD's entry in the array at PHV$GL_REFCBAS_LW, the number of reasons the PHD should remain in memory.

6. MMG$FREWSLE calls MMG_STD$DELWSLEX_64, in module PAGEFAULT.

   MMG_STD$DELWSLEX_64 decrements PFN$W_PT_VAL_CNT in the PFN database record for the page table page that mapped this page to indicate the page table maps one less valid page. If that count goes to –1, it also decrements PHD$L_PTCNTVAL to indicate one less page table page mapping valid pages. It decrements either PCB$L_PPGCNT or PCB$L_GPGCNT, depending on page type. It clears the WSLE and returns.

7. MMG$FREWSLE returns to its caller.

5.3.6 TB Invalidation

As described in Chapter 1, a translation buffer is a CPU component that caches the results of recent successful virtual address translations of valid pages. Each TB entry caches one translation: a virtual page number and, minimally, its corresponding PFN, address space match, and protection bits. The size and organization of a TB are CPU-specific. Some CPUs have both an ITB and a DTB.

The operating system is responsible for flushing no longer correct entries from the TB. For example, it must invalidate a TB entry corresponding to a no longer valid PTE that maps a page being deleted or removed from a process's working set. It must also invalidate the TB entry for a valid page whose protection is changing.

A TB entry whose address space match (ASM) bit is set represents a physical page shared at the same virtual address in all processes. In practice, only system space pages and shared page table space have the ASM bit set. On a symmetric multiprocessing (SMP) system, such a shared page can be represented in multiple processors' TBs and must therefore be invalidated in the TBs of all processors.

Because a process with a single kernel thread runs on only one processor at a time, its process-private TB entries, those with a clear ASM bit, need be invalidated only on the TB of the processor on which the kernel thread is running. The kernel threads of a multithreaded process, however, can run on multiple processors. Such a process's pages can be represented in multiple processors' TBs and must therefore be invalidated on all processors on which its kernel threads are currently executing. (Chapter *Scheduling* describes how the assignment of address space numbers to kernel threads prevents stale TB entries on processors on which its kernel threads are *not* currently executing.)

Executive modules typically invalidate TB entries through one of the following macros, which are provided for MACRO-32, BLISS, and C:

- TBI_DATA_64—Invalidate a single DTB entry for a page whose fault-on-execute bit is still set.
- TBI_SINGLE—Invalidate a single entry from both the ITB and the DTB.
- TBI_ALL—Invalidate all TB entries.

Note that the TBI_SINGLE_64 macro provided by versions prior to OpenVMS Version 7.0 is no longer available; TBI_SINGLE can now handle a 32-bit or 64-bit address.

As described in Chapter 1, the executive sets the fault-on-execute bit in the PTE of each page faulted as the result of a data fetch. If an attempt is made to execute an instruction from the page, a fault-on-execute exception occurs. The exception service routine clears the fault-on-execute bit. When the fault-on-execute bit is still set for a page whose TB entries must be invalidated, the executive invokes the TBI_DATA_64 macro because there can be no ITB entry for the page.

Both TBI_SINGLE and TBI_DATA_64 include an argument to specify the virtual address to be invalidated and an argument to specify the address of the PCB associated with the process context. Each has an ENVIRON argument whose default value is MP. Other values for the ENVIRON argument are THIS_CPU_ONLY, ASSUME_SHARED, and ASSUME_PRIVATE.

If one of these macros is invoked with the ENVIRON argument specified as THIS_CPU_ONLY, the macro merely generates a CALL_PAL MTPR instruction whose processor register depends on the macro. (Chapter 1 lists the processor registers associated with TB invalidation.) These macros are invoked this way when the virtual address is known to be one whose ASM bit is clear and private to a single-thread process or when the virtual address is known to be CPU-specific.

To invalidate a TB entry for a page that is shared, the executive invokes the appropriate TB invalidate macro and explicitly specifies the ENVIRON argument as ASSUMED_SHARED. In response, code is generated that transfers control to the subroutine in module TBI_ROUTINES corresponding to the macro, either MMG[_STD]$TBI_SINGLE or MMG[_STD]$TBI_DATA_64. Each of these subroutines tests whether SMP is enabled and, if not, merely executes a CALL_PAL MTPR instruction specifying the appropriate processor register. If SMP is enabled, each subroutine calls MP_INVALIDATE or MP_INVALIDATE_DATA, in module TBI_ROUTINES. Chapter *Symmetric Multiprocessing* describes MP_INVALIDATE, MP_INVALIDATE_DATA, and the means by which one processor notifies the other members to flush one or all TB entries.

To invalidate a TB entry for a page that may or may not be shared, the executive calls the appropriate TB invalidate macro and specifies the ENVIRON argument as MP or implicitly specifies it by omitting the argument. Code is generated that tests whether SMP is enabled and, if not, executes a CALL_PAL MTPR instruction. If SMP is enabled, the generated code tests whether the address is actually in shared space (system or shared page table space):

- If so, it calls MMG[_STD]$TBI_SINGLE or MMG[_STD]$TBI_DATA_64.
- If not, it tests whether the process has multiple kernel threads. If so, it calls MMG[_STD]TBI_SINGLE_THREADS or MMG[_STD]$TBI_DATA_64_THREADS, in module TBI_ROUTINES. If the process is single-threaded, it generates a CALL_PAL MTPR instruction.

MMG[_STD]$TBI_SINGLE_THREADS and MMG[_STD]$TBI_DATA_64_THREADS call MP_INVALIDATE_THREADS or MP_INVALIDATE_DATA_THREADS, in module TBI_ROUTINES. These two subroutines differ from MP_INVALIDATE and MP_INVALIDATE_DATA in that they notify only those processors on which kernel threads of this process are active.

To invalidate a TB entry for a page that is not shared, the executive calls the appropriate TB invalidate macro and explicitly specifies the ENVIRON argument as ASSUMED_PRIVATE. Code is generated that tests whether the process has multiple kernel threads and, if so, transfers control to either MMG[_STD]$TBI_SINGLE_THREADS or MMG[_STD]$TBI_DATA_64_THREADS. If not, the generated code executes a CALL_PAL MTPR instruction.

The *OpenVMS Alpha Guide to Upgrading Privileged-Code Applications* contains further information on the use of the TB invalidate macros.

5.4 Working Set Limit Adjustment

The working set limit is the maximum number of WSLEs the process is allowed to use. A process's working set limit (see Table 5.2) varies over its lifetime as a result of events such as image execution and exit, dynamic working set limit adjustment, and swapper trimming.

The working set limit can be altered with the $ADJWSL system service, described in Section 5.4.1. Requested explicitly by the process, the system service can alter the working set limit up to WSEXTENT.

The service can also be requested automatically on behalf of the process, for example, when the process tries to expand its address space. The executive checks whether after adding page tables to map the new space, the dynamic working set list would have enough room for the fluid working set (PHD$L_WSFLUID) plus the worst-case number of page table pages required to map it, to allow the process to perform useful work. If this check fails, and the process's working set limit is smaller than its quota, the executive increases the process's limit.

Similarly, when a process tries to lock pages into memory or into its working set list, explicitly through system service or implicitly through requesting direct I/O, the executive checks that the space left for dynamic WSLEs is sufficient. If not and if the limit is smaller than the quota, the executive increases the process's limit.

The $ADJWSL service is also requested on behalf of the process by the quantum-end routine when it performs automatic working set limit adjustment. Through this means, the maximum size to which the working set limit can grow is WSQUOTA, unless there are sufficient pages on the free page list (more than the SYSGEN parameter BORROWLIM). In that case, automatic working set limit adjustment can enlarge the limit up to WSEXTENT.

After the working set limit is increased, if there are more than the SYSGEN parameter GROWLIM pages on the free page list, the executive allows the process to use the extended limit by adding more pages to its working set without removing already valid entries. Adding pages to a process's working set decreases the probability that the process will incur a page fault.

Section 5.4.3 describes the automatic working set limit adjustment mechanism.

5.4.1 $ADJWSL System Service

The $ADJWSL system service is requested to alter the process's working set limit by a number of pagelets. Its procedure, EXE$ADJWSL in module SYSADJWSL, runs in kernel mode, at interrupt priority level (IPL) 2 and above. EXE$ADJWSL first converts its input argument from pagelets to pages, rounding up if necessary. It then determines whether the request is to increase or reduce the limit.

To increase the limit, EXE$ADJWSL first checks whether the process has multiple kernel threads. If so, it acquires the MMG spinlock, raising IPL to IPL$_MMG, to serialize access to fields such as PHD$L_LAST and PHD$L_WSSIZE with the page fault service routine. Although the need to acquire the inner mode semaphore prevents more than one kernel thread in a process from executing a memory management system service, one kernel thread could be executing EXE$ADJWSL while another incurred a page fault.

EXE$ADJWSL then checks and possibly reduces the size of the increase. The new limit must be less than or equal to the value of the SYSGEN parameter WSMAX, converted from units of pagelets to pages; less than or equal to the process's extended maximum working set limit; and within the system's physical memory capacity (available pages minus the minimum size of the free page list).

If the new working set limit is within the current capacity of the working set list, EXE$ADJWSL computes a new value for PHD$L_EXTDYNWS and returns. Otherwise, EXE$ADJWSL must call MMG$ALCPHD, in module PHDUTL, to increase the working set list capacity. If EXE$ADJWSL acquired the MMG spinlock, it releases it and drops IPL to 2 before calling MMG$ALCPHD.

MMG$ALCPHD tests whether there is a gap between the high-address end of the working set list and the low-address end of the PST that is large enough for the working set list expansion. If not, it tries to compress enough unused entries from the low-address end of the PST to accommodate the expansion. If that also fails, MMG$ALCPHD tries to shift the PST to higher addresses by moving it to as yet unused pages of the PHD. As previously described, the PHD cannot be expanded in this manner if the number of pages in the nonpageable part of the current PHD is half the size of the process's WSQUOTA.

If expanded working set list pages are created, they must be locked into the working set list. It is possible that locking all the expansion pages at once would leave insufficient extra dynamic entries in the existing working set list. However, if the working set list were partially expanded, the number of dynamic entries would increase, allowing more expansion pages to be locked. Thus, expanding the working set limit may require multiple iterations.

MMG$ALCPHD returns the number of entries by which it increased the capacity of the working set list. If no increase was possible, it returns zero.

If the process has multiple kernel threads, EXE$ADJWSL reacquires the MMG spinlock, raising IPL to IPL$_MMG.

If MMG$ALCPHD added any new entries, EXE$ADJWSL changes PHD$L_WSNEXT to point to the first of the newly added WSLEs and clears the WSLEs to initialize them. It adds the number of new WSLEs to both PHD$L_WSLAST and PHD$L_WSSIZE. It recalculates PHD$L_EXTDYNWS and returns to its requestor.

To decrease the limit, EXE$ADJWSL first acquires the MMG spinlock, raising IPL to IPL$_MMG, to block swapper trimming and possible quantum-end working set limit adjustment. It sets MMG$V_NO_MPL_FLUSH in MMG$GL_FREWFLGS to delay modified page writing during MMG$SHRINKWS's execution (see Chapter 4). This enables multiple requests to accumulate before the modified page list is processed, thus improving performance. It calls MMG$SHRINKWS, in module SYSADJWSL.

MMG$SHRINKWS checks and possibly reduces the size of the decrease. The new limit must allow for a dynamic portion of the working set list that can accommodate at least SYSGEN parameter MINWSCNT WSLEs plus the number of page tables locked in the working set. In addition, PHD$L_EXTDYNWS cannot be reduced below zero.

MMG$SHRINKWS modifies the working set limit. If the process's working set size is already less than or equal to the new limit, it simply returns to EXE$ADJWSL. Otherwise, MMG$SHRINKWS repeatedly calls MMG$FREWSLE (see Section 5.3.1), in module PAGEFAULT, for each page to be removed from the process's working set. The reduced list can be sparse, that is, can contain unused and unusable WSLEs; the working set capacity is not necessarily decreased with the working set limit. Control returns to EXE$ADJWSL.

If MMG$FREWSLE generated modified page write requests, EXE$ADJWSL requests a modified page flush (see Chapter 4) and changes the kernel thread's state to miscellaneous wait on resource RSN$_MPWBUSY. If MMG$FREWSLE did not generate additional requests, but returned a status from SCANDEADPT indicating the kernel thread should be stalled, EXE$ADJWSL changes the kernel thread's state to miscellaneous wait on the specified resource, for example, RSN$_MPWBUSY. When EXE$ADJWSL returns, the system service dispatcher will wait the kernel thread in the access mode from which the $ADJWSL request was made.

EXE$ADJWSL releases the spinlock, recalculates PHD$L_EXTDYNWS, and returns.

5.4.2 SET WORKING_SET Command

The DCL command SET WORKING_SET enables the user to alter the default working set limit (DFWSCNT), the normal maximum working set limit (WSQUOTA), or the extended maximum working set limit (WSEXTENT). None of these can be set to a value larger than the authorized extended maximum working set limit (WSAUTHEXT). For OpenVMS VAX compatibility, the command's qualifiers are expressed in units of pagelets.

Altering the default limit affects the working set list reset operation performed by the routine MMG$IMGRESET, in module PHDUTL, which is called at image exit. Altering the normal maximum working set limit affects the maximum working set limit when physical memory is not plentiful. It changes the upper limit for future $ADJWSL system service requests.

With the /[NO]ADJUST qualifier to this command, a user can also disable or reenable automatic working set limit adjustment. Use of that qualifier sets or clears the process control block (PCB) status longword bit PCB$V_DISAWS.

5.4.3 Automatic Working Set Limit Adjustment

As described in Section 5.4, the executive adjusts working set limit through an explicit $ADJWSL request or as a side effect of image exit, address expansion, or page locking. In addition it also provides automatic working set limit adjustment to keep a process's page fault rate within limits set by one of several SYSGEN parameters. Note that no such adjustment takes place for real-time processes or for a process that has disabled automatic working set limit adjustment through the DCL command SET WORKING_SET/NOADJUST. The executive can also use automatic working set limit adjustment to reclaim an extension to the working set of a low-priority process.

Table 5.3 shows the parameters that control automatic working set limit adjustment. All the SYSGEN parameters listed in this table are dynamic and can be altered without rebooting the system.

Automatic working set limit adjustment takes place as part of the quantum-end routine (see Chapter *Scheduling*).

Table 5.3 Process and System Parameters Used by Automatic Working Set Limit Adjustment

Description	Location or Name	Comments
Total amount of CPU time charged to this process	PHD$L_CPUTIM	Updated by interval timer interrupt service routine
Amount of CPU time at last adjustment check	PHD$L_TIMREF	Updated by quantum-end routine when adjustment check is made; Altered when process is placed into a wait
Total number of page faults for this process	PHD$L_PAGEFLTS	Updated each time this process incurs a page fault
Number of page faults at last adjustment check	PHD$L_PFLREF	Updated by quantum-end routine when adjustment check is made
Most recent page fault rate for this process	PHD$L_PFLTRATE	Recorded at each adjustment check; Compared to PFRATH and PFRATL
Process automatic working set limit adjustment flag	PCB$V_DISAWS in PCB$L_STS	When set, disables adjustment for process

Table 5.3 *(continued)* **Process and System Parameters Used by Automatic Working Set Limit Adjustment**

Description	Location or Name	Comments
Amount of CPU time process must accumulate before page fault rate check is made	AWSTIME[1]	
Lower limit page fault rate	PFRATL[1]	When 0, disables adjustment based on page fault rate for entire system
Number of pagelets by which to decrease working set limit	WSDEC[1]	Also, amount to reclaim from low-priority process with extended working set
Lower bound in pagelets for decreasing working set list size	AWSMIN[1]	Do not adjust if PCB$L_PPGCNT is less than or equal to this
Upper limit page fault rate	PFRATH[1]	
Number of pagelets by which to increase working set limit	WSINC[1]	When 0, disables adjustment for entire system
Free page list size that allows growth of working set	GROWLIM[1]	Add new page to working set only if free page list has more pages than this value
Free page list size that allows extension of working set limit	BORROWLIM[1]	Extend working set limit beyond WSQUOTA only if free page list has more pages than this value; When –1, disables working set limit extension for entire system

[1]This value is a SYSGEN parameter.

The quantum-end routine, SCH$QEND in module RSE, adjusts the working set limit in the following steps:

1. It makes the following checks. If any of these conditions is true, SCH$QEND performs no adjustment.

   — If the kernel thread's priority is in the real-time range, adjustment of this process is disabled.

   — If the user has entered the DCL command SET WORKING_SET/NOADJUST, PCB$V_DISAWS is set and automatic working set limit adjustment for the process has been disabled.

   — If PHD$V_NO_WS_CHNG is set, the executive is deleting this process and its address space, and adjustment is irrelevant.

— If the WSINC parameter is set to zero, the adjustment is disabled on a systemwide basis.

2. If the process's kernel thread or threads have not been executing long enough since the last adjustment (if the difference between accumulated CPU time, PHD$L_CPUTIM, and the time of the last adjustment attempt, PHD$L_TIMREF, is less than the SYSGEN parameter AWSTIME), no adjustment based on page fault rate is made. SCH$QEND proceeds with step 5.

   If the process has accumulated enough CPU time, the reference time is updated (PHD$L_CPUTIM is copied to PHD$L_TIMREF), and the rate checks are made.

   Between adjustment checks, PHD$L_TIMREF is also altered whenever a kernel thread in the process is placed in a wait. As described in Chapter *Scheduling*, when a kernel thread goes into a wait, the SYSGEN parameter IOTA is charged against its quantum. To balance the quantum charge, IOTA is subtracted from PHD$L_TIMREF, so that the last check for adjustment appears to have taken place longer ago than it really did and AWSTIME is more quickly reached. This subtraction helps ensure the expansion of the working set limit of a process that is faulting heavily. Without it, a process that undergoes many page fault waits could reach quantum end without having accumulated AWSTIME worth of CPU time and thus not be considered for automatic working set limit adjustment.

3. SCH$QEND calculates the current page fault rate. The philosophy for automatic working set limit adjustment is based on two premises. If the page fault rate is low enough, the system can reclaim physical memory from the process, by reducing its working set limit, without harming the process by causing it to fault heavily. If the page fault rate is too high, the process can benefit from a larger working set limit because it will incur fewer faults without degrading the system.

4. If the current page fault rate is too high (greater than or equal to PFRATH), SCH$QEND checks whether the working set limit should be increased.

   — If the working set size is less than 75 percent of the current working set limit, the working set limit is not expanded.

   — If the current working set limit is below WSQUOTA, it is expanded by WSINC, converted to pages.

   — If the working set limit is greater than or equal to WSQUOTA, the number of pages on the free page list is compared to the SYSGEN parameter BORROWLIM.

     If there are BORROWLIM or more pages on the free page list, the working set limit is increased by WSINC, converted to pages. It can be increased to a maximum limit of WSEXTENT.

     If there are fewer than BORROWLIM pages on the free page list, the working set limit is not increased.

     Setting BORROWLIM to –1 disables working set limit expansion above WSQUOTA for the entire system.

Once the working set limit has been expanded, newly faulted pages may be added to the working set. The page fault handler adds pages to the working set above WSQUOTA only when there are more than the SYSGEN parameter GROWLIM pages on the free page list.

SCH$QEND proceeds with step 6.

5. If WSDEC is zero, shrinking the working set by automatic working set limit adjustment is disabled and no adjustment occurs. If WSDEC is nonzero, two types of decrease to the working set limit are possible.

   First, if the current page fault rate is low enough (less than PFRATL), the working set limit is shrunk by WSDEC, converted to pages. However, if the contents of PCB$L_PPGCNT are less than or equal to AWSMIN, no adjustment takes place. This decision is based on the assumption that many of the pages in the working set are global pages and therefore the system will not benefit (and the process may suffer) if the working set limit is decreased.

   Note that PFRATL is zero by default. This default value effectively disables this method of working set limit reduction in favor of swapper working set trimming. The rationale for this change is explained at the end of this list.

   Second, even if a meaningful interval has not elapsed for computing a page fault rate, the process's working set limit will be shrunk, whatever its page fault rate and whatever the value of PFRATL, if all the following are true:

   — The process has had a pixscan priority boost in its last 32 execution quantums (PCB$L_PIXHIST is nonzero). Chapter *Scheduling* describes the pixscan mechanism. That the process had a pixscan boost implies that it is a low-priority process.

     Note, however, that in a multithreaded process, each kernel thread has its own priority; PCB$L_PIXHIST is therefore not necessarily representative of the process as a whole. For the working set list of a multithreaded process to be shrunk, the additional condition must be met that no other kernel thread in this process is current.

   — The free page list contains fewer than GROWLIM pages.

   — The process's working set limit is larger than WSQUOTA.

   Its working set limit will be decreased by the smaller of WSDEC, converted to pages, and the amount by which its working set limit exceeds WSQUOTA. This mechanism reclaims working set growth beyond WSQUOTA, which is regarded as temporary growth to be permitted only when sufficient memory is available.

6. The actual working set limit adjustment is accomplished by a kernel mode AST that requests the $ADJWSL system service. The AST parameter passed to this AST is the amount of previously determined increase or decrease. This step is required because the system service must be called from process context (at IPL 0) and SCH$QEND is executing in system context in response to the IPL$_TIMERFORK software timer interrupt.

Two problems are inherent in the quantum-end scheme of automatic working set limit adjustment: processes that are compute-intensive will reach quantum end many times, and images that have been written to be efficient with respect to page faults (and incur a low page fault rate) will qualify for working set limit reduction, because their page fault rate is lower than PFRATL. In both these cases, working set limit reduction is not desirable. In contrast, swapper trimming (summarized in Section 5.2.3 and detailed in Chapter 6) selects processes starting with those that are less likely to need large working sets.

Working set limit reduction based on page fault rate at quantum end is disabled by setting the default value of PFRATL to zero. Consequently, swapper trimming and the image exit reset are the primary methods used to reduce working set limit. In contrast to automatic working set limit reduction, swapper trimming shrinks the working set limit (and size) only when free pages are needed. The executive also uses automatic working set limit adjustment at quantum end to reclaim extensions from the working sets of low-priority processes.

5.5 Proactive Memory Reclamation from Periodically Waking Processes

Proactive memory reclamation, also known as the ticker, is enabled when the low bit of SYSGEN parameter MMG_CTLFLAGS is set, as it is by default. If enabled, the mechanism becomes active only when the free page list is less than twice FREEGOAL and modified page writing would not make up the difference. The mechanism reduces the working set size of long-waiting processes and periodically waiting processes with normal (non-real-time) priorities.

The Synchronize ($SYNCH) and Hibernate ($HIBER) system services as well as the event flag wait services, such as Wait for Logical OR of Event Flags ($WFLOR), are responsible for implementing the policy of proactive memory reclamation from periodically waking processes. Each of the services checks whether the mechanism is active, whether the kernel thread being waited has a normal priority, and whether the process has accumulated 30 seconds of wait time (PCB$L_ACC_WAITIME) since the last time its execution history was checked. In the case of a multithreaded process, each kernel thread must have a normal priority and be in HIB, LEF, or CEF state. If all the conditions are met, each service procedure calls EXE$CHK_WAIT_BHVR, in module RSE.

EXE$CHK_WAIT_BHVR takes the following steps:

1. It checks whether the process has any outstanding direct I/O and, if so, returns immediately. Direct I/O completion is typically fast and is likely to change the process's scheduling state.

2. It checks whether the process has a high ratio of wait time to execution time since this routine was last called to check this process. If the process's accumulated CPU time is at least 1 percent of its wait time, the routine continues with step 5.

3. It tests whether the process has disabled automatic working set adjustment (PCB$V_DISAWS in PCB$L_STS is set) or whether the executive is deleting this process and its address space (PHD$V_NO_WS_CHNG in PHD$L_FLAGS is set). If either is true, EXE$CHK_WAIT_BHVR continues with step 5.
4. Holding the MMG spinlock, it tries to reduce the process's working set size by 25 percent. It does not alter the working set limit.
5. It copies the accumulated CPU time from PHD$L_CPUTIME to PCB$L_CPUTIME_REF for use the next time EXE$CHK_WAIT_BHVR is executed. It clears PCB$L_ACC_WAITIME and returns.

5.6 Lock Pages in Working Set System Services

A process requests these system services to lock a virtual page into its process working set and thus prevent page faults from occurring on references to the page. Locking a page into the working set guarantees that when a kernel thread of the process is current, the locked page is always valid. These services have obvious benefit for time-critical applications and other situations in which a program must access code or data without incurring a page fault.

These system services are also requested by process-based kernel mode routines to ensure the validity of code and data pages accessed above IPL 2. Page faults at IPLs above 2 are prohibited; if one occurs, the page fault handler generates the fatal bugcheck PGFIPLHI.

Pages locked into a process working set do not necessarily remain resident in physical memory when no kernel threads of the process are current; the entire working set might be outswapped. To guarantee residency of the pages, a process must request either the $LCKPAG[_64] system service or both the $LKWSET[_64] and the Set Process Swap Mode ($SETSWM) system services.

$LKWSET is the traditional service for locking pages into the working set list. $LKWSET_64, added in OpenVMS Alpha Version 7.0, enables a process to lock a page whose address cannot be expressed in 32 bits.

5.6.1 $LKWSET System Service

The $LKWSET system service procedure, EXE$LKWSET in module SYSLKWSET, executes in kernel mode. It takes the following steps:

1. It creates and initializes scratch space on the stack and raises IPL to 2.
2. It tests the accessibility of the INADR argument and maximizes the ACMODE argument with the mode of the service requestor.
3. If necessary and possible, EXE$LKWSET increases the working set limit to have sufficient extra dynamic entries to accommodate the pages to be locked and a page table page for each such page.

If the process has disabled working set limit adjustment, or if its working set limit is already larger than its quota, no increase is possible. As a result, MMG$LCKULKPAG may be able to lock only a limited number of pages.

4. EXE$LKWSET calls MMG$CREDEL, in module SYSCREDEL, specifying MMG$LCKULKPAG, in module SYSLKWSET, as the per-page service-specific routine. Chapter 3 describes the memory management stack scratch space, the actions of MMG$CREDEL, and its invocation of the specified service-specific routine.
5. When MMG$CREDEL returns, EXE$LKWSET restores the previous IPL and returns to its requestor with the status from MMG$CREDEL.

To lock a page into the working set, MMG$LCKULKPAG, with its alternative entry point MMG_STD$LCKULKPAG, takes the following steps:

1. It tests whether the page is readable from the system service requestor's access mode. If the page is inaccessible, it returns the error status SS$_ACCVIO, which becomes the status returned by the system service.
2. It acquires the MMG spinlock, raising IPL to IPL$_MMG.
3. It examines the L3PTE that maps the page. If the page or any of the higher level page table pages is not valid, MMG$LCKULKPAG stores the address of the page to be locked in PCB$Q_KEEP_IN_WS, releases the MMG spinlock, faults the page, resets PCB$Q_KEEP_IN_WS to –1, and continues with step 2.
4. It compares the page owner access mode with the mode of the system service requestor. If the page is owned by a more privileged mode, the requestor is not allowed to alter its state, and MMG$LCKULKPAG releases the MMG spinlock and returns the error status SS$_PAGOWNVIO.
5. It tests whether the window bit is set in the L3PTE and, if so, immediately returns the success status SS$_WASSET. A virtual page whose L3PTE's window bit is set is always valid and is not described by a WSLE, so no further action is appropriate.
6. MMG$LCKULKPAG examines the PFN$L_PAGE_STATE field in the page's PFN database record to determine if the page type is process or read-only global. If either, it continues with step 7. If the page is a writable global page from a memory-resident or Galaxywide global section, it immediately returns the success status SS$_WASSET. A virtual page from such a global section is always valid and is not described by a WSLE, so no further action is appropriate.

   If the page is not one of these types, it releases the MMG spinlock and returns the error status SS$_NOPRIV; a process is not permitted to lock any other type of page into its working set. In particular, it may not lock global writable pages because when a process is outswapped, the swapper must be able to remove global writable pages from the working set. The removal avoids any ambiguity at inswap concerning the location of the most recent copy of a global writable page.

7. MMG$LCKULKPAG gets the working set list index (WSLX) for a process page from its PFN$L_WSLX field. WSLX information is not kept for a global page; instead, MMG$LCKULKPAG must scan the process's working set list to locate the entry for the page.
8. MMG$LCKULKPAG examines the WSLE. If the page is already locked into the working set, the routine releases the MMG spinlock and returns the success status SS$_WASSET.
9. Otherwise, it checks whether the page has been locked into memory and, if so, continues with step 10. If not, it checks that PHD$L_EXTDYNWS is at least 2 (to allow for the page table page as well as the page being locked). This ensures that the process will have enough dynamic WSLEs after the page is locked into its working set. If not, it releases the MMG spinlock and returns the error status SS$_LKWSETFUL.
10. It sets the WSL$V_WSLOCK bit in the WSLE of the newly locked page.
11. If the page has already been locked into memory, it is within the user-locked region of the working set list, and MMG$LCKULKPAG continues with step 12. More typically, it must reorganize the working set list, pictured in Figure 5.2, so that the locked page's entry is in the user-locked region of the working set list, following the PHD$L_WSLOCK pointer. MMG$LCKULKPAG accomplishes this reorganization by exchanging the newly locked WSLE with the entry pointed to by PHD$L_WSDYN and incrementing PHD$L_WSDYN to point to the next entry in the list. If PHD$L_WSDYN pointed to a valid WSLE, it exchanges the contents of the PFN$L_WSLX_QW fields for the two valid pages; otherwise, it updates the PFN$L_WSLX_QW field for the newly locked page.
12. MMG$LCKULKPAG increments PFN$W_PT_LCK_CNT in the PFN database record for the page table page mapping the locked page. If this is the first locked page mapped by this page table and the page maps no window pages, it also increments PHD$L_PTCNTLCK, the number of page table pages mapping locked WSLEs.
13. It checks that PHD$L_WSNEXT is still pointing into the dynamic part of the working set list (and not at the former PHD$L_WSDYN, which is now in the user-locked region), moving it if necessary to point to the same WSLE as PHD$L_WSLAST.
14. It recalculates PHD$L_EXTDYNWS.
15. It releases the MMG spinlock and returns to MMG$CREDEL.

5.6.2 $LKWSET_64 System Service

The $LKWSET_64 system service procedure, EXE$LKWSET_64 in module SYS_LKWSET_64, executes in kernel mode. $LKWSET_64 resembles the $LKWSET system service, but all its address and length arguments are 64 bits. Thus it can be used to lock pages from P0, P1, or P2 space. EXE$LKWSET_64 takes the following steps:

1. It checks the number of arguments with which the service was requested and, if incorrect, returns either the error status SS$_INSFARG or SS$_TOO_MANY_ARGS.
2. It checks that output arguments are accessible and, if not, returns the error status SS$_ACCVIO.
3. It maximizes the ACMODE argument.
4. It rounds down the START_VA_64 argument to the nearest page boundary. It rounds up the LENGTH_64 to an integral number of pages large enough to include the rounded-down starting address and the ending address implied by the original START_VA_64 and LENGTH_64 arguments.
5. It raises IPL to 2 to block AST delivery.
6. EXE$LKWSET_64 determines the address of the RDE corresponding to the START_VA_64 argument. If none corresponds, it returns error status SS$_ACCVIO or SS$_NOT_PROCESS_VA, depending on the argument's value.
7. If necessary and possible, EXE$LKWSET_64 increases the working set limit to have sufficient extra dynamic entries to accommodate the pages to be locked and the maximum number of page tables to map them. A P0 or P1 space page could need an additional L3PT; a P2 space page could need an additional L2PT and L3PT.

   If the process has disabled working set limit adjustment, or if its working set limit is already larger than its quota, no increase is possible. As a result, only a limited number of pages may be locked.
8. EXE$LKWSET_64 loops, calling MMG_STD$LCKULKPAG for each page (see Section 5.6.1). If the region's addresses are ascending, it begins with the lowest address in the range to be locked. If the addresses descend, it begins with the highest address.
9. It stores return information in the RETURN_VA_64 and RETURN_LENGTH_64 arguments.
10. EXE$LKWSET_64 returns the status from MMG_STD$LCKULKPAG to its requestor.

5.7 Lock Pages in Memory System Services

The operations of the $LCKPAG[_64] system service are similar to those of the $LKWSET[_64] system service. However, the $LCKPAG[_64] service guarantees permanent residency for the specified virtual address range in addition to performing an implicit working set lock of those pages. The pages remain resident until the process specifies them in an unlock page system service request. Because this operation permanently allocates a system resource, physical memory, it requires the privilege PSWAPM.

$LCKPAG is the traditional service for locking pages into the working set list. $LCKPAG_64, added in OpenVMS Alpha Version 7.0, enables a process to lock a page whose address cannot be expressed in 32 bits.

5.7.1 $LCKPAG System Service

Executing in kernel mode, the $LCKPAG system service procedure, EXE$LCKPAG in module SYSLKWSET, tests whether the current security persona has the privilege PSWAPM and, if not, returns the error status SS$_NOPRIV. It raises IPL to 2 and increases the working set limit as necessary and possible.

It calls MMG$CREDEL, specifying MMG$LCKULKPAG as the per-page service-specific routine. MMG$LCKULKPAG is called with a flag that specifies the page is to be locked into memory rather than into the working set.

Although the results of requesting the $LKWSET and the $LCKPAG services are similar, the following differences exist:

- The WSLE of a page locked into memory has the WSL$V_PFNLOCK bit set rather than the WSL$V_WSLOCK bit.
- The PHD of a process that maps a page locked into memory must be locked into memory itself to ensure the residency of the page table page mapping the locked page.
- A global writable page that is not permanently resident can be locked into memory, although it cannot be explicitly locked into the working set.
- In locking a global page into memory, MMG$LCKULKPAG increments PFN$L_GBL_LCK_CNT in its physical page's PFN database record.
- If this is the first time a particular global page is locked into memory or if this is a process page not in use as a buffer object, MMG$LCKULKPAG increments MMG$GL_PFNLOCK_PAGES to indicate one more PFN-locked page, and decrements PFN$GL_PHYPGCNT to indicate one less page of physical memory available for general use. MMG$LCKULKPAG then calls EXE$CHKFLUPAGES, in module MEMORYALC, to confirm that enough physical memory remains available for general use.

EXE$CHKFLUPAGES subtracts the minimum sizes of the free and modified page lists from PFN$GL_PHYPGCNT and checks that the result is large enough to accommodate a reasonably large inswap. If not, it returns an error. In response to the error, MMG$LCKULKPAG decrements MMG$GL_PFNLOCK_PAGES; increments PFN$GL_PHYPGCNT; if the page is global, decrements PFN$L_GBL_LCK_CNT; and returns the error status SS$_LCKPAGFUL. That error is passed back to the service requestor.

5.7.2 $LCKPAG_64 System Service

The $LCKPAG_64 system service procedure, EXE$LCKPAG_64 in module SYS_LKWSET_64, resembles EXE$LKWSET_64 (see Section 5.6.2) with the following significant differences:

- EXE$LCKPAG_64 tests whether the current security persona has the privilege PSWAPM and, if not, returns the error status SS$_NOPRIV.
- It calls MMG_STD$LCKULKPAG with a flag that specifies the page is to be locked into memory rather than into the working set.

5.8 Unlock Pages System Services

These system services unlock pages from either the working set or physical memory.

$ULWSET and $ULKPAG are the traditional services for unlocking pages from the working set list and memory. $ULWSET_64 and $ULKPAG_64, added in OpenVMS Alpha Version 7.0, enable a process to unlock pages whose addresses cannot be expressed in 32 bits.

5.8.1 $ULWSET and $ULKPAG System Services

The two 32-bit system service procedures are EXE$ULWSET and EXE$ULKPAG, both in SYSLKWSET. Both, executing in kernel mode, call MMG$CREDEL with MMG$LCKULKPAG as the per-page service-specific routine. Both execute at IPL 0; working set trimming and adjustment do not interfere with unlocking pages.

MMG[_STD]$LCKULKPAG is called with one flag that specifies the operation is an unlock and a second flag that specifies whether the page is to be unlocked from the working set or from memory. It takes the following steps to unlock each page:

1. Its first steps are identical to steps 1 through 7 described for MMG$LCKULKPAG in Section 5.6.1.
2. MMG$LCKULKPAG examines the WSLE. If the page is not locked into the working set, the routine releases the MMG spinlock and returns the success status SS$_WASCLR.

3. If the page is a global page being unlocked from memory, MMG$LCKULKPAG decrements PFN$L_GBL_LCK_CNT. If the count goes to zero or if this is a process page being unlocked from memory, it decrements MMG$GL_PFNLOCK_PAGES, and if the page is not in use as a buffer object, it also increments PFN$GL_PHYPGCNT.
4. Otherwise, depending on the operation requested, it clears the appropriate WSLE bit (WSL$V_WSLOCK or WSL$V_PFNLOCK).
5. If one of the lock bits is still set, it goes on to step 7. Otherwise, it decrements PHD$L_WSDYN and swaps the WSLE of the page being unlocked with the one pointed to by PHD$L_WSDYN, thus making the unlocked WSLE the first one in the dynamic region. If PHD$L_WSDYN pointed to a valid WSLE, it exchanges the contents of the PFN$L_WSLX_QW fields for the two valid pages; otherwise, it updates the PFN$L_WSLX_QW field for the newly unlocked page.

   MMG$LCKULKPAG decrements PFN$W_PT_LCK_CNT in the PFN database record of the page table page mapping the locked page. If the count goes to –1 and PFN$W_PT_WIN_CNT is also –1, it also decrements PHD$L_PTCNTLCK, the number of page table pages mapping locked WSLEs.
6. It recalculates PHD$L_EXTDYNWS.
7. It releases the MMG spinlock and returns to MMG$CREDEL.

5.8.2 $ULWSET_64 and $ULKPAG_64 System Services

The two 64-bit system service procedures are EXE$ULWSET_64 and EXE$ULKPAG_64, both in SYS_LKWSET_64. Their argument validation resembles that of EXE$LKWSET_64 (see Section 5.6.2).

Each loops, calling MMG_STD$LCKULKPAG once per page with one flag that specifies the operation is an unlock and a second flag that specifies whether the page is to be unlocked from the working set or from memory.

Section 5.8.1 describes MMG_STD$LCKULKPAG's actions to unlock pages.

5.9 Purge Working Set System Services

A process requests these system services to remove all virtual pages in a specified address range from its working set. A process might request this service if a certain set of routines or data were no longer required. By voluntarily removing entries from the working set, a process can exercise some control over the working set list replacement algorithm, increasing the chances for frequently used pages to remain in the working set.

OpenVMS requests this service on behalf of a process when it requests the $PROCESS_AFFINITY or $PROCESS_CAPABILITIES service to change its home resource affinity domain (RAD) and sets CAP$M_PURGE_WS_IF_NEW_RAF in the FLAGS argument.

$PURGWS is the traditional service for removing pages from the working set list. $PURGE_WS, added in OpenVMS Alpha Version 7.0, enables a process to remove pages from address ranges that cannot be expressed in 32 bits.

The executive uses the $PURGWS system service as part of the image startup sequence (see Chapter *Image Activation and Exit*) to ensure that a program starts its execution without unnecessary pages such as command language interpreter command processing routines in its working set.

5.9.1 $PURGWS System Service

The $PURGWS system service procedure, EXE$PURGWS in module SYSPURGWS, runs in kernel mode. It takes the following steps:

1. It creates and initializes the stack scratch space and raises IPL to 2.
2. It calls MMG$CREDEL (see Chapter 3), specifying PURGWSPAG, in module SYSPURGWS, as the per-page service-specific routine.
3. EXE$PURGWS returns the status from MMG$CREDEL to its requestor.

PURGWSPAG immediately calls MMG$PURGWSSCN, in module SYSPURGWS, which takes the following steps:

1. It acquires the MMG spinlock, raising IPL to IPL$_MMG.
2. It scans the dynamic region of the working set list, examining each WSLE.
   - If the WSLE is not valid or is locked into the working set or memory, or if the address of the associated virtual page does not fall within the boundaries specified by the system service requestor, MMG$PURGWSSCN goes on to the next entry.
   - Otherwise, MMG$PURGWSSCN calls MMG$FREWSLX_64, described in Section 5.3.5, to take steps to release the WSLE and change the state of the page.
3. When MMG$PURGWSSCN reaches the end of the dynamic region, it releases the MMG spinlock, restoring the entry IPL, and returns.

5.9.2 $PURGE_WS System Service

The $PURGE_WS system service procedure, EXE$PURGE_WS in module SYS_PURGWS_64, runs in kernel mode.

It takes the following steps:

1. It rounds down the START_VA_64 argument to the nearest page boundary. It rounds up the LENGTH_64 to an integral number of pages large enough to include the rounded-down starting address and the ending address implied by the original START_VA_64 and LENGTH_64 arguments.
2. It raises IPL to 2 to block AST delivery.

3. It calls MMG_STD$PURGWSPAG_64, in module SYSPURGWS, which initializes the arguments to call MMG$PURGWSSCN, described in Section 5.9.1.
4. EXE$PURGE_WS restores the previous IPL and returns SS$_NORMAL.

5.10 Keeping a Page in the Working Set List

Occasionally it is desirable or necessary to fault a page into the working set and have it remain valid, perhaps for improved or more predictable performance. Code executing in kernel mode at elevated IPL, however, has a different concern. Because a page fault at IPL 3 or above results in a PGFIPLHI fatal bugcheck, a code thread executing at elevated IPL must ensure the residency of all code, data, and linkage section pages it accesses.

The issues related to the residency of particular pages in process and system working set lists include

- Specifying the pages of interest
- For elevated IPL execution, ensuring that all relevant pages are resident
- Keeping the pages in the working set

This section summarizes the first issue briefly; its focus is on the others.

Specifying the particular pages generally requires identifying the starting and ending addresses symbolically or identifying the starting address symbolically and specifying the length of the area of interest. How simple these steps are depends on whether the area of interest contains data, code, and its associated linkage section, or all three. It also depends on whether the language in which the source modules are written supports such capabilities.

In general, data pages are easier to specify and can be identified through data cell names at the beginning and end of the data. The organization of code written in any language cannot be taken for granted: a compiler may reorder code, convert routine invocations to in-line code, and so on. This makes it difficult to identify the boundaries of code to be made resident. Moreover, the linkage section associated with the code must also be made resident.

A number of events can lead to replacement paging or the removal of pages from a process's working set list:

- Execution in the process's context of a code thread of any access mode that incurs page faults, whether mainline code running in one or multiple kernel threads, a procedure in a shareable image, inner access mode service (Record Management Services, system service, or command language interpreter callback), AST thread, or condition handler
- Execution of a code thread that directly locks an invalid page into memory or the working set list or indirectly locks buffer pages by requesting direct I/O operations

- Quantum-end automatic working set limit adjustment of a process with a current kernel thread
- Swapper trimming of a process with no current kernel threads
- Proactive memory reclamation from the working set of a process with long-waiting kernel threads or a periodically waking kernel thread about to go into a wait

For a kernel thread to fault a page into its process's working set list and have it remain there, it must either ensure that the page is not a candidate for replacement paging or prevent all the events previously listed that lead to replacement paging.

The most straightforward measure, available in any access mode, is to lock the page with the $LKWSET[_64] system service. As a result, the page's WSLE is placed in the user-locked region of the working set list and is not a candidate for replacement paging. The page remains in the working set list regardless of the scheduling state of kernel threads in the process and throughout any outswap and inswap. The only page type for which this mechanism fails is a global writable page. The executive prohibits locking global writable pages into the working set list to avoid ambiguity at inswap concerning the location of the most recent version of the page. To ensure the residency of a global writable page, a process must lock the page into memory. Note that locking a global page into memory does not prevent process page faults for it.

For kernel mode code, typically the issue is one of preventing any page fault during elevated IPL execution. Kernel mode code, whether running as part of an image or as part of the executive, may be able to request the $LKWSET[_64] system service to lock pages into a process working set list. The $LKWSET[_64] system service, however, cannot be used to lock pages into the system working set list.

Code that runs at elevated IPL must also make its associated linkage section and any other data resident. Another issue for elevated IPL code is that the compiler may generate calls to Run-Time Library or other language support routines. These routines must also be made resident and furthermore must be appropriate for execution in kernel mode at elevated IPL.

OpenVMS Alpha provides two sets of MACRO-32 macros to facilitate locking P0 and P1 space code and linkage section pages into the working set list. One set is for use with image code intended to be locked for the duration of the image's execution. The other set is for use with code to be locked and unlocked. The two sets of macros should not be mixed in one image.

The first set consists of the macros $LOCKED_PAGE_START and $LOCKED_PAGE_END, which delimit the area to be locked by creating special program sections (PSECTs) for the code and its associated linkage section, and the macro $LOCKED_PAGE_INIT, which should be invoked from within initialization code in the image to generate the appropriate $LKWSET requests.

The other set of macros consists of $LOCK_PAGE and $UNLOCK_PAGE, which delimit the code to be locked. These macros can be invoked multiple times within an image. All delimited code is placed into a separate PSECT, and the linkage section associated with that code is also placed into a separate PSECT. Code generated by the

$LOCK_PAGE macro makes $LKWSET requests for both the code and linkage section areas, and code generated by the $UNLOCK_PAGE macro makes the corresponding $ULWSET requests.

These macros are described in more detail in *OpenVMS MACRO-32 Porting and User's Guide*. Both sets of macros are primarily intended for elevated IPL execution. Care must be taken to ensure that the delimited code does not call Run-Time Library or other procedures. The MACRO-32 compiler for OpenVMS Alpha generates calls to routines to emulate certain VAX instructions. An image that uses these macros must link against the system base image (using the /SYSEXE qualifier) to resolve references to emulation routine symbols with the routines supplied in a nonpageable executive image. These macros may not be suitable for all applications.

Example 5.1 shows an example of how to lock code and linkage section PSECTs from a C program.

Example 5.1 Locking C Code and Linkage into the Working Set

```
$
$ CC /OBJECT=TEST /list=test /machine SYS$INPUT:

#pragma module test_code "v1.0"

/*
//  Define the references to the linkage and code psects
*/
#pragma extern_model save
#pragma extern_model strict_refdef "$$C$LINKAGE_BEGIN" noshr
void *__linkage_begin ;
#pragma extern_model restore

#pragma extern_model save
#pragma extern_model strict_refdef "__C$LINKAGE_END" noshr
void *__linkage_end ;
#pragma extern_model restore

#pragma extern_model save
#pragma extern_model strict_refdef "$$C$CODE_BEGIN" shr
void *__code_begin ;
#pragma extern_model restore

#pragma extern_model save
#pragma extern_model strict_refdef "__C$CODE_END" shr
void *__code_end ;
#pragma extern_model restore


#include <stdio.h>

void test_routine()
  {
  printf("Test Routine") ;
  }

main(void)
  {
  int *lp;
```

Example 5.1 ***(continued)*** **Locking C Code and Linkage into the Working Set**

```
  printf("The addresses of the linkage section are:\n");
  printf("    begin: %08p       end: %08p\n",
    &__linkage_begin, &__linkage_end);

  printf("The addresses of the code section are:\n");
  printf("    begin: %08p       end: %08p\n",
    &__code_begin, &__code_end);

  printf("The address of main(linkage) is: %08p\n", main);
  printf("The address of test_routine(linkage) is %08p\n", test_routine);

  lp = (int*) &main;
  printf("The address of main(code) is: %08p\n", (void *) lp[2] );

  lp = (int*) &test_routine ;
  printf("The address of test_routine(code) is %08p\n", (void *) lp[2] ) ;

  return 1;
  }
$
$ LINK /MAP=TEST_CODE /CROSS/FULL/EXE=TEST_CODE TEST -
        + SYS$INPUT:/OPT
!
! Match code and linkage section psect attributes
!
psect= $$C$CODE_BEGIN,PIC,CON,REL,LCL,  SHR,  EXE,NOWRT,NOVEC,  MOD
psect= __C$CODE_END,PIC,CON,REL,LCL,  SHR,  EXE,NOWRT,NOVEC,  MOD
psect=$$C$LINKAGE_BEGIN,NOPIC,CON,REL,LCL,NOSHR,NOEXE,NOWRT,NOVEC,MOD
psect=__C$LINKAGE_END,NOPIC,CON,REL,LCL,NOSHR,NOEXE,NOWRT,NOVEC,MOD
$
$
```

The example program creates variables in specifically named PSECTs. The linker collects PSECTs with identical attributes into the same image section; it orders the PSECTs alphabetically by their names. The PSECT names specified by the program are names that will sort before and after the standard C compiler code and linkage section PSECT names. The example specifies linker options to give the delimiting PSECTs attributes identical to those of the standard C compiler code and linkage section PSECTs.

The result is that the variables are at the beginning and end of the code and linkage section image sections, and their addresses can be supplied as starting and ending addresses to the $LKWSET service.

In OpenVMS versions prior to OpenVMS Version 7.0, kernel mode code used bit PHD$V_NO_WS_CHNG as an alternative mechanism. The general sequence was to raise IPL to 2, set the bit, and fault the page or pages into the working set list. Setting this bit blocked swapper trimming, automatic working set limit adjustment, and proactive memory reclamation. This bit still exists, but with limited use: it is set only by process deletion code that runs after the process has been reduced to a single kernel thread, and it is tested only by the routines that initiate swapper trimming and automatic working set limit adjustment. Because this mechanism does not block working set replacement paging, it could not be extended to a process with multiple kernel threads.

The PHD$V_NO_WS_CHNG mechanism has been superseded by a mechanism that involves fields PCB$Q_KEEP_IN_WS and PCB$Q_KEEP_IN_WS2. These fields are initialized to an invalid process address, –1, and reset to that value after use. Kernel mode code records in these fields the starting and ending addresses of a range of virtually contiguous addresses that must remain in the working set. It then faults the page or pages into the working set, acquires the MMG spinlock to prevent further changes to the working set, and writes –1 to the fields to reset them. Use of the fields is synchronized through the inner mode semaphore (see Chapter *Kernel Threads*); only one kernel thread at a time can hold the semaphore and use them. Use of this mechanism is reserved to Hewlett-Packard Company; any other use is unsupported.

MMG$FREWSLE, the routine that removes entries from the working set list (see Section 5.3.1), reads these fields: if a page to be removed from the working set falls within the addresses in these fields, it leaves that page in the working set. MMG$FREWSLE is used for replacement paging, swapper trimming, automatic working set limit reduction, and proactive memory reclamation. Typically, it does not execute holding the inner mode semaphore. Two different methods are used to synchronize its reading the fields with a kernel thread's writing them:

- Acquiring the MMG spinlock

  MMG$FREWSLE executes holding the MMG spinlock. Kernel mode code that uses the PCB$Q_KEEP_IN_WS mechanism, such as MMG$LCKULKPAG (see Section 5.6.1), can acquire the MMG spinlock before writing the fields to block concurrent execution by MMG$FREWSLE.

- Testing bit PHD$V_FREWSLE_ACTIVE in PHD$L_FLAGS

  MMG$FREWSLE sets bit PHD$V_FREWSLE_ACTIVE and executes a memory barrier before reading PCB$Q_KEEP_IN_WS and PCB$Q_KEEP_IN_WS2 and clears the bit when done. MMG$IOLOCK, for example, called to lock a direct I/O buffer into memory (see Chapter *I/O System Services*), uses this method. It first writes the PCB fields and then tests whether the process has multiple kernel threads and, if so, executes a memory barrier and tests PHD$V_FREWSLE_ACTIVE. If the bit is set, it spins waiting for it to be cleared by MMG$FREWSLE before faulting the page into the working set.

Use of this mechanism is reserved to Hewlett-Packard Company; any other use is unsupported.

None of these alternatives is suitable for keeping pages in the system working set list, pages such as paged pool or pageable data in executive images. The $LKWSET[_64] system service rejects an attempt to lock system pages. The PCB$Q_KEEP_IN_WS mechanism is also not suitable because multiple threads of execution executing kernel mode code in multiple processes could concurrently attempt to lock system pages, and there is nothing to serialize their uses of the fields.

For kernel mode code that needs to fault S0/S1 space pages into the system working set list and have them remain there, two routines are provided. (Currently, all uses of S2 space are nonpageable.) The routine MMG$LOCK_SYSTEM_PAGES, in module LOCK_SYSTEM_PAGES, can lock pages into the system working set. For each page to be locked, the routine takes the following steps:

1. It faults the page.
2. It acquires the MMG spinlock.
3. It tests whether the page is still valid and, if not, releases the spinlock and returns to step 1.
4. It increments PFN$L_SHRCNT in the PFN database record for the physical page occupied by the virtual page, gets the WSLX from the PFN$L_WSLX_QW field, and sets the WSL$V_WSLOCK bit in the WSLE in the system working set list.
5. It releases the MMG spinlock.

When the caller no longer requires the pages to be resident, it calls MMG$UNLOCK_SYSTEM_PAGES, in module LOCK_SYSTEM_PAGES, which clears the WSL$V_WSLOCK bit and decrements the PFN$L_SHRCNT field for each page.

Locking many pages into a working set list is not always possible or desirable. In cases where elevated IPL execution is not an issue, a process can do the following to minimize page faults once the desired pages are in the working set:

- Reduce the chance of swapper trimming by entering the DCL command SET WORKING_SET/QUOTA=*authquota* and /EXTENT=*authquota*, where *authquota* is the authorized normal maximum working set limit. Unless the process is about to be outswapped, this prevents first-level swapper trimming by ensuring that the working set limit is not above the authorized maximum limit. A process about to be outswapped may have its working set size reduced to SWPOUTPGCNT.
- Disable automatic working set limit adjustment and second-level swapper trimming by entering the DCL command SET WORKING_SET/NOADJUST. This also blocks proactive memory reclamation from a process whose kernel thread or threads are classified as periodically waking.
- Lock itself into the balance set by requesting the Set Process Swap Mode ($SETSWM) system service in case, as a result of its execution characteristics, it is classified as a long-waiting process and becomes subject to proactive memory reclamation.
- Do not enable multiple kernel threads in the process and do execute a constrained sequence of already resident code that touches already resident data and linkage section pages. In general, such code must block AST delivery, cause no exceptions, signal no conditions, and call no procedures outside the address space already resident.

5.11 Relevant Source Modules

Source modules described in this chapter include

[CLIUTL]SETMISC.B32
[LIB]UAFDEF.SDL
[LIB]WSLDEF.SDL
[LOGIN]INITUSER.B32
[SYS]LOCK_SYSTEM_PAGES.MAR
[SYS]PAGEFAULT.MAR
[SYS]PHDUTL.MAR
[SYS]RSE.MAR
[SYS]SYS_LKWSET_64.C
[SYS]SYS_PURGWS_64.C
[SYS]SYSADJWSL.MAR
[SYS]SYSLKWSET.MAR
[SYS]SYSPURGWS.MAR
[SYS]TBI_ROUTINES.MAR
[SYSBOOT]SYSBOOT64.B64

Chapter 6
The Swapper

A time to cast away stones and a time to gather stones together . . .

Ecclesiastes 3:5

The amount of physical memory on the system is not a hard limit to the number of processes in the system. The OpenVMS Alpha operating system effectively extends physical memory by keeping a subset of active processes resident. It maximizes the number of such processes by limiting the number of pages that each process has in memory at any given time. Processes not resident in memory reside on mass storage in swap files; that is, they are outswapped.

The swapper process is the systemwide physical memory manager. Its responsibilities include maintaining an adequate supply of physical memory and ensuring that the highest priority computable kernel threads are resident in memory.

This chapter summarizes the top-level flow through the swapper process and concentrates on its inswap and outswap operations. Chapter 4 describes how the swapper writes modified pages to their backing store.

6.1 Overview

This section reviews some basic swapper concepts.

6.1.1 Swapper Responsibilities

The swapper has several main responsibilities:

- Ensuring that the balance set contains the most important processes
- Maintaining a minimum free page list size
- Maintaining a maximum modified page list size

Its first responsibility is to ensure that the currently resident kernel threads are the highest priority computable kernel threads in the system. When a nonresident kernel thread becomes computable, the swapper must bring its process back into memory if the kernel thread's priority and the available memory allow.

The swapper maintains the number of free pages (the sum of pages on the free and zeroed page lists) above the threshold established by the SYSGEN parameter FREELIM. Free physical pages are needed for resolving page faults and inswapping processes with computable kernel threads. The swapper reclaims memory to keep the number of free pages above FREELIM by means of four operations, described in more detail in subsequent sections:

1. The swapper deletes process headers (PHDs) of already deleted processes. It outswaps any PHDs and page tables that are associated with previously outswapped process bodies and that are eligible for outswap.
2. It calls the modified page writer routine to write modified pages.
3. It shrinks the working sets of one or more resident processes.
4. If necessary, the swapper selects an eligible process for outswap, shrinks its working set, and removes that process from memory. The table that determines outswap selection also determines the order in which processes are selected for working set reduction.

The swapper stops reclaiming pages when the number of free pages exceeds the SYSGEN parameter FREEGOAL.

The swapper ensures that there are fewer pages on the modified page list than the threshold established by the SYSGEN parameter MPW_HILIMIT. When the modified page list grows above this limit, the swapper calls the modified page writer routine to write the contents of some modified pages to their backing store and to move the physical pages to the free page list.

6.1.2 System Events That Trigger Swapper Activity

The swapper spends its idle time hibernating. Executive components that detect a need for swapper activity wake the swapper by calling routine SCH$SWPWAKE, in module RSE. In addition, SCH$SWPWAKE is called once a second from system timer code.

SCH$SWPWAKE performs a series of checks to determine whether there is a real need for the swapper to run. If so, it awakens the swapper. If not, it simply returns. Performing these checks in SCH$SWPWAKE rather than in the swapper process itself avoids the overhead of two needless context switches.

Table 6.1 lists the system events that trigger a possible need for swapper activity, the module containing the routine that detects each need, and the action the swapper takes in response.

Table 6.1 Events That May Cause the Swapper to Be Awakened

System Event	Routine Name (Module)	Swapper Action
Kernel thread that is outswapped becomes computable	SCH$CHSE (RSE)	The swapper attempts to make its process resident.
Quantum end	SCH$QEND (RSE)	The swapper may be able to perform an outswap previously blocked by initial quantum flag setting or kernel thread priority.
Modified page list exceeds upper limit	MMG$DALLOCPFN, MMG$INS_PFNH/T (ALLOCPFN)	The swapper writes modified pages.
Free page list drops below low limit	MMG$REM_PFN[H] (ALLOCPFN)	The swapper increases the free page count, taking the steps summarized in Section 6.1.1.
Balance set slot of deleted process becomes available	DELETE_IN_SYS_ CONTEXT (SYSDELPRC)	The swapper can delete the PHD and may be able to perform a previously blocked inswap.
PHD reference count goes to zero	MMG$DECPHDREF[1] (PAGEFAULT)	The swapper can outswap a PHD and page tables to join the previously outswapped process body.
Powerfail recovery	EXE$RESTART_ CONT (POWERFAIL)	The swapper queues a power recovery AST to any process that requested one.
System timer subroutine executes once a second	EXE$TIMEOUT (TIMESCHDL)	The swapper is awakened if there is any work for it.

The swapper can be awakened in another, more indirect way: clearing the cell that contains the modified page list high limit so that a subsequent test for whether the list size exceeds its high limit will fail. The routine MMG$PURGE_MPL, in module WRTMFYPAG, uses this method. This routine, called to request the writing of modified pages, is described in Chapter 4.

6.1.3 Swapper Implementation

The swapper is implemented as a separate process whose single kernel thread has a priority of 16, the lowest real-time priority. It is selected for execution like any other kernel thread in the system.

The swapper executes entirely in kernel mode. All swapper code resides in system space. Except for some initialization code, all swapper code is in module SWAPPER. The swapper's stack and almost all its data are also in system space. The swapper has one page of P1 space to eliminate the need for a number of special-case checks for swapper process context.

With the removal of process page tables from system space in OpenVMS Alpha Version 7.0, the swapper has no virtual access to another process's page tables. In the course of inswap and outswap, it therefore temporarily adopts the address space of the target process to access its page tables. That all its code and most of its data are in system space enables the swapper to access them from any set of process page tables.

The swapper serves as a convenient process context for several system functions. In particular, during system initialization it performs those initialization tasks that require process context and must be performed prior to the creation of any other process, for example, initializing paged pool and creating the SYSINIT process. Chapter *Operating System Initialization and Shutdown* describes these functions of the swapper.

In addition, the file system uses the swapper as a process context for the execution of certain asynchronous system trap (AST) procedures. Clusterwide file system cache coherency and volume locking are implemented through system-owned file system locks (see Chapter *Lock Management* and Appendix *Lock and Resource Use by OpenVMS Components*). When one VMScluster node's lock blocks a second node's progress, the second requests execution of a blocking routine on the first. Running in system context on the first node, the blocking routine queues an AST to the swapper process. Running in process context on the first node, the AST procedure can request standard system services to convert the associated lock to a less restrictive mode or dequeue it.

6.2 Swapper Use of Memory Management Data Structures

Chapter 2 describes the memory management data structures used by both the page fault handler and the swapper. The discussion here reviews those structures and adds descriptions of the structures used exclusively by the swapper.

6.2.1 Process-Private Structures

The information used by the swapper in managing the details of inswapping or outswapping is contained in the following structures:

- Working set list of the process to be outswapped or inswapped
- Process-private page tables
- Process header BAK array

The working set list describes the portion of a process's virtual address space that must be written to the swap file or otherwise dealt with when the process is outswapped. When the process is inswapped, the working set list describes the process pages in the swap file. The swapper's scan of the working set list at outswap is discussed in Section 6.5.

The working set list does not supply the swapper with all the information necessary to outswap a process. Other information about a virtual page is contained in its page table entry (PTE) or in the page frame number (PFN) database record for that physical page. Each working set list entry (WSLE) effectively points to a PTE that contains a PFN. When outswapping, the swapper copies the PTE contents to a quadword array called the swapper map (see Section 6.2.2). It then inserts the contents of the PFN$Q_BAK field for this physical page into the PTE, dissociating the process from the physical memory that its virtual page occupied.

In the course of outswapping, the swapper links the process's level 2 page tables (L2PTs) and level 3 page tables (L3PTs) together using first the PFN$Q_BAK array elements and then the PTEs that map the page tables (see Section 6.5.3.5).

PHD pages are also part of a process's working set. These pages reside in system space; system space level 3 page table entries (L3PTEs) map the balance set slot in which the PHD resides. As part of outswapping, the swapper dissociates the PHD pages from their L3PTEs so that it can reuse the balance set slot. Thus, unlike those of process pages, PHD pages' L3PTEs are not available to hold these pages' backing store addresses while they are outswapped.

Instead, when a process is outswapped, the contents of the PFN$Q_BAK field for each PHD page currently in the working set are stored in the corresponding array element in the PHD page BAK array (see Chapter 2). When the process is inswapped, the PHD page array can be scanned and the BAK contents copied from the array back into the PFN$Q_BAK fields of the PFN database records for the physical pages that contain the PHD.

Entries representing the process's page tables can be scattered throughout the working set list. As the swapper scans the working set list to prepare the process body for outswap, it links the page tables into lists through the low longwords of their PFN$Q_BAK fields: each page table's PFN$Q_BAK contains the working set list position, or WSLX, of the next page table in the list.

The swapper follows the chains in preparing the page tables for outswap and, subsequently, in reestablishing the process's page tables after inswap. Sections 6.5.3.5 and 6.6.3 have more information.

6.2.2 Swapping I/O Data Structures

At system initialization, the swapper allocates physical pages for the swapper map and system space L3PTEs to map it. The swapper map is an array of quadwords whose address is stored in the global cell SWP$GL_MAP. The number of quadwords in the array is the number of pages equivalent to the value of the SYSGEN parameter WSMAX, which is in units of pagelets.

The swapper map is a pseudo page table. It describes the working set of a process to be outswapped or inswapped. Each entry represents one page in the working set. The swapper map can describe only one outswap or one inswap operation at a time.

At outswap, for each page in the working set, the swapper reads the PTE that maps it and stores the PFN of that page in an element of the swapper map. It passes the address of the beginning of the swapper map to the I/O system as the system virtual address of the L3PTE that maps the first page of the I/O buffer. The swap image is output from these pages to the swap file. Thus, the swapper map transforms a collection of virtually noncontiguous pages into virtually contiguous pages that can be transferred in one or more I/O requests.

At inswap, the swapper allocates physical pages of memory for the working set being inswapped and records their PFNs in the swapper map. It passes the address of the beginning of the swapper map to the I/O system as the system virtual address of the L3PTE that maps the first page of the I/O buffer. The swap image is input into these pages. As the swapper rebuilds the process's working set list and page tables, it copies the PFN from each swapper map entry to the appropriate system or process PTE.

Like the page fault handler, the swapper makes standard I/O requests. During system initialization, it allocates an I/O request packet (IRP) to be used for swap I/O. Because most disk drivers execute as kernel processes (see Chapter *Software Interrupts*), the swapper also allocates a kernel process block and physical memory for a kernel process stack. The preallocation prevents any possible deadlock when an outswap is requested to free memory because there are not enough free pages.

To perform an inswap or outswap, the swapper initializes some of the IRP fields that will be interpreted in a special manner by the I/O postprocessing routine. It then calls one of the swapper I/O entry points in module SYSQIOREQ (EXE$BLDPKTSWPR or EXE$BLDPKTSWPW) that fills in an appropriate function code and queues the packet to the appropriate disk driver. Tables 4.2 to 4.4 show how the IRP is used by the swapper for its I/O activities.

As described in Chapter 4, the swapper also uses preallocated IRPs for modified page writing.

Certain swapper operations complete asynchronously. The swapper maintains two bits in the cell SCH$GL_SIP as signals of ongoing operation: when set, SCH$V_SIP means that an inswap or outswap is in progress and is described by the swapper map; when set, SCH$V_MPW means that modified page writes are in progress.

6.2.3 Swap File Data Structures

The system maintains a page file control block for each page and swap file in the system. Figure 2.31 shows the layout of this data structure and describes its fields. Both page and swap files can be used for swapping if SYSGEN parameter NOPGFLSWP is clear. If it is set, only swap files can be used for swapping. By default it is clear.

During system initialization, the SYSINIT process opens the primary swap file SYS$SPECIFIC:[SYSEXE]SWAPFILE.SYS, if it exists, and initializes its page file control block. When any additional swap file is installed (with the SYSGEN command INSTALL), SYSGEN initializes its page file control block.

In early versions of VAX VMS, the executive required that there be a swap slot large enough to outswap the process at its current size, up to the maximum of its authorized quota. When a process was created, space for its working set was assigned in the first swap file with enough free space. When the process working set grew too large for the swap space, a replacement swap slot was allocated. When the working set limit was adjusted at image reset, a smaller swap slot was allocated. Each swap slot consisted of virtually contiguous blocks within a single swap file.

In VAX VMS Version 5, swap space allocation changed considerably, reflecting the fact that processes are outswapped relatively infrequently and that they are typically outswapped with shrunken working sets. Swap space is not assigned until a process is selected for outswap, subsequent to any swapper trimming. The executive attempts to allocate virtually contiguous space in a single swap or page file. If that fails, however, it allocates multiple file extents in a number of swap and page files. (A file extent is a group of consecutively numbered logical blocks.) This approach requires less dedicated swap file space than did early VAX VMS versions and results in less fragmentation of swap and page files. The overhead of allocating and deallocating seldom-used swap space has been eliminated.

Based on VAX VMS Version 5, the OpenVMS Alpha executive allocates swap space in a similar manner; the one difference is that swap space is allocated in units of pages rather than disk blocks.

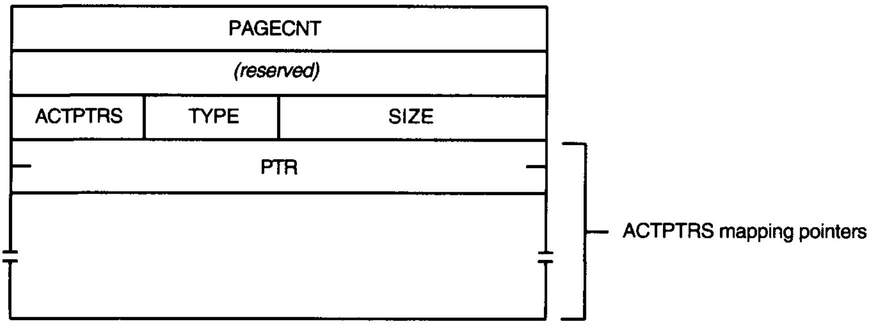
When a process is outswapped, its process control block (PCB) remains resident. In particular, two fields in the PCB of an outswapped process contain information necessary to inswap the process: PCB$L_WSSWP, the location of its swap space, and PCB$L_SWAPSIZE, the low 31 bits of which represent the swap space's size in pages.

The value in PCB$L_WSSWP has several interpretations, depending on the value in PCB$L_SWAPSIZE:

- When a process is first created, its PCB$L_WSSWP is zeroed to indicate to the swapper that this process must be initialized from the shell.
- The high-order bit set in PCB$L_SWAPSIZE indicates that the swap space consists of a single extent. The upper byte of PCB$L_WSSWP is a longword index into the page-and-swap-file vector (see Figure 2.31). The indexed element of the array contains the address of the page file control block that describes the process's swap file. The other three bytes specify the starting page number of the swap space.
- If the high-order bit of PCB$L_SWAPSIZE is clear, PCB$L_WSSWP contains the system virtual address of a nonpaged pool data structure called a page file map (PFLMAP). Whenever the swap space consists of more than one extent, the swapper allocates a PFLMAP and initializes one pointer for each extent.

Figure 6.1 shows the layout of a PFLMAP. PFLMAP$L_PAGECNT is the total number of pages described in all the PFLMAP's pointers. PFLMAP$W_SIZE and PFLMAP$B_TYPE are the standard dynamic data structure fields. A PFLMAP has space for 64 pointers. PFLMAP$B_ACTPTRS is the number of pointers actually in use. The pointers begin at offset PFLMAP$Q_PTR.

Figure 6.1 Layout of a Page File Map (PFLMAP)



Each pointer is a quadword. Its first longword contains a swap file index and starting page number, just like the contents of PCB$L_WSSWP for a single-extent swap space. The second longword contains the number of pages in the extent. Bit 31 is set in the second longword of the last pointer to flag it as the end.

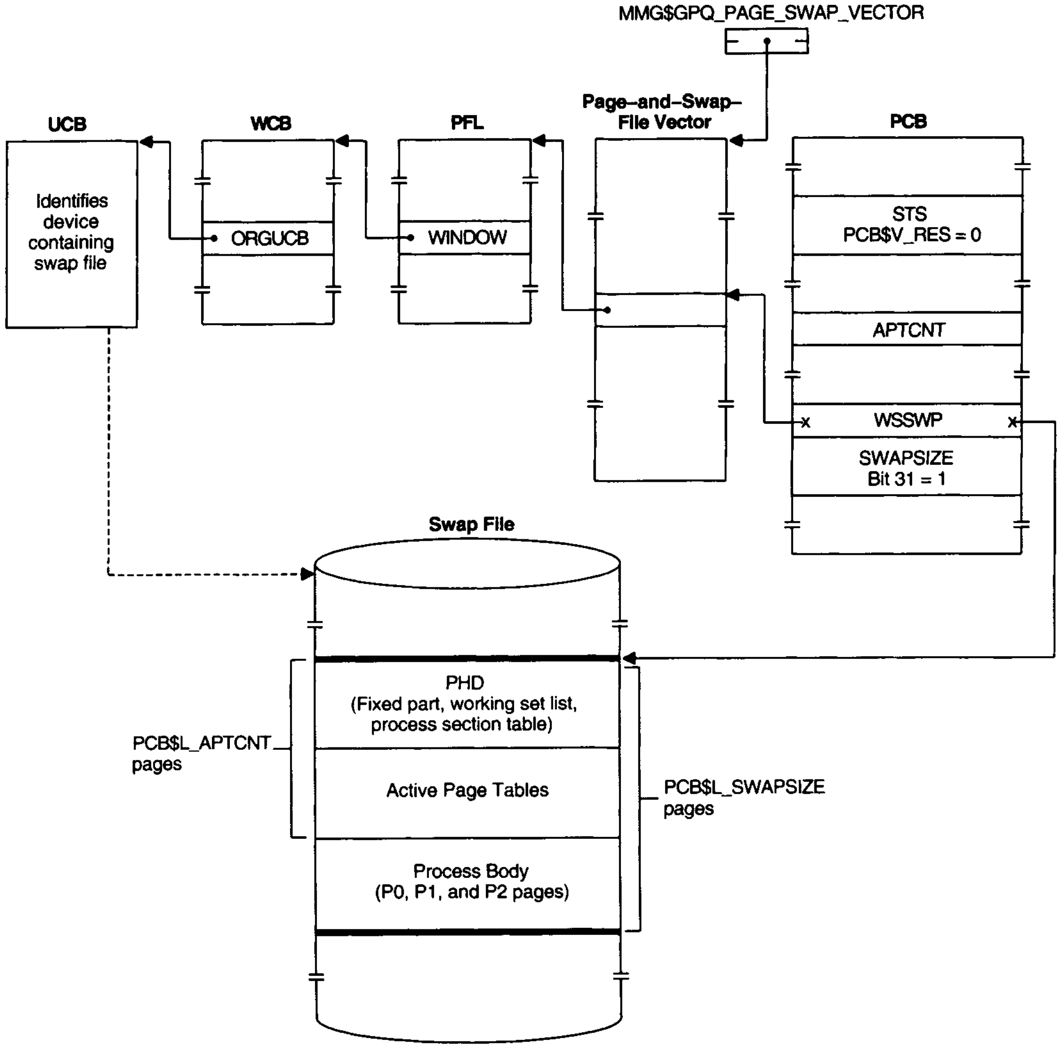
In the case of a single-extent swap space, PCB$L_SWAPSIZE contains the size of the slot, with bit 31 set to indicate it is the only pointer. Thus, the executive can treat the quadword beginning at PCB$L_WSSWP as a pointer with the same form as one in a PFLMAP.

Figure 6.2 shows the relations among the data structures involved in swap file use and also the structure of a single-extent swap space. The upper byte of PCB$L_WSSWP indexes the page-and-swap-file vector array element that contains the address of the page file control block for that swap file. The page file control block field PFL$L_WINDOW contains the address of the window control block (WCB) describing the location of the swap file on a mass storage medium. The field WCB$L_ORGUCB contains the address of the unit control block (UCB) for that device.

Within the swap file, the process's slot begins at the page whose number is in the low three bytes of PCB$L_WSSWP. It must contain room for the PHD, process-private page tables, and the process body (the P0, P1, and P2 space pages belonging to the process). The total size of the swap space is contained in PCB$L_SWAPSIZE. It is the smallest multiple of system cell SWP$GW_SWPINC large enough to accommodate the process's working set size, which is the sum of PCB$L_PPGCNT and PCB$L_GPGCNT, its process-private and global page counts. During system initialization, SWP$GW_SWPINC is set to the same value as the modified page write cluster factor, SYSGEN parameter MPW_WRTCLUSTER.

The field PCB$L_APTCNT contains the number of pages of space reserved for the PHD and page tables. This field has no meaning for a resident process; the swapper calculates its value when scanning the working set list of a process about to be outswapped. They are positioned in the following order: PHD pages, level 1 page table (L1PT), L2PTs, and L3PTs.

Figure 6.2 Swap File Database



6.3 Swapper Main Loop

The swapper does not determine why it was awakened. Every time it is awakened, it tends to all the tasks for which it is responsible. The main loop of the swapper consists of the following steps:

1. It calls local routine BALANCE, which tests the number of free pages.
   - If there are sufficient free pages, but there are deleted PHDs to clean up, BALANCE calls local routine OUTSWAP to clean up a deleted PHD.

   - If there are insufficient free pages and the size of the modified page list is large enough, BALANCE requests the writing of modified pages to make up the deficit; otherwise, it calls OUTSWAP, which may trigger the shrinking of process working sets in addition to cleaning up a deleted PHD and possibly outswapping a process.

   Section 6.3.1 describes BALANCE in more detail.

2. The swapper calls the modified page writer routine, MMG$WRTMFYPAG, in module WRTMFYPAG, which initiates modified page writing in response to any pending requests. For example, if the size of the modified page list exceeds its current upper limit, modified pages are written until the size of the list falls below the SYSGEN parameter MPW_LOWAITLIMIT. Chapter 4 describes the modified page writer.

3. The swapper calls local routine SWAPSCHED to identify the highest priority computable outswapped kernel thread. If there is none, SWAPSCHED returns. Otherwise, it calculates the size of that process's working set and tests whether there are enough free pages to accommodate it without reducing the number of free pages below its minimum, SYSGEN parameter FREELIM.

   - If there are enough pages, SWAPSCHED calls local routine INSWAP (see Section 6.6) to initiate the inswap.
   - If there are not enough pages, SWAPSCHED calls local routine OUTSWAP (see Section 6.3.3) to make up the free page deficit.

   Section 6.3.2 discusses SWAPSCHED in more detail.

4. Because the swapper is a system process that executes fairly frequently, it is a convenient vehicle for testing whether a powerfail recovery has occurred and, if so, notifying all processes that have requested power recovery AST notification through the Set Power Recovery AST ($SETPRA) system service. This mechanism is currently unused because of lack of hardware support for powerfail recovery.

5. Finally, the swapper puts itself into the hibernate state, after checking its wake pending flag. If any thread of execution, including the swapper itself in one of its routines, has requested swapper activity since the swapper began execution, the hibernate is skipped and the swapper goes back to step 1.

6.3.1 The BALANCE Routine

Figure 6.3 shows the basic decisions and flow of the BALANCE routine. In the figure, FREECNT refers to the contents of SCH$GL_FREECNT, the sum of the number of pages on the free and zeroed page lists, and MFYCNT refers to the contents of SCH$GL_MFYCNT, the number of pages on the modified page list. The numbers in the figure correspond to those in the following list:

BALANCE takes the following steps:

❶ BALANCE acquires the MMG and SCHED spinlocks, raising interrupt priority level (IPL) to IPL$_MMG.

❷ It subtracts the desired size of the free page list, the SYSGEN parameter FREEGOAL, from the number of free pages (the contents of SCH$GL_FREECNT). It stores the difference in R3 as its working copy of the free page deficit. If the number of free pages is larger than FREEGOAL, BALANCE goes on to step 6.

❸ If the number of free pages is smaller than FREEGOAL, BALANCE tests whether modified page writing is already in progress. If it is, BALANCE checks whether enough modified pages are being written to make up the difference between the number of free pages and SYSGEN parameter FREEGOAL. (The number of free pages will be replenished to a target size of FREEGOAL pages.)

— If so, it continues with step 6.

— If not, it continues with step 8.

❹ If modified page writing is not in progress, BALANCE tests whether the modified page list contains as many pages as the SYSGEN parameter MPW_THRESH. If the threshold has been reached, BALANCE further tests that the difference between the list's current size and its low limit (the SYSGEN parameter MPW_LOLIMIT) is large enough to satisfy the deficit. That is, the modified page list must contain enough pages to pass both tests before the swapper can replenish the free page list from it. If the modified page list is not large enough, BALANCE goes to step 8.

❺ If the modified page list is large enough, BALANCE calls MMG$PURGE_MPL, in module WRTMFYPAG, to request that enough pages be written from the modified page list to make up the free page deficit. (Chapter 4 describes MMG$PURGE_MPL and the modified page writer.) BALANCE releases the spinlocks and returns.

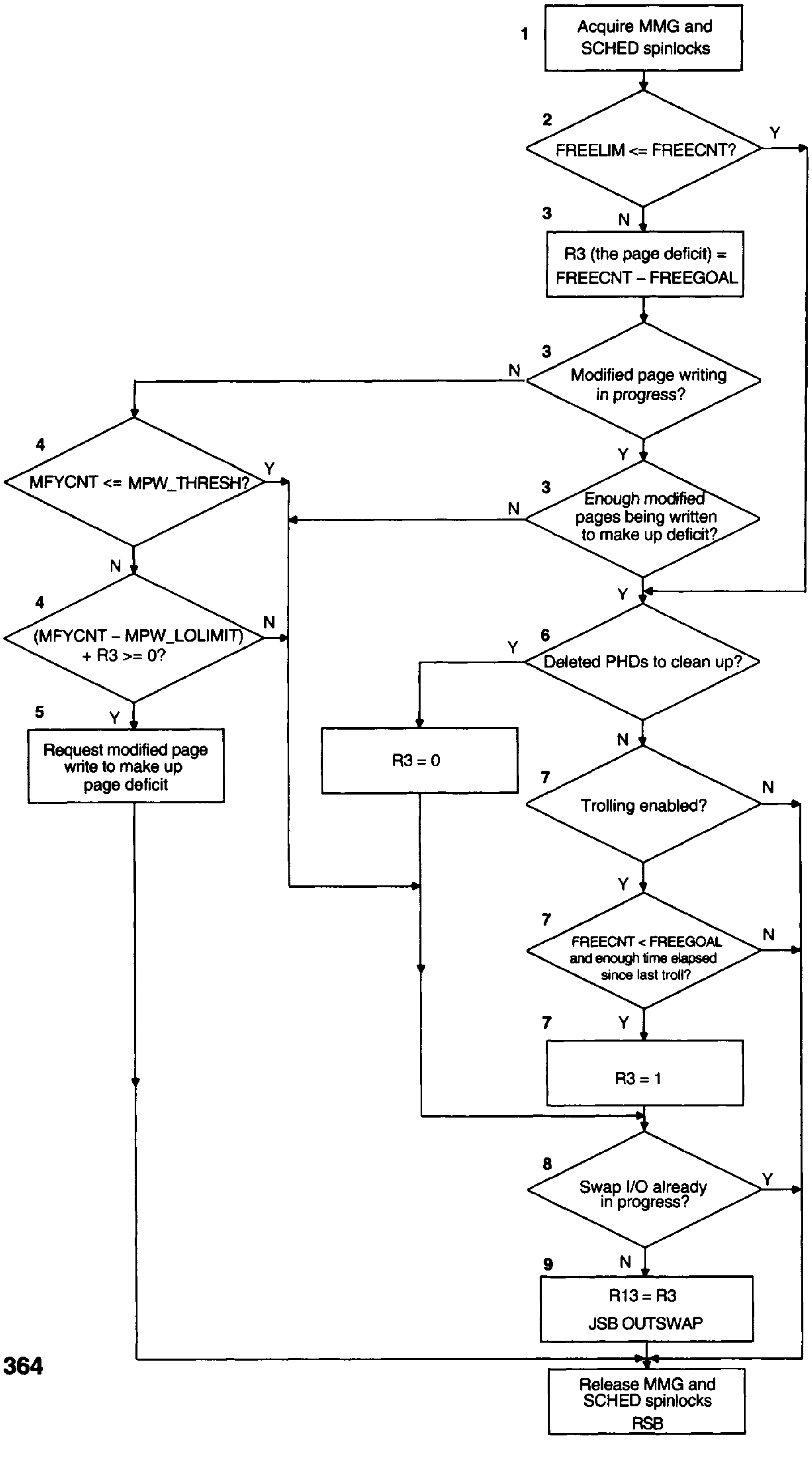
❻ BALANCE tests whether there are any PHDs belonging to deleted processes from which to reclaim memory and, if so, clears R3 and continues with step 8.

❼ If there are no deleted PHDs, BALANCE tests bit 1 in SYSGEN parameter MMG_CTLFLAGS to see if the mechanism known as trolling is enabled. If trolling is enabled, BALANCE tests whether there are fewer free pages than FREEGOAL and whether enough time has elapsed since the last troll attempt. If both are true, it initializes R3 to 1 to indicate that OUTSWAP should troll for a suitable process to outswap proactively.

❽ BALANCE tests and sets SCH$V_SIP in SCH$GL_SIP. If the swapper already has an inswap or outswap in progress, BALANCE releases the spinlocks and returns.

❾ If no swap I/O is in progress, BALANCE transfers to routine OUTSWAP, with R13 a copy of R3 and SWP$GB_ISWPRI set to zero. Section 6.3.3 discusses OUTSWAP and the meaning of its arguments.

Figure 6.3 BALANCE Operations


1
Acquire MMG and SCHED spinlocks
2
FREELIM <= FREECNT?
Y
3
N
R3 (the page deficit) = FREECNT – FREEGOAL
3
N
Modified page writing in progress?
Y
3
N
Enough modified pages being written to make up deficit?
Y
4
MFYCNT <= MPW_THRESH?
Y
N
4
(MFYCNT – MPW_LOLIMIT) + R3 >= 0?
N
5
Y
Request modified page write to make up page deficit
6
Y
Deleted PHDs to clean up?
R3 = 0
N
7
Trolling enabled?
N
Y
7
FREECNT < FREEGOAL and enough time elapsed since last troll?
N
7
Y
R3 = 1
8
Swap I/O already in progress?
Y
9
N
R13 = R3
JSB OUTSWAP
Release MMG and SCHED spinlocks
RSB

6.3.2 The SWAPSCHED Routine and Selection of Inswap Process

To select the outswapped process with the highest priority computable kernel thread, SWAPSCHED takes the following steps:

1. It acquires the MMG spinlock.
2. It tests and sets bit SCH$V_SIP in SCH$GL_SIP. If the bit was already set, indicating that the swapper map is in use, SWAPSCHED releases the spinlock and returns.

   Otherwise, it acquires the SCHED spinlock to synchronize access to the scheduling database.
3. It selects the highest priority nonempty computable outswap (COMO) queue. It removes a kernel thread from that queue, if one exists, to inswap its process.

   The scheduling subsystem maintains 64 quadword listheads for COMO kernel threads, one for each software priority (see Chapter *Scheduling*). These queues are identical to the 64 queues of the computable resident (COM) kernel threads. The steps taken by the swapper to decide which kernel thread to inswap parallel those taken by the rescheduling interrupt service routine (see Chapter *Scheduling*) to select the next kernel thread for execution.
4. If there is no COMO kernel thread, SWAPSCHED clears SCH$V_SIP, releases the spinlocks, and returns.
5. If a COMO kernel thread exists and there are enough pages for its working set, SWAPSCHED calls INSWAP to read the kernel thread's process into memory.
6. If a COMO kernel thread exists but there are insufficient pages for its working set, SWAPSCHED attempts an optimization aimed at minimizing swapping on systems with more compute-bound processes than can fit into available memory. It makes two checks. One is whether the kernel thread's priority is no higher than the SYSGEN parameter DEFPRI, the default kernel thread priority. The other is whether less time than the SYSGEN parameter SWPRATE (a time interval with a default value of 5 seconds) has elapsed since the last inswap of a process with a kernel thread priority as low as DEFPRI. If both are true, SWAPSCHED abandons the inswap.

   Otherwise, it sets SWP$GB_ISWPRI to the priority of the inswap kernel thread and R13 to the complement of the free page deficit and calls OUTSWAP to reclaim enough memory for the inswap.

Whenever enough pages become available, the swapper executes the INSWAP routine (see Section 6.6.2) to initiate reading the outswapped process with the highest priority kernel thread into memory. Later, after the inswap I/O request completes, the swapper rebuilds the working set list and process page tables. The swapper calls routine SCH$CHSEP, in module RSE, to change the state of the newly inswapped process's kernel threads. Section 6.6 describes these steps in more detail. The newly inswapped kernel thread will be scheduled when the processor (or a member of a symmetric multiprocessing system) is available and the kernel thread is the highest priority computable resident kernel thread.

6.3.3 The OUTSWAP Routine

The swapper executes the OUTSWAP routine to perform one or more tasks related to memory reclamation. OUTSWAP is entered with the MMG and SCHED spinlocks held. It has one explicit argument, the contents of R13, the desired function:

- A value of 0 means OUTSWAP is to free a deleted PHD or outswap a PHD and page tables to join their outswapped process body.
- A value of 1 means that OUTSWAP is to outswap a suitable process proactively.
- A value of 80000000_{16} means OUTSWAP is to free a balance set slot, either by outswapping a PHD and page tables or, less immediately, by outswapping a process body.
- Any other negative value is the complement of the free page deficit that OUTSWAP is to make up in any way possible.

OUTSWAP has one implicit argument, SWP$GB_ISWPRI, which contains zero or the priority of the inswap candidate. SCH$OSWPSCHED, called by OUTSWAP, compares this priority to that of certain kernel threads to determine if they are suitable candidates for shrinking or outswapping. Because an internal priority of zero represents the highest priority, when SWP$GB_ISWPRI is zero all those kernel threads are considered suitable. Section 6.4 provides details on the selection of shrink and outswap candidates.

OUTSWAP takes the following steps:

1. If R13 contains the value 1, OUTSWAP continues with step 7.
2. Otherwise, it first attempts to reclaim memory by releasing the PHD of a previously deleted process or by outswapping the PHD and page tables of a previously outswapped process. It scans the PHD reference count array for a suitable header.
3. If OUTSWAP finds a PHD with a zero reference count, it tests the corresponding PHV$GL_PIXBAS array element.

   — If it contains –1, the process has been deleted and the swapper can release its PHD slot and its L1PT

   In routine DELPHD, the swapper dissociates the process from all assigned page files. DELPHD scans the system space L3PTEs that map the slot, releases any valid pages to the free page list, and deallocates any page file backing store associated with any invalid pages. When done, it calls MMG$DINS_PRCPGFLS, in module PAGEFILE, in case a pending page file deinstallation can be carried out now that all page file backing store associated with the process has been released.

DELPHD inserts the L1PT into the free page list. It invalidates all translation buffer (TB) entries to remove stale translations representing the deleted PHD slot. It clears the PHV$GL_PIXBAS array element and changes the PHD reference count to –1. It returns to OUTSWAP, which returns to its caller.

— If the corresponding PHV$GL_PIXBAS array element contains a positive value, the process has been outswapped and its PHD and page tables can be outswapped as well, as described in Section 6.5.4.2. After the I/O is initiated, control returns to OUTSWAP's caller.

4. If the PHD has a nonzero reference count and belongs to an outswapped process, OUTSWAP determines whether the page tables map any pages locked in memory by testing bits PCB$V_PHDLOCK and PCB$V_FREDLOCK in PCB$L_STS2. If so, the PHD and page tables cannot be outswapped, and OUTSWAP returns to step 2 to scan for another PHD.

   If the page tables map no locked pages, OUTSWAP usually records the slot number of the process and returns to step 2 to continue the scan, in case there is a deleted PHD to clean up. To avoid always picking the same slot to outswap, one time in eight OUTSWAP does not record the slot number of the first candidate.

5. After scanning all the slots without finding one that contains the PHD of a deleted process, OUTSWAP checks whether it has found a PHD belonging to an outswapped process. If so, it takes the steps described in Section 6.5.4.1 to attempt to sever all the connections between the PHD, page tables, and memory so the PHD and page tables can be outswapped. If the reference count goes to zero, outswap of the PHD and page tables is initiated and control returns to OUTSWAP's caller.

   If the reference count does not go to zero, the page tables probably map modified pages, which must be written first. OUTSWAP calls MMG$PURGE_MPL, in module WRTMFYPAG, to request that any modified pages mapped by that process's page tables be written when modified page writing is initiated.

   OUTSWAP returns to step 2, to scan for another PHD.

6. If OUTSWAP scans all the balance set slots without finding a PHD to release or outswap, it tests R13.

   — If the argument is zero, OUTSWAP returns to its caller.

   — If the argument is negative, OUTSWAP continues with the next step.

7. OUTSWAP calls SCH$OSWPSCHED, in module OSWPSCHED. Depending on the contents of R13, SCH$OSWPSCHED may shrink one or more working sets, select a process to outswap, or both. Unless the process has disabled automatic working set limit adjustment or is a real-time process, its working set limit is shrunk with its working set. Section 6.4 describes its operations.

Whenever SCH$OSWPSCHED shrinks a working set, it checks whether the free page deficit has been made up. If the deficit has not yet been made up, it makes checks similar to those previously described to determine whether writing the modified page list is appropriate and whether it would satisfy the deficit. If it would, SCH$OSWPSCHED calls MMG$PURGE_MPL to request that enough modified pages be written to make up the free page deficit.

When SCH$OSWPSCHED selects a process to outswap, it first shrinks the process's working set. In previous OpenVMS versions, the process's working set was shrunk to its normal maximum working set quota (WSQUOTA). As of OpenVMS Version 7.2, SCH$OSWPSCHED tries to shrink the working set to the number of pages represented by SYSGEN parameter SWPOUTPGCNT (which is in units of pagelets).

This change in behavior reduces the amount of physical memory needed to inswap a process and thus the maximum number of fluid physical pages that OpenVMS must maintain to inswap. Previously, OpenVMS had to maintain enough fluid pages to inswap a process with a working set list whose size was the number of pages represented by WSMAX. On a system with a large value for WSMAX, this requirement could severely limit the use of physical memory, resulting in failures to expand nonpaged pool, the lock ID table, and the system page table.

If the deficit has still not been made up by shrinking the process or if a balance set slot is needed for a process to be inswapped, SCH$OSWPSCHED then allocates swap space for the process's working set and reports a SWPOUT scheduling event to change the process's kernel threads' scheduling states from resident ones to outswapped ones.

8. If SCH$OSWPSCHED returns with an identified outswap candidate, OUTSWAP takes the steps described in Section 6.5 to outswap that process. After initiating the I/O to outswap the process body, OUTSWAP returns to its caller. Later, after the process body outswap I/O completes, the process header may be outswapped as well.

   If SCH$OSWPSCHED returns without an identified outswap candidate, OUTSWAP simply returns to its caller.

6.4 Selection of Shrink and Outswap Processes

When the swapper needs physical memory or a balance set slot, it calls the routine SCH$OSWPSCHED. The swapper specifies that it needs a certain number of pages of memory, that it needs a balance set slot, or that a suitable process, if any, should be swapped proactively. SCH$OSWPSCHED can shrink the working sets of selected processes, select a process to be outswapped, or perform both operations.

When bit 1 of SYSGEN parameter MMG_CTLFLAGS is set, the mechanism known as trolling is enabled. As its first and possibly only action, SCH$OSWPSCHED searches for a suitable process to outswap proactively. The search is driven by the TROLL table, described in Section 6.4.1. Section 6.4.2 describes how this table is used.

If bit 1 of MMG_CTLFLAGS is clear, or if the trolling routine found no suitable process to outswap, SCH$OSWPSCHED searches more extensively for processes to shrink or swap. Its search is also table-driven. Section 6.4.1 describes the OSWPSCHED table, and Section 6.4.3, how the table is used.

If SCH$OSWPSCHED is entered to troll or to free a balance set slot, it attempts to shrink a suitable process's working set to the number of pages represented by the SYSGEN parameter SWPOUTPGCNT.

If SCH$OSWPSCHED is entered to reduce the free page deficit, it can perform two levels of shrinking: in first-level trimming, it shrinks an extended working set back to the normal maximum working set limit (WSQUOTA); in second-level trimming, it attempts to shrink a working set to the number of pages represented by SWPOUTPGCNT. Before performing any second-level trimming, it performs first-level trimming of all suitable processes. (Chapter 5 describes the distinction between working set size, limit, and quota.)

Whenever it gains free pages from shrinking a working set, it checks whether there are enough pages on the free and modified page lists to satisfy the swapper's need. If enough pages are available, SCH$OSWPSCHED returns. If SCH$OSWPSCHED selects a process to be outswapped, it returns to the swapper, which is responsible for the actual outswap.

6.4.1 The OSWPSCHED and TROLL Tables

This section describes both the traditional OSWPSCHED table and the table used by the trolling routine. Because the table that drives the trolling routine is a subset of the OSWPSCHED table, the OSWPSCHED table is described first.

The OSWPSCHED table is divided into sections, each specifying one or more resident kernel thread scheduling states and a set of conditions associated with each state. Table 6.2 lists the individual entries and sections in the OSWPSCHED table. States in the same section are considered equivalent. Selection of shrink and outswap candidates depends on the factors named in the column heads of Table 6.2.

A kernel thread in the scheduling state computable to be scheduled (COM TBS) is a computable class-scheduled thread whose class has run out of quantum. Chapter *Scheduling* describes scheduling states, class scheduling, and the TBS and other state queues.

Table 6.2 OSWPSCHED Table

State	I/O	Priority	Initial Quantum	Long Wait	Dormant	Flags
SUSP	No buffered	n/a	n/a	n/a	n/a	Swap (SWAPASAP)
SUSP	Buffered	n/a	n/a	n/a	n/a	Second (SWPOGOAL)[2]
COM	n/a	n/a	n/a	n/a	Yes	First only (LVL1_ TRIM)
HIB LEF	n/a No direct	n/a n/a	n/a n/a	Yes Yes	n/a n/a	Second[2] Second[2]
CEF	No direct	n/a	n/a	n/a	n/a	Second[2]
HIB LEF	n/a No direct	n/a n/a	n/a n/a	No No	n/a n/a	Second[2] Second[2]
COM TBS	n/a	Yes[1]	Yes	n/a	No	First only
FPG COLPG	n/a n/a	Yes Yes	n/a n/a	n/a n/a	n/a n/a	
MWAIT	n/a	n/a	n/a	n/a	n/a	
CEF LEF	Direct Direct	Yes Yes	Yes Yes	n/a n/a	n/a n/a	

[1] This constraint is not present in the OSWPSCHED table; however, it is present in the algorithm and thus shown here.

[2] This flag is obsolete, but still present.

Table 6.2 *(continued)* OSWPSCHED Table

State	I/O	Priority	Initial Quantum	Long Wait	Dormant	Flags
PFW	n/a	Yes	Yes	n/a	n/a	
COM	n/a	Yes[1]	Yes	n/a	No	

[1] This constraint is not present in the OSWPSCHED table; however, it is present in the algorithm and thus shown here.

In general, SCH$OSWPSCHED scans the scheduling queues in the order shown in the State column. It checks whether any kernel thread in that state queue satisfies the conditions in the second through sixth columns. If a kernel thread satisfies those conditions, its process may be a candidate for shrinking and possibly for swapping.

In the case of a multithreaded process, each kernel thread must meet the scheduling state constraints in the table for the process to be suitable for being shrunk or outswapped. As SCH$OSWPSCHED scans the table, it keeps track of how many kernel threads in each multithreaded process it has encountered in scheduling queues on this pass of the table so that it can determine if all threads meet the constraints.

The conditions in the table entries discriminate among kernel threads, based on their likelihood of becoming computable in a short while and the effects of shrinking or swapping their processes. When the system needs to reclaim physical memory, process working sets extended in times of plentiful memory are shrunk first.

In general, the intent is to prevent the outswap of a process with a kernel thread that is about to become computable when the only reason for the swap is to bring a process with a kernel thread of equal priority into memory. Overall system performance may be improved by shrinking processes rather than swapping them. However, a process with kernel threads in some states may be affected less by being swapped than by having its working set limit reduced.

Descriptions of the various conditions follow:

- I/O. A table entry in this column can specify No direct, Direct, No buffered, Buffered, and n/a.

  When a kernel thread is in a local event flag (LEF) or common event flag (CEF) scheduling state, and its process has an outstanding direct I/O request, there is a high probability that the kernel thread is waiting for the direct I/O to complete. If so, the kernel thread will soon become computable and thus be a less desirable shrink or outswap candidate. SCH$OSWPSCHED therefore distinguishes between kernel threads with and without outstanding I/O requests.

Suspension affects all kernel threads in a process. A suspended kernel thread, by default, can receive kernel and executive ASTs. To prevent a suspended process from being outswapped and one of its kernel threads then becoming computable again as the result of buffered I/O completion, the table distinguishes between suspended kernel threads with and without outstanding buffered I/O requests.

In this column, n/a means that the existence of either type of outstanding I/O request is irrelevant. No test is made for either.

- Priority. A table entry in this column can specify Yes or n/a.

  Yes in this column means that SCH$OSWPSCHED compares the priority of the highest priority computable kernel thread in a process to be inswapped with that of any kernel thread whose process may be shrunk or outswapped. A process with a kernel thread that is computable or likely to be computable soon is not considered a candidate, unless the kernel thread's priority is less than or equal to that of the potential inswap process, stored in global location SWP$GB_ISWPRI. (The swapper zeros SWP$GB_ISWPRI before calling SCH$OSWPSCHED to make up a free page list deficit.)

  In this column, n/a means no test is made.

- Initial Quantum. A table entry in this column can specify Yes or n/a.

  Yes in this column means that SCH$OSWPSCHED rejects a process that is in its initial memory residency quantum. A process with a kernel thread likely to become computable soon is not considered a candidate for second-level trimming or outswapping if it is within its initial memory residency quantum. If SWP$GB_ISWPRI is less than or equal to 47, indicating the inswap candidate is real-time, the constraint is ignored. The intent is to leave the process in memory long enough to do useful work, after the system has expended the overhead of inswapping it. This reduces the possibility of swap thrashing, a condition in which the system spends more time swapping in and out than in process execution.

  In this column, n/a means that SCH$OSWPSCHED does not test whether the process is in its initial quantum.

- Long Wait. A table entry in this column can specify Yes, No, or n/a.

  Either Yes or No in this column means that SCH$OSWPSCHED determines whether a kernel thread has been waiting in an LEF or hibernate (HIB) state longer than the SYSGEN parameter LONGWAIT. Yes means that for a kernel thread to be a candidate, it must be in a long wait. A kernel thread that has been waiting a long time is likely to wait longer still; one that has been waiting a short time is more likely to become computable soon. For example, a kernel thread waiting for terminal input longer than a LONGWAIT interval is likely to remain in LEF longer still.

  No in this column means that the kernel thread must not have been waiting a long time; n/a means that SCH$OSWPSCHED does not test for this condition.

- Dormant. A table entry in this column can specify Yes, No, or n/a.

Either Yes or No in this column means that SCH$OSWPSCHED determines whether a computable kernel thread is dormant, that is, one whose priority is less than or equal to the SYSGEN parameter DEFPRI and that has been on a COM or COMO queue for longer than the SYSGEN parameter DORMANTWAIT. Yes in this column means that the kernel thread must be dormant to be a candidate. A process with dormant kernel threads is considered a very good candidate to be shrunk. An example of such a process is one with a compute-bound kernel thread with a priority too low to get CPU time.

This condition expedites the shrinking and outswap of a process such as a low-priority batch job. While the process's kernel thread or threads run at night on a lightly loaded system, its working set is expanded and it can acquire extensive physical memory, but once interactive users log in, the process's kernel threads cannot get CPU time.

No in this column means the kernel thread must not be dormant to be a candidate; n/a means that SCH$OSWPSCHED does not test for this condition.

This older mechanism for dealing with dormant kernel threads persists in case the system manager has disabled the newer, preferred mechanism: the combination of PIXSCAN priority boost and quantum-end working set trimming. Chapter 5 contains information on quantum-end trimming, and Chapter *Scheduling* describes the PIXSCAN mechanism.

When SCH$OSWPSCHED finds a candidate process, its subsequent action depends on the flags shown in the last column and described in Section 6.4.3.

In addition to conditions imposed by the table entries, there are several implicit constraints on the suitability of a particular process to be shrunk or outswapped:

- A process cannot be outswapped if it has locked itself into the balance set.
- The working set limit of a process that has disabled automatic working set adjustment cannot be reduced, although its working set may be reduced.
- The working set limit of a real-time process cannot be shrunk below WSQUOTA, although its working set may be reduced.
- If the executive is deleting the process and its address space (bit PHD$V_NO_WS_CHNG in PHD$W_FLAGS), the working set cannot be shrunk or outswapped.
- If the executive has temporarily blocked movement of the PHD (by setting bit PHD$V_LOCK_HEADER in PHD$L_FLAGS), the process cannot be swapped.
- A process that is already outswapped cannot be shrunk or outswapped.

The TROLL table consists of three entries within one section. Its first entry specifies the SUSP scheduling state. Its other two entries are HIB and LEF, with the LONGWAIT flag set for each. The actual order of these two entries varies, depending on which queue had the longest waiting kernel thread the last time the trolling routine executed.

6.4.2 Trolling

Trolling is triggered by a combination of circumstances, including fewer pages on the free page list than FREEGOAL (see Section 6.3.1). The trolling routine, TROLLER, in module OSWPSCHED, looks for a process to outswap proactively, that is, before the swapper needs a balance set slot to accommodate a process to be inswapped.

In the case of a multithreaded process, each kernel thread must meet the scheduling state constraints in the table for the process to be suitable for being shrunk or outswapped. As TROLLER scans the table, it keeps track of how many kernel threads in each multithreaded process it has encountered in scheduling queues on this invocation.

TROLLER takes the following steps:

1. It first tests whether the free page list has fewer pages than FREELIM. If so, the test for whether a kernel thread has been waiting a long enough time will be based on half the value of the LONGWAIT parameter.
2. Scanning the SUSP wait queue, it tests each kernel thread's process to check that the process has not locked itself into the balance set and that the executive has set neither PHD$V_LOCK_HEADER nor PHD$V_NO_WS_CHNG in PHD$L_FLAGS. If all these constraints are met, TROLLER has found a candidate to outswap and continues with step 5.
3. If TROLLER failed to find a process with suspended kernel threads to outswap, it scans the HIB and LEF wait queues. Which one it scans first depends on which had the longest waiting kernel thread the last time TROLLER executed. A suitable kernel thread in either scheduling state must have been waiting long enough and must meet the constraints just listed.
4. If it processes the entire TROLL table and finds no candidate, it returns a failure status to SCH$OSWPSCHED.
5. When it finds a candidate to outswap, TROLLER reduces the process's working set, but not its limit, to the number of pages represented by SWPOUTPGCNT.
6. It allocates swap space for the outswap candidate.
7. TROLLER scans both the HIB and LEF wait queues to determine the longest waiting swappable kernel thread in either state and calculates how soon that kernel thread could meet the LONGWAIT constraint TROLLER established at its entry.
8. It recalculates the next time at which the trolling routine should be entered as the later of 5 seconds from the current time and the time at which the oldest swappable HIB or LEF kernel thread will have waited long enough to meet the LONGWAIT constraint. Because the trolling routine is automatically entered every time SCH$OSWPSCHED is, the result of this calculation represents a maximum interval between trolls.

9. If necessary, it switches the second and third entries in the TROLL table so that the queue that currently has the oldest kernel thread will be scanned first on TROLLER's next execution.
10. It returns to SCH$OSWPSCHED with the address of the PCB of the outswap candidate.

SCH$OSWPSCHED clears PCB$V_RES in PCB$L_STS for that process, reports a SWPOUT event for the process to change its scheduling state, resets the swap failure count, and returns the process's PCB address to OUTSWAP.

6.4.3 Passes Through the OSWPSCHED Table

SCH$OSWPSCHED scans the scheduling database looking for processes to be shrunk or outswapped. The search for a candidate process is table-driven.

SCH$OSWPSCHED makes two passes through the table. On its first pass, it potentially traverses all sections of the table, performing first-level trimming (to WSQUOTA) of any candidate processes.

If, however, it has been entered with a request to outswap a process to free a balance set slot, it shrinks the working set of the first candidate process that has not locked itself into the balance set to the number of pages represented by SWPOUTPGCNT, selects that process as an outswap candidate, and returns its PCB address to the swapper. If the process is a real-time process or one that has disabled automatic working set limit adjustment, its working set size is reduced but its limit is not changed.

If SCH$OSWPSCHED has been entered to satisfy a free page deficit, it continues reclaiming memory from working sets that had been extended until it reaches the end of the table, reclaims enough free pages to satisfy the deficit, or finds a process to be outswapped. A suitable outswap candidate is one whose kernel threads meet the scheduling state and conditions of a table entry that includes the SWAPASAP flag and that has not locked itself into the balance set.

If SCH$OSWPSCHED reaches the end of the table without satisfying the deficit or locating an outswap candidate, it makes a second pass through the table, starting its scan at the beginning of the table. If it has been entered to satisfy a free page deficit, it performs second-level trimming.

In second-level swapper trimming, SCH$OSWPSCHED can scan each section of the table twice. In scanning the table, it ignores any entry that has the LVL1_TRIM flag set. First, if the entry contains the SWPOGOAL flag, SCH$OSWPSCHED shrinks the working set of a process selected by this entry (unless the process has disabled automatic working set adjustment). The working set is reduced, if possible, to the number of pages represented by SWPOUTPGCNT. If the process is a real-time process or one that has disabled automatic working set limit adjustment, its working set size is reduced but its limit is not changed.

If the deficit is not satisfied, SCH$OSWPSCHED continues scanning through processes selected by the table section. When it gets to the end of the section, it restarts at the beginning of the section, looking for a process to outswap. When SCH$OSWPSCHED gets to the end of the section for the second time, it goes to the next section. The pass ends when the deficit is satisfied or a process is found to outswap. If outswapping a process does not satisfy the deficit, eventually the swapper will reexecute the OUTSWAP and SCH$OSWPSCHED routines.

The swapper maintains a swap failure counter that records the number of times it has failed to locate a candidate to shrink or swap. This count is maintained across calls of SCH$OSWPSCHED. It is intended to loosen the constraints in situations where the normal conditions have failed to produce candidates. When this count reaches a value equal to SWPFAIL, the swapper ignores certain constraints when selecting a process to shrink or outswap: it ignores the initial quantum condition for all processes and the priority constraint for all processes except COM ones. The counter is reset each time an outswap candidate is successfully located.

When the swapper scans a series of processes in a particular scheduling queue, the scan begins with the least recently queued entry (at the tail of the queue). This starting point ensures that the longer a process has been in a wait queue, the more chance it has of being shrunk or swapped. (A process is inserted into a wait queue at the front of the list, unlike most queues.)

To determine whether all kernel threads of a multithreaded process meet the table entry contstraints, SCH$OSWPSCHED keeps track of how many kernel threads in each multithreaded process it has encountered in scheduling queues on this pass of the table:

1. It increments SWP$L_SEQ_NUMBER at the beginning of each pass through the OSWPSCHED table.
2. The first time in a pass through a table that it encounters any of a process's kernel threads, it records SWP$L_SEQ_NUMBER in PCB$L_SWP_SEQ and copies the number of kernel threads in the process to PCB$L_SWP_KT.
3. For each kernel thread it encounters, it records SWP$L_SEQ_NUMBER in KTB$L_SWP_SEQ to indicate that it has encountered this kernel thread on this table pass. If the kernel thread is a suitable candidate, it decrements PCB$L_SWP_KT.
4. When PCB$L_SWP_KT is decremented to zero, all the kernel threads of the process meet the constraints and the process is a suitable candidate to be shrunk or outswapped.

6.5 Outswap Operation

Outswap is described before inswap because it is easier to explain inswap in terms of what the swapper puts into the swap file. The swapper does not remove processes from the balance set indiscriminately. In general, unless trolling is enabled, the swapper tries to satisfy the free page deficit first by shrinking working sets, deleting or outswapping PHDs and page tables, and writing modified pages. The swapper outswaps a process if one of the following conditions is true:

- Trolling is enabled, and an existing process's kernel threads meet the trolling constraints.
- The steps just described fail to free enough pages.
- SCH$OSWPSCHED encounters a process whose kernel threads meet the constraints of a table entry with the SWAPASAP flag.
- The system needs a balance set slot (PHD slot).

6.5.1 Selection of an Outswap Candidate

As described in Section 6.4, the outswap selection is driven by an ordered table of scheduling states and associated conditions. The swapper selects a process less likely to benefit from remaining in memory. Once a candidate is selected, the swapper allocates swap space and prepares the working set of that process for outswap.

6.5.2 Allocation of Swap Space

Section 6.2.3 describes how the swapper calculates the amount of swap space needed. To allocate the space, the swapper calls MMG_STD$ALLOC_SWAP_SPACE, in module PAGE_FILE. MMG_STD$ALLOC_SWAP_SPACE first allocates a PFLMAP from nonpaged pool.

Looking for possibly large quantities of available swap space can be time-consuming. MMG_STD$ALLOC_SWAP_SPACE uses the PFL$L_DIR_CLUSTER array in each PFL (see Figure 2.31) to minimize the effort. As described in Chapter 2, each array element represents a number of adjacent set bits in the directory bitmap. The elements represent increasing powers of 2: the first element counts the number of adjacent set bits; the second element, the number of pairs of adjacent set bits; the third element, the number of groups of four adjacent set bits; and so on.

The maximum number of adjacent bits represented is 2^6, or 64. Since each bit in the directory bitmap represents 16 bits in the storage bitmap, a count in this cluster entry represents 1,024 adjacent free pages. Descending array elements represent occurrences of 512, 256, and so on, pages.

To avoid costly but possibly futile scans of the bitmaps, the swapper calculates the minimum acceptable swap space extent as the total space needed, split into the maximum possible number of extents. It rounds the result up to the next power of 2.

It then begins to scan the page-and-swap-file vector (see Figure 2.31). If SYSGEN parameter NOPGFLSWP is zero, it can examine both page and swap file PFLs; otherwise, it can examine only swap file PFLs.

In each PFL it examines, it looks at the PFL$L_DIR_CLUSTER array entry that represents the minimum acceptable extent. (As described in Chapter 2, each array entry represents a number of set bits in the directory bitmap for that file. Each set bit represents 16 adjacent set bits in the storage bitmap.) If that array entry is nonzero, it records in the PFLMAP the index of the PFL in the page-and-swap-file vector and the size of that minimum extent.

It continues to record minimum extents from that PFL in the PFLMAP until it has accounted for all the set bits or until it has accumulated enough extents for the needed swap space. If it has accounted for all the set bits but not accumulated enough extents, it goes on to the next PFL.

Once MMG_STD$ALLOC_SWAP_SPACE has accumulated enough extents, it then calls MMG_STD$ALLOC_PAGSWP_PAGES, in module PAGE_FILE, potentially once for each extent, passing it the PFL index, the total size of the swap space needed, and the size of that extent.

Each time MMG_STD$ALLOC_PAGSWP_PAGES is called, it locates the available cluster that would satisfy the minimum extent size in the specified PFL. Starting with that cluster, it allocates as much contiguous space as it can, up to the total size of the swap space.

MMG_STD$ALLOC_SWAP_SPACE initializes a mapping pointer in the PFLMAP to correspond to the space allocated from that PFL, condensing the equivalent extents into the one mapping pointer. If the total space has not yet been allocated, it calls MMG_STD$ALLOC_PAGSWP_PAGES to allocate from the next file.

6.5.3 Outswap of the Process Body

The swapper outswaps the process body (P0, P1, and P2 pages) separately from the PHD and page tables for the following reasons:

- Fields in the PHD and page tables (most notably WSLEs and PTEs) are modified as the working set list is processed.
- The PHD and page tables may not be swappable at the same time as the body because of outstanding I/O, pages on the modified page list, or some other reason.

Even though the PHD and page tables are outswapped separately, space in the swap file is reserved for them at the beginning of the swap slot.

6.5.3.1 Scanning the Working Set List

To prepare the process body for outswap, the swapper scans the process's working set list. It must examine each page in the working set list to determine if any special action is required. The swapper looks at a combination of the page type (found in the WSLE as well as the PFN$L_PAGE_STATE field in the PFN database record) and the valid bit.

A page in the working set can be in one of the following three states:

- The page is valid.
- The page is currently being read into memory. The swapper treats page reads like any other I/O in progress when swapping a process.
- The process PTE contains a global page table index (GPTX), and the indexed global page table entry (GPTE) indicates a transition state. The swapper handles global pages in a special manner when outswapping a process.

Table 6.3 lists all combinations of page type, state, and valid bit setting that the swapper encounters and the action it takes for each. Several combinations are discussed further in the following sections. One type of page not discussed further is a page locked into memory, one whose WSLE PFNLOCK bit is set. Apart from setting PCB$V_PHDLOCK in the process's PCB$L_STS2 as an indication that its PHD and page tables cannot be outswapped, the swapper ignores such pages; they remain in memory, and no other action is required.

Table 6.3 Scan of Working Set List of Outswap Process

Page Type WSLE<3:1>	Page Validity L3PTE	Action of Swapper for This Page
Process page	Valid	Outswap page. If there is outstanding I/O and the page is being modified, store in its PFN record the number of the swap file page where the updated page contents should be written when the I/O completes. If the page is part of a buffer object, decrement its page table's share count.
Process page	Transition	(Page state = Read in Progress) Treat as page with I/O in progress. Special action may be taken at inswap or by the modified page writer. (Page state = Read Error) Drop from working set. No other transition states are possible for a page in the working set.
System page	n/a	It is impossible for a system page to be in a process working set. The swapper generates the fatal IVWSETLIST bugcheck.

Table 6.3 *(continued)* **Scan of Working Set List of Outswap Process**

Page Type WSLE<3:1>	Page Validity L3PTE	Action of Swapper for This Page
Global read-only	Transition	If the process L3PTE contains a GPTX, then the global page table must contain a transition L3PTE. The page is dropped from the process working set.
Global read-only	Valid	If share count = 1, then outswap. If share count > 1, drop from working set unless the page is locked in the working set. It is highly likely that a process can fault such a page later without I/O. This check avoids multiple copies of the same page in the swap file.
Global writable	n/a	Drop from working set. At inswap, it would be difficult to determine whether the page in memory is more up-to-date than the swap file copy.
Page table page	n/a	Not part of the process body. However, in the single scan of the working set list the swapper builds chains of L2 and L3 page tables for later processing.

The basic step the swapper takes as it scans the working set list is to add a description of each swappable page to its swapper map. As a result of this pseudo page table, the virtually noncontiguous pages in the process's working set appear virtually contiguous to the I/O system (see Figures 6.5 and 6.8).

To access the process-private page tables of the process being outswapped, the swapper switches hardware context, temporarily adopting that process's address space. This change in behavior is required because, as of OpenVMS Alpha Version 7.0, process-private page tables are mapped only in process-private address space.

For each page in the working set, the swapper performs the following steps:

1. It locates the PTE that maps the page from the virtual page number in the WSLE.
2. It determines any special action, based on page validity and page type.
3. For a process body page to be outswapped, it copies the PFN from the L3PTE to the swapper map.
4. It records the modify bit (logical OR of the L3PTE modify bit and PFN$L_PAGE_STATE field saved modify bit) in the WSLE.
5. For a valid process page, it sets the delete contents bit in the PFN$L_PAGE_STATE field. This bit causes the page to be placed at the head of the free page list when its reference count goes to zero (normally, when the swap write completes).
6. It updates page table reference counts (see Chapter 4).

Note that the swapper does not explicitly restore each process body L3PTE to the contents of its physical page's PFN$Q_BAK field. The contents will be replaced when the page is released (after the swap write completes and all other references to the page are eliminated).

6.5.3.2 Pages Within Buffer Objects

If the swapper encounters a page whose reference count is greater than 1, it checks whether bit PFN$V_BUFOBJ is set in PFN$L_PAGE_STATE. If so, the page is part of a buffer object. The swapper changes the page's state to release pending and decrements the share count and PFN$W_PT_VAL_CNT of the page table page that maps it so that the PHD and page tables can be released. It increments PCB$L_BUFOBJ_CNT, the number of buffer object and PFN-locked pages.

6.5.3.3 Pages with Direct I/O in Progress

If, in the swapper's scan of the working set list, it encounters a modified page with outstanding I/O, it stores in the page's PFN$W_SWPPAG field the location in the swap file where that page belongs and sets PFN$V_SWPPAG_VALID in PFN$L_PAGE_STATE. The page will be swapped along with the rest of the process body to reserve a place for it in the swap file.

If the I/O operation is a write (from memory to mass storage) and the page was not otherwise modified, the contents currently being written to the swap file are good. The page will be inserted into the free page list when the I/O operation completes.

If the I/O operation is a read (or if it is a write and some other action has caused the page to be modified), the physical page will be placed into the modified page list when the I/O completes. The modified page writer takes special action for a modified page whose PFN$V_SWPPAG_VALID bit is set. That is, it writes the page to the swap file page whose number is in PFN$W_SWPPAG rather than to its normal backing store address.

6.5.3.4 Global Pages

Global pages are also given special treatment at outswap. A writable global page is dropped from the working set before the process is outswapped. The task of determining whether the contents that are swapped are up-to-date when the process is brought back into memory is more complicated than simply refaulting the page (often without I/O) when the process is swapped back into memory.

A global read-only page is swapped only if its global share count is 1. In all other cases, the page is typically dropped from the working set and must be refaulted (most likely without I/O) after the process is inswapped. (Global read-only pages, however, that are locked into the working set are not dropped from it.) Global transition pages are also dropped from the working set.

6.5.3.5 PHD Pages and Page Table Pages

As the swapper scans the working set list, it records the locations of all the process-private page tables it encounters so that it can process them later without rescanning the working set list. It builds one list of all L3PTs other than those mapping buffer objects and another list of all L2PTs. These page table pages remain resident while the process body is outswapped so that they can be updated to reflect pages written to the swap file and any pages dropped from the working set list. The swapper also builds one list of all L3PTs that map buffer objects; these page table pages will remain resident.

At the beginning of the working set list scan, the swapper clears PHDL_L3PT_WSLX, PHDL_L2PT_WSLX, and PHD$L_BUFOBJ_WSLX, the listheads for the three lists. When the swapper encounters the first page table of each type, it stores the WSLX of that page table in the appropriate listhead.

When it encounters the second page table of a particular type, it stores the WSLX of that page table in the low longword of the first page table's PFN$Q_BAK. This scheme takes advantage of the fact that there is no useful information in the low longword of the BAK field for page table pages. The WSLX of any subsequent page table is stored in the low longword of the previous page table's PFN$Q_BAK.

When the swapper has scanned the entire working set list, it clears the low longword of the last page table's PFN$Q_BAK to terminate the list.

During the final preparation for header outswap, the PFN$Q_BAK field for a page table page is written to the PTE that maps that page table. This enables the page table linkage to survive outswap and inswap.

The swapper increments PCB$L_APTCNT for the L1PT, each L2PT, and each L3PT that does not map a buffer object. For each L3PT that maps a buffer object, it increments PCB$L_BUFOBJ_CNT. It updates page table reference counts (see Chapter 4) for the page table page that maps each page table page.

For each PHD page it finds in the working set list, it increments PCB$L_APTCNT and subtracts the PHD address from the virtual address stored in the WSLE, converting it to an offset from the beginning of the PHD. If the PHD is outswapped, it will most likely be inswapped to a different balance set slot, and any PHD virtual address stored in a WSLE would have to be recalculated. It is not necessary to record the locations of PHD pages: they are the first pages locked into the working set list.

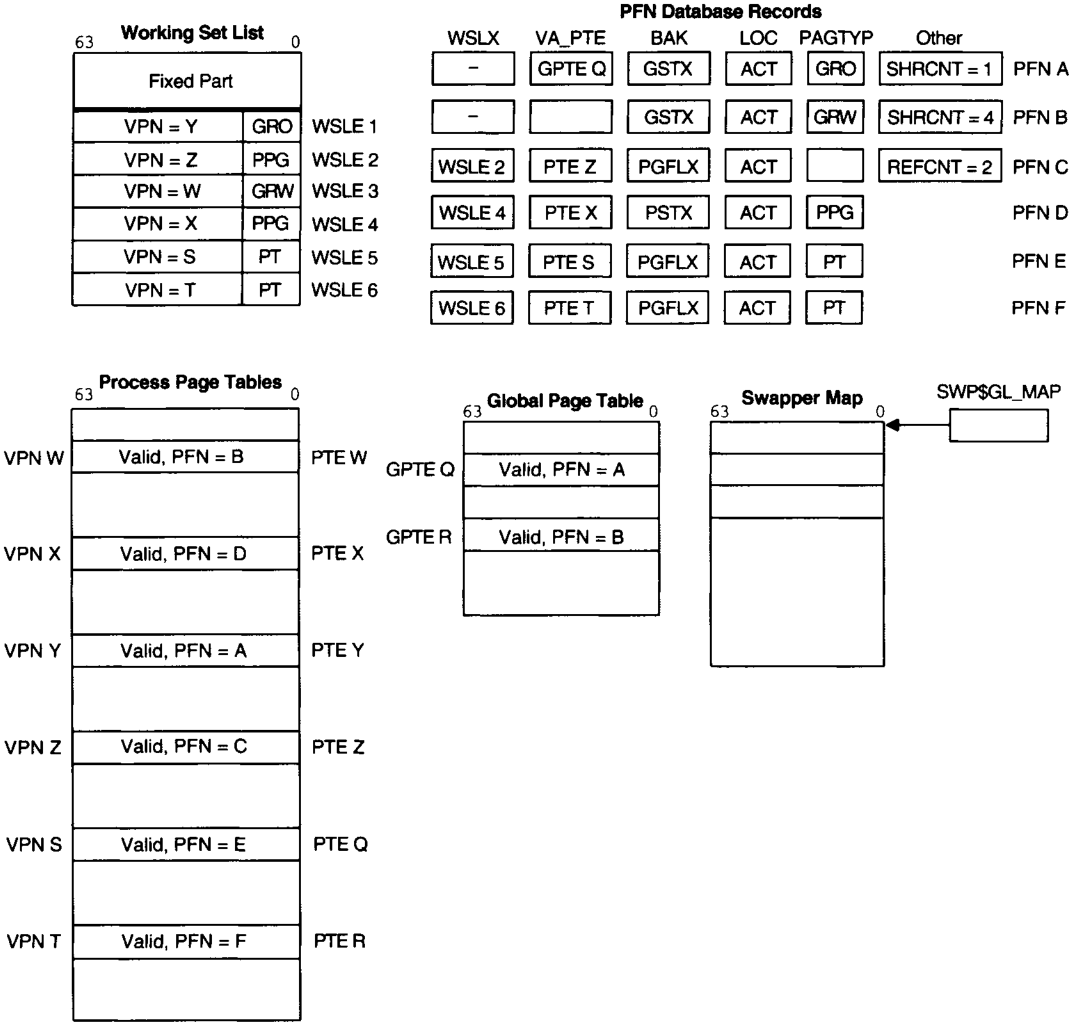
6.5.3.6 Example of a Process Outswap

Figures 6.4 to 6.6 show some of the cases the swapper encounters while it is scanning the process's working set list. The key information about each page is a combination of the L3PTE validity and the page type. The order of the scan is defined by the order of the working set list. Note that the example is simplified; in particular, it omits the L1PT and L2PTs.

Figure 6.4 shows excerpts from the working set list, the process page tables, and the associated PFN database records before the swapper begins its working set scan. Figure 6.5 shows the modified working set list and the swapper map after the working set list scan but before the I/O request is initiated. Figure 6.6 shows the state of the L3PTEs after the swap write has completed and the physical pages have been released.

In these figures, the term VA_PTE represents the combination of PFN database fields PFN$L_PT_PFN and PFN$Q_PTE_INDEX, and the term WSLX stands for the PFN database field PFN$L_WSLX_QW.

Figure 6.4 Example Working Set List before Outswap Scan



Working Set List
63
0
Fixed Part
VPN = Y | GRO | WSLE 1
VPN = Z | PPG | WSLE 2
VPN = W | GRW | WSLE 3
VPN = X | PPG | WSLE 4
VPN = S | PT | WSLE 5
VPN = T | PT | WSLE 6
PFN Database Records
WSLX VA_PTE BAK LOC PAGTYP Other
- | GPTE Q | GSTX | ACT | GRO | SHRCNT = 1 | PFN A
- | | GSTX | ACT | GRW | SHRCNT = 4 | PFN B
WSLE 2 | PTE Z | PGFLX | ACT | | REFCNT = 2 | PFN C
WSLE 4 | PTE X | PSTX | ACT | PPG | PFN D
WSLE 5 | PTE S | PGFLX | ACT | PT | PFN E
WSLE 6 | PTE T | PGFLX | ACT | PT | PFN F
Process Page Tables
63
0
VPN W | Valid, PFN = B | PTE W
VPN X | Valid, PFN = D | PTE X
VPN Y | Valid, PFN = A | PTE Y
VPN Z | Valid, PFN = C | PTE Z
VPN S | Valid, PFN = E | PTE Q
VPN T | Valid, PFN = F | PTE R
Global Page Table
63
0
GPTE Q | Valid, PFN = A
GPTE R | Valid, PFN = B
Swapper Map
63
0
SWP$GL_MAP

1. WSLE 1 is a global read-only page. The VPN field of the WSLE locates the L3PTE. The PFN field of the L3PTE locates the PFN database record associated with this

physical page. In particular, the share count for this page is 1. (This process is the only process that currently has this page in its working set.) The swapper writes this page out as part of the swap image for this process. Thus, PFN A is the first page in the swapper's map (see Figure 6.5). It marks the page for deletion.

When the outswap I/O completes, the swapper will clear PFN$L_PT_PFN and PFN$Q_PTE_INDEX and place the page at the head of the free page list (see Figure 6.6).

2. WSLE 2 is a process page that also has I/O in progress (a reference count of 2.) This page will be swapped; its PFN is shown in the swapper's map.

Figure 6.5 Example Working Set List after Outswap Scan

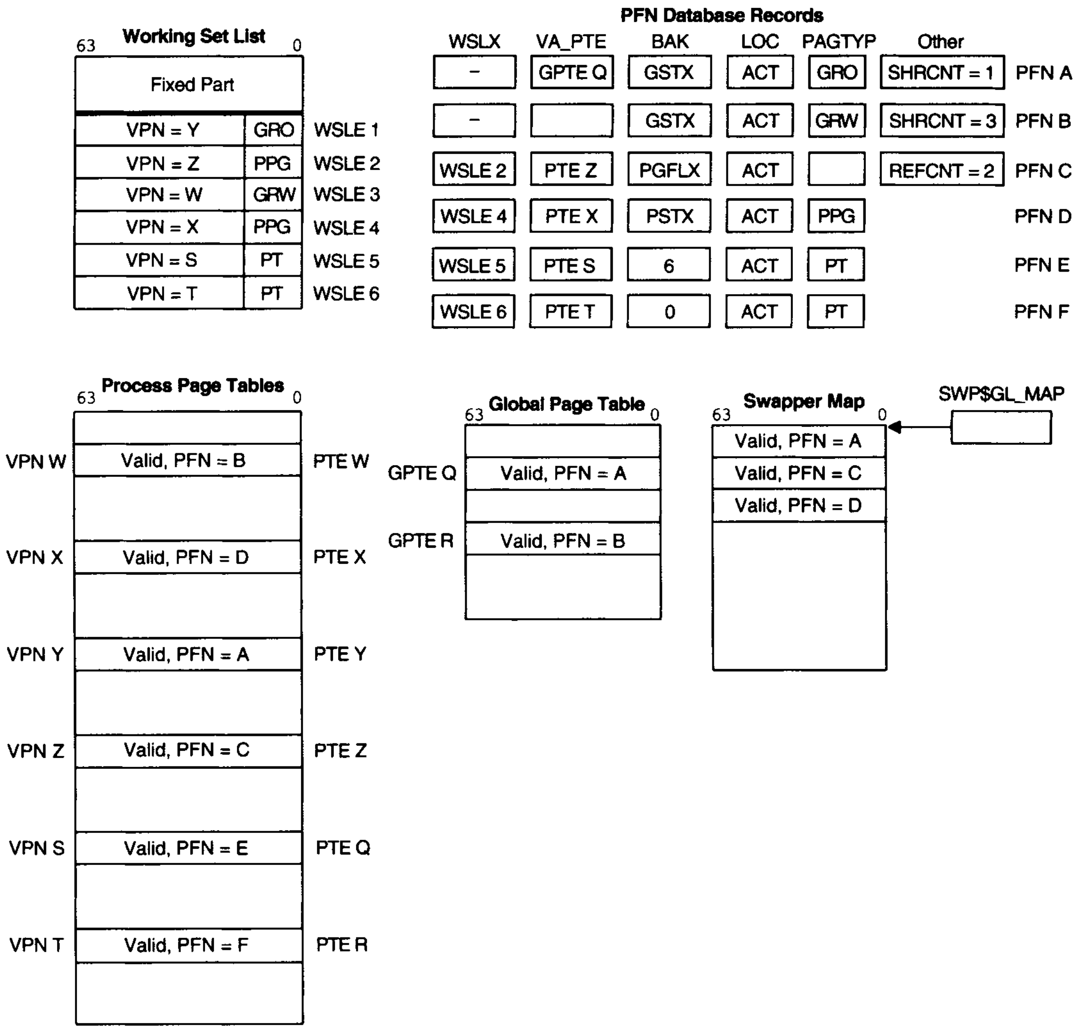
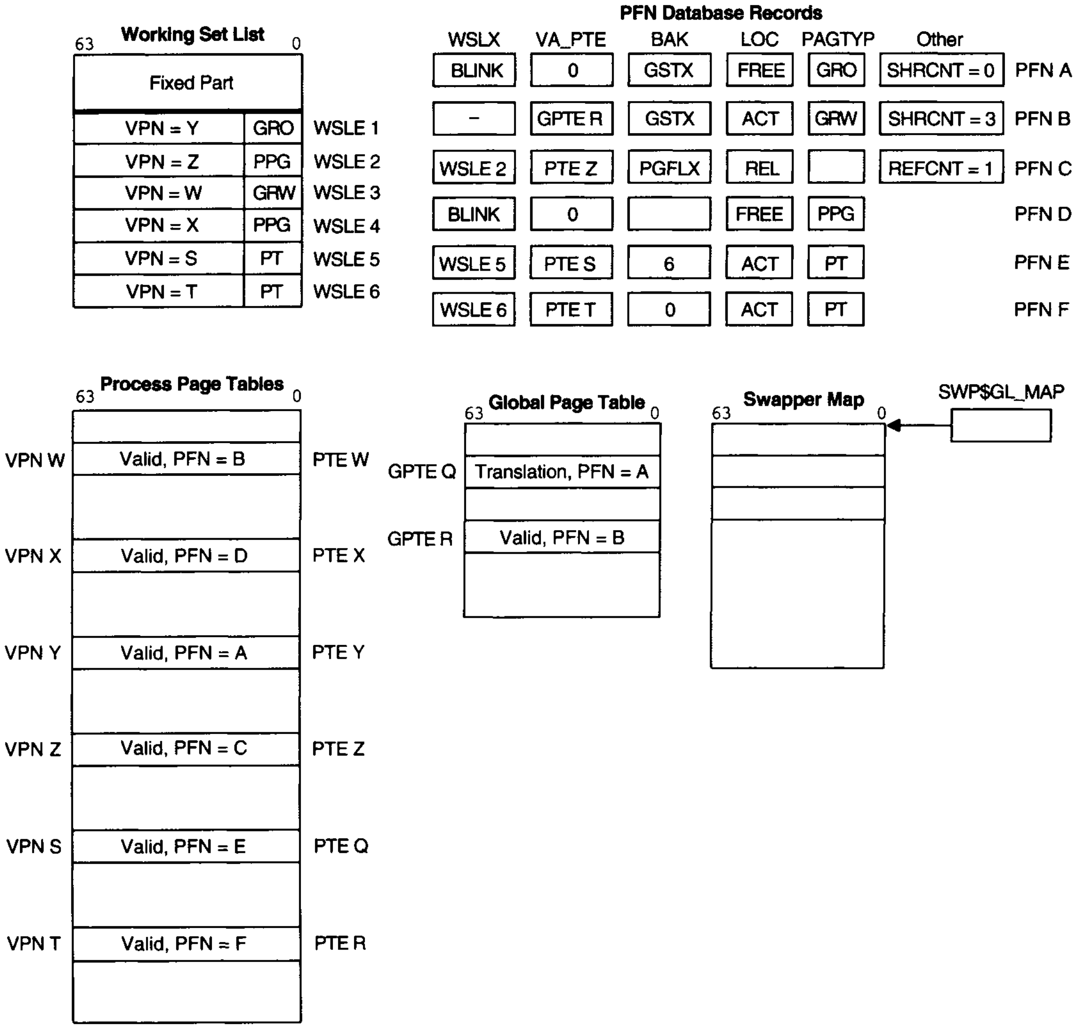

Working Set List
63
0
Fixed Part
VPN = Y | GRO | WSLE 1
VPN = Z | PPG | WSLE 2
VPN = W | GRW | WSLE 3
VPN = X | PPG | WSLE 4
VPN = S | PT | WSLE 5
VPN = T | PT | WSLE 6
PFN Database Records
WSLX | VA_PTE | BAK | LOC | PAGTYP | Other
- | GPTE Q | GSTX | ACT | GRO | SHRCNT = 1 | PFN A
- | | GSTX | ACT | GRW | SHRCNT = 3 | PFN B
WSLE 2 | PTE Z | PGFLX | ACT | | REFCNT = 2 | PFN C
WSLE 4 | PTE X | PSTX | ACT | PPG | PFN D
WSLE 5 | PTE S | 6 | ACT | PT | PFN E
WSLE 6 | PTE T | 0 | ACT | PT | PFN F
Process Page Tables
63
0
VPN W | Valid, PFN = B | PTE W
VPN X | Valid, PFN = D | PTE X
VPN Y | Valid, PFN = A | PTE Y
VPN Z | Valid, PFN = C | PTE Z
VPN S | Valid, PFN = E | PTE Q
VPN T | Valid, PFN = F | PTE R
Global Page Table
63
0
GPTE Q | Valid, PFN = A
GPTE R | Valid, PFN = B
Swapper Map
63
0
Valid, PFN = A
Valid, PFN = C
Valid, PFN = D
SWP$GL_MAP

Figure 6.6 Changes after Swapper's Write Completes



PFN Database Records

WSLX	VA_PTE	BAK	LOC	PAGTYP	Other	
BLINK	0	GSTX	FREE	GRO	SHRCNT = 0	PFN A
–	GPTE R	GSTX	ACT	GRW	SHRCNT = 3	PFN B
WSLE 2	PTE Z	PGFLX	REL		REFCNT = 1	PFN C
BLINK	0		FREE	PPG		PFN D
WSLE 5	PTE S	6	ACT	PT		PFN E
WSLE 6	PTE T	0	ACT	PT		PFN F

Working Set List (63 … 0)

Fixed Part		
VPN = Y	GRO	WSLE 1
VPN = Z	PPG	WSLE 2
VPN = W	GRW	WSLE 3
VPN = X	PPG	WSLE 4
VPN = S	PT	WSLE 5
VPN = T	PT	WSLE 6

If the page was previously modified (if either the L3PTE modify bit or saved modify bit in PFN$L_PAGE_STATE is set), the address in the swap file where the page belongs is stored in the PFN$W_SWPPAG field and bit PFN$V_SWPPAG_VALID is set in PFN$L_PAGE_STATE. A set PFN$V_SWPPAG_VALID bit causes the page to be placed into the modified page list when it is released. If the process is still outswapped when the modified page writer writes this page, the page will be written to the page reserved for it in the swap file.

The page is marked for deletion. If the I/O is still outstanding when the outswap completes, the page state is changed to release pending. Later, when the I/O completes and the PFN$L_REFCNT reaches zero, the page will be placed at the head of the free page list and its PFN$L_PT_PFN and PFN$Q_PTE_INDEX fields cleared.

3. WSLE 3 is a global writable page. The page is dropped from the process working set (see Figure 6.5); the process L3PTE contents are replaced with the GPTX of GPTE R, and the share count for PFN B is decremented. Notice that PFN B is not included in the swapper map.
4. WSLE 4 is an ordinary process page. The page is added to the swapper map (PFN D) and it is marked for deletion. The deletion will actually occur after the swapper's write operation completes.
5. WSLE 5 is a process L3PT that does not map a buffer object. The swapper links it into the list at PHD$L_L3PT_WSLX. Its PFN is not included in the swapper map because it is not part of the process body.
6. WSLE 6 is also a process L3PT that does not map a buffer object. It is also linked into the L3PT list, but the low longword of its PFN$Q_BAK field contains 0 because it is the last L3PT in the list.

6.5.4 Outswap of the Process Header and Page Tables

The PHD and page tables are not outswapped until after the process body has been successfully written to the swap file. Before they can be outswapped, ties between physical pages and the process page tables must be severed, including pages that were in the working set and written to the swap file and also pages that are in some transition state, notably pages on the free and modified page lists.

6.5.4.1 Partial Outswap

After the process body has been outswapped, the PHD and page tables can be outswapped if the PHD reference count is zero. The PHD reference count (see Figure 2.30) represents the number of reasons (transition pages, active page table pages, and so on) the page tables and thus the PHD cannot be outswapped.

The swapper checks the PHD reference count. If the reference count is zero, the swapper outswaps the PHD and tables immediately, taking the steps in Section 6.5.4.2. If the reference count is nonzero, the header and page tables cannot be outswapped. This circumstance of a body outswap not being followed immediately by the outswap of the header and page tables is referred to as a partial outswap.

Later in a subsequent invocation of OUTSWAP (see Section 6.3.3), when the swapper locates a PHD from a partially outswapped process, it takes whatever actions are required to remove the ties that bind the PHD and page tables to physical memory. First, it eliminates any transition PTEs whose physical pages are on the free page list.

It locates a transition PTE by scanning the free page list for a process or process page table page mapped by the page table hierarchy defined by the L1PT associated with the PHD being examined. It starts its scan at the back of the list with the most recently queued entries, on the assumption that the transition pages are more frequently in the back half of the list. (Pages associated with deleted virtual address space are placed at the front of the list.)

Whenever it finds such a page, it calls MMG$DEL_CONTENTS_PFN, in module ALLOCPFN, which restores the backing store information in the physical page's PFN$Q_BAK field to the process PTE, reinitializes the PFN database record to indicate the page is not attached to any virtual page, moves the page from its current location to the head of the free page list, and decrements the corresponding page table page share count.

Because the free page list is only one of several transition states, the scan of the free page list may not free the PHD for removal. Pages may be in some other transition state. A page in a transition state that represents some form of I/O in progress (release pending, read in progress, write in progress) is left alone because there is nothing that the swapper can do until the I/O completes. After the free page list is scanned, if the process still has transition pages, the swapper calls MMG$PURGE_MPL to request that all modified pages be written that are in the PHD or that are mapped by this process's page tables. A modified page written to its backing store is released to the free page list. Later, after the pages have been selectively purged from modified page list, the swapper will scan the free page list again.

If the swapper succeeds in releasing a PHD with the previously described free page list scan, it can take the steps described in the next section to outswap the PHD.

6.5.4.2 Preparing the Process Header and Page Tables for Outswap

Once the reference count for the PHD reaches zero, it can be outswapped and the balance set slot freed. The outswap of the PHD and page tables is similar to the outswap of a process body, in that the PFNs corresponding to the PHD and page table pages are inserted into the swapper map to form a virtually contiguous transfer for the I/O subsystem. The PHD pages are first, followed by the L1PT, the L2PTs, and the L3PTs. Any L3PT pages that map buffer objects are omitted.

There are several differences, however, between the outswap of a PHD and page tables and the outswap of a process body. When a process body is outswapped, the header that maps that body is still resident. When the swapper's write completes and each physical page is being deleted, the contents of the PFN$Q_BAK field in the PFN database record for each page are restored to the process L3PTE.

PHD pages are mapped by system space PTEs for that balance set slot. The system space PTEs are not available to hold the PFN$Q_BAK field contents because they will be used by the next occupant of this balance set slot. Instead, the PHD page BAK array (see Section 6.2.1) serves this purpose. As the PHD is processed for outswap, the contents of the PFN$Q_BAK field for each active header page are stored in the corresponding PHD page BAK array element.

Lower level page table pages are mapped by upper level page table pages. The swapper stores the PFN$Q_BAK field contents in the PTE that maps a page table page. This records the forward links in the L3PT and L2PT chains for use during inswap processing.

Routine RELPHD, in the SWAPPER module, prepares the PHD to be outswapped. Before calling it, the swapper switches to the mapping context of the outswapped process so that its page tables can be accessed. It does this by storing the contents of the outswapped process's page table base register (PTBR), the current kernel stack pointer, and the current address space number (ASN) in an alternative hardware privileged context block (HWPCB) and switching context to that HWPCB.

RELPHD takes the following steps:

1. It scans the list of L3PTs that map buffer objects. For each one it finds, it updates the reference counts of the page table page that maps it (see Chapter 4).
2. It scans the list of L3PTs that do not map buffer objects. For each one it finds, it updates the reference counts of the page table page that maps it. It stores the contents of PFN$Q_BAK into the PTE that maps the page table page. It stores the PFN occupied by the L3PT in the swapper map. It clears PFN$L_PT_PFN and PFN$Q_PTE_INDEX of the associated PFN to sever the connection between that PFN and the PTE that mapped it.

   It scans the list of L2PTs, taking the same actions for each L2PT. The L3PTs must be processed first, while the L2PTs that map them are still valid.
3. It scans the system space L3PTEs that map the balance set slot. For each valid one, it stores the PFN$Q_BAK contents of the associated PFN into the corresponding PHD BAK array element and clears the PFN$L_PT_PFN and PFN$Q_PTE_INDEX fields to sever the connection between that PFN and the system space L3PTE.
4. It copies the local event flags from the PHD to the PCB (see Chapter *Event Flags*).
5. It stores the index of the PHD slot in PCB$L_PHD and clears PCB$V_PHDRES in PCB$L_STS as indications that the process no longer has a resident PHD. It also clears KTB$Q_PHYPCB and KTB$L_VIRPCB in the initial kernel thread's KTB to indicate that the hardware PCB address must be recalculated after the process is inswapped.
6. It stores the PFN of the L1PT in the swapper map and clears the PFN's PFN$L_PT_PFN and PFN$Q_PTE_INDEX fields.
7. It switches back to the address space of the swapper process and queues a write request to outswap the PHD and page tables.

Once the header and page tables are successfully outswapped, routine RELEASE_PROCESS_HEADER, in module SWAPPER, runs. It reinitializes the system space L3PTEs that mapped the PHD and flushes stale translations from the translation buffer. It releases each outswapped header and page table page to the front of the free page list, reinitializing its PFN database record. It initializes the PHD reference count to −1 and clears the PHV$GL_PIXBAS element corresponding to the slot. If the process has a single kernel thread, it clears PCB$L_PHD. Otherwise, it clears KTBL_PHD, KTBL_VIRPCB, and KTB$Q_PHYPCB for each kernel thread. The balance set slot is now available for further use.

6.6 Inswap Operation

The inswap is exactly the opposite of the outswap operation. The swapper brings the PHD, active page tables, and process body back into physical memory. It then uses the contents of the working set list to rebuild the process page tables, an operation that primarily involves updating each valid PTE to reflect the new PFN used by that PTE. As each page is processed, the swapper can resolve any special case that existed when the process was outswapped.

6.6.1 Selection of an Inswap Candidate

As described in Section 6.3.2, the swapper selects a process for inswap, much as the scheduling subsystem selects a candidate for execution. The following processes are candidates for inswap:

- Newly created processes
- Processes with a kernel thread in some outswapped wait state that was just made computable
- Processes that were outswapped with a kernel thread in the computable state

The process with the highest priority COMO kernel thread is the one selected for inswap.

6.6.2 Preparation for Inswap

Before inswapping a process, the swapper must locate a free balance set slot for the process's PHD and allocate pages of physical memory for its working set. In the case of a partial outswap, it is possible that the PHD and page tables will not have been swapped in the time between the outswap and subsequent inswap of the process body. In the corresponding partial inswap, the swapper need not allocate a balance set slot and bring the PHD and page tables into memory because they are already resident.

In routine SWAPSCHED, the swapper calculates the number of pages required as the sum of PCB$L_PPGCNT and PCB$L_GPGCNT. If the PHD and page table pages are still resident, SWAPSCHED subtracts the number of header and page table pages (PCB$L_APTCNT) from the number of pages to be allocated. SWAPSCHED also subtracts PCB$L_BUFOBJ_CNT, the number of buffer object pages, and pages locked in memory. It tests whether the number of free pages is large enough for the required number of pages to be allocated. If not, it calls OUTSWAP, specifying the number of free pages to be reclaimed. Sometime later, after the outswap is completed, the swapper will try to inswap again, selecting a candidate from the highest priority COMO queue.

If the number of free pages is large enough, SWAPSCHED calls INSWAP to inswap the process body and, if necessary, the PHD and page tables. If the PHD has been outswapped, INSWAP scans the PHD reference count array for a balance set slot with a negative reference count. If it fails to find one, it calls OUTSWAP (see Section 6.3.3), specifying that a process should be outswapped to free a balance set slot. Sometime later, after the outswap is completed, the swapper will try to inswap again, selecting a candidate from the highest priority COMO queue.

If INSWAP finds a free balance set slot, it zeros the PHD reference count for that slot, stores the low word of the process's ID in the corresponding PHV$GL_PIXBAS array element, and stores in PCB$L_PHD the byte offset of the slot from the beginning of the balance set slot area.

It then allocates as many free physical pages as required to accommodate the process's working set. If the process has a home resource affinity domain (RAD), it allocates pages from that RAD's memory. If the process does not have a home RAD, but this is a nonuniform memory access (NUMA) platform and RAD support enabled, INSWAP tests bit RIH$V_SPECIAL in parameter RAD_SUPPORT.

- If the bit is clear, INSWAP uses the default allocation method, selecting the next RAD in the round-robin with both memory and CPU.
- If the bit is set, INSWAP uses the method specified for swapper allocation, allocating from the current RAD, the base RAD, or the next RAD in the round-robin. If home RAD allocation was specified, because the process has no home RAD, it uses the next RAD in the round-robin.

  The RAD from which the pages are allocated becomes the process's home RAD.

INSWAP updates the PFN database record for each page by incrementing the page's reference count and setting its state to active. It initializes a swapper map entry with the PFN of each allocated page.

INSWAP records the PCB of the inswap process in SWP$GL_INPCB and resets the swap failure count. It initiates the inswap I/O.

6.6.3 Inswap of the Process Header and Page Tables

After the inswap I/O completes, routine SETUP, in module SWAPPER, executes.

If the PHD and page tables were outswapped, SETUP must reestablish them in memory before the process body can be reconstructed. SETUP must adjust any process data tied to a specific balance set slot (that is, specific system virtual or physical addresses) to reflect the PHD's new location.

SETUP takes the following steps:

1. It adds the address of the beginning of the balance set slots to the contents of PCB$L_PHD, which were the byte offset of the slot.

2. It tests PCB$V_PHDRES in PCB$L_STS to see whether the PHD and page tables remained resident. If so, it switches to the mapping context of the inswapped process so that it can access the process's page tables (see Section 6.5.4.2) and continues with step 8. It continues to run in the mapping context of the inswapped process until the final processing of the inswap (see Section 6.6.4.5).
3. Otherwise, it must reestablish the PHD in the balance set slot and reinitialize the page tables. The swapper does this work in local routine FILLPHD, called from SETUP. FILLPHD takes the following steps:
    a. It initializes each system space L3PTE that maps a PHD page in the balance set slot with the PFN from the swapper map, a protection of ERKW, and set valid, fault-on-execute, no-execute, and address space match bits. If MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, it also sets the no-TB-miss-memory-barrier-required bit (see Chapter 1) in the L3PTE.
    b. It updates the PFN database record for that page of memory with backing store information from the PHD BAK array.
    c. The PHD pages are at the beginning of the process's swap image. FILLPHD can identify any empty pages from data in the BAK array. Empty pages can result from a gap between the process section table and working set list and from empty entries at the end of the working set list.
    d. FILLPHD locates the L1PT as the first page in the swap image following the PHD. The L2PTs and L3PTs immediately follow.

       To reinitialize the page tables, FILLPHD first stores the PFN of the L1PT in PHD$Q_PTBR, recording the base of the process's page table hierarchy. It stores the PFN in the L1PTE that self-maps the L1PT (see Figure 1.9).
    e. It switches to the mapping context of the inswapped process so that it can access the process's page tables as it initializes them (see Section 6.5.4.2). It continues to run in the mapping context of the inswapped process until the final processing of the inswap (see Section 6.6.4.5). It initializes the PFN database for the L1PT.
    f. It processes the L2PTs, following the chain from PHD$L_L2PT_WSLX. It determines the virtual address of each one from the WSLE and then the virtual address of the L1PTE that maps it. It initializes the L1PTE with the next PFN from the swapper map, a protection of ERKW, and set valid, fault-on-execute, and no-execute bits. If MMG$V_NO_MB is set in the MMG_CTLFLAGS SYSGEN parameter, it also sets the no-TB-miss-memory-barrier-required bit. It initializes the PFN database for the L2PT pages.
    g. It processes the L3PTs in the same manner.
    h. It processes the list of buffer object pages and pages locked in memory, modifying the PFN database field PFN$L_PT_PFN for each to reflect the actual location of the page table that maps the page and updating the page table's page table reference counts (see Chapter 4).

4. SETUP copies the local event flags from the PCB to the PHD (see Chapter *Event Flags*).

5. It stores the index of the slot in PHD$GL_PHVINDEX.

6. It sets PCB$V_PHDRES in PCB$L_STS as an indication that the PHD is resident.

7. SETUP sets PHD$V_NOACCVIO in PHD$L_FLAGS as an indication that the header has just been inswapped, possibly to a different balance set slot, and that the first reference the process makes to another balance set slot could be the result of a swap at an inopportune time. Chapter 4 describes how the page fault handler tests and clears this bit.

8. If soft RAD affinity support is enabled (bit RIH$V_AFFINITY set in SYSGEN parameter RAD_SUPPORT), it sets KTB$V_SOFT_RAD_AFFINITY in the primary kernel thread's KTB$L_FLAGS, and CPB$V_SOFT_RAD_AFFINITY in its KTB$L_CAPABILITIES, KTB$L_PERMANENT_CAPABILITIES, and KTB$L_CAPABILITY. It clears KTB$L_SRA_SKIP_COUNT.

   Soft RAD affinity is a mechanism that biases a process to run on CPUs in its home RAD. Running on a particular CPU and RAD to select the next process to run, the scheduling subsystem skips over processes from other RADs. There is, however, a maximum number of times a process can be skipped before being run. A CPU that would otherwise go idle runs an off-RAD process.

9. If necessary, SETUP recalculates the physical address of the HWPCB and stores it in PCB$Q_PHYPCB, and stores the HWPCB's virtual address in the initial kernel thread's KTB$Q_VIRPCB.

10. If this is a process with multiple kernel threads, it stores the address of the PHD in each thread's KTB$L_PHD, stores the addresses of its HWPCB in KTB$Q_PHYPCB and KTB$L_VIRPCB, and stores the addresses of the process's L1PT in each thread's HWPCB.

    If soft RAD affinity support is enabled, it initializes the other threads' KTBs as in step 8. It also stores the home RAD in each thread's KTB$L_HOME_RAD.

11. It initializes PCB$L_PRVCPU to –1 to ensure that when any kernel thread in the process is next executed, it is assigned a new address space number. This step eliminates the need to flush stale process-private translation buffer entries.

12. SETUP initializes the P1 PTEs that double-map the PHD pages.

    This P1 mapping provides invariant addresses for the nonpageable part of the PHD. The system space mapping is subject to change with outswap and inswap: if the header is outswapped, it is likely to be inswapped into a different balance set slot. Chapter 2 describes the conventions for accessing the PHD.

    The P1 window to the PHD has the following implications:

    — The physical pages that are doubly mapped are not kept track of through reference counts. However, these header pages are a permanent part of the process working set.

— The P1 page table page that maps these pages must also be a permanent member of the process working set.

6.6.4 Rebuilding the Process Body

After the PHD and the page tables are in a known state, the process body can be restored to the state it was in before the process was outswapped.

6.6.4.1 Rebuilding the Working Set List and Process Page Tables

Rebuilding the process body involves scanning both the swapper map and the process working set list. Recall that at outswap the processing of each page was determined by a combination of page type and validity. On inswap, the key to the processing of each page is the contents of the PTE, located by the virtual address field in the WSLE. An approximation of swapper activity for each page is as follows:

1. The swapper locates the L3PTE from the virtual address in the WSLE.
2. In the usual case, the original contents of the L3PTE are stored in the PFN$Q_BAK field, and the PFN from the swapper map entry is inserted into the now valid L3PTE.
3. If, for some reason, a copy of the page already exists in memory (for example, if the page was locked into memory with the $LCKPAG[_64] system service), that copy is put into the process working set. The duplicate page from the swapper map is released to the front of the free page list.

At inswap, the swapper determines what action to take for each particular page in the working set list from the contents of the L3PTE. Table 6.4 details the different cases the swapper can encounter.

Table 6.4 Rebuilding the Working Set List and the Process Page Tables

Type of Page Table Entry	Action of Swapper for This Page
L3PTE is valid.	Page was not released at outswap. If the page was locked into memory, or is part of a buffer object, no action is required.
L3PTE indicates a transition page, probably because of outstanding I/O when process was outswapped.	Fault transition page into process working set. Release duplicate page that was just inswapped.

Table 6.4 *(continued)* **Rebuilding the Working Set List and the Process Page Tables**

Type of Page Table Entry	Action of Swapper for This Page
L3PTE contains a GPTX. Page must be global read-only because global read/write pages were dropped from the working set at outswap time.	Swapper action is based on the contents of the GPTE: · If the GPTE is valid, copy the PFN in the GPTE to the process L3PTE and release the duplicate page. · If the GPTE indicates a transition page, make the GPTE valid, add that physical page to the process working set, and release the duplicate page. · If the GPTE indicates a GSTX, keep the page just inswapped and make it the master page in the GPTE as well as the slave page in the process L3PTE.
L3PTE contains a page file index or a process section table index.	These are the usual contents for a page that did not have outstanding I/O or other page references when the process was outswapped. The PFN in the swapper map is inserted into the process page table. Its PFN database record is initialized.

If the virtual address field represents a system space address, the WSLE describes a page in the PHD. The swapper must calculate the new system virtual address corresponding to that page and modify the WSLE.

If the virtual address field represents a page table space address, the WSLE describes a process-private page table page. If its WSLE$V_PFNLOCK bit is set, the page table page maps window pages or page table pages that map window pages, and the swapper has to adjust the PFN's window count or share count:

- For an L3PT, the swapper scans the L3PTEs for those mapping a PFN-mapped page, a memory-resident page, or a Galaxywide section page. For each such page, the swapper increments the L3PT's PFN$W_PT_WIN_CNT. When the count transitions from –1 to 0, if PFN$W_PT_LCK_CNT is still –1, indicating the L3PT maps no pages locked in the working set or in memory, the swapper increments PHD$L_PTCNTLCK, the number of locked page table pages.
- For an L2PT, the swapper scans the L2PTEs for those mapping memory-resident shared L3PTs. For each such page, the swapper increments the L2PT's PFN$W_PT_WIN_CNT. When the count transitions from –1 to 0, if PFN$W_PT_LCK_CNT is still –1, the swapper increments PHD$L_PTCNTLCK, the number of locked page table pages.

Regardless of the state of the WSLE$V_PFNLOCK bit, the swapper must increment the PFN$L_SHRCNT in the page table page that maps each L2PT and L3PT in the working set list. When the share count transitions from 0 to 1, the swapper locks the mapping page table page into the working set list, increments PHD$L_PTCNTACT to indicate another active page table page, and increments the PHD's entry in the array at PHV$GL_REFCBAS_LW, the number of reasons the PHD should remain in memory.

6.6.4.2 Pages with I/O in Progress when Outswap Occurred

Pages that had I/O in progress when the process was outswapped were written to the swap file anyway to reserve space. If the page was previously unmodified, it would have been put into the free page list when both the swap write and the outstanding write operation completed. If the page was previously modified, it would have been put into the modified page list when both the swap write and the outstanding write operation completed (because bit PFN$V_SWPPAG_VALID was set).

In either case, it is possible for the process to be inswapped before one of these physical pages is reused. The swapper uses the physical page that is already contained in the process L3PTE (as a transition page) and releases the duplicate physical page from the swapper map to the front of the free page list.

In the case of a page on the free page list, this decision is simply one of convenience. For a page on the modified page list, the contents of the page in the swap image are out-of-date, and the swapper must use the physical page that is already in memory.

6.6.4.3 Resolution of Global Read-Only Pages

The only type of global page that can be in the swap file is a global read-only page that had a share count of 1 when the process was outswapped (or a page that was explicitly locked). All other global pages were dropped from the process working set before the process was outswapped.

There are two cases that the swapper can find when rebuilding the process page tables. At inswap, the process L3PTE for a global read-only page always contains a GPTX. The swapper's treatment of the page is determined by the contents of the GPTE indexed by the GPTX:

- If no other process has mapped the global page, the GPTE contains a GSTX. The swapper stores the PFN from the swapper map in both the process L3PTE and the GPTE.
- If some other process referenced the global page while this process was outswapped, the GPTE can indicate a valid or a transition page. In either case, the swapper releases the duplicate page to the free page list and stores the PFN from the GPTE in the process PTE. If the page is in transition, the swapper makes it valid.

6.6.4.4 Example of an Inswap Operation

Figures 6.7 to 6.9 show an inswap operation that illustrates some of the special cases the swapper encounters when inswapping a process body. Note that this example is not related to the outswap example shown in Figures 6.4 to 6.6. In this example the process body has been outswapped, but not the PHD and page tables. In these figures, the term VA_PTE represents the combination of PFN database fields PFN$L_PT_PFN and PFN$Q_PTE_INDEX, and the term WSLX stands for the PFN database field PFN$L_WSLX_QW.

Figure 6.7 Working Set List and Swapper Map before Physical Page Allocation

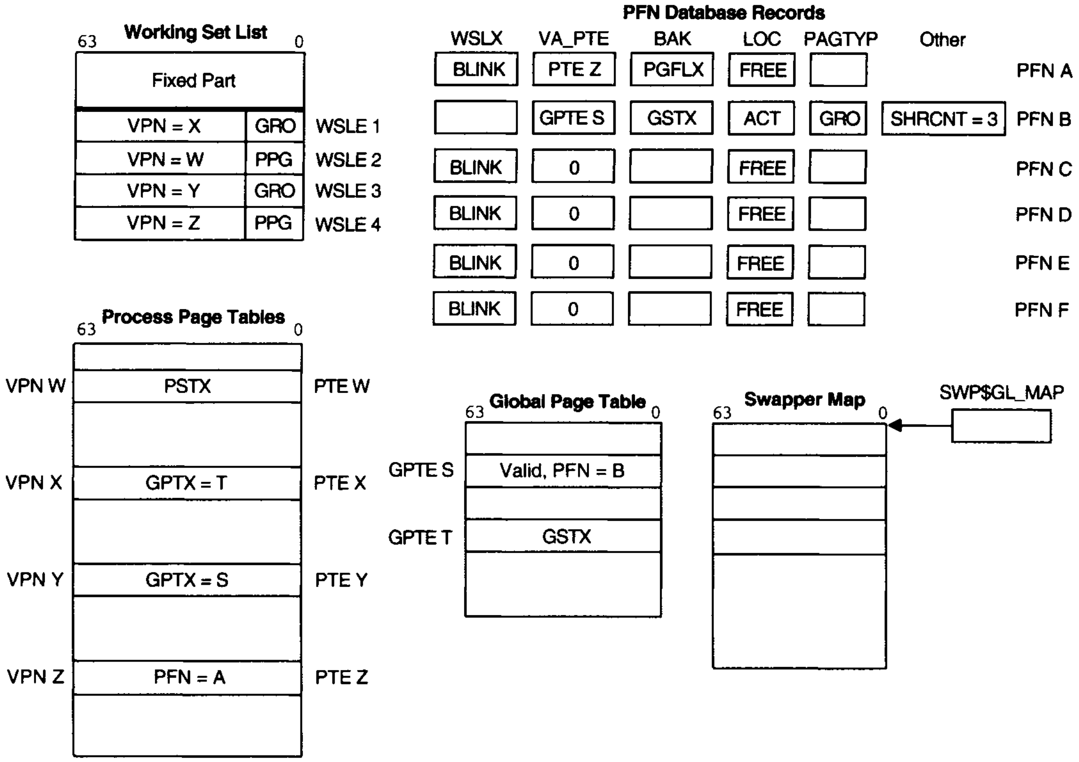


Figure 6.7 shows the state of the PHD and page tables after the process has been selected to be inswapped.

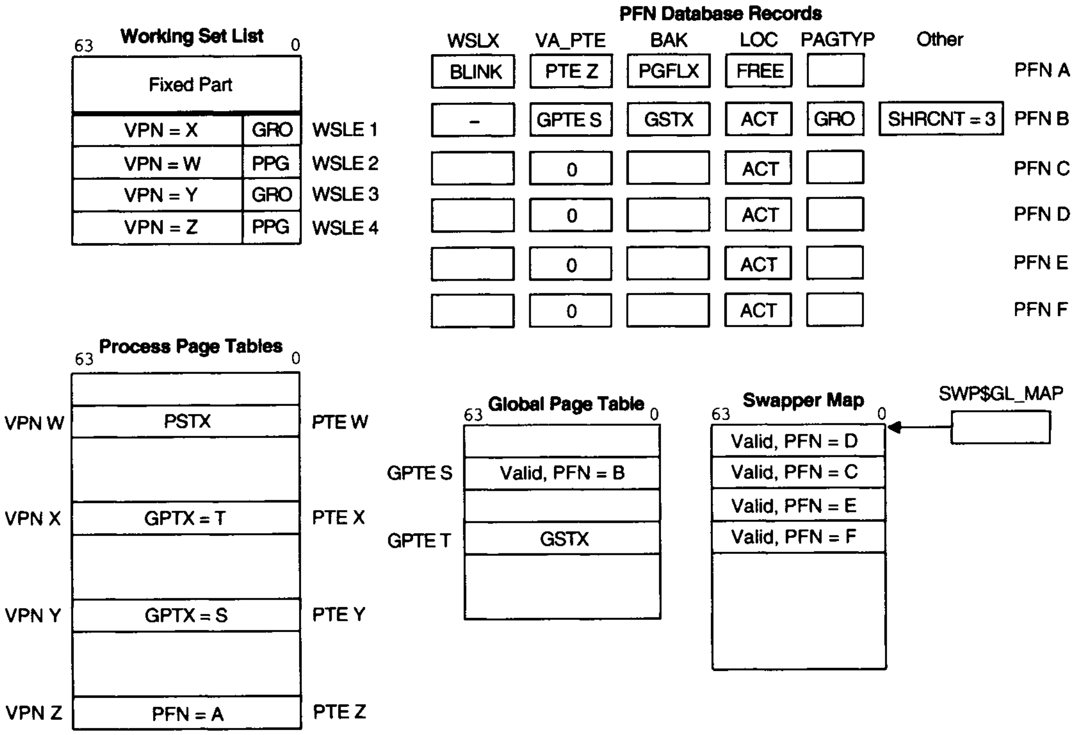
Figure 6.8 shows that four physical pages have been allocated to contain the four working set pages that the example describes. Figure 6.9 shows the rebuilt process page tables and the PFN database changes that result from rebuilding the working set and process page tables.

1. WSLE 1 locates virtual page number X. This L3PTE contains a GPTX. The referenced GPTE (GPTE T) contains a GSTX, indicating that the GPTE is not valid.

PFN D is inserted into the L3PTE. The swapper also inserts PFN D into the GPTE, sets the GPTE valid bit (see Figure 6.9), and updates the PFN database record for physical page D to reflect its new state.

2. WSLE 2 is a process page mapped by L3PTE W (see Figure 6.8). This L3PTE contains a process section table index. The L3PTE is updated to contain PFN C, and the PSTX is stored in the PFN$Q_BAK field for that page (see Figure 6.8). Other PFN record fields are updated accordingly.

Figure 6.8 Working Set List and Swapper Map after Physical Page Allocation



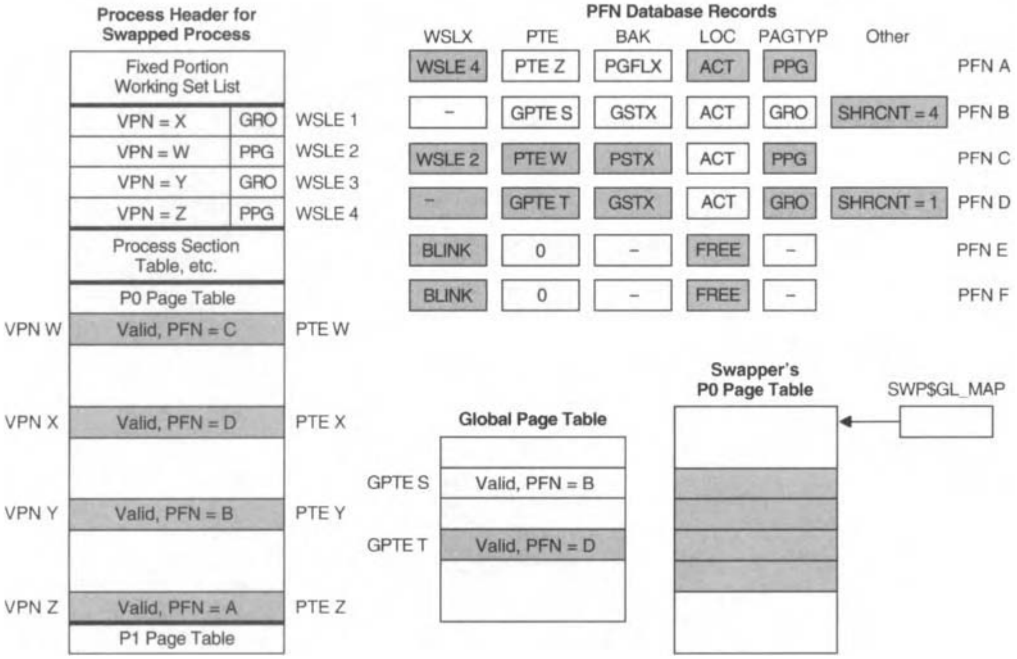
3. WSLE 3, which locates L3PTE Y, is exactly like the first, as far as the process data is concerned. However, the GPTE (GPTE S) is valid, indicating that another copy of this page already exists. This could occur only if another process had faulted the page while this process was outswapped.

The duplicate page (PFN E) is released to the front of the free page list. The process L3PTE is altered to contain the physical page that already exists (PFN B), and the share count for that page is incremented (from 3 to 4).

4. WSLE 4 resembles WSLE 2. However, the process L3PTE indicates a transition page. This implies that the header in this example was never outswapped.

The action taken here is similar to step 3, where a duplicate global page was discovered. The page just read (PFN F) is released to the head of the free page list. The transition page (PFN A) is faulted back into the process working set by removing the page from the free page list, changing its state to active, and setting the valid bit in the L3PTE.

Figure 6.9 Working Set List and Rebuilt Page Tables



6.6.4.5 Final Processing of the Inswap Operation

After the working set list has been scanned and the process page tables rebuilt, several other steps must be taken before the process is executable. After switching back to its own address space, the swapper calls local routine SETAST_CONTEXT. The swapper then invalidates the translation buffer to remove any stale translations of the balance set slot. As soon as the swapper releases the MMG and SCHED spinlocks, the computable kernel threads of the inswapped process are eligible to be scheduled.

SETAST_CONTEXT takes these steps:

1. It sets the resident bit, PCB$V_RES, and the initial quantum bit, PCB$V_INQUAN, in PCB$L_STS.
2. It calculates contents for the AST summary register (ASTSR) and stores them in the HWPCB. (ASTs may have been queued to the process while it was outswapped. The HWPCB, which contains a copy of the ASTSR, was not available while the header was not resident.)

If the process has multiple kernel threads, SETAST_CONTEXT must do this for each kernel thread. Each kernel thread has its own set of AST queues and its own HWPCB.

Additionally, SETAST_CONTEXT must take into account the queues in the PCB of ASTs that require the inner mode semaphore. It first checks whether the inner mode semaphore is currently owned by any kernel thread. If not, it determines the highest priority AST that requires the inner mode semaphore, identifies for which kernel thread it was intended, and modifies the inner mode semaphore to reflect that kernel thread as owner. It updates the ASTSR copy of the inner mode semaphore's owner to reflect the state of the PCB queues. Chapter *Kernel Threads* describes the inner mode semaphore, the different sets of AST queues, and AST delivery in a multithreaded process.

3. Each kernel thread gets a new quantum in KTB$L_QUANT and, optionally, a new thread quantum in KTB$L_TQUANT.
4. SETAST_CONTEXT calls SCH$CHSEP to change each kernel thread's scheduling state as appropriate, for example, to COM from COMO or to HIB from HIBO.
5. It clears bit PCB$V_PHDLOCK in PCB$L_STS2 (see Section 6.5.3.1).
6. It deallocates the process's swap space and clears PCB$L_WSSWP and PCB$L_SWAPSIZE to show that the process has no swap space allocated.
7. It clears SCH$V_SIP in SCH$GL_SIP.

6.7 Relevant Source Modules

Source modules described in this chapter include

[LIB]PFLMAPDEF.SDL
[SYS]OSWPSCHED.MAR
[SYS]SWAPPER.MAR
[SYS]SWAPPER_INIT.MAR

This Page Intentionally Left Blank

Chapter 7

Pool Management

In this bright little package, now isn't it odd?
You've a dime's worth of something known only to God!

Edgar Albert Guest, *The Package of Seeds*

The OpenVMS Alpha operating system creates and uses many data structures in the course of its work. Although it creates some of them at system initialization, it creates most when they are needed and destroys them when their useful life is finished. It maintains distinct areas of virtual address memory, called pools, in which it allocates and deallocates dynamic data structures. Each pool has different characteristics. This chapter describes these memory areas, their uses, and their allocation and deallocation algorithms.

Section 7.1 summarizes the various pools. Section 7.2 discusses dynamic data structures. Section 7.3 describes the structures and mechanisms of the variable-length pools, and Section 7.4, those of the fixed-length pools. Subsequent sections describe the various pools in detail.

7.1 Summary of Pool Areas

Almost all executive data structures created after system initialization are volatile; they are allocated on demand and deallocated when no longer needed. These data structures typically have a common header format (see Section 7.2). Their memory requirements vary in a number of ways:

- Pageability—Data structures accessed by code running at interrupt priority level (IPL) 2 or below can be pageable; data structures accessed at higher IPLs cannot.

- Virtual location—Some data structures are local to one process, mapped in process-private address space; others must be mapped in system space, accessible to multiple processes and to system context code.

- Physical location—Some data structures are accessed by I/O adapters and must be in addresses within I/O bus and adapter physical addressing limits.

On a nonuniform memory access (NUMA) platform, such as an AlphaServer GS160, some physical memory is local to the CPU. The CPU can access local memory in its own resource affinity domain (RAD) more quickly than nonlocal memory.

- Protection—Many dynamic data structures are created and modified only by kernel mode code, but some data structures are accessed by outer modes.

The executive provides different storage areas to meet the memory requirements of dynamic data structures, based on two different allocation schemes: variable-length allocation and fixed-length allocation.

There are several pools of storage for variable-length allocation:

- A nonpageable system space pool, known as nonpaged pool
- Under some circumstances, a nonpageable system space pool called bus-addressable pool (BAP)
- On a NUMA system, multiple sections of nonpaged pool in different sections of physical memory
- A pageable system space pool, known as paged pool
- A pageable process-private space pool, known as the process allocation region

The executive also provides lookaside lists of fixed-length packets. A lookaside list is a linked list of equal-size packets, each of which is ready for allocation through a quick unlinking operation. Lookaside lists enable faster allocation and deallocation of the most frequently used sizes and types of storage. Throughout this chapter, *packet* refers to a preformed, fixed-length allocation, and *block* refers to a variable-length allocation.

The executive provides the following lookaside lists:

- Nonpaged pool lookaside lists, with element sizes starting from 64 bytes and going up to 8,192 bytes in 64-byte increments
- A kernel process block (KPB) lookaside list out of nonpaged pool
- On a NUMA system, multiple sets of nonpaged pool lookaside lists in different sections of physical memory
- An S2 space list of nonpageable 256-byte packets for lock management resource blocks and lock blocks
- A process quota block (PQB) lookaside list out of paged pool
- A process-private kernel request packet (KRP) lookaside list out of P1 space for each process

The pool areas are summarized in Table 7.1.

Table 7.1 Comparison of Different Pool Areas

System Space	
Nonpaged Pool Variable-Length Region	
Protection	ERKW
Synchronization technique	POOL spinlock
Type of list	Variable-length blocks; singly linked absolute list
Allocation	Multiple of 64 bytes; mask is EXE$M_NPAGGRNMSK[1]
Minimum request size	1 byte
Characteristics	Nonpageable; expandable; RAD-specific
Bus-Addressable Pool Variable-Length Region	
Protection	ERKW
Synchronization technique	POOL spinlock
Type of list	Variable-length blocks; singly linked absolute list
Allocation	Multiple of 64 bytes; mask is EXE$M_NPAGGRNMSK[1]
Minimum request size	1 byte
Characteristics	Nonpageable; expandable
Nonpaged Pool Lookaside Lists	
Protection	ERKW
Synchronization technique	Load-locked/store-conditional mechanism
Type of list	Fixed-length packets; singly linked absolute list
Allocation	Multiple of 64 bytes; mask is EXE$M_NPAGGRNMSK[1]
Minimum request size	1 byte
Characteristics	Nonpageable; packets are initially allocated out of the nonpaged pool variable-length region and deallocated to these lists; RAD-specific
Bus-Addressable Pool Lookaside Lists	
Protection	ERKW
Synchronization technique	Load-locked/store-conditional mechanism
Type of list	Fixed-length packets; singly linked absolute list
Allocation	Multiple of 64 bytes; mask is EXE$M_NPAGGRNMSK[1]
Minimum request size	1 byte

[1] See Section 7.3 for a description of allocation masks.

Table 7.1 ***(continued)*** **Comparison of Different Pool Areas**

System Space	
Bus-Addressable Pool Lookaside Lists	
Characteristics	Nonpageable; packets are initially allocated out of the bus-addressable pool variable-length region and deallocated to these lists
Nonpaged Pool KPB Lookaside List	
Protection	ERKW
Synchronization technique	Load-locked/store-conditional mechanism
Type of list	Fixed-length packets; singly linked absolute list
Allocation	KPB$C_LENGTH
Minimum request size	KPB$C_LENGTH
Characteristics	Nonpageable; packets are initially allocated out of the non-paged pool variable-length region and deallocated to this list
S2 Space Lock Management Lookaside List	
Protection	ERKW
Synchronization technique	LCKMGR spinlock
Type of list	Fixed-length packets; doubly linked absolute list
Allocation	256 bytes
Standard request size	RSB$C_LENGTH and LKB$C_LENGTH
Characteristics	Nonpageable; expandable
Paged Pool	
Protection	ERKW
Synchronization technique	EXE$GL_PGDYNMTX mutex
Type of list	Variable-length blocks; singly linked absolute list
Allocation	Multiple of 16 bytes; mask is EXE$M_PAGGRNMSK[1]
Minimum request size	1 byte
Characteristics	Pageable
Paged Pool PQB Lookaside List	
Protection	ERKW
Synchronization technique	Self-relative queue operations
Type of list	Fixed-length packets; doubly linked self-relative queue

[1] See Section 7.3 for a description of allocation masks.

Table 7.1 ***(continued)*** **Comparison of Different Pool Areas**

	System Space
	Paged Pool PQB Lookaside List
Allocation	PQB$C_LENGTH
Minimum request size	PQB$C_LENGTH
Characteristics	Pageable; PQBs are initially allocated out of paged pool and deallocated to this list
	Process-Private Space
	Process Allocation Region
Protection	UREW
Synchronization technique	Access mode and IPL
Type of list	Variable-length blocks; singly linked absolute list
Allocation	Multiple of 16 bytes; mask is EXE$M_P1GRNMSK[1]
Minimum request size	1 byte
Characteristics	Pageable; expandable into P0 space
	P1 Space KRP Lookaside List
Protection	URKW
Synchronization technique	Access mode and absolute queue operations
Type of list	Fixed-length packets; doubly linked absolute queue
Allocation	CTL$C_KRP_SIZE
Minimum request size	CTL$C_KRP_SIZE
Characteristics	Pageable

[1] See Section 7.3 for a description of allocation masks.

7.2 Dynamic Data Structures

Traditionally, most dynamic data structures have the common header format shown in Figure 7.1:

- For a data structure allocated from 32-bit space, the first two longwords are available to link the data structure into a list or queue.
- The third longword contains the size, type, and (optional) subtype fields at byte offsets 8, 10, and 11.

Figure 7.1 Format of Dynamic Data Structures

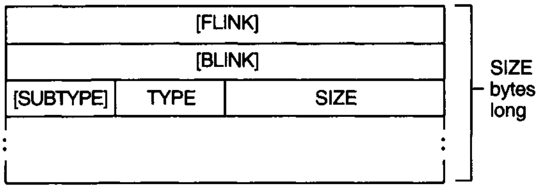


Figure 7.2 Format of Dynamic Data Structures

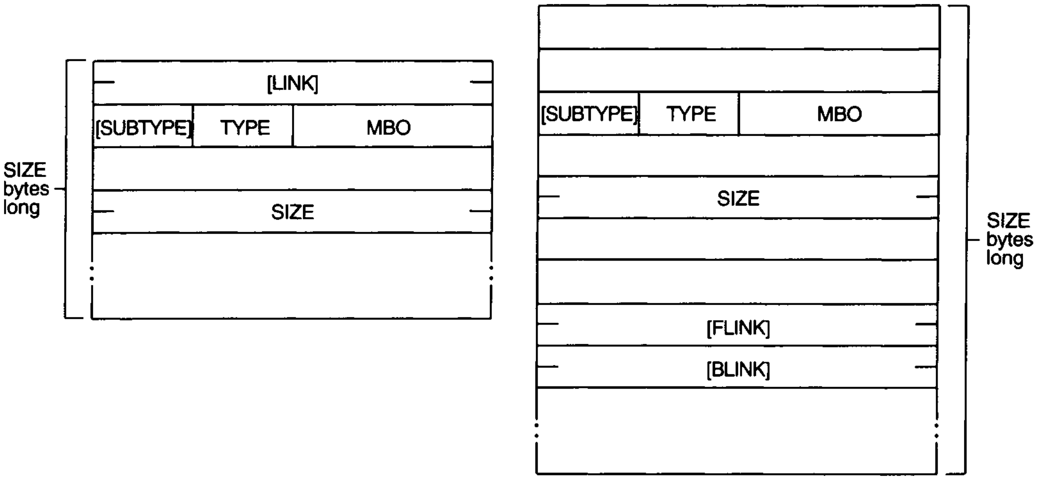


Figure 7.2 shows two alternative header formats. Like the traditional header format, these formats have size, type, and subtype fields in standard locations. The chief differences are the must-be-one (MBO) field at the same offset as the traditional size and a new 64-bit size field. Because data structures are always an even number of bytes, the MBO field enables the System Dump Analyzer (SDA) and other code to distinguish the traditional header format from the newer ones.

A 64-bit size can describe an arbitrarily large structure. Moreover, it facilitates address arithmetic with 64-bit structure addresses.

An additional difference is that 64-bit links are recommended. This minimizes recoding if a structure is moved to 64-bit space. The first format can be used for structures inserted into a singly linked list, and the second for doubly linked lists.

When a dynamic data structure is deallocated to the variable-length list, the size field specifies how much storage is being returned. For fixed-length packet deallocations, the size field selects the lookaside list into which the packet will be placed. Note, however, that the standard pool deallocation routines assume the traditional format and expect the size to be in the word at offset 8.

The type field enables system components to distinguish different data structures and to confirm that a piece of dynamic storage contains the expected data structure type. Codes that have numeric values greater than or equal to DYN$C_SUBTYPE are subtypable codes. Each subtypable code refers to a generic function. Different data structures related to the same generic function have the same value in the type field but different values in the subtype field. The subtype field is at offset *xxx*$B_SUBTYPE within a subtypable data structure.

For example, the system block (SB) and the path block (PB) are data structures used by system communication services (SCS). Both structures have the value DYN$C_SCS in their type field; the SB has the value DYN$C_SCS_SB in its subtype field, whereas the PB has the value DYN$C_SCS_PB in its subtype field.

The SDA utility uses the type, subtype, and size fields to produce a formatted display of a dynamic data structure and to determine the portions of variable-length pool that are in use.

The macro $DYNDEF defines the possible values for the type and subtype fields. Table 7.2 lists type values.

Table 7.2 Data Structure Type Definitions

Symbolic Name	Code	Structure Type
DYN$C_ADP	1	Adapter control block
DYN$C_ACB	2	AST control block
DYN$C_AQB	3	ACP queue block
DYN$C_CEB	4	Common event block
DYN$C_CRB	5	Controller request block
DYN$C_DDB	6	Device data block
DYN$C_FCB	7	File control block
DYN$C_FRK	8	Fork block
DYN$C_IDB	9	Interrupt dispatch block
DYN$C_IRP	10	I/O request packet
DYN$C_LOG	11	Reserved
DYN$C_PCB	12	Process control block
DYN$C_PQB	13	Process quota block
DYN$C_RVT	14	Relative volume table
DYN$C_TQE	15	Timer queue entry
DYN$C_UCB	16	Unit control block
DYN$C_VCB	17	Volume control block
DYN$C_WCB	18	Window control block
DYN$C_BUFIO	19	Buffered I/O buffer

Table 7.2 *(continued)*   Data Structure Type Definitions

Symbolic Name	Code	Structure Type
DYN$C_TYPAHD	20	Terminal type-ahead buffer
DYN$C_GSD	21	Global section descriptor
DYN$C_MVL	22	Magnetic tape volume list
DYN$C_NET	23	Network message block
DYN$C_KFE	24	Known file entry
DYN$C_MTL	25	Mounted volume list entry
DYN$C_BRDCST	26	Broadcast message block
DYN$C_CXB	27	Complex chained buffer
DYN$C_NDB	28	Network node descriptor block
DYN$C_SSB	29	Logical link subchannel status block
DYN$C_DPT	30	Driver prologue table
DYN$C_JPB	31	Job parameter block
DYN$C_PBH	32	Performance buffer header
DYN$C_PDB	33	Performance data block
DYN$C_PIB	34	Performance information block
DYN$C_PFL	35	Page file control block
DYN$C_PFLMAP	36	Page file mapping window
DYN$C_PTR	37	Pointer control block
DYN$C_KFRH	38	Known file resident image header
DYN$C_DCCB	39	Data cache control block
DYN$C_EXTGSD	40	Extended global section descriptor
DYN$C_SHMGSD	41	Reserved
DYN$C_SHB	42	Reserved
DYN$C_MBX	43	Mailbox control block
DYN$C_IRPE	44	Reserved
DYN$C_SLAVCEB	45	Reserved
DYN$C_SHMCEB	46	Reserved
DYN$C_JIB	47	Job information block
DYN$C_TWP	48	Terminal driver write packet ($TTYDEF)
DYN$C_RBM	49	Reserved
DYN$C_VCA	50	Disk volume cache block
DYN$C_CDB	51	X25 low-end system (LES) channel data block
DYN$C_LPD	52	X25 LES process descriptor
DYN$C_LKB	53	Lock block

Table 7.2 ***(continued)*** **Data Structure Type Definitions**

Symbolic Name	Code	Structure Type
DYN$C_RSB	54	Resource block
DYN$C_LCKRQ	55	Lock manager request packet
DYN$C_RSHT	56	Resource hash table
DYN$C_CDRP	57	Class driver request packet
DYN$C_ERP	58	Error log packet
DYN$C_CIDG	59	CI datagram buffer
DYN$C_CIMSG	60	CI message buffer
DYN$C_XWB	61	DECnet logical link context block
DYN$C_WQE	62	DECnet work queue block
DYN$C_ACL	63	Access control list queue entry
DYN$C_LNM	64	Logical name block
DYN$C_FLK	65	Fork lock request block
DYN$C_RIGHTSLIST	66	Rights list
DYN$C_KFD	67	Known file directory
DYN$C_KFPB	68	Known file pointer block
DYN$C_CIA	69	Compound intrusion analysis block
DYN$C_PMB	70	Page fault monitor control block
DYN$C_PFB	71	Page fault monitor buffer
DYN$C_CHIP	72	Internal check protection block
DYN$C_ORB	73	Object rights block
DYN$C_QVAST	74	Reserved
DYN$C_MVWB	75	Mount verification work buffer
DYN$C_UNC	76	Universal context block
DYN$C_DCB	77	DECnet control block for chained I/O
DYN$C_VCRP	78	VAX communication request packet
DYN$C_SPL	79	Spinlock control block
DYN$C_ARB	80	Access rights block
DYN$C_LCKCTX	81	Lock context block
DYN$C_BOD	82	Buffer object descriptor
DYN$C_FTRD	83	FTDRIVER read request packet
DYN$C_DDTM_EVENT	84	DDTM event notification block
DYN$C_DFLB	85	Dump file locator block
DYN$C_PTC	86	Portable Operating System Interface (POSIX) terminal control
DYN$C_OCB	86	Object class block (security)

Table 7.2 ***(continued)*** **Data Structure Type Definitions**

Symbolic Name	Code	Structure Type
DYN$C_CPCB	87	Common process control block
DYN$C_HWPCB	88	Hardware privileged context block
DYN$C_GCB	89	Glyph control block
DYN$C_RDPB	90	Resource domain pointer block
DYN$C_RDDB	91	Resource domain data block
DYN$C_SCDRP	92	SCSI class driver request packet
DYN$C_TQE_ACB	93	Timer queue entry/AST control block
DYN$C_NSAB	94	Security audit block
DYN$C_DEA	95	Deaccess audit pending block
DYN$C_SUBTYPE	96	Beginning of subtypable codes
DYN$C_SCS	96	SCS control block
DYN$C_CI	97	CI port structure
DYN$C_LOADCODE	98	Loadable code
DYN$C_INIT	99	Structure set up by INIT
DYN$C_CLASSDRV	100	Class driver structure
DYN$C_CLU	101	VMScluster structure
DYN$C_PGD	102	Paged pool structure
DYN$C_DECW	103	DECwindows structure
DYN$C_VWS	104	Reserved
DYN$C_DSRV	105	Disk server structure
DYN$C_MP	106	Multiprocessing-related structure
DYN$C_NSA	107	Nondiscretionary security audit structure
DYN$C_CWPS	108	Clusterwide process services
DYN$C_VP	109	Reserved
DYN$C_SHAD	110	Volume shadowing structures
DYN$C_VCC	111	Virtual I/O cache structure
DYN$C_OVRS	112	OpenVMS NT registry server
DYN$C_DDTM	113	Digital distributed transaction manager structures
DYN$C_SMI	114	System management integrator structure
DYN$C_TSRV	115	Tape server structure
DYN$C_LAVC	116	VMScluster structure
DYN$C_DECNET	117	DECnet structure
DYN$C_PSX	118	POSIX structure
DYN$C_QMAN	119	Queue manager structure

Table 7.2 *(continued)* Data Structure Type Definitions

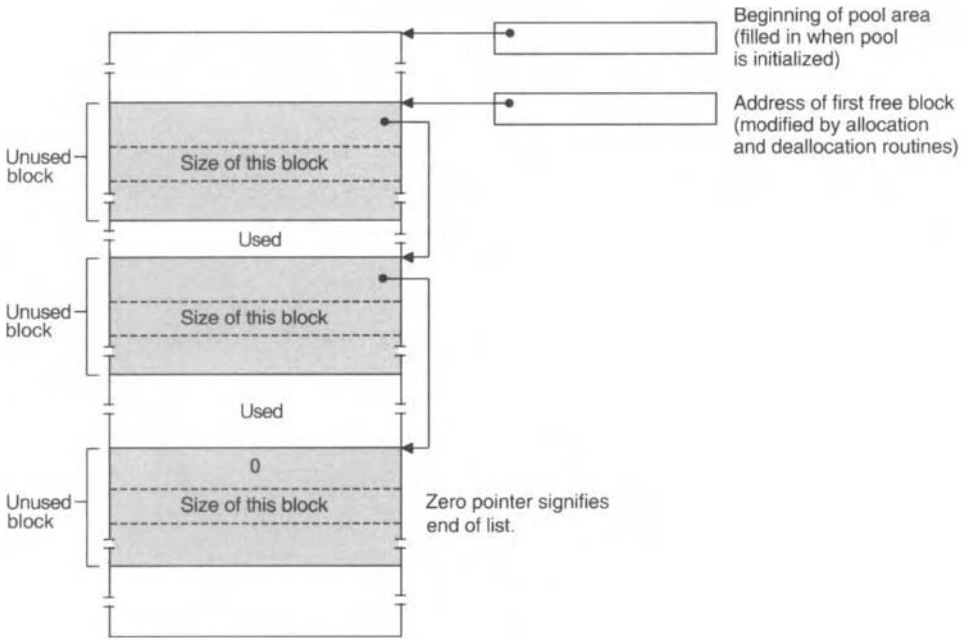
Symbolic Name	Code	Structure Type
DYN$C_SM	120	Storage manager structure
DYN$C_MISC	121	Miscellaneous type
DYN$C_RC	122	Redundant array of inexpensive disks (RAID) structure
DYN$C_IPC	123	Interprocess communication services structures
DYN$C_FILE_SYSTEM	124	File system structures
DYN$C_F64	125	Reserved
DYN$C_FILES_64	126	Reserved
DYN$C_SECURITY	127	Security structures
DYN$C_SHRBUFIO	128	Shared memory buffered I/O structures
DYN$C_LNMC	129	Logical name cache blocks
DYN$C_ICC	130	Intracluster communications structures
DYN$C_GLX	131	Galaxy structures
DYN$C_CTD	132	Galaxy AST control block
DYN$C_LCK	133	Lock management structures
DYN$C_QSRV	134	Reserved
DYN$C_SYS_EVENT	135	System event notification structures
DYN$C_SMCI	136	Shared memory cluster interconnect structures
DYN$C_SPLX	137	Spinlock-related extensions
DYN$C_WBM	138	Write bitmap structures

7.3 Variable-Length Pools

Pools that permit allocation of variable-length blocks have a common structure. Each pool has a global location containing the virtual address of the beginning of the pool and a listhead containing the virtual address of the first unused block in the pool. The first two longwords of each unused block describe the block. As illustrated in Figure 7.3, the first longword in a block contains the address of the next unused block in the list. The second longword contains the size in bytes of the unused block inclusive of the first two longwords. Each successive unused block is found at a higher address. Thus, the unused blocks in each pool area form a singly linked, memory-ordered list. The shaded areas in the figure represent unused blocks.

All pool areas are initially page-aligned. The allocation routines for the variable-length pools round the requested size up to the next multiple of 16 or 64 bytes to impose a granularity on both the allocated and unused areas. The granularity of nonpaged and bus-addressable pool allocation is 64 bytes; the granularity of the other pools is 16 bytes. The symbol EXE$M_*xxx*GRNMSK is a mask that indicates allocation

Figure 7.3 Layout of Unused Areas in Variable-Length Pools


Beginning of pool area (filled in when pool is initialized)
Address of first free block (modified by allocation and deallocation routines)
Unused block
Size of this block
Used
Unused block
Size of this block
Used
0
Unused block
Size of this block
Zero pointer signifies end of list.

granularity, where *xxx* is NPAG for nonpaged and bus-addressable pool, PAG for paged pool, and P1 for the process allocation region. For increased maintainability, any code that needs these values should use the symbol rather than a hard-coded value.

Table 7.3 summarizes variable-length allocation listheads and routines. In the table, all routines are in module MEMORYALC and contents of locations are dynamic unless marked otherwise.

Each variable-length pool has its own set of allocation and deallocation routines. The various routines call the lower level routines EXE[_STD]$ALLOCATE and EXE[_STD]$DEALLOCATE, in module MEMORYALC, which support the structure common to the variable-length lists. Each routine has two arguments: the address of the pool listhead and the size of the data structure to be allocated or deallocated. These general-purpose routines are also used for several other pools, including symbol table space of the Digital command language (DCL) interpreter.

7.3.1 Variable-Length Block Allocation

When the low-level allocation routine EXE[_STD]$ALLOCATE is called, it searches from the beginning of the list until it encounters an unused block large enough to satisfy the request. If the fit is exact, the allocation routine simply adjusts the previous pointer to point to the next free block. If the fit is not exact, it subtracts the allocated size from the original size of the block, puts the new size into the remainder of the block, and adjusts the previous pointer to point to the remainder of the block. That is, if the fit is not exact, the low-address end of the block is allocated, and the high-address end is placed back in the list. The two possible allocation situations (exact and inexact fit) are illustrated in Figure 7.4. The shaded areas in the figure represent unused blocks.

Table 7.3 Variable-Length Allocation Listheads and Routines

System Space	
Nonpaged Pool Variable-Length Regions	
Beginning address	@MMG$GL_NPAGEDYN[1]
First free block's address	@EXE$GL_NONPAGED[2]
Expansion area's address	@MMG$GL_NPAGNEXT
Allocation routines	EXE$ALONPAGVAR,[3] EXE$ALONONPAGED,[3] EXE$ALONONPAGED_ALN,[3] EXE$ALLOCATE_POOL[4,5]
Deallocation routines	EXE$DEANONPAGED,[3] EXE$DEANONPGDSIZ,[3] EXE$DEALLOCATE_POOL[4,5]
Bus-Addressable Pool Variable-Length Region	
Beginning address	@MMG$GQ_BAP[1]
First free block's address	@EXE$GQ_BAP_VARIABLE
Allocation routine	EXE$ALLOCATE_POOL[4,5]
Deallocation routine	EXE$DEALLOCATE_POOL[4,5]
Paged Pool	
Beginning address	@MMG$GL_PAGEDYN[1]
First free block's address	@EXE$GL_PAGED
Allocation routine	EXE$ALOPAGED
Deallocation routine	EXE$DEAPAGED
Process-Private Space	
Process Allocation Region	
First free block's address	@CTL$GQ_ALLOCREG, @CTL$GQ_P0ALLOC
Allocation routines	EXE$ALOP1IMAG, EXE$ALOP1PROC, EXE$ALOP0IMAG

[1] The static contents of this location are recorded during system initialization.

[2] The listhead for a per-RAD nonpaged pool is in its dynamically allocated LSTHDS structure (see Section 7.5.1).

[3] This routine is in module MEMORYALC_DYN.

[4] EXE$ALLOCATE_POOL and EXE$DEALLOCATE_POOL operate on nonpaged pool or bus-addressable pool, depending on input arguments.

[5] This routine is in module MEMORYALC_POOL.

Table 7.3 *(continued)* Variable-Length Allocation Listheads and Routines

	Process-Private Space
	Process Allocation Region
Deallocation routine	EXE$DEAP1

The first part of Figure 7.4 (Initial Condition) shows a section of paged pool; MMG$GL_PAGEDYN, which points to the beginning of paged pool; and EXE$GL_PAGED, which points to the first available block of paged pool. In this example, allocated blocks of memory are identified only by the total number of bytes in use, with no indication of the number and size of the individual data structures within each block.

The second part of Figure 7.4 (80 Bytes Allocated) shows the structure of paged pool after the allocation of an 80-byte block. Note that the discrete portions of 96 bytes and 48 bytes in use and the 80 bytes that were allocated are now combined to show a 224-byte block of paged pool in use.

The third part of Figure 7.4 (48 Bytes Allocated) shows an alternative scenario, the structure of paged pool after the allocation of a 48-byte block. The 48 bytes were taken from the first unused block large enough to contain it. Because this allocation was not an exact fit, an unused 32-byte block remains.

7.3.2 Variable-Length Block Deallocation

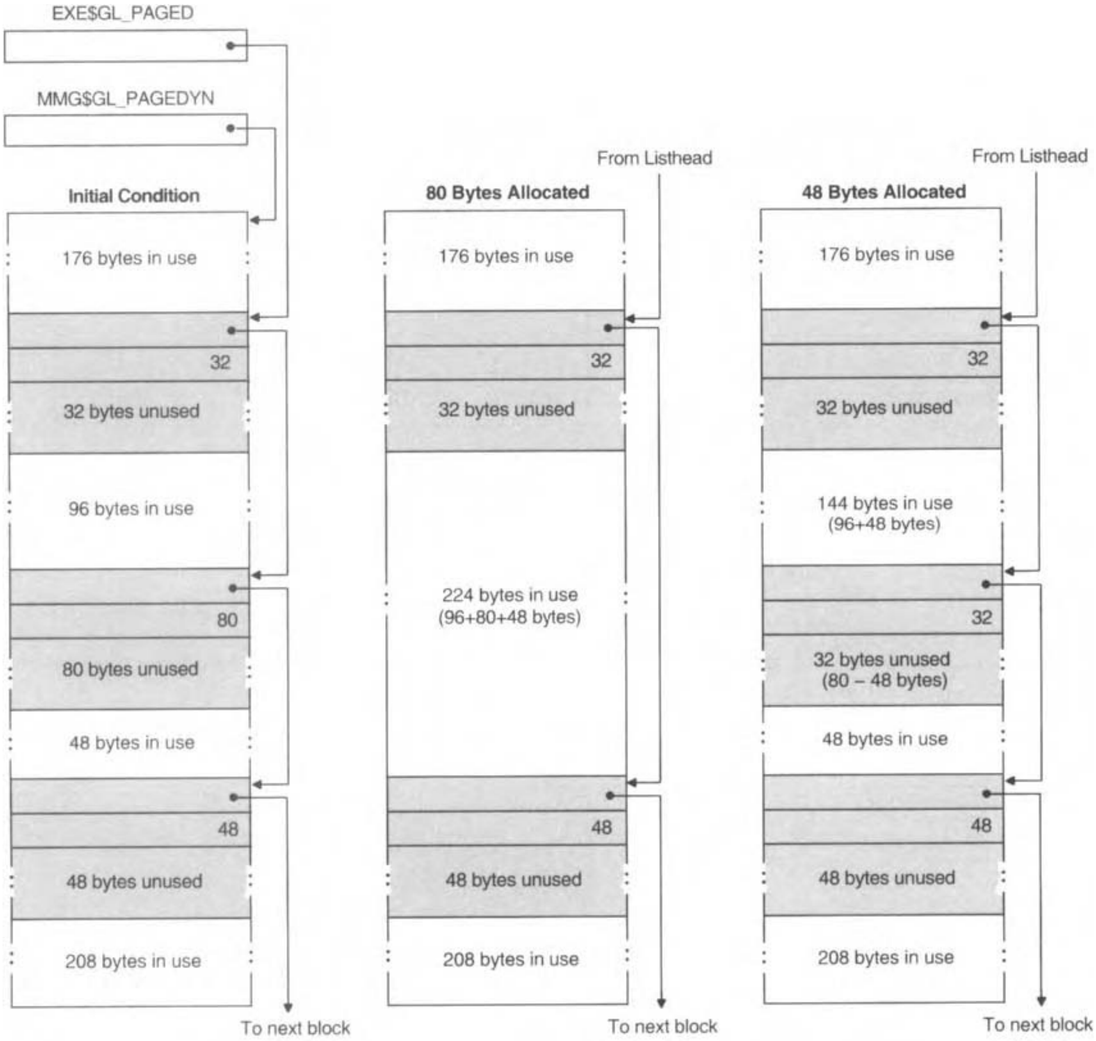
When a block is deallocated, it must be inserted into the list according to its address. EXE[_STD]$DEALLOCATE follows the unused area pointers until it encounters a block whose address is higher than the address of the block to be deallocated. If the deallocated block is adjacent to another unused block, the two blocks are merged into a single unused area.

This merging, or agglomeration, can occur at the end of the preceding unused block or at the beginning of the following block (or both). Because merging occurs automatically as a part of deallocation, there is no need for any externally triggered routine to consolidate pool fragmentation.

Figure 7.5 shows three sample deallocations, two of which illustrate merging. The first part of the figure (Initial Condition) shows an area of paged pool containing logical name blocks for three logical names: ADAM, GREGORY, and ROSAMUND. These three logical name blocks are bracketed by two unused portions of paged pool, one 64 bytes long, the other 176 bytes long.

The second part of Figure 7.5 (ADAM Deleted) shows the result of deleting the logical name ADAM. Because the logical name block was adjacent to the high-address end of an unused block, the blocks are merged. The size of the deallocated block is simply added to the size of the unused block. No pointers need to be adjusted.

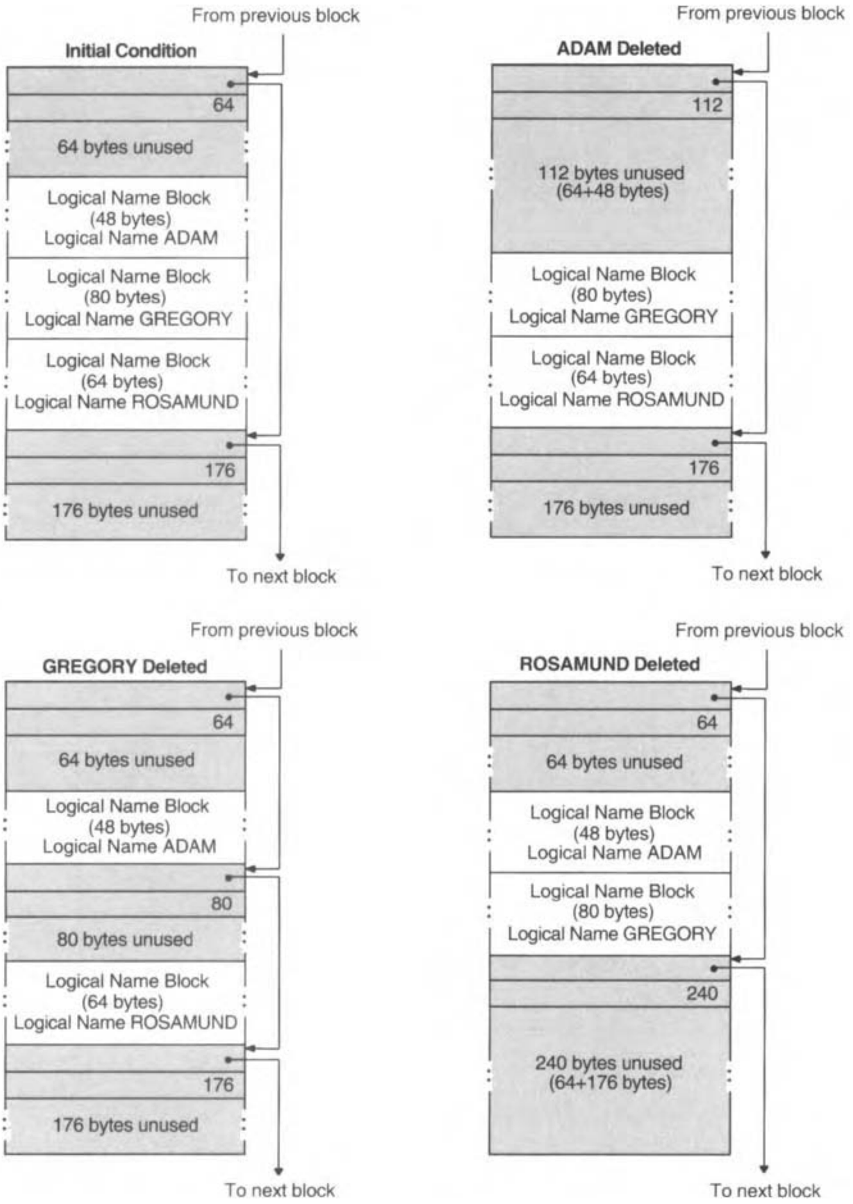
Figure 7.4 Examples of Variable-Length Block Allocation



The structure shown in the third part of Figure 7.5 (GREGORY Deleted) shows an alternative scenario, the result of deleting the logical name GREGORY. The pointer in the unused block of 64 bytes is altered to point to the deallocated block; a new pointer and size longword are created within the deallocated block.

The fourth part of Figure 7.5 (ROSAMUND Deleted) shows the result of deleting the logical name ROSAMUND. In this case, the deallocated block is adjacent to the low-address end of an unused block, so the blocks are merged. The pointer to the next unused block that was previously in the adjacent block is moved to the beginning of the newly deallocated block. The pointer in the unused block of 64 bytes is altered

Figure 7.5 Examples of Variable-Length Block Deallocation



to point to the merged block. The following longword is loaded with the size of the merged block (240 bytes).

7.4 Fixed-Length Lists

Fixed-length lists, also known as lookaside lists, consist of fixed-length packets available for allocation. Fixed-length lists expedite the allocation and deallocation of the most commonly used sizes and types of storage. In contrast to variable-length allocation, fixed-length allocation is very simple. There is minimal overhead searching for a sufficiently large block of free memory to accommodate a specific request.

The OpenVMS Alpha operating system uses both doubly linked lists and a type of singly linked list for fixed-length packet lists. It uses two types of doubly linked lists:

- Each element of an absolute queue contains the addresses of the previous and next elements in the list. Some absolute queues have longword addresses and can only contain elements within 32-bit address space, whereas others have quadword addresses.
- Each element of a self-relative queue contains the displacements to the previous and next elements in the list. Self-relative queues currently used for lookaside lists have longword displacements, limiting the pool from which elements are formed to 4 GB.

The Alpha architecture implements queue insertions and removals through privileged architecture library (PALcode) routines. While PALcode routines automatically provide synchronization, the CPU overhead to call them is high enough that alternatives have been created:

- The mechanism for the singly linked lookaside lists provides an efficient way to insert and remove packets atomically without using PALcode routines.
- The routines that allocate and deallocate from quadword absolute queues rely on higher-level synchronization by their callers and directly modify forward and backward links.

Table 7.4 summarizes fixed-length allocation listheads and routines. In the table, each routine is in module MEMORYALC and contents of locations are dynamic unless marked otherwise.

Table 7.4 Fixed-Length Allocation Listheads and Routines

System Space	
Nonpaged Pool Lookaside Lists	
Type of list	Singly linked absolute list

Table 7.4 ***(continued)*** **Fixed-Length Allocation Listheads and Routines**

	System Space
	Nonpaged Pool Lookaside Lists
Listhead address	EXE$GS_NPP_BASE_LSTHDS + LSTHDS$Q_LISTHEADS + *listhead_offset* (see Figures 7.9 and 7.11)[1,2]
Allocation routines	EXE[_STD]$ALONONPAGED,[3] EXE[_STD]$ALLOCBUF, EXE[_STD]$ALLOC*xyz*,[4] EXE$ALLOCATE_POOL[5]
Deallocation routines	EXE$DEANONPAGED,[3] EXE$DEANONPGDSIZ,[3] EXE$DEALLOCATE_POOL[5]
	Bus-Addressable Pool Lookaside Lists
Type of list	Singly linked absolute list
Listhead address	EXE$GS_BAP_BASE_LSTHDS + LSTHDS$Q_LISTHEADS + *listhead_offset* (see Figure 7.9)[1,2]
Allocation routine	EXE$ALLOCATE_POOL[5]
Deallocation routine	EXE$DEALLOCATE_POOL[5]
	S2 Space Lock Management Lookaside List
Type of list	Doubly linked absolute list
Listhead address	LCK$AR_POOLZONE_REGION (see Figure 7.8)
Allocation routine	EXE$POOL[ZONE]_ALLOCATE[6]
Deallocation routine	EXE$POOL[ZONE]_DEALLOCATE[6]
	KPB Lookaside List
Type of list	Singly linked absolute list
Listhead address	IOC$GQ_KPBLAL
Allocation routine	EXE$KP_ALLOCATE_KPB[7]
Deallocation routine	EXE$KP_DEALLOCATE_KPB[7]

[1] The address of the lookaside listhead for a specific size is static. Given the packet size, this address can be computed using the formula *listhead_offset* = (*packet_size*/40_{16}) * 8.

[2] The listheads for a per-RAD nonpaged pool are in its dynamically allocated LSTHDS structure (see Section 7.5.1).

[3] This routine is in module MEMORYALC_DYN.

[4] *xyz* is the name of a data structure, such as PCB (process control block) or JIB (job information block).

[5] This routine is in module MEMORYALC_DYN_64.

[6] This routine is in module POOL_ZONES.

[7] This routine is in module KERNEL_PROCESS.

Table 7.4 *(continued)* **Fixed-Length Allocation Listheads and Routines**

	System Space
	PQB Lookaside List
Type of list	Self-relative queue
Listhead address	EXE$GQ_PQBIQ
	Process-Private Space
	KRP Lookaside List
Type of list	Absolute queue
Beginning address	@CTL$A_KRP
First free block's address	@CTL$GL_KRPFL
Last free block's address	@CTL$GL_KRPBL

7.4.1 Doubly Linked Lookaside Lists

Insertion and removal of an element from the head or tail of a queue through PALcode routines are atomic:

- For an absolute queue, each such modification is atomic with respect to any other threads of execution on the same processor.
- For a self-relative queue, each such modification is atomic with respect to all other threads of execution on all members of a symmetric multiprocessing (SMP) system.

Chapter *Synchronization Techniques* contains further information on queues and synchronizing access to them.

Figure 7.6 (Initial Condition) shows the general form of a fixed-length list that is either a self-relative queue or an absolute queue.

A packet is allocated by removing the first element from the front of the list (see Figure 7.6, Packet Removed from Head). A packet is deallocated by inserting it at the back of the list (see Figure 7.6, Packet Inserted at Tail).

7.4.2 Singly Linked Lookaside Lists

Shown in Figure 7.7, this newer type of lookaside list is singly linked and absolute. Its listhead is a naturally aligned quadword. The first longword of the list contains the address of the first packet, or zero if the list is empty. The second longword is a sequence number used in synchronizing access to the list. Packets are always allocated from and deallocated to the front of this kind of list. The first longword of each packet contains the address of the next packet in the list; the first longword of the last packet contains zero.

Figure 7.6 Fixed-Length Packet Allocation and Deallocation from a Queue

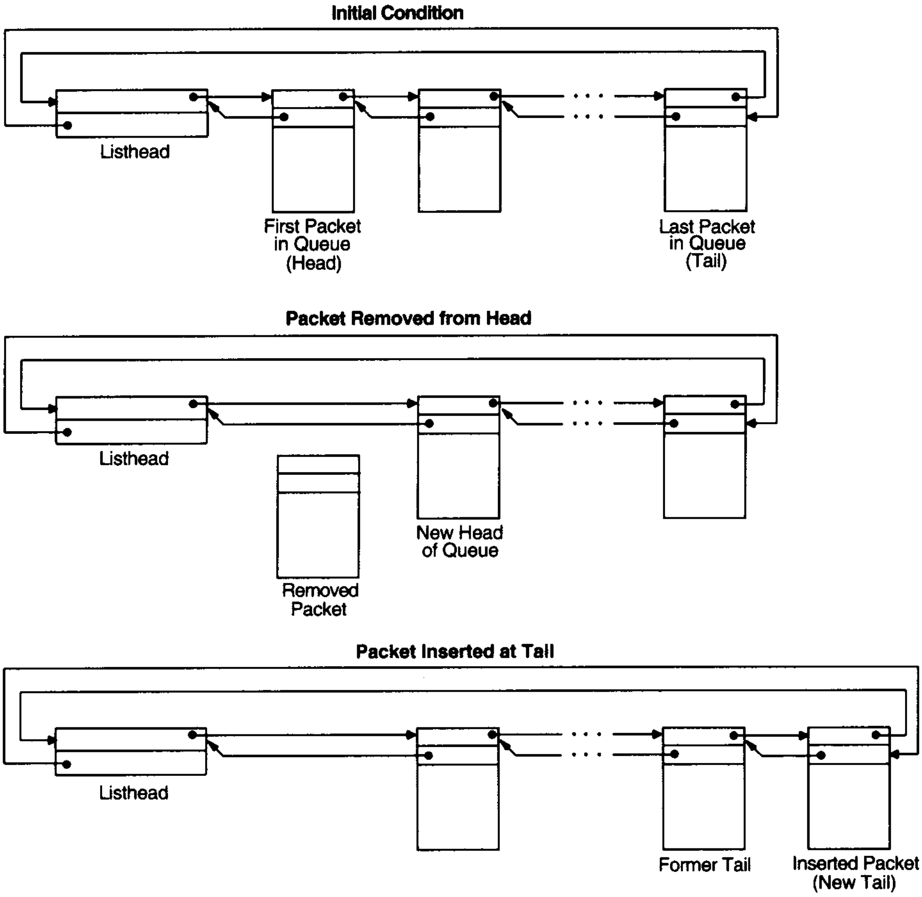
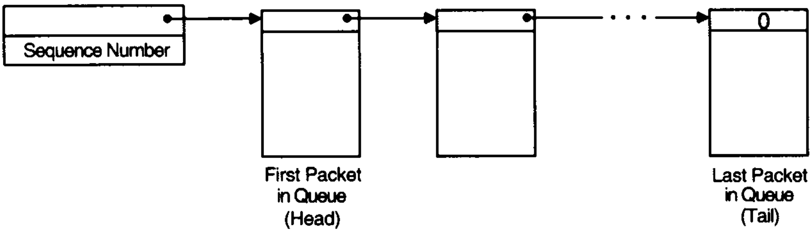


Figure 7.7 Singly Linked Lookaside List



Routines EXE$LAL_REMOVE_FIRST[_AND_COUNT] and EXE$LAL_INSERT_FIRST[_AND_COUNT], in module LOOK_ASIDE_LIST, allocate and deallocate packets from this list. The insertions and removals are atomic with respect to all threads of execution on all SMP system members without the use of a spinlock.

7.4.2.1 Singly Linked List Deallocation

To deallocate a packet, EXE$LAL_INSERT_FIRST and EXE$LAL_INSERT_FIRST_AND_COUNT take the following steps:

1. Each routine copies the address of the first packet in the list from the listhead to the forward link of the packet being deallocated.
2. It executes a memory barrier (MB) instruction to ensure that the first write is visible before the next write.
3. It executes a load-locked instruction (LDL_L) to refetch the address of the first packet from the listhead.
4. If that address has changed, it restarts the insertion at step 1. Otherwise, it conditionally stores (STL_C) the address of the packet being deallocated in the listhead.

   If the store operation fails, another thread of execution has interrupted this one (or accessed the list concurrently on another SMP system member); in that case, each routine restarts the insertion at step 1.
5. If the store operation succeeds, EXE$LAL_INSERT_FIRST_AND_COUNT increments the packet counter associated with this lookaside list.
6. Each routine returns to its caller.

7.4.2.2 Singly Linked List Allocation

Because the store-conditional instruction will fail if a memory reference occurs between the load and the store, allocating a packet is somewhat more complex than deallocating one. To allocate a packet, EXE$LAL_REMOVE_FIRST and EXE$LAL_REMOVE_FIRST_AND_COUNT take the following steps:

1. Each routine loads both the sequence number and address of the first packet in the list. If the list is empty, it returns a failure status to its caller.
2. It executes an MB instruction to ensure the first read is visible before the next.
3. It loads the address of the second packet in the list from the forward link of the first.
4. It executes a load-locked (LDQ_L) instruction to refetch the sequence number and address of the first packet. If either has changed, it restarts the removal at step 1.
5. It forms the new contents of the listhead as the incremented sequence number and address of the second packet.
6. It conditionally stores (STQ_C) these contents. If the store operation fails, it restarts the removal at step 1.
7. If the store operation succeeds, the routine confirms that the forward pointer of the packet just allocated is the same as the address it loaded in step 3. If the addresses are the same, EXE$LAL_REMOVE_FIRST_AND_COUNT decrements the packet counter associated with this lookaside list.

If the addresses are not the same, it generates the fatal bugcheck BADQHDR. This sanity check has a high probability of detecting the unlikely event that between steps 1 and 3, 2^{31} other accesses occurred to the list (so that the sequence number wrapped around to itself) and the first packet in the list at step 1 was again the first packet in the list at step 4.

8. Each routine returns to its caller with the address of the allocated packet.

7.4.3 Pool Zone Lookaside Lists

A mechanism new with OpenVMS Alpha Version 7.2 enables a kernel mode component to create its own system space pool. The pool consists of one or more zones. Each page of each zone is divided into fixed-size packets, and each zone can have a different packet size. The pages that make up a zone are not required to be physically or virtually contiguous. This mechanism was added primarily to create lookaside lists in S2 space.

7.4.3.1 Pool and Zone Creation

To create the pool, the component creates a data structure to describe the pool. The structure consists of a POOLZONE_REGION structure plus a POOLZONE substructure for each zone. The structure may be allocated from nonpaged pool or created as part of an executive image. The component initializes the POOLZONE_REGION header with the number of zones and the addresses of page allocation and deallocation routines for that pool. The component then calls EXE$POOLZONE_CREATE, in module POOL_ZONES, once for each zone, starting with the smallest packet size and continuing in order by packet size.

EXE$POOLZONE_CREATE is passed a pointer to the POOLZONE_REGION structure, the packet size, initial region size in pages, and maximum region size. It allocates and maps physical pages of memory for the initial zone size, using the page allocation routine specified in the POOLZONE_REGION structure. The page allocation routine can, for example, call MMG_STD$ALLOC_SYSTEM_VA_MAP, in module SYSVA_ALLOC, which performs these tasks, making the appropriate changes to the memory management database.

EXE$POOLZONE_CREATE initializes a POOLZONE_PAGE data structure at the beginning of each page of memory and links each POOLZONE_PAGE into the front of a listhead in its zone's POOLZONE structure. It splits each page into fixed-size packets. It clears the longword that contains the standard SIZE, TYPE, and SUBTYPE fields in each packet; optionally fills the packet with the pool poison deallocation pattern (see Section 7.15); and links the packet into a listhead in the POOLZONE_PAGE structure.

7.4.3.2 Data Structures

Figure 7.8 shows the pool zone data structures and their relationships. These data structures are defined only by a C header file, and their field names are lowercase. The POOLZONE_REGION structure is typically allocated from nonpaged pool and includes one POOLZONE structure for each zone in the pool.

In the POOLZONE_REGION structure, type, subtype, and size have the typical meanings. Type and subtype are component-specific. Lock management code, for example, uses a type of DYN$C_LCK and a subtype of DYN$C_LCK_POOLZONE. Fields zonepage_alloc_rtn and zonepage_dealloc_rtn contain addresses of routines within the kernel mode component that allocate and deallocate physical pages and their mappings. The field zone_count contains the number of POOLZONEs that follow.

In the POOLZONE structure, zonepage_flink and zonepage_blink form a queue listhead for POOLZONE_PAGEs that are part of this zone. Field packet_size contains the size in bytes of packets in this zone. Field max_pages is the maximum number of POOLZONE_PAGEs in this zone. Field free_count is the number of available packets in this zone.

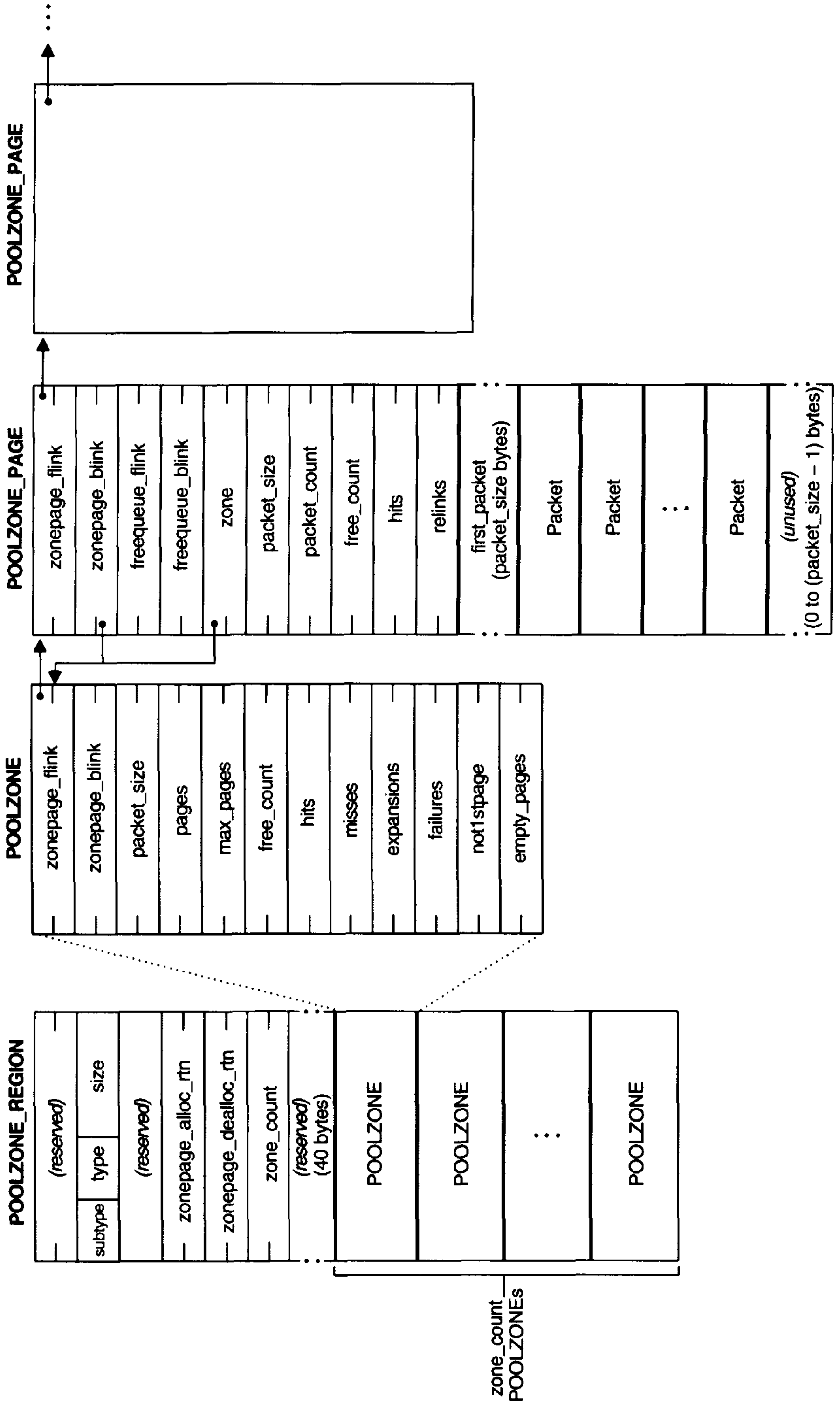
Field misses records allocation attempts when the zone initially has no available packets. Field hits records allocation attempts when the zone does have available packets. Fields expansions and failures record successful and failed attempts to expand the zone. The field not1stpage, which is maintained only by the monitor version of SYSTEM_PRIMITIVES (see Section 7.13), is the number of failures to find a packet on the first page in the POOLZONE_PAGE queue. Field empty_pages counts the number of pages in the zone from which no packets have been allocated.

In the POOLZONE_PAGE structure, zonepage_flink and zonepage_blink link the page into the associated zone's queue. Fields freequeue_flink and freequeue_blink form the listhead of available packets in this page. Field zone contains the address of the page's associated POOLZONE structure. Field packet_size contains the size in bytes of this page's packets. Field packet_count contains the number of packets on this POOLZONE_PAGE, and field free_count, the number of available packets.

Field hits records the number of successful allocations from this page. Field relinks records the number of times this POOLZONE_PAGE has been moved from its position on the POOLZONE_PAGE list. A page with no available packets is moved to the end of the POOLZONE_PAGE list. When a packet is deallocated to such a page, it is moved to the front of the POOLZONE_PAGE list. Field relinks is maintained only by the monitor version of SYSTEM_PRIMITIVES (see Section 7.13).

The rest of the page is divided into packets of size packet_size. Field first_packet is the offset of the first packet in the page.

Figure 7.8 Pool Zone Data Structures



7.4.3.3 Allocation and Zone Expansion

To allocate a packet, a kernel mode component calls EXE$POOLZONE_ALLOCATE or EXE$POOL_ALLOCATE, both in module POOL_ZONES.

EXE$POOLZONE_ALLOCATE is passed the addresses of the POOLZONE_REGION and the POOLZONE structures. It checks whether the zone has any free pages and, if not, expands the zone by allocating and initializing another POOLZONE_PAGE. Thus the zone can be expanded, up to its maximum number of pages.

EXE$POOLZONE_ALLOCATE then checks whether the first page in that pool zone has any free packets and, if so, removes the first packet from the list and returns its address to its caller. If not, it checks the next POOLZONE_PAGE in that zone, continuing until it finds one with a free packet. When it removes the last packet from a POOLZONE_PAGE, it removes the POOLZONE_PAGE from the list and reinserts it at the end of the list, to shorten the search for a free packet on subsequent allocations.

EXE$POOL_ALLOCATE is passed the address of the POOLZONE_REGION structure and the number of bytes to be allocated. It examines the POOLZONE substructures to find the first zone whose packet size is large enough to accommodate the request. It calls EXE$POOLZONE_ALLOCATE to allocate a packet from that list, optionally records the allocation in the nonpaged pool history buffer (see Section 7.15.2), and optionally checks that the packet's poison pattern is intact (see Section 7.15).

7.4.3.4 Deallocation

To deallocate a packet, a kernel mode component calls either EXE$POOLZONE_DEALLOCATE or EXE$POOL_DEALLOCATE, both in module POOL_ZONES.

EXE$POOLZONE_DEALLOCATE is passed the addresses of the POOLZONE structure and the packet. It rounds the packet address back to the page boundary to form the address of the associated POOLZONE_PAGE structure and inserts the packet on its free list. If the page previously had no free packets, EXE$POOLZONE_DEALLOCATE removes the POOLZONE_PAGE from the POOLZONE_PAGE queue and reinserts it at the front of the queue.

EXE$POOL_DEALLOCATE is passed the address of the packet to be deallocated. It rounds the address back to a page boundary to form the address of the associated POOLZONE_PAGE structure and follows its pointer to the POOLZONE structure. Optionally, it records the deallocation in the nonpaged pool history buffer (see Section 7.15.2). Optionally, it fills the packet with the pool poison deallocation pattern (see Section 7.15). It calls EXE$POOLZONE_DEALLOCATE to deallocate the packet.

7.4.3.5 Reclamation

EXE$POOLZONE_PURGE, in module POOL_ZONES, deallocates POOLZONE_PAGEs from which no packets have been allocated. It is called with pointers to a POOLZONE_REGION, POOLZONE, and a target number of POOLZONE_PAGEs to reclaim. It scans the POOLZONE_PAGE queue for a POOLZONE_PAGE all of whose packets are available. It calls the page deallocation routine specified by ZONEPAGE_REGION field zonepage_dealloc_rtn.

7.5 Nonpaged Pool

Nonpaged dynamic memory, commonly known as nonpaged pool, contains data structures used by components that typically run in system context, such as unit control blocks and I/O request packets. These parts of the operating system can only access system space. Furthermore, they execute at IPLs above 2, where page faults are not permitted.

Nonpaged dynamic memory also contains data structures that are shared by multiple processes and that may be accessed above IPL 2. Nonpaged pool is the most heavily used of the pool areas.

The protection on nonpaged pool is ERKW, allowing it to be read from executive and kernel modes but written only from kernel mode.

Nonpaged pool consists of a variable-length list and a number of fixed-length lookaside lists. The lookaside lists provide for the most frequently allocated nonpaged pool data structures. Section 7.5.4 discusses allocation in detail.

The OpenVMS Alpha executive provides 128 (IOC_C_NUMLISTS, defined by $NPOOL_DATADEF) lookaside lists for packets ranging in size from 64 to 8,192 (IOC_C_MAXLISTPKT) bytes in increments of 64 bytes. These lookaside lists are of the singly linked absolute type. Section 7.5.1 describes these lists in more detail.

In the case of a NUMA system, each RAD with memory can have its own variable-length and fixed-length nonpaged pool lists. Section 7.6 provides further details.

In addition to the traditional type of nonpaged pool, certain systems also have a bus-addressable nonpaged pool (see Section 7.7).

A nonpaged pool allocation routine attempting to service a request first rounds up the requested size to the next multiple of 64 (EXE$M_NPAGGRNMSK + 1) and checks the listhead corresponding to the requested size. If there is no packet on that list or if the requested size is larger than 8,192 bytes, the allocation routine allocates pool from the variable-length list. Thus, all packets on nonpaged pool lookaside lists originate in the nonpaged pool variable-length region.

A nonpaged pool deallocation routine does not return pool directly to the variable-length list. Rather, the deallocation routine inserts it into the lookaside list corresponding to the packet's size unless the size is larger than 8,192 bytes.

Packets do not remain on the lookaside lists forever. They are either consumed by later allocation requests or returned to the variable-length list through a process called pool reclamation. When there is no packet on a request size's corresponding list and there is insufficient memory in the nonpaged pool variable-length region, the executive initiates pool reclamation. It also initiates pool reclamation periodically to ensure sufficient memory on the nonpaged pool variable-length list. Section 7.5.6 describes nonpaged pool reclamation.

When a nonpaged pool request cannot be satisfied even after pool reclamation, the executive attempts to expand nonpaged pool. Section 7.5.7 describes nonpaged pool expansion.

In addition to nonpaged pool lookaside lists, the executive provides a lookaside list of KPBs, used primarily by device driver fork processes. KPBs, initially allocated from nonpaged pool, are deallocated to the KPB lookaside list. The KPB allocation routine attempts to allocate a KPB from this list as a faster alternative to general nonpaged pool allocation. Each KPB points to an associated kernel process stack. Allocation and initialization of a kernel process stack is a time-consuming process. Maintaining the KPBs on a separate lookaside list allows the executive to reuse KPBs and their associated stacks. Chapter *Software Interrupts* describes kernel processes and KPBs.

7.5.1 Data Structures

The implementation of nonpaged pool has been generalized to enable special pool types to be created. OpenVMS Version 7.2 added support for bus-addressable nonpaged pool (BAP). As part of the generalization, various system cells that described nonpaged pool were moved to data structures, and new routines used these data structures to determine their actions. For compatibility, the original nonpaged pool allocation and deallocation routines are still provided.

BAP and the base RAD's nonpaged pool are each described by an NPOOL and a LSTHDS data structure. These structures are static, created during compilation of module SYSTEM_DATA_CELLS. Figure 7.9 shows these structures and the relationship between them.

This section describes the fields common to the structures for both types of pool, as well as fields specific to the nonpaged pool NPOOL. Section 7.7.1 describes fields specific to BAP, and Section 7.6.1 describes extensions for support of per-RAD nonpaged pool.

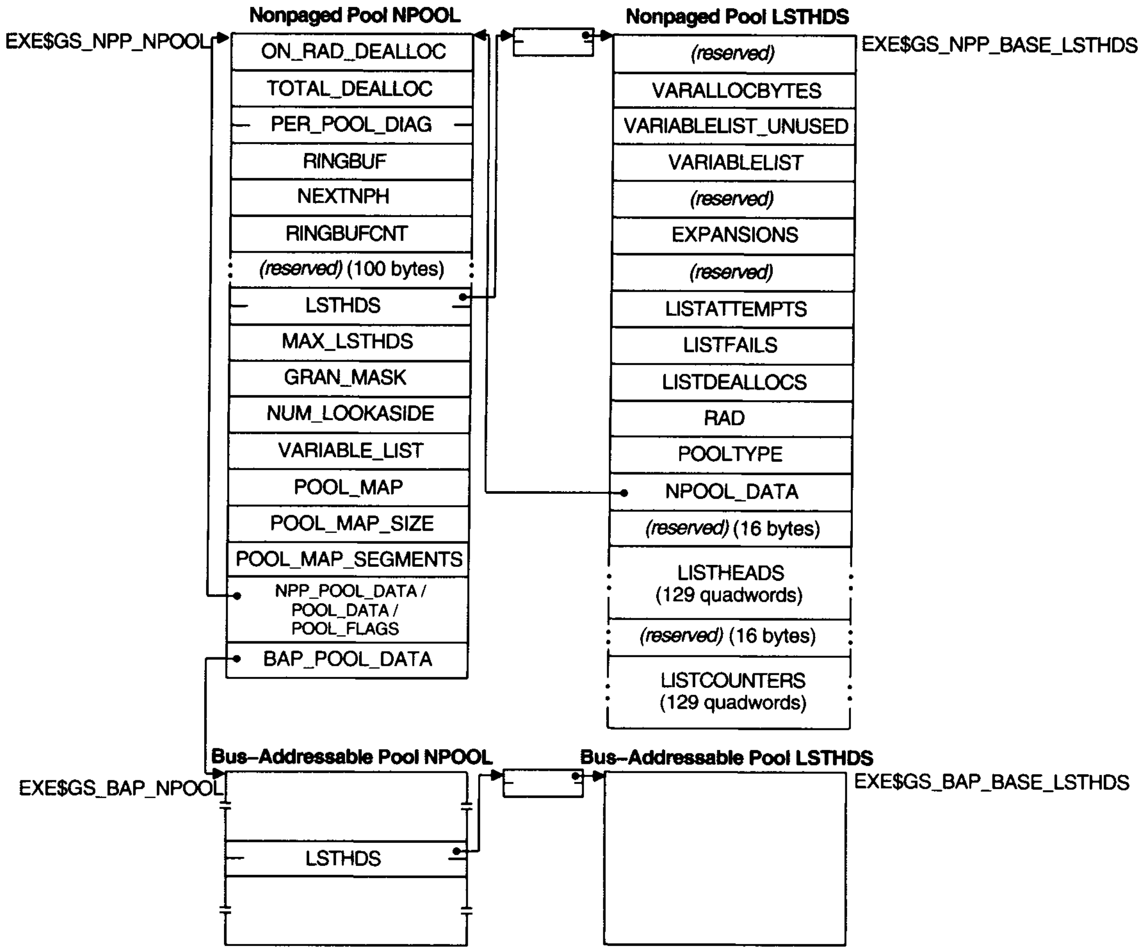
In the nonpaged pool NPOOL structure, NPOOL$PS_RINGBUF contains the address of the pool history ring buffer, and NPOOL$L_RINGBUFCNT, the number of pool history buffers in it. NPOOL$PS_NEXTNPH contains a pointer to the next history buffer to be used. The standard nonpaged pool ring buffer records both nonpaged and bus-addressable pool history, as well as pool zone history. Section 7.15.2 contains more information.

In each type of NPOOL structure, NPOOL$PS_POOL_MAP contains the address of a list of descriptors of segments that make up this pool. NPOOL$L_POOL_MAP_SIZE contains the size of the list in bytes, and NPOOL$L_POOL_MAP_SEGMENTS, the number of descriptors in it. Each descriptor consists of four quadwords: the first contains the address of the segment; the second, its size in bytes; the third, the address of the end of the segment; and the fourth, a longword with the number of the associated RAD.

In the nonpaged pool structure, NPOOL$L_BAP_POOL_DATA contains the address of the BAP NPOOL structure.

NPOOL$AR_LSTHDS points to a block containing one or more addresses of LSTHDS structures. This indirection enables the same code to be used in systems that have per-RAD nonpaged pool and those that do not.

Figure 7.9 Nonpaged and Bus-Addressable Pool Data Structures



NPOOL$L_GRAN_MASK specifies the granularity of allocation for this pool. It is initialized to EXE$M_NPAGGRNMSK for both types of pool.

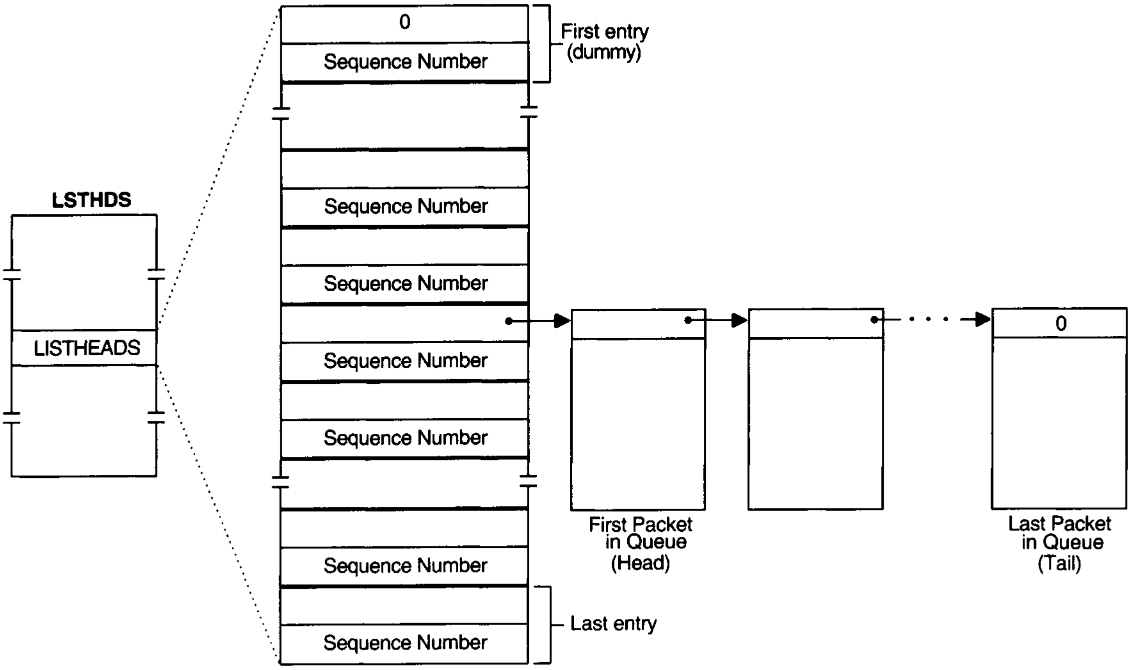
NPOOL$L_NUM_LOOKASIDE specifies the number of lookaside lists for this pool. It is initialized to IOC_C_NUMLISTS for both types of pool.

NPOOL$PS_VARIABLE_LIST contains the address of the listhead for the variable-length pool of this type but is not used.

A LSTHDS structure contains the actual nonpaged pool lookaside lists. Figure 7.10 shows the array of lookaside lists and an example lookaside list.

LSTHDS$AR_LISTATTEMPTS, LSTHDS$AR_LISTFAILS, and LSTHDS$AR_LISTDEALLOCS point to statistics buffers, which are described in Section 7.14, along with LSTHDS$L_VARALLOCBYTES.

Figure 7.10 Nonpaged Pool Lookaside Lists



LSTHDS$PS_VARIABLELIST contains the address of the variable-length listhead for this type of pool.

LSTHDS$L_EXPANSIONS records the number of times this pool has been expanded.

LSTHDS$L_POOLTYPE identifies the pool, either MMG$K_POOLTYPE_NPP or MMG$K_POOLTYPE_BAP.

LSTHDS$PS_NPOOL_DATA points to the associated NPOOL structure.

LSTHDS$Q_LISTHEADS is an array of 129 (IOC_C_NUMLISTS + 1) lookaside listheads (see Figure 7.10). For both BAP and the base RAD's nonpaged pool, this array is created as a zeroed array of longwords during compilation of module SYSTEM_DATA_CELLS. The extra entry enables one-based indexing of the array as a function of the packet size. The lookaside lists are not prepopulated at system initialization. Instead, when a block of pool is deallocated, if its size corresponds to a lookaside list size, the block is inserted on that lookaside list.

LSTHDS$Q_LISTCOUNTERS is a corresponding array of counters. Each element in the array is the count of packets on the corresponding lookaside list. The counts are not necessarily precise because some lookaside list insertions and removals do not update the counts and because the counts are not kept atomically. Used during pool reclamation, the counters are self-correcting.

7.5.2 Uses of Nonpaged Pool

Nonpaged pool is created during early stages of system initialization. The following executive data structures are allocated from nonpaged pool:

- Buffered I/O buffers
- I/O data structures, such as I/O request packets, unit control blocks, controller request blocks, adapter control blocks, window control blocks, file control blocks, class driver data blocks, and class driver request packets
- Synchronization data structures, such as common event blocks and dynamic spinlocks
- Process data structures, such as process control blocks, job information blocks, and kernel thread blocks
- Kernel process blocks
- Other miscellaneous systemwide data structures, such as timer queue entries

7.5.3 Initialization

SYSGEN parameters NPAGEDYN and NPAGEVIR specify the size of nonpaged pool. NPAGEDYN is the initial size of nonpaged pool in bytes. NPAGEVIR is the maximum size, in bytes, to which it can expand.

New with OpenVMS Alpha Version 7.3, parameter NPAGECALC allows for automatic calculation of NPAGEDYN. Its default value at initial system boot is 1, but running AUTOGEN changes it to 0. The SYSGEN or SYSBOOT USE DEFAULT command changes it back to 1.

During system initialization, if SYSGEN parameter NPAGECALC is 1, SYSBOOT calculates default values for these parameters based on the amount of physical memory. The default calculated value for NPAGEDYN is 512 KB plus 1 page per 128 pages of memory, up to a maximum of 128 MB. The default value for NPAGEVIR is 8 MB plus 1 page for each 32 pages of memory, up to a maximum of 256 MB. Both parameters are rounded down to a number representing an integral number of pages.

As described in Section 7.7, SYSBOOT may adjust the initial and maximum sizes of nonpaged pool to include BAP.

SYSBOOT also adjusts the initial size of nonpaged pool if per-RAD pool is needed on this system. If bit RIH$V_RAD_POOL (bit 6) is set in SYSGEN parameter RAD_SUPPORT, SYSBOOT rounds up parameter NPAGERAD, the number of bytes to reserve for nonbase-RAD pool, to an integral number of pages and subtracts it from the initial size of nonpaged pool. It then adjusts the initial size to a minimum of 4 MB.

SYSBOOT allocates a slice of the nonpaged system data huge page for the initial size of nonpaged pool (see Chapter 1). SYSBOOT also reserves enough virtual address space contiguous to this region for nonpaged pool to expand to its maximum size.

SYSBOOT initializes the nonpaged pool variable-length list and the global locations EXE$GL_NONPAGED, MMG$GL_NPAGEDYN, and MMG$GL_NPAGNEXT.

Later in system initialization, INI$INITIALIZE_POOL, in module MEMORYALC_POOL, initializes the NPOOL structure that describes nonpaged pool. In particular, it fills in a pool map descriptor to describe the initial nonpaged pool segment.

7.5.4 Allocation

A number of routines in module MEMORYALC allocate traditional nonpaged pool. Some of these routines, such as EXE[_STD]$ALLOCPCB or EXE[_STD]$ALLOCTQE, allocate pool for a particular type of data structure, filling in its size and type. Some routines, intended for use only within process context, conditionally place the kernel thread into a resource wait (see Section 7.5.8 and Chapter *Scheduling*) for resource RSN$_NPDYNMEM if pool is unavailable. All these routines call EXE$ALONONPAGED, in module MEMORYALC_DYN, the general traditional nonpaged pool allocation routine.

Another general nonpaged pool allocation routine is EXE$ALLOCATE_POOL, in module MEMORYALC_POOL. It can also allocate BAP or pool local to a specific RAD. It is called with arguments specifying pool type, requested size, RAD, and alignment.

Because allocation from and deallocation to a lookaside list are so much faster than the equivalent operations involving the variable-length list, EXE[_STD]$ALONONPAGED and EXE$ALLOCATE_POOL check to determine whether a requested block can be allocated from one of the lookaside lists. Each allocates requests from the variable-length list only if the requested size is larger than 8,192 (IOC_C_MAXLISTPKT) bytes or if the lookaside list corresponding to the requested size is empty.

A consumer of nonpaged pool must use an appropriate executive procedure for allocation and deallocation. Direct allocation from or deallocation to a nonpaged pool lookaside list is not allowed. That is, directly manipulating a lookaside list through the EXELAL_REMOVE_FIRST/EXELAL_INSERT_FIRST routines or through the load-locked/store-conditional mechanism is not allowed.

EXE[_STD]$ALONONPAGED and EXE[_STD]$ALONPAGVAR are entry points to the same procedure, EXE$ALONONPAGED_INT in module MEMORYALC_DYN, which allocates nonpaged pool by performing the following steps:

1. If per-RAD pool is in use, it calls EXE$ALLOCATE_POOL with pool type MMG$K_POOLTYPE_NPP, to allocate a block of the requested size from this system's per-RAD nonpaged pool. Section 7.6.3 describes how EXE$ALLOCATE_POOL handles RAD-specific requests. (The traditional routines can allocate only base RAD pool.) EXE$ALONONPAGED_INT returns the status from EXE$ALLOCATE_POOL and the address of the packet, if any, to its caller.

2. Otherwise, it rounds up the requested size to the nearest multiple of 64 (EXE$M_NPAGGRNMSK + 1).

3. If the rounded value of the requested size is larger than 8,192 bytes, it proceeds with step 5.

4. It calls EXE$LAL_REMOVE_FIRST_AND_COUNT (described in Section 7.4) to allocate the first packet from the lookaside list corresponding to the requested size. If a packet was successfully allocated, EXE$ALONONPAGED_INT returns to its caller with a success status and the address of the packet. Otherwise, it continues.
5. If the current IPL is greater than IPL$_POOL and system initialization has completed (BOOSTATE$V_SWAPPER flag in EXE$GL_STATE is set), it returns to its caller with the error status SS$_INSFMEM. Otherwise, it continues. Allocating from the variable-length list at above IPL$_POOL is permissible only during system initialization.
6. It calls EXE$ALONPAGVAR_INT, in module MEMORYALC_DYN, to allocate a nonpaged pool block of the requested size and returns the status from EXE$ALONPAGVAR_INT and the address of the packet, if any, to its caller.

EXE$ALONPAGVAR_INT allocates pool only from the variable-length list. It performs the following steps:

1. It rounds the request size up to a multiple of 64 (EXE$M_NPAGGRNMSK + 1).
2. It increments PMS$GL_NPAGDYNREQ, which tracks the number of allocation requests for variable-length pool (see Table 7.5).
3. It acquires the POOL spinlock, raising IPL to IPL$_POOL.
4. It calls the lower level routine EXE$ALLOCATE, described in Section 7.3.
5. EXE$ALONPAGVAR_INT releases the POOL spinlock, restoring the previous IPL. If EXE$ALLOCATE succeeded, EXE$ALONPAGVAR_INT returns the size and address of the allocated block.
6. If the allocation failed, it checks whether pool reclamation was already performed for this request. If not, it calls EXE$RECLAIM_POOL_AGGRESSIVE, in module MEMORYALC_POOL (see Section 7.5.6). Upon return from EXE$RECLAIM_POOL_AGGRESSIVE, regardless of whether pool was reclaimed, EXE$ALONPAGVAR_INT retries pool allocation beginning with step 3.
7. If pool reclamation was already attempted for this request, it instead calls EXE$EXTENDPOOL, in module MEMORYALC_POOL, to attempt pool expansion (see Section 7.5.7).

   If the expansion succeeds, EXE$ALONPAGVAR_INT repeats the allocation attempt. If pool expansion fails because pool has been expanded to its maximum size, it calls EXE$FLUSHLISTS, in module MEMORYALC_DYN (see Section 7.5.6).

   If, despite the expansion and flushing effort, the nonpaged pool request cannot be satisfied, EXE$ALONPAGVAR_INT increments PMS$GL_NPAGDYNREQF and updates PMS$GL_NPAGDYNFPAGES (see Table 7.5) and returns the error status SS$_INSFMEM to its caller.

Since nonpaged pool allocation granularity is 64 (EXE$M_NPAGGRNMSK + 1) bytes and nonpaged pool begins at a page boundary, all nonpaged pool packets and blocks are guaranteed to be at least 64-byte aligned.

A consumer requiring greater than 64-byte alignment can call either the routine EXE$ALONONPAGED_ALN, in module MEMORYALC_DYN, or the routine EXE$ALLOCATE_POOL, in module MEMORYALC_POOL. Each attempts nonpaged pool allocation to the specified alignment constraint.

To allocate traditional nonpaged pool, EXE$ALLOCATE_POOL takes the following steps:

1. If the requested size is 0, the monitor version (see Section 7.13) generates the fatal bugcheck BADALORQSZ.
2. EXE$ALLOCATE_POOL determines the address of the LSTHDS for this pool type and RAD.
3. If the caller specified alignment requirements, EXE$ALLOCATE_POOL continues with step 10.
4. Otherwise, if the requested size is smaller than 8,192 (IOC_C_MAXLISTPKT) bytes, it tries to remove a packet from the lookaside list corresponding to that size and pool type. If successful, it returns to its caller.
5. If the request is larger or the lookaside list was empty, it acquires the POOL spinlock to synchronize access to the variable-length list associated with this pool type.
6. It calls EXE_STD$ALLOCATE to allocate pool from the variable-length list.
7. It releases the POOL spinlock.
8. If the pool was allocated successfully, EXE$ALLOCATE_POOL returns to its caller.
9. If the request cannot be allocated, EXE$ALLOCATE_POOL tries one after another of the following techniques, reattempting the pool allocation each time:

   — Aggressive pool reclamation from lookaside lists (see Section 7.5.6)

   — Flushing the lookaside lists (see Section 7.5.6)

   — Expanding pool (see Section 7.5.7)

   If all these attempts to regain pool are unsuccessful, EXE$ALLOCATE_POOL returns error status SS$_INSFMEM to its caller.
10. If the caller requested a specific alignment and if the request is smaller than 8,192 bytes, EXE$ALLOCATE_POOL makes three attempts to remove a packet from the corresponding lookaside list that meets the requested alignment and that does not cross a page boundary.

11. If the lookaside list allocation failed or the request is too large, it calls STD_ALLOCATE_ALN, in module MEMORYALC, which allocates a block from the variable-length list that meets the requested alignment and that does not cross a page boundary.

    If STD_ALLOCATE_ALN is unsuccessful, EXE$ALLOCATE_POOL attempts to reclaim pool as described in step 9.

Sections 7.6.3 and 7.7.3 describe how EXE$ALLOCATE_POOL handles RAD-specific and bus-addressable pool allocations.

7.5.5 Deallocation

To deallocate nonpaged pool, a consumer of nonpaged pool calls EXE[_STD]$DEANONPAGED or EXE[_STD]$DEANONPGDSIZ, in module MEMORYALC_DYN, or EXE$DEALLOCATE_POOL, in module MEMORYALC_POOL. EXE[_STD]$DEANONPGDSIZ is used to deallocate a pool block larger than 64 KB or a block with one of the new header formats shown in Figure 7.2.

EXE[_STD]$DEANONPAGED tests whether per-RAD pool is in use. If so, it calls EXE$DEALLOCATE_POOL and returns. Otherwise, it determines the size of the block being returned and calls EXE$DEANONPGDSIZ.

EXE[_STD]$DEANONPGDSIZ returns the deallocated block either to one of the lookaside lists or to the variable-length region, performing the following steps:

1. If per-RAD pool is in use, it calls EXE$DEALLOCATE_POOL and returns.
2. If per-RAD pool is not in use, the monitor version of the routine (see Section 7.13) tests that the size being returned is nonzero and that the starting address is on a pool allocation granularity boundary. If either is false, it generates the fatal BADDALRQSZ bugcheck.
3. It rounds up the deallocation request size to a multiple of 64 (EXE$M_NPAGGRNMSK + 1).
4. If the rounded deallocation size is less than or equal to 8,192 (IOC_C_MAXLISTPKT) bytes, it determines the appropriate listhead and calls EXE$LAL_INSERT_FIRST_AND_COUNT to return the deallocated packet to the front of that list. It then returns to its caller.
5. If the rounded deallocation size is larger than 8,192 bytes, EXE$DEANONPGDSIZ acquires the POOL spinlock, raising IPL to IPL$_POOL; calls EXE$DEALLOCATE_POOL, the lower level routine described in Section 7.3; and then releases the POOL spinlock, restoring the previous IPL.

EXE$DEALLOCATE_POOL is called with arguments that include pool type and RAD. It returns the deallocated block either to one of the lookaside lists or to the variable-length region for that pool type and RAD. It determines the address of the appropriate LSTHDS.

If the packet is smaller than 8,192 bytes, EXE$DEALLOCATE_POOL inserts it on the listhead appropriate to the pool type and size. Otherwise, it acquires the POOL spinlock, calls EXE_STD$DEALLOCATE to deallocate the pool to the appropriate variable-length list, and releases the spinlock.

7.5.6 Reclamation

Also called adaptive nonpaged pool management, pool reclamation simplifies system management by automatically adapting to varying workloads, thereby eliminating a number of SYSGEN parameters used with the earlier style of pool management.

As previously described, nonpaged pool deallocation routines insert a packet on a lookaside list rather than returning it to the variable-length list. Returning a packet to a lookaside list enables faster allocation of packets of that size. On a running system, however, the demanded allocation sizes are somewhat unpredictable. As more packets are put on lookaside lists, remaining space on the variable-length list gets smaller. Without a process for reclaiming space from unused lookaside list packets, nonpaged pool exhaustion or excessive fragmentation can occur, slowing down or preventing further allocation.

Through a process called nonpaged pool reclamation, packets from nonpaged pool lookaside lists are moved to the associated nonpaged pool variable-length list. Reclamation can be gentle or aggressive. The executive performs gentle reclamation periodically. It performs aggressive reclamation when an allocation request cannot be satisfied from either the appropriate lookaside list or the associated variable-length list.

Reclamation is not possible if there are no packets on any of the lookaside lists.

Three SYSGEN parameters control reclamation:

- NPAG_AGGRESSIVE, the percentage of packets remaining on a list after aggressive reclamation, by default 50
- NPAG_GENTLE, the percentage of packets remaining on a list after gentle reclamation, by default 85
- NPAG_INTERVAL, the number of seconds between gentle reclamations, by default 30

Nonpaged pool reclamation is initiated by calling either EXE$RECLAIM_POOL_GENTLE or EXE$RECLAIM_POOL_AGGRESSIVE, in module MEMORYALC_POOL.

EXE$RECLAIM_POOL_GENTLE is called every NPAG_INTERVAL seconds as a repeating system timer routine (see Chapter *Time Support*). Because NPAG_GENTLE is a dynamic parameter, EXE$RECLAIM_POOL_GENTLE examines it each time it is entered. If NPAG_INTERVAL is negative, it resets the timer interval to 1 minute and simply returns. This enables the system manager to disable nonpaged pool reclamation temporarily and later reenable it by changing NPAG_INTERVAL to a positive number. Otherwise, it recalculates the next timer interval from the current value of NPAG_GENTLE and, if less than 1 second, modifies it to be 1 second.

Each time EXE$RECLAIM_POOL_GENTLE is entered, it reclaims pool from two lookaside lists in one LSTHDS structure. It divides the set of lists in the structure in half. Entered the first time, it reclaims pool from the first lookaside list in each half, namely, the listhead at array index 1 and the listhead at array index 1 plus the quotient of IOC_C_NUMLISTS and 2. The next time it is entered, it reclaims pool from the listheads at index 2 and index 2 plus the quotient of IOC_C_NUMLISTS and 2.

After processing all the lookaside lists in one LSTHDS, EXE$RECLAIM_POOL_GENTLE continues with lookaside lists in another LSTHDS if one exists:

- A NUMA system with multiple RADs may have multiple sections of nonpaged pool, each with its own LSTHDS.
- A system with separate BAP has a LSTHDS to describe it.

Every time EXE$RECLAIM_POOL_GENTLE processes all the lookaside lists in all the LSTHDSs, it calls EXE$KP_RECLAIM_KPB, in module KERNEL_PROCESS, to reclaim one KPB if there are at least two on the lookaside list. Chapter *Software Interrupts* describes KPBs and KPB reclamation.

EXE$RECLAIM_POOL_AGGRESSIVE is called by EXE$ALONPAGVAR_INT and EXE$ALLOCATE_POOL when either routine determines that there is insufficient space in a nonpaged pool variable-length list to satisfy a request. It reclaims pool from each nonempty lookaside list associated with that variable-length list. If the pool is not BAP, it also calls EXE$RECLAIM_KPB to reclaim a KPB.

EXE$RECLAIM_POOL_GENTLE and EXE$RECLAIM_POOL_AGGRESSIVE both call EXE$TRIM_POOL_LIST, in module MEMORYALC_POOL, to do the actual reclamation. EXE$TRIM_POOL_LIST is called with the packet size to be trimmed, the percent of packets to remain on the list, and the address of the LSTHDS containing the listheads.

EXE$TRIM_POOL_LIST takes the following steps:

1. It calculates the index of the lookaside listhead corresponding to the packet size.
2. It sets a time limit of one quarter the value of the SMP_SPINWAIT SYSGEN parameter. This sets an effective time limit for gentle reclamation of one half the value of SMP_SPINWAIT.
3. It acquires the POOL spinlock, raising IPL to IPL$_POOL, to synchronize access to the variable-length list.
4. It reads the counter array element corresponding to that listhead to determine how many packets are in the list and calculates how many should remain on the list after trimming.
5. It removes a packet from the list and calls EXE_STD$DEALLOCATE to return it to the variable-length list associated with that LSTHDS.

   It continues returning packets to pool until one of the following occurs:

   — It has trimmed the list to the desired percentage.

— The time limit has elapsed.

— There are no more packets on the list.

This unlikely circumstance occurs only if the counter contains a larger number of packets than are actually on the list. The counter is not necessarily accurate because it is possible for kernel mode code to remove a packet from the list without decrementing the counter and because the counts are not kept atomically.

6. It updates the counter array element, releases the POOL spinlock, and returns.

EXE$FLUSHLISTS is called by EXE$ALONPAGVAR_INT when both reclamation and pool expansion have failed to produce enough nonpaged pool to satisfy the current request. It performs much the same operations as the reclaim-pool routines and EXE$TRIM_POOL_LIST, with the following differences:

1. It removes all packets from lookaside lists whose packets are as large as or larger than the requested allocation size.
2. It sets an execution time limit of two thirds the value of the SMP_SPINWAIT SYSGEN parameter. After deallocating 100 packets, it checks whether the time limit has elapsed. If so, it returns a failure status. If not, it continues deallocating packets until there are no more, the time limit elapses, or a large enough piece of now-available pool has been agglomerated to satisfy the request. (Recall that EXE$DEALLOCATE maintains the variable-length list as an ordered list, agglomerating adjacent blocks as necessary to reduce fragmentation.)

   The intent of limiting the time spent flushing the lists is to minimize the possibility of spinwait timeouts when a routine holding a spinlock tries to allocate nonpaged pool.
3. After having reclaimed pool from all lookaside lists that are large enough without agglomerating a large enough packet, EXE$FLUSHLISTS calls EXE_STD$KP_RECLAIM_KPB to return available kernel process blocks. It then calls SCS_STD$URGENT_RECLAMATION, in module [CLUSTER]SCSFASTPATH, to try to reclaim BAP.

7.5.7 Expansion

Dynamic nonpaged pool expansion creates additional nonpaged pool as it is needed. At system initialization, SYSBOOT allocates space in the nonpaged system data huge page for the initial size of nonpaged pool and reserves enough contiguous virtual address space for nonpaged pool to expand up to NPAGEVIR pages. When an attempt to allocate nonpaged pool fails, the pool can be expanded by allocating more physical memory for it and altering the system page table accordingly. Note that expanded pool is not within the nonpaged system data huge page, although it is virtually contiguous to it.

When pool reclamation does not yield sufficient space to satisfy an allocation request, EXE$ALONPAGVAR_INT, EXE$ALONPAGED_ALN, and EXE$ALLOCATE_POOL call EXE$EXTENDPOOL, in module MEMORYALC_POOL, to attempt to expand pool. EXE$EXTENDPOOL calls EXE$EXTEND_NPP, also in module MEMORYALC_POOL, specifying that nonpaged pool should be expanded by four pages.

To synchronize allocation of physical memory and alteration of the system page table, EXE$EXTEND_NPP must acquire the MMG spinlock.

EXE$EXTEND_NPP first checks whether the current CPU already holds the MMG spinlock. If not, it checks whether it can acquire it: if it was entered from an interrupt service routine running above IPL$_SYNCH, the MMG spinlock's IPL, or is running on a CPU that owns any higher ranking spinlock, it cannot acquire the MMG spinlock. If either is true, it creates an IPL$_QUEUEAST fork process to expand nonpaged pool at some later time and returns an allocation failure status to its caller.

Whether running in the environment of the original allocation request or in the fork process, EXE$EXTEND_NPP then confirms that sufficient reserved virtual address is left for the requested expansion and calls EXE_STD$CHKFLUPAGES, in module MEMORYALC, to check that the physical pages can be allocated without reducing the number of physical pages available to processes below the minimum required. Pool expansion must leave sufficient available physical pages to accommodate the sum of the following:

- Space to inswap a reasonably large process, that is, the least of the following:
  - Maximum theoretically possible swap image SWP$GL_SWAP_IMAGE_SIZE_MAX (64K – 1)
  - Four times SYSGEN parameter SWPOUTPGCNT in pages
  - SYSGEN parameter WSMAX in pages
- The modified page list low limit (SYSGEN parameter MPW_LOLIMIT)
- The free page list low limit (SYSGEN parameter FREELIM)

If the memory sufficiency check fails, EXE$EXTEND_NPP attempts to broadcast a message to the operator's console, logs an expansion failure event (see Section 7.14), and returns the error status SS$_INSFMEM to its caller.

If the check succeeds, EXE$EXTEND_NPP calls MMG_STD$ALLOC_PFN_MAP_SYSTEM_VA, in module SYSVA_ALLOC, to allocate and map physical memory. It updates the page frame number (PFN) database for the allocated pages and places their PFNs in the next available system space level 3 page table entries (L3PTEs), beginning with the one corresponding to the address in MMG$L_NPAGNEXT. In each L3PTE it sets the valid, address space match, and modify bits.

If any expansion occurred, EXE$EXTEND_NPP acquires the POOL spinlock, calls EXE_STD$DEALLOCATE to add the new virtual pages to nonpaged pool, adds a descriptor for them to the nonpaged pool map, logs an expansion success event (see Section 7.14), and releases the POOL spinlock. If EXE$EXTEND_NPP acquired the MMG spinlock, it releases that spinlock also.

If any expansion occurred, it updates MMG$L_NPAGNEXT and reports that the resource RSN$_NPDYNMEM is available for any waiting kernel threads.

Nonpaged pool expansion provides a degree of dynamic system tuning. The penalty for undersizing NPAGEDYN is the increased overhead in allocating requests that cause expansion. An additional penalty is the performance loss associated with not having the expanded pages within a huge page (and thus its granularity hint region). Chapter 1 explains how huge pages improve system performance.

The penalties for oversizing NPAGEVIR are one quadword (the L3PTE) for each unused page and one associated unusable page of system virtual address space. If NPAGEVIR is too small, kernel threads may be placed into a resource wait state, waiting for nonpaged pool to become available.

Less dynamic than nonpaged pool expansion, the AUTOGEN facility can adjust SYSGEN parameters that govern the initial size of nonpaged pool according to a given system's workload, as outlined in Section 7.14.

Nonpaged pool expands, but it does not contract. No mechanism returns PFNs from nonpaged pool to the free page list. The nonpaged pool region returns to its original size only at the next bootstrap, if NPAGEDYN has not changed.

7.5.8 Synchronization

As described in Section 7.4.2, the load-locked/store-conditional mechanism synchronizes accesses to nonpaged pool lookaside lists.

The POOL spinlock serializes access to nonpaged pool variable-length lists. Acquiring the POOL spinlock raises IPL to IPL$_POOL. The allocation, deallocation, reclamation, and expansion routines for nonpaged pool acquire and release the POOL spinlock.

Device drivers running at fork level frequently allocate dynamic storage. The POOL spinlock ranks higher than all fork locks and the MAILBOX spinlock. This allows a CPU executing a driver fork process to acquire the POOL spinlock while owning the MAILBOX or any of the IOLOCK*x* fork locks. However, a CPU executing at device IPL may not acquire the POOL spinlock because device IPL is higher than IPL$_POOL.

Each nonpaged pool allocation routine that runs in process context, such as EXE[_STD]$ALLOCCEB or EXE[_STD]$ALLOCIRP, calls EXE$ALONONPAGED without acquiring the SCHED spinlock. If this attempt to allocate pool is successful, the routine has avoided the overhead of SCHED spinlock acquisition and release.

If EXE$ALONONPAGED fails to allocate the pool, the process context nonpaged pool allocation routine tests bit PCB$V_SSRWAIT in PCB$L_STS. If it is set, the routine returns a failure status to its caller. Otherwise, it acquires the SCHED spinlock, raising IPL to IPL$_SCHED and synchronizing access to the scheduling database, and calls EXE$ALONONPAGED again. If the second allocation attempt fails, the allocation routine calls a scheduling routine to place the kernel thread into a resource wait state, waiting for RSN$_NPDYNMEM.

A kernel thread in such a wait state will be made computable whenever RSN$_NPDYNMEM is declared available. In earlier versions of VAX VMS, the resource was declared available each time nonpaged pool was deallocated. Because resource waits occur less frequently than deallocations, OpenVMS Alpha reduces overhead by avoiding this declaration at deallocation. Instead, the resource is declared available once a second by EXE$TIMEOUT, in module TIMESCHDL (see Chapter *Time Support*). It is also declared available by EXE$EXTEND_NPP whenever nonpaged pool is expanded.

Code executing as the result of an interrupt at IPL$_SCHED or above typically deallocates nonpaged pool through routine COM[_STD]$DRVDEALMEM, in module MEMORYALC.

COM[_STD]$DRVDEALMEM deallocates a packet by simply calling EXE$DEANONPGDSIZ under any of the following circumstances:

- The packet is no larger than 8,192 (IOC_C_MAXLISTPKT) bytes. Such a packet is returned to the lookaside list. Access to the lookaside list is synchronized using special instructions.
- IPL is below IPL$_POOL.

If COM[_STD]$DRVDEALMEM is called from IPL$_POOL or above, however, it transforms the block that is to be deallocated into a fork block (see Chapter *Software Interrupts*) and requests an IPL$_QUEUEAST software interrupt. The code that executes as the IPL$_QUEUEAST fork process (the saved procedure value in the fork block) simply calls EXE$DEANONPAGED to deallocate the block. If the block is less than the size of a fork block, COM[_STD]$DRVDEALMEM generates the nonfatal bugcheck BADDALRQSZ.

By convention, process context code that allocates a nonpaged pool data structure executes at IPL 2 or above as long as the data structure's existence is recorded solely in a temporary process location, such as in a register or on the stack. Running at IPL 2 blocks AST delivery and prevents the possible loss of the pool if the process were to be deleted.

7.6 Per-RAD Pool

On a NUMA system, each RAD with physical memory may have its own section of nonpaged pool. The RADs' pool sections are virtually adjacent, but each is made up of physical memory local to its RAD. This enables code executing on a particular RAD to allocate nonpaged pool from memory with a faster access time.

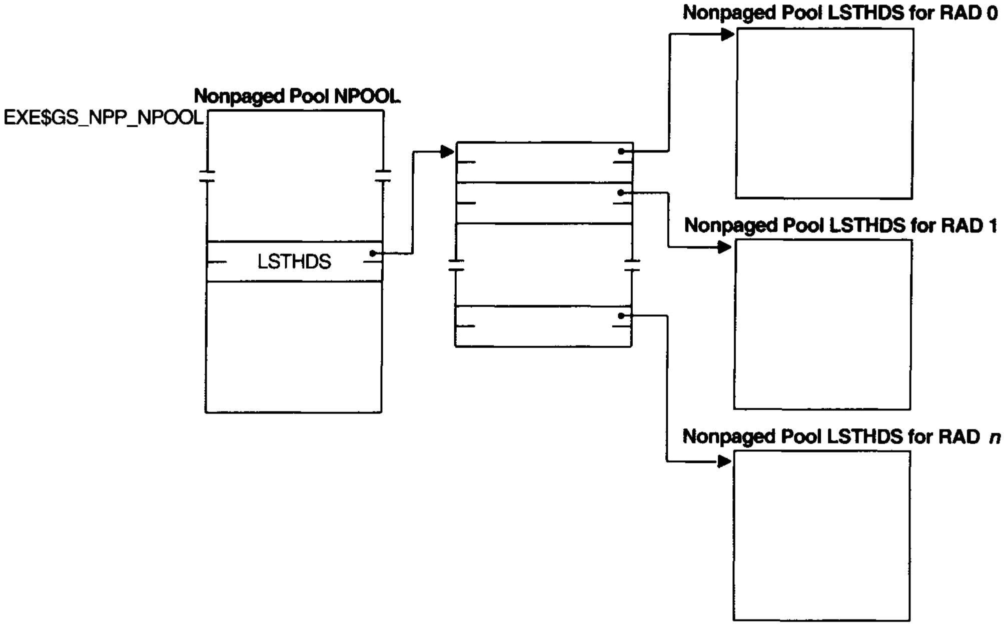
7.6.1 Data Structures

Each per-RAD pool section has its own variable-length and lookaside lists and thus its own LSTHDS structure. The LSTHDS structure is created in S2 space mapped to physical memory local to the RAD whose pool it describes. In each LSTHDS structure, LSTHDS$PS_NPOOL_DATA points to the nonpaged pool NPOOL structure.

As shown in Figure 7.11, the nonpaged pool NPOOL$AR_LSTHDS contains the address of an array of LSTHDS structure pointers, indexed by RAD. Each entry contains the address of the corresponding RAD's LSTHDS. The array is in the base RAD section of nonpaged pool.

NPOOL$L_MAX_LSTHDS contains the highest RAD number for which a nonpaged pool section has been created.

Figure 7.11 Per-RAD Nonpaged Pool Data Structures


Nonpaged Pool NPOOL
EXE$GS_NPP_NPOOL
LSTHDS
Nonpaged Pool LSTHDS for RAD 0
Nonpaged Pool LSTHDS for RAD 1
Nonpaged Pool LSTHDS for RAD *n*

Several fields in the nonpaged pool NPOOL support per-RAD pools. NPOOL$L_ON_RAD_DEALLOC records the number of times a piece of pool is deallocated from code running on the RAD associated with that pool. NPOOL$L_TOTAL_DEALLOC records the number of deallocations of per-RAD pool. These counts are kept only by the monitor version of SYSTEM_PRIMITIVES (see Section 7.13).

LSTHDS$L_RAD identifies the RAD of the physical memory that makes up this pool.

7.6.2 Initialization

During system initialization, INI$INITIALIZE_POOL, in module MEMORYALC_POOL, determines how many RADs have memory and the size of their initial nonpaged pool sections.

Each initial nonpaged pool section is at least one page large. If NPAGERAD is larger, the size of the initial section is NPAGERAD divided by the number of RADs with memory and rounded up to an integral number of pages.

For each RAD with memory, INI$INITIALIZE_POOL allocates that many physical pages from memory local to the RAD and maps them at the address in MMG$GL_NPAGNEXT, the next virtual addresses adjacent to existing nonpaged pool. It updates MMG$GL_NPAGNEXT. Initially, all the per-RAD pool is on the variable-length list. INI$INITIALIZE_POOL adds a descriptor to the nonpaged pool NPOOL pool map to describe this RAD's initial pool. If this is the monitor version of SYSTEM_PRIMITIVES, it allocates the three lookaside list statistics arrays from per-RAD pool.

It sets POOL$GL_USING_RAD_POOLS to 1 to indicate that at least one nonbase RAD pool exists.

7.6.3 Allocation

When per-RAD pool is enabled, by default all allocation is specific to the current RAD. This improves system performance by enabling all unmodified allocations to get memory from the RAD on which the allocator is executing. It is also possible to allocate nonpaged pool local to a specific RAD.

EXE$ALLOCATE_POOL calculates the address of the LSTHDS to be used:

- If EXE$ALLOCATE_POOL is called with no RAD specification and there is physical memory associated with the current RAD, it uses the LSTHDS for the current RAD.
- If there is no memory associated with the current RAD, it uses the LSTHDS for the RAD from which the system was booted.
- If a RAD was specified, it uses that RAD's LSTHDS.

If it cannot remove a packet from the appropriate per-RAD lookaside list to satisfy the request, it allocates per-RAD variable-length pool.

If enough per-RAD pool is not available, EXE$ALLOCATE_POOL expands that RAD's pool basically as described in Section 7.5.7. The major difference, however, is that it allocates physical memory associated with the specified RAD.

If, for some reason, the expansion fails and pool cannot be allocated from the specified RAD's pool, EXE$ALLOCATE_POOL attempts allocation from another RAD's pool. If necessary, it performs aggressive reclamation, flushes the lists, and tries to expand that RAD's pool. If none of those actions is successful, it goes on to another RAD. If it cannot allocate pool from any RAD's pool, it returns the error status SS$_INSFMEM to its caller.

7.6.4 Deallocation

When per-RAD pool is in use, EXE$DEALLOCATE_POOL must determine to which per-RAD pool a particular block should be deallocated. From the page table entry (PTE) that maps the beginning of the block, it extracts the PFN and then determines the RAD of that PFN. Depending on the size of the block, it deallocates the pool to the appropriate lookaside list or to the variable-length list of that per-RAD pool.

7.7 Bus-Addressable Pool

Bus-addressable pool is nonpageable pool mapped into physically contiguous physical memory whose address range is within I/O bus and 32-bit adapter physical addressing limits. A driver for a device or adapter that cannot access the maximum possible physical address range allocates BAP for structures and buffers to be accessed by the device or adapter.

On a particular system, if all devices and adapters can access all the physical memory, BAP is merged into nonpaged pool, and requests for BAP are satisfied from the nonpaged pool variable-length list and lookaside lists.

If a system has a Peripheral Component Interconnect (PCI) adapter and physical memory addresses above 1 GB, or an Extended Memory Interconnect (XMI) adapter and physical memory addresses above 4 GB, it needs separate BAP.

In general, BAP is handled in the same manner as nonpaged pool. The routines that allocate, deallocate, and reclaim are similar to those for nonpaged pool except that their flow varies with pool type so that they can be used for BAP, standard nonpaged pool, or some future type of pool. Section 7.7.1 describes the data structures related to BAP; Section 7.7.2, its initialization; and Section 7.7.3, the differences between allocating bus-addressable and nonpaged pool.

7.7.1 Data Structures

NPOOL$AR_LSTHDS contains zero; there is only one BAP LSTHDS.

NPOOL$Q_PER_POOL_DIAG points to a history buffer that records attempts to register requirements for BAP. Each entry contains the minimum and maximum physical addresses and the minimum and maximum pool requested.

NPOOL$L_POOL_FLAGS contains flags that describe BAP:

- NPOOL$V_NOT_NPP, when set, means the structure describes something other than standard nonpaged pool. The flag is set in the NPOOL data structure for BAP, whether or not the pool is within nonpaged pool.
- NPOOL$V_POOL_SEPARATE, when set, means that BAP is not within nonpaged pool. This bit is meaningful only for the BAP NPOOL$L_POOL_FLAGS field.
- NPOOL$V_POOL_WITHIN_NPP, when set, means that BAP is within nonpaged pool.
- NPOOL$V_MINIMUM_MODE, when set, means that system initialization completed without having initialized any BAP, either internal or external to standard nonpaged pool, but that a later call to EXE$REGISTER_POOL_INFO resulted in initializing BAP.

7.7.2 Initialization

SYSBOOT determines whether BAP is needed on this system by examining the following SYSGEN parameters:

- NPAG_BAP_MIN—minimum amount of BAP required
- NPAG_BAP_MAX—maximum amount of BAP required
- NPAG_BAP_MIN_PA—lowest physical address allowed within BAP
- NPAG_BAP_MAX_PA—highest physical address allowed within BAP

By default the first three of these parameters are 0, and NPAG_BAP_MAX_PA is –1. The parameters are altered indirectly through device drivers calling EXE$REGISTER_POOL_INFO, in module MEMORYALC_POOL, to specify their needs for BAP. EXE$REGISTER_POOL_INFO records their needs in cells read by AUTOGEN. The information recorded is cumulative, with the minimum and maximum physical addresses representing the lowest and highest addresses registered.

AUTOGEN transforms the cells' contents into values to be used for the SYSGEN parameters on a subsequent boot.

If the BAP SYSGEN parameters are set to their default values, SYSBOOT does not create BAP or enlarge nonpaged pool.

If the parameters are not default, SYSBOOT determines whether standard nonpaged pool could meet the requirements by comparing NPAG_BAP_MIN_PA and NPAG_BAP_MAX_PA to the range of physical addresses on the system. (Nonpaged pool expands using free physical pages of memory, so it could theoretically occupy any physical memory.) If NPAG_BAP_MIN_PA and NPAG_BAP_MAX_PA cover the entire range of physical memory present, SYSBOOT adjusts NPAGEDYN and NPAGEVIR by the amount of BAP needed: it adds the value of NPAG_BAP_MIN to NPAGEDYN and the value of NPAG_BAP_MAX to NPAGEVIR. Merging BAP with nonpaged pool makes the system more adaptable to pool requests.

If nonpaged pool cannot meet the requirements, SYSBOOT allocates a slice from the executive data huge page for use as BAP. It stores its address in EXE$GQ_BAP_VARIABLE. If the slice's physical pages do not meet BAP requirements, SYSBOOT resets the bus-addressable SYSGEN parameters to their default values and alters STARTUP_P1 and STARTUP_P3 so as to trigger AUTOGEN to run after system initialization. This unlikely case can occur when the physical memory configuration has been altered substantially, particularly on a Galaxy platform. After drivers have registered requirements for BAP, AUTOGEN reads the requirements, alters the SYSGEN parameters, and reboots the system.

INI$INITIALIZE_POOL, in module MEMORYALC_POOL, determines whether BAP is separate and initializes various data structures accordingly. In particular, it sets NPOOL$V_POOL_SEPARATE or NPOOL$V_POOL_WITHIN_NPP in NPOOL$L_POOL_FLAGS. If BAP is within nonpaged pool, the bus-addressable NPOOL and LSTHDS structures are not used.

If BAP is separate, its NPOOL and LSTHDS structures are used in allocating and deallocating BAP. INI$INITIALIZE also allocates statistics buffers for its lookaside lists from nonpaged pool.

After system initialization, if a driver registers BAP requirements that cannot be met by what, if anything, has been initialized, EXE$REGISTER_POOL_INFO tries to accommodate the request:

- If no BAP has been initialized and the requirements can be met by standard nonpaged pool, EXE$REGISTER_POOL_INFO initializes various structures to indicate that BAP is within nonpaged pool. It sets NPOOL$V_MINIMUM_MODE and NPOOL$V_POOL_WITHIN_NPP.

  If the requirements cannot be met by standard nonpaged pool, EXE$REGISTER_POOL_INFO calls EXE_STD$ALONONPAGED_LIM, in module MEMORYALC_DYN_64, to allocate nonpaged pool occupying pages of physical memory that the driver's device can access. If successful, EXE$REGISTER_POOL_INFO deallocates the pool to the bus-addressable variable-length list, fills in a pool map descriptor entry for it on the bus-addressable list, and removes an entry for it from the appropriate nonpaged pool map. It sets NPOOL$V_MINIMUM_MODE and NPOOL$V_POOL_SEPARATE.

- If BAP has been initialized, but the new requirements exceed the maximum pool size, EXE$REGISTER_POOL_INFO tries to expand BAP with nonpaged pool occupying suitable physical pages. BAP expansion is most likely to succeed during the early life of the system, before pool in physical pages that meet the address constraints is allocated to other uses. Because pool registration usually occurs early, BAP expansion is likely to succeed.

  If successful, EXE$REGISTER_POOL_INFO deallocates the pool to the bus-addressable variable-length list, removes it from the standard nonpaged pool map, and fills in a pool map descriptor entry for it on the bus-addressable list.

7.7.3 Allocation

A kernel mode component allocates BAP by calling EXE$ALLOCATE_POOL, in module MEMORYALC_POOL, passing it a pool type of MMG$K_POOLTYPE_BAP, the request size, and the alignment requirements.

When requested to allocate BAP, EXE$ALLOCATE_POOL takes the steps described in Section 7.5.4, with the following differences:

1. If BAP has not been initialized and is not within nonpaged pool, EXE$ALLOCATE_POOL returns SS$_BADPARAM to its caller. (The monitor version generates the fatal bugcheck BADALORQSZ.) This circumstance can result if SYSGEN parameters have their default values and no drivers that needed BAP registered their requirements in previous system boots.
2. If BAP has been initialized and is separate from nonpaged pool, pool will be allocated from a BAP lookaside list or the bus-addressable variable-length list.

3. If BAP was not created as a separate pool at system initialization, and the requested size and alignment cannot fit on a single page, EXE$ALLOCATE_POOL checks that the block allocated is physically contiguous. (If BAP was created as a separate pool, it is guaranteed to be physically contiguous.) If not, it inserts the block on a holding queue rather than deallocate it immediately and makes up to nine additional attempts to allocate a physically contiguous block. Before returning, it deallocates the blocks on the holding queue.
4. If the request cannot be allocated, EXE$ALLOCATE_POOL tries one after another of the following techniques, reattempting the pool allocation each time:
   — Aggressive pool reclamation from lookaside lists (see Section 7.5.6)
   — Flushing the lookaside lists (see Section 7.5.6)
   — Expanding traditional pool if BAP is within nonpaged pool (see Section 7.5.7)

     If BAP is separate from nonpaged pool, it expands BAP by allocating from nonpaged pool the request size rounded up to a page boundary. It tries to allocate pool occupying physical pages that meet the BAP address constraints. If the attempt fails, it expands nonpaged pool and tries again.

     If the attempt succeeds, it deallocates the pool to the BAP variable-length list, removes that pool segment from the nonpaged pool map, and adds it to the BAP pool map.

   — Calling callback routines specified by components that registered use of BAP to recover previously allocated pool and reflushing the lookaside lists

7.8 Lock Management Lookaside List

During system initialization, LCK$POOLZONE_INIT, in module LOCK_UTILS, creates a lookaside list in S2 space for use by lock management routines. In particular, it creates a POOLZONE_REGION data structure (see Figure 7.8), stores its address in LCK$AR_POOLZONE_REGION, and creates the S2 space pool zone.

Lock management routines allocate resource blocks (RSBs) and lock blocks (LKBs) (see Chapter *Lock Management*) from the pool zone's lookaside list.

LCK$POOLZONE_INIT determines the size of the lookaside list packets by taking the larger of RSB$K_LENGTH and LKB$K_LENGTH and rounding up to the next 32-byte boundary. In OpenVMS Alpha Version 7.3, the size is 256 bytes. It determines the initial number of pages in the zone by halving the lesser of the number of pages of physical memory available and the number of pages required to accommodate LOCKIDTBL and RESHASHTBL packets. The maximum number of pages in the zone is twice the number required for LOCKIDTBL_MAX packets.

Access to the lookaside list is synchronized with the LCKMGR spinlock. The lock management routines that allocate and deallocate RSBs and LCKs acquire this spinlock before calling EXE$POOL_ALLOCATE and EXE$POOL_DEALLOCATE.

Routine LCK$CHECK_POOLZONE, in module LOCK_UTILS, is responsible for reclamation from this pool zone. The zone is never shrunk below its initial page allocation. LCK$CHECK_POOLZONE is called by LCK$CHK_CACHES, which runs once a second.

LCK$CHECK_POOLZONE determines how many pages are currently in the region and how many have at least one available packet. It determines the smaller of the number of empty pages and the number of expansion pages.

1. If the pool zone has not expanded beyond its initial size, no pages are reclaimed. The routine returns.
2. If there are 10 or fewer empty/expansion pages and if, on average, the zone's pages have at least one available packet per page, LCK$CHECK_POOLZONE returns one empty page.
3. If there are between 10 and 100 empty/expansion pages, it returns one empty page.
4. If there are more than 100 empty/expansion pages, it returns 3 percent of the empty pages.

7.9 Extended File Cache Lookaside Lists

Extended File Cache (XFC) code creates three S2 space lookaside lists for its own use:

- Permanently allocated pool—the lesser of 4 MB and 1 percent of physical memory
- Dynamically allocated pool that can be reclaimed on demand
- Dynamically allocated pool that cannot be reclaimed on demand

The POOLZONE_REGION structures (see Figure 7.8) that describe these pools are within the structure Xfc$vabAnchor, defined in the SYS$XFCACHE[_MON].EXE executive image.

During system initialiation, routines XfcMemmgtPermanentAreaInit and XfcMemmgtDynamicAreaInit, in module [XFC]XFC_MEMMGT, create a pool zone in each pool for each of the following uses:

- Permanently allocated pool based in physical memory reserved through the Reserved Memory Registry entry named VCC$MIN_CACHE_SIZE
  - Barrier structures (BARs), currently unused
  - Secondary extent cache blocks (SECBs)
  - Primary extent cache blocks (PECBs)
  - Cache volume blocks (CVBs)
  - Cache file blocks (CFBs)
  - I/O statistics collection structures (IOSIZEs)

- Dynamically allocated and reclaimable pool
  - BARs
  - SECBs
  - PECBs
  - IOSIZEs
- Dynamically allocated and nonreclaimable pool
  - CVBs
  - CFBs

All the zones in these pools are created with zero initial pages.

7.10 Paged Pool

Paged dynamic memory, commonly known as paged pool, contains data structures that are used by multiple processes and that are not required to be permanently memory-resident. Its protection is ERKW, allowing it to be read from executive and kernel modes but written only from kernel mode.

During system initialization, SYSBOOT reserves system space for paged pool. The SYSGEN parameter PAGEDYN specifies the size of this area in bytes. By default paged pool is created as a set of demand zero pages. BOO$INIT_POOL, in module [SYSBOOT]SYSBOOT, places its starting address in both EXE$GL_PAGED and MMG$GL_PAGEDYN. System initialization code running in the context of the swapper process initializes the pool as one data structure encompassing the entire pool. That initialization incurs a page fault and thus requires process context.

If SYSGEN parameter POOLPAGING is set to zero, BOO$INIT_POOL instead creates paged pool as permanently allocated pages taken from the nonpaged system data huge page. A nonpageable paged pool facilitates debugging code whose data structures come from paged pool.

Process context kernel mode code calls the routine EXE[_STD]$ALOPAGED to allocate paged pool and the routine EXE[_STD]$DEAPAGED to deallocate paged pool. These routines, both in module MEMORYALC, call the lower level variable-length allocation and deallocation routines described in Section 7.3.

EXE[_STD]$DEAPAGED tests that the size being returned is nonzero and that the starting address is on a pool allocation granularity boundary. If either is false, it generates the nonfatal BADDALRQSZ bugcheck.

If an allocation request cannot be satisfied, EXE[_STD]$ALOPAGED returns to its caller with a failure status. The caller may return an error, for example, SS$_INSFMEM, to the user program, or the caller may place the kernel thread into a resource wait state, waiting for resource RSN$_PGDYNMEM.

Whenever paged pool is deallocated, EXE[_STD]$DEAPAGED calls SCH$RAVAIL, in module MUTEX, to declare the availability of paged pool for any waiting kernel thread. Chapter *Scheduling* describes resource waits.

Unused paged pool requires little system overhead: one L3PTE per page of pool and one corresponding reserved page of system virtual address space. Because paged pool is created as demand zero L3PTEs (see Chapter 2), it expands on demand through page faults.

Because this area is pageable, code that accesses it must run at IPL 2 or below while accessing it. Elevated IPL, therefore, cannot be used for synchronizing access to the paged pool list or to any data structures allocated from it. The EXE$GL_PGDYNMTX mutex serializes access to the paged pool list. Both EXE[_STD]$ALOPAGED and EXE[_STD]$DEAPAGED lock this mutex for write access.

By convention, process context code that allocates a paged pool data structure executes at IPL 2 as long as the data structure's existence is recorded solely in a temporary process location, such as in a register or on the stack. Running at IPL 2 blocks AST delivery and prevents the possible loss of the pool if the process were to be deleted.

The following data structures are located in paged pool:

- The shareable logical name tables and logical name blocks
- The Files-11 Extended QIO Processor (XQP) I/O buffer cache, which is used for data such as file headers, index file bitmap blocks, directory file data blocks, and quota file data blocks
- Global section descriptors, which are used when a global section is mapped or unmapped
- Mounted volume list entries, which associate a mounted volume name with its corresponding logical name and unit control block address
- Access control list elements, which specify what access to an object is allowed for different classes of users
- Object rights blocks that are accessed at IPL 2 and below
- Data structures required by the Install utility to describe known images

  Any image that is installed has a known file entry created to describe it. Some frequently accessed known images also have their image headers permanently resident in paged pool. These data structures are described in more detail in Chapter *Image Activation and Exit.*
- PQBs, which are temporarily used during process creation to store the quotas and limits of the new process

  PQBs, initially allocated from paged pool, are not deallocated back to the paged pool list. Instead, they are queued to a lookaside list, the self-relative queue at global label EXE$GQ_PQBIQ. Process creation code attempts to allocate a PQB by removing an element from this queue as a faster alternative to general paged pool allocation.

7.11 Process Allocation Region

The process allocation region contains variable-length data structures that are used only by a single process and are not required to be permanently memory-resident. (Process allocation region pages are pageable.) Its protection is UREW, allowing executive and kernel modes to write it and any access mode to read it.

The process allocation region consists of a P1 space variable-length pool and may include a P0 space variable-length pool as well. The P0 space allocation pool is useful only for image-specific data structures that do not need to survive image exit. The P1 space pool can be used for both image-specific data structures and data structures that must survive the rundown of an image, such as logical name tables.

During process startup EXE$PROCSTRT reserves P1 address space for the process allocation region. The SYSGEN parameter CTLPAGES specifies the number of pagelets in the P1 pool. Free space in the P1 process allocation region is maintained in a singly linked, memory-ordered list, as described in Section 7.3. EXE$PROCSTRT initializes the pool and its listhead, CTL$GQ_ALLOCREG. There is no global pointer that locates the beginning of the process allocation region.

Executive or kernel mode code running in process context calls EXE[_STD]$ALOP1PROC, EXE[_STD]$ALOP1IMAG, or EXE[_STD]$ALOP0IMAG to allocate space from the process allocation region, and EXE[_STD]$DEAP1 to deallocate a data structure to the region. These routines are in module MEMORYALC. When the data structure must be allocated from the P1 pool, EXE[_STD]$ALOP1PROC is used. When the data structure is image-specific, EXE[_STD]$ALOP1IMAG or EXE[_STD]$ALOP0IMAG is used.

EXE[_STD]$ALOP1IMAG and EXE[_STD]$ALOP0IMAG differ in which region they first attempt the allocation. EXE[_STD]$ALOP1IMAG tries the P1 region first, whereas EXE[_STD]$ALOP0IMAG tries the P0 region first. If EXE[_STD]$ALOP1IMAG finds that there is insufficient space, or EXE[_STD]$ALOP0IMAG finds that allocation in the P0 region is disallowed, each attempts to allocate from the other region. Neither routine can allocate from P1 space if the P1 process allocation region reaches a threshold of use specified by the SYSGEN parameter CTLIMGLIM. If the current image is one that was linked with the NOP0BUFS option, allocation from P0 space is prevented. If the allocation fails, these routines return the SS$_INSFMEM error status.

Additionally, EXE[_STD]ALOP1IMAG first checks whether a main image has been activated. If not, it branches to EXE$ALOP1PROC to avoid allocating any P0 space that might later be necessary for image activation.

The CTLIMGLIM limit does not apply to EXE[_STD]$ALOP1PROC. The latter may allocate space until the P1 allocation region is exhausted. The arithmetic difference between CTLPAGES and CTLIMGLIM guarantees a minimum number of pagelets exclusively for EXE[_STD]$ALOP1PROC. It only allocates space from the P1 region. If an allocation fails, it returns the error status SS$_INSFMEM.

Free space in the P0 process allocation region is maintained in a singly linked, memory-ordered list, as described in Section 7.3. During compilation of the SHELL*xx*K module, where *xx* is the system page size of 8, 16, 32, or 64 KB, the P0 process allocation region listhead, CTL$GQ_P0ALLOC, is initialized to zero. The image rundown routine deletes P0 space and zeros the listhead.

If not prevented by the presence of the NOP0BUFS linker option, EXE[_STD]$ALOP1IMAG and EXE[_STD]$ALOP0IMAG create and expand the P0 process allocation region by calling the routine MMG$EXPREG, in module SYSCREDEL. This routine functions much like the Expand Program/Control Region ($EXPREG) system service. EXE[_STD]$ALOP1IMAG and EXE[_STD]$ALOP0IMAG expand the P0 region as needed to satisfy allocation requests, but always by at least one virtual page. Each time one of these routines expands the P0 region, it calls EXE$DEALLOCATE to link the new space into the free list.

The current image and other executive routines may also expand the P0 virtual address space for their own purposes. Depending on the sequence of these expansions, multiple P0 allocation region expansions can result in a noncontiguous P0 allocation region. Note that this contrasts with the paged, nonpaged, and P1 allocation pools, which are always virtually contiguous.

EXE[_STD]$ALOP1PROC, EXE[_STD]$ALOP1IMAG, and EXE[_STD]$ALOP0IMAG store the address of the appropriate listhead in a register and call EXE$ALLOCATE to perform the variable-length allocation described in Section 7.3.1. EXE$DEAP1 determines whether the block being deallocated is from the P0 or P1 space pool and calls EXE$DEALLOCATE with the address of the appropriate listhead.

For a single-threaded process, no special synchronization mechanism is needed for the process allocation region. However, the allocation and deallocation routines change to kernel mode and execute at IPL 2, effectively blocking any other mainline or AST code from executing and perhaps attempting a simultaneous allocation from or deallocation to the process allocation region.

In the case of a process with multiple kernel threads, an additional mechanism is needed to synchronize mutiple kernel threads' allocations and deallocations. In addition to running at IPL 2, the allocation and deallocation routines lock CTL$GQ_POOL_MUTEX, a process-private mutex, for write access.

The following data structures are located in the process allocation region:

- The process-private logical name tables and logical name blocks
- Image control blocks, built by the image activator to describe what images have been activated in the process
- Rights database identifier blocks, which contain Record Management Services context (internal file and stream identifiers) for the rights database file
- A context block in which the Breakthrough ($BRKTHRU) system service maintains status information as the service asynchronously broadcasts messages to the terminals specified by the user

- Process scan context blocks, used by the Process Scan ($PROCESS_SCAN) system service, described in Chapter *Process Control and Communication*

There is enough room in the process allocation region for privileged application software to allocate process-specific data structures of reasonable size.

7.12 KRP Lookaside List

The KRP lookaside list is a P1 space list for process-private kernel mode data structures that are not required to be permanently memory-resident. The list is a doubly linked absolute queue, whose listhead contains the addresses of the first and last blocks in the list. The protection on this storage area is URKW, allowing it to be read from any mode but modified only from kernel mode.

Address space for this list is defined at compilation time of the SHELL*xx*K module, which defines the fixed part of P1 space. Two global symbols, CTL$C_KRP_COUNT and CTL$C_KRP_SIZE, control the number of KRP packets created and the size of each packet. In OpenVMS Alpha Version 7.3, ten packets of 768 bytes each are created. Routine EXE$PROCSTRT, in module PROCSTRT, initializes the list, forming packets and inserting them into the list at CTL$GL_KRPFL and CTL$GL_KRPBL.

A KRP is used as pageable storage, local to a kernel mode subroutine. KRPs should be used only for temporary storage that is deallocated before the subroutine returns. The most common use of KRPs is to store an equivalence name returned from a logical name translation.

Allocation and deallocation to this list is through CALL_PAL INSQUEL and CALL_PAL REMQUEL PALcode instructions. There is no need for synchronization other than that provided by the PALcode operations. Because KRPs are used only for storage local to the execution of a procedure, a failure to allocate a KRP is very unexpected and indicates a serious error rather than a temporary resource shortage. Kernel mode code that is unsuccessful at allocating from this list thus generates the fatal bugcheck KRPEMPTY.

7.13 Alternative Versions of Modules and Images

Some executive modules and images have alternative versions. An alternative version might contain code used only for debugging, performance monitoring, or field testing. For example, module MEMORYALC_DYN is conditionally compiled to produce two object modules: MEMORYALC_DYN_MIN and MEMORYALC_DYN_MON. Various other modules in the [SYS] facility, including MEMORYALC_POOL, MEMORYALC, and MEMORYALC_DYN_64, are conditionally compiled in a similar manner to produce two versions, one with the _MIN suffix and the other with the _MON suffix. The object file with the _MON suffix contains additional debugging and performance-monitoring code that is not present in the _MIN version. The former version is often referred to as the monitor version. Sections 7.14 and 7.15 describe this additional code.

MEMORYALC_*_MON modules are linked into the executive image SYSTEM_PRIMITIVES.EXE, and MEMORYALC_*_MIN modules are linked into the executive image SYSTEM_PRIMITIVES_MIN.EXE. The values of SYSGEN parameters POOLCHECK and SYSTEM_CHECK determine which image is loaded. If either is nonzero, SYSBOOT loads SYSTEM_PRIMITIVES.EXE; otherwise, it loads SYSTEM_PRIMITIVES_MIN.EXE. After the monitor version is loaded, some, but not all, checking can be disabled by clearing the POOLCHECK parameter.

If both POOLCHECK and SYSTEM_CHECK are zero at system initialization, SYSTEM_PRIMITIVES_MIN.EXE is loaded. Although POOLCHECK is a dynamic parameter, if it is zero at system initialization, setting it nonzero later has no effect.

7.14 Collecting Pool Allocation Statistics

The executive requires adequate pool space to operate properly. Inadequate pool space can contribute to poor system performance and, in extreme cases, can cause the system to become totally unresponsive. The AUTOGEN facility has a feedback mechanism that, based on data gathered by various operating system components, can adjust SYSGEN parameter values to a given system's workload.

Many of the pool allocation and expansion routines described in this chapter record nonpaged and paged pool allocation and failure statistics. (An allocation request that results in a pool expansion is not classified as a failure; pool expansion is assumed to be a routine event.) From these statistics, AUTOGEN's feedback mechanism can calculate new values for the SYSGEN parameters that control the system's paged and nonpaged pool sizes. The statistics used by AUTOGEN are kept by both versions of the SYSTEM_PRIMITIVES executive image.

A variable-length list (paged or nonpaged) allocation fails when no sufficiently large free block is found and, in the case of nonpaged pool, the list cannot be expanded. The routines that detect the allocation failure keep a total of the number of pages that fail to be allocated. They collect three categories of statistics for paged pool and variable-length nonpaged pool:

- Total number of allocation attempts
- Number of allocation failures
- Total number of pages that could not be allocated

Table 7.5 lists the data collected and the routines responsible for updating the data cells. The program AGEN$FEEDBACK.EXE (part of the MANAGE facility) reads these data cells during the SAVPARAMS phase of AUTOGEN.COM. See the *OpenVMS System Manager's Manual* for a description of AUTOGEN's operational phases and instructions for running it.

Table 7.5 Paged and Nonpaged Pool Allocation Statistics

Statistic	Location	Maintained by
	Nonpaged Pool	
Number of successful expansions	PMS$GL_NPAGDYNEXPS	EXE$EXTEND_NPP
Number of expansion failures	PMS$GL_NPAGDYNEXPF	EXE$EXTEND_NPP
Number of allocation attempts	PMS$GL_NPAGDYNREQ	EXE$ALLOCATE_POOL EXE$ALONPAGVAR EXE$ALONONPAGED_ALN EXE$ALONONPAGED_LIM
Number of allocation failures	PMS$GL_NPAGDYNREQF	EXE$ALLOCATE_POOL EXE$ALONPAGVAR EXE$ALONONPAGED_ALN EXE$ALONONPAGED_LIM
Unused	PMS$GL_NPAGDYNF	n/a
Total number of pages that failed to be allocated	PMS$GL_NPAGDYNFPAGES	EXE$ALONPAGVAR EXE$ALONONPAGED_ALN EXE$ALONONPAGED_LIM
	Paged Pool	
Number of allocation attempts	PMS$GL_PAGDYNREQ	EXE$ALOPAGED
Number of allocation failures	PMS$GL_PAGDYNREQF	EXE$ALOPAGED
Number of 10-second intervals with allocation failures	PMS$GL_PAGDYNF	EXE$ALOPAGED
Total number of pages that failed to be allocated	PMS$GL_PAGDYNFPAGES	EXE$ALOPAGED

In addition to the data and routines listed in Table 7.5, the routines in the monitor version of SYSTEM_PRIMITIVES record information about nonpaged and bus-addressable pool lookaside list performance. LSTHDS$AR_LISTATTEMPTS, LSTHDS$AR_LISTFAILS, and LSTHDS$AR_LISTDEALLOCS each point to an array of 128 (IOC_C_NUMLISTS) longwords, a longword for each lookaside list. LSTHDS$AR_LISTATTEMPTS records attempts to allocate from each list, LSTHDS$AR_LISTFAILS records failures to allocate from each list, and LSTHDS$AR_LISTDEALLOCS records deallocations to each list.

If BAP is separate from nonpaged pool, there are separate statistics arrays for BAP lookaside lists. If per-RAD pool is in use, each section of pool has its own statistics arrays.

These routines also record how many bytes of variable-length pool have been allocated in LSTHDS$L_VARALLOCBYTES.

The pool zone allocation and deallocation routines record statistics in fields in POOLZONE and POOLZONE_PAGE structures. Section 7.4.3.2 describes those fields and identifies those kept only by the SYSTEM_PRIMITIVES image. This information is intended for use by the kernel mode component that created the pool zone; AUTOGEN cannot make use of it.

7.15 Detecting Pool Corruption

Certain pool misuses can lead to obscure problems if left unchecked. The operating system implements two mechanisms to help troubleshoot pool corruption problems:

- Pool poisoning occurs dynamically, as packets or blocks are allocated and deallocated, and can result in timely detection of fatal errors.
- Pool history facilitates troubleshooting of problems after a crash has occurred.

Both mechanisms are optional and enabled through SYSGEN parameter POOLCHECK or SYSGEN parameter SYSTEM_CHECK. They are not permanently enabled because of their effect on system performance.

7.15.1 Pool Poisoning

The pool poisoning mechanism can detect pool misuses such as

- Continued use of a block of pool after it is deallocated
- Use of uninitialized fields in a block of allocated pool
- Use of a block of pool that was not allocated

The mechanism applies to the variable-length pools (paged, nonpaged, and bus-addressable pool, and the process allocation region) and to the nonpaged pool, bus-addressable, and pool zone lookaside lists. It involves

- Filling deallocated pool with a unique pattern, called the FREE or "poison" pattern
- Checking that the poison pattern is intact in pool being allocated and generating the fatal bugcheck POOLCHECK if the pattern is not intact
- Filling allocated pool with a second pattern, called the ALLO pattern

This section describes the POOLCHECK SYSGEN parameter, explains the mechanism's workings, and lists some limits to its ability to detect corruption.

7.15.1.1 POOLCHECK Parameter

The dynamic SYSGEN parameter POOLCHECK consists of four eight-bit fields (see Table 7.6 and Figure 7.12). The fields, whose names begin with PCHECK$B_, are defined by the macro $POOLCHECKDEF. The bits in the PCHECK$B_FLAGS byte enable and disable pool filling and most checking, and specify which pools are affected. The rest of this section describes the individual bits. The PCHECK$B_FREE and PCHECK$B_ALLO bytes specify the patterns written into pool when the space is deallocated and allocated. The PCHECK$B_SIZE_TO_CHECK byte controls block or packet size checking at deallocation.

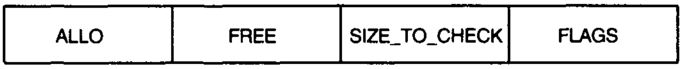
The default value of POOLCHECK is zero. Note that its value should be changed only for a specific purpose, such as debugging a device driver; there is a severe performance penalty when this parameter is nonzero.

Although POOLCHECK is dynamic, in order for the monitor version of SYSTEM_PRIMITIVES to be loaded, either POOLCHECK or SYSTEM_CHECK must be nonzero at system initialization.

Table 7.6 POOLCHECK Parameter FLAGS Bits

Bit	Name	Meaning if Set
0	POISON	Fill with FREE pattern on deallocation
1	CHECK	On allocation, check for FREE pattern and fill with ALLO pattern; enable pool checking
2–5		Undefined
6	DEALLO_ SIZE	Check deallocation size against size allocated (unused)
7	P1	Perform pool-checking operations for process allocation region also

Figure 7.12 POOLCHECK Parameter



Bits in PCHECK$B_FLAGS put the mechanism into one of several states:

- Do not fill or check blocks
- Fill blocks only upon deallocation
- Fill blocks upon deallocation; check and fill blocks upon allocation

Bits 0 and 7 enable the filling of blocks during deallocation. Bit 0 enables the filling, with the PCHECK$B_FREE pattern, of blocks deallocated to the variable-length paged, nonpaged, and bus-addressable pools, and to the nonpaged pool, bus-addressable, and pool zone lookaside lists. Bits 0 and 7 together enable the filling of blocks deallocated to the process allocation region.

When set in combination with the other bits, bit 1 enables the checking and filling of blocks during allocation. If set with bit 0, it enables the checking and filling, with the PCHECK$B_ALLO pattern, of blocks allocated from the variable-length paged and nonpaged pools and from the nonpaged pool lookaside lists. If set with bit 7, it enables the checking and filling of blocks allocated from the process allocation region.

The PCHECK$B_SIZE_TO_CHECK field determines whether each block or packet size is checked when it is deallocated:

- If PCHECK$B_SIZE_TO_CHECK contains 0, no checking is done.
- If PCHECK$B_SIZE_TO_CHECK is 255, all sizes are checked.
- Any other value in PCHECK$B_SIZE_TO_CHECK identifies a specific size to be checked. The value is multiplied by 64 and compared to the size of the block being deallocated. If the two sizes are equal, the size is checked.

If POOLCHECK is zero but SYSTEM_CHECK is nonzero, pool filling and checking are done with the default allocation (aaaa) and deallocation patterns (dddd). All pools and lists are filled and checked.

To check the size, routine CHECK_DEALLOCATION_SIZE, in module MEMORYALC_POOL_MON, searches the history buffer for the most recent entry for a block or packet at this address. If one is found and the allocated size in it does not match the size to be deallocated, the system generates a fatal POOLCHECK bugcheck. Because there are circumstances in which the size deallocated is intentionally different from the size allocated, Hewlett-Packard Company recommends that you enable size checking only when you are looking for a specific problem.

7.15.1.2 Pool-Poisoning Routine

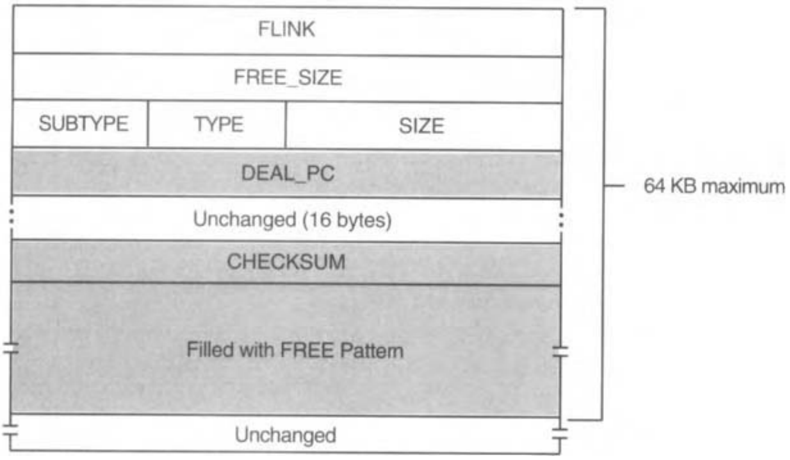
The routine POISON_PACKET, in module MEMORYALC, is called to fill pool space with a predictable pattern under several circumstances:

- Space is deallocated by EXE[_STD]$DEANONPAGED, EXE[_STD]$DEANONPGDSIZ, EXE[_STD]$DEALLOCATE, EXE$DEALLOCATE_POOL, in module MEMORYALC_DYN_64, or EXE$POOL_DEALLOCATE.
- A pool zone is expanded by EXE$POOLZONE_EXPAND, in module POOL_ZONES.
- A deallocated variable-length block is agglomerated with free blocks.
- Space is returned to variable-length pool by EXE[_STD]$ALLOCATE as a result of an inexact fit.
- Space is added to variable-length nonpaged pool as a result of pool expansion.

The macro $PFREEDEF defines offsets to a free block or packet of pool. Figure 7.13 shows the effects of pool poisoning on a free piece of pool. The shading in the figure indicates the fields modified by poisoning. The first nine longwords form a header, of which the first eight typically remain unchanged by pool poisoning:

- The first three longwords typically contain the forward pointer to the next free block; the size of the block, if it is a variable-length block; and the original size, type, and subtype fields.
- When POISON_PACKET is called by EXE[_STD]$DEANONPAGED and EXE[_STD]$DEANONPGDSIZ to poison a packet returning to a lookaside list, the fourth longword of the header contains the return address of the deallocation routine's caller. When POISON_PACKET is called by EXE[_STD]$DEALLOCATE, that is, for a variable-length block or a packet trimmed off a lookaside list, this longword contains stale data that is still potentially useful in crash dump analysis.
- The next two quadwords contain stale data that is unused by POISON_PACKET.

Figure 7.13 Format of Poisoned Pool Space



If enabled by the previously described bits, POISON_PACKET poisons deallocated pool as follows:

1. If its address is within paged pool or nonpaged pool, it checks that its ending address is within the pool upper boundary. It touches the beginning and end of the deallocated pool to catalyze any page fault that would lead to a later PGFIPLHI bugcheck or any access violation that would lead to a later crash.
2. It calculates a checksum by adding (ignoring any carry) the following:
   - — FREE pattern byte
   - — The deallocated block's address

- — Contents of the longword beginning at PFREE$W_SIZE
- — Contents of the longword at PFREE$L_DEAL_PC
- — Contents of the longword beginning at EXE$GQ_BOOTTIME + 1

It stores the checksum in the longword at offset PFREE$L_CHECKSUM of the block.

Under certain circumstances, it is possible for the contents of memory to be preserved from one bootstrap of the operating system to the next. The last longword used in calculating the checksum enables the checking routine to differentiate between stale poisoned pool and pool space poisoned during this bootstrap of the operating system.

3. It initializes the remainder of the space, up to a maximum of 64 KB, with the FREE pattern.

7.15.1.3 Pool-Checking Routine

The routine CHECK_PACKET, in module MEMORYALC, checks pool space. It is called by

- EXE[_STD]$ALLOCATE, when allocating variable-length pool space from paged pool, nonpaged pool, or the process allocation region
- EXE[_STD]$ALONONPAGED and EXE[_STD]$ALONONPAGED_ALN, when allocating a lookaside packet
- EXE$POOL_ALLOCATE, when allocating a packet from a pool zone lookaside list
- EXE$ALLOCATE_POOL, when allocating nonpaged or bus-addressable pool from a lookaside list
- EXE_STD$ALONONPAGED_LIM, when allocating a lookaside list packet whose physical address is below a caller-specified minimum

CHECK_PACKET calculates the expected checksum using the algorithm described in Section 7.15.1.2. If the expected checksum does not match that found in the PFREE$L_CHECKSUM longword, CHECK_PACKET assumes the block is unpoisoned and makes no further checks. (Since POOLCHECK is a dynamic SYSGEN parameter, it is possible that pool poisoning was disabled for a time, resulting in unpoisoned blocks on the free list. Alternatively, the block may have been poisoned during a previous bootstrap.)

If the checksum matches, CHECK_PACKET examines the remainder of the block for the FREE pattern. If the FREE pattern is not intact, it generates the fatal bugcheck POOLCHECK after pushing a reason code onto the stack. Table 7.7 summarizes these reason codes.

Table 7.7 POOLCHECK Bugcheck Reason Codes

Value	Meaning
0	Packet is corrupted
1, 2	Unused
3	Paged block extends outside of paged pool
4	Nonpaged block extends outside of nonpaged pool
5	P1 space allocation attempted at too high an IPL
6	Block could not be agglomerated
7	Deallocation and allocation were not the same size

If the FREE pattern is intact or if the checksum did not match, CHECK_PACKET fills the entire block (including the first nine longwords) with the ALLO pattern.

7.15.1.4 Constraints on the Pool-Checking Mechanism

Some circumstances can circumvent the pool-checking mechanism:

- Allocation and deallocation of lookaside list packets by any routine directly, rather than through the appropriate executive routines, bypass the filling and checking performed by the previously described routines.
- Any corruption of pool space that corrupts the third, fourth, or ninth (checksum) longword effectively disables checking for that block.
- Checking occurs only at allocation time. Corruption that occurs after a block is allocated is not detected.
- When a block being deallocated to variable-length pool is merged with a free block above or below it, the entire resulting free block is filled. This masks any corruption that may have previously occurred in an adjacent free block.
- The mechanism fills and checks a maximum of 65,500 bytes (64 KB less the nine-longword header).

Disabling and reenabling pool poisoning with the same FREE pattern can lead to false POOLCHECK bugchecks. If EXE$DEALLOCATE concatenates a variable-length block to the bottom of a poisoned free block while pool poisoning is disabled, only the top part of the resulting free block contains the FREE pattern. If pool checking is subsequently enabled with the same FREE pattern and this free block is allocated, CHECK_PACKET interprets it as being corrupt.

The book *Writing OpenVMS Alpha Device Drivers in C* provides detailed suggestions for using pool checking and for analyzing POOLCHECK bugchecks.

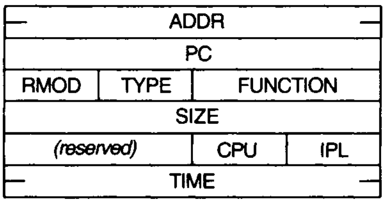
7.15.2 Pool History

The pool history mechanism records information about nonpageable pool allocations and deallocations in a nonpaged pool ring buffer. The information pertains to nonpaged pool, per-RAD nonpaged pool sections, bus-addressable pool, and pool zone lookaside lists. If the system crashes as a result of pool corruption, information about the most recent allocations and deallocations can be displayed using the SDA utility.

The pool history mechanism is enabled by bootstrapping the system with a nonzero value for either the SYSTEM_CHECK or POOLCHECK SYSGEN parameter. SYSTEM_PRIMITIVES.EXE contains the code described in this section.

During system initialization, the executive image's initialization routine, INI$INIT_MEMORYALC_DYN, in module MEMORYALC_DYN, allocates a block of nonpaged pool for a pool history ring buffer. The size of the buffer is determined by SYSGEN parameter NPAG_RING_SIZE, whose default value is 2,048 history buffers. It stores the address of this block in NPOOL$PS_RINGBUF and NPOOL$PS_NEXTNPH, and the number of entries in NPOOL$L_RINGBUFCNT.

Figure 7.14 Layout of Nonpaged Pool History Buffer Entry



Function Value	Meaning/Caller
0	Nonpaged pool lookaside list allocation
1	Nonpaged pool variable-length region allocation
2	EXE$DEANONPAGED nonpaged pool
3	EXE$DEANONPGDSIZ nonpaged pool
4	EXE$ALLOCATE_POOL nonpaged pool lookaside list
5	EXE$ALLOCATE_POOL aligned nonpaged pool
6	EXE$DEALLOCATE_POOL nonpaged pool
7	EXE$DEALLOCATE_POOL nonpaged pool of specified size
8	EXE$ALLOCATE_POOL bus-addressable lookaside list
9	EXE$ALLOCATE_POOL aligned bus-addressable pool
10	EXE$DEALLOCATE_POOL bus-addressable pool
11	EXE$DEALLOCATE_POOL bus-addressable pool of specifed size
12	EXE$POOL_ALLOCATE lookaside list
13	EXE$POOL_DEALLOCATE lookaside list
14	Failure to allocate bus-addressable pool
15	EXE$ALLOCATE_POOL nonpaged pool variable-length list
16	EXE$ALLOCATE_POOL bus-addressable variable-length list
17	EXE$ALONONPAGED_ALN nonpaged pool
18	Nonpaged pool expansion
19	Bus-addressable pool expansion

The layout of a pool history buffer entry is shown in Figure 7.14. The macro $NPHDEF (defined in module [LIB]NPOOL_DATA) defines the offsets to the fields in this structure.

Various pool allocation and deallocation routines call procedure UPDATE_RINGBUF, in module MEMORYALC_DYN_64, as part of their operation. UPDATE_RINGBUF maintains the nonpaged pool history ring buffer.

Each time it is called, UPDATE_RINGBUF updates NPOOL$PS_NEXTNPH to point to the next available history buffer. If all history buffers have been used, it initializes NPOOL$PS_NEXTNPH with the contents of IOC$AR_RINGBUF, the beginning of the history buffer area. UPDATE_RINGBUF synchronizes access to the ring buffer through a combination of raising to IPL 31 and acquiring a private spinlock.

UPDATE_RINGBUF records the following information in the history buffer:

- The return address of the caller of the allocation or deallocation routine
- Address of the packet or block being allocated or deallocated
- Size, type, and subtype of data structure being allocated or deallocated
- A value indicating the type of allocation or deallocation
- The ID of the CPU on which it is running
- The IPL at which it was entered
- The current time

The SDA command SHOW POOL/RING_BUFFER displays information stored in the history buffers. Note that this command cannot display useful information on a running system because of the dynamic nature of pool; it is used mainly in crash dump analysis.

7.16 Relevant Source Modules

Source modules described in this chapter include

[LIB]DYNDEF.SDL
[LIB]NPOOL_DATA.SDL
[LIB]PFREEDEF.SDL
[LIB]POOLCHECKDEF.SDL
[LIB]RIHDEF.SDL
[LIB_H]POOL_ZONES.H
[SYS]LDR_MEM_ALLOC.B64
[SYS]LDR_MEM_INIT.B64
[SYS]LOCK_UTILS.C
[SYS]LOOK_ASIDE_LIST.MAR
[SYS]MEMORYALC.MAR
[SYS]MEMORYALC_DYN.B32
[SYS]MEMORYALC_DYN_64.C
[SYS]MEMORYALC_POOL.C
[SYS]POOL_ZONES.C
[SYS]PROCESS_PAGE_DEFINITIONS.MAR
[SYS]SYSBOOT64.B64
[XFC]XFC_MEMMGT.C
[XFC]XFCDEF.H

Appendix A
Selected Acronyms

These acronyms are selected from those that appear in this book. This list is not exhaustive; for instance, acronyms for facilities, programs, and instructions are not included.

Acronym	Meaning
ACB	AST control block
ACL	access control list
ASN	address space number
AST	asynchronous system trap
ASTSR	AST summary register
BAP	bus-addressable pool
BAR	barrier structure
BOD	buffer object descriptor
CCB	channel control block
CEF	common event flag wait (scheduling state)
CFB	cache file block
COM	computable (scheduling state)
COMO	computable outswapped (scheduling state)
CRF	copy-on-reference
CVB	cache volume block
DCL	Digital command language
DIOBM	direct I/O buffer map
DTB	data stream translation buffer

Acronym	Meaning
FIFO	first-in/first-out
FP	frame pointer (register)
FRED	floating-point register and execution data structure
GB	gigabyte
GPT	global page table
GPTE	global page table entry
GPTX	global page table index
GSD	global section descriptor
GST	global section table
GSTE	global section table entry
GSTX	global section table index
HIB	hibernate wait (scheduling state)
HIBO	hibernate wait outswapped (scheduling state)
HWPCB	hardware privileged context block
HWRPB	hardware restart parameter block
ID	identification
IOSIZE	I/O statistics collection structure
IPL	interrupt priority level
IRP	I/O request packet
ITB	instruction stream translation buffer
JIB	job information block
KB	kilobyte
KPB	kernel process block
KRP	kernel request packet
KTB	kernel thread block
L1PT	level 1 page table
L1PTE	level 1 page table entry
L2PT	level 2 page table
L2PTE	level 2 page table entry
L3PT	level 3 page table
L3PTE	level 3 page table entry

Acronym	Meaning
LDRHP	loader huge page descriptor
LDRIMG	loader image data block
LEF	local event flag wait (scheduling state)
LKB	lock block
MB	megabyte, memory barrier
MPW IRP	modified page writer I/O request packet
NUMA	nonuniform memory access
ORB	object rights block
PALcode	privileged architecture library code
PB	path block
PC	program counter (register)
PCB	process control block
PCI	Peripheral Component Interconnect
PFL	page file control block
PFLMAP	page/swap file mapping window block
PFN	page frame number
PHD	process header
PID	process identifier
PMAP	PFN memory map
PMM	physical memory map
PQB	process quota block
PS	processor status (register)
PSECT	program section
PST	process section table
PSTE	process section table entry
PSTX	process section table index
PTBR	page table base register
PTE	page table entry
QBB	quad building block
RAD	resource affinity domain
RDE	region descriptor entry

Selected Acronyms

Acronym	Meaning
RMD	reserved memory descriptor
RSB	resource block
RVT	relative volume table
SB	system block
SBB	system building block
SCS	system communication services
SECB	secondary extent cache block
SMP	symmetric multiprocessing
SPT	system page table
SPTE	system page table entry
SYSPTBR	system page table register
TB	terabyte, translation buffer
TBCHK	translation buffer check (register)
TQE	timer queue entry
UAF	user authorization file
UCB	unit control block
VBN	virtual block number
VCB	volume control block
VIRBND	virtual address boundary (register)
VLM	very large memory
VPN	virtual page number
VPTB	virtual page table base (register)
WCB	window control block
WSLE	working set list entry
WSLX	working set list index
XFC	Extended File Cache
XMI	Extended Memory Interconnect
XQP	Extended QIO Processor

INDEX

This Page Intentionally Left Blank