Tru64 UNIX

NUMA Overview

Part Number: AA-NUMAG-DE

September 2002

Operating System and Version: Tru64 UNIX Version 5.1 or higher

This document introduces the operating system features that support Non-Uniform Memory Access (NUMA) and explains when and how they are used.

Hewlett-Packard Company
Palo Alto, California

© 2002 Hewlett-Packard Company

UNIX® and The Open Group™ are trademarks of The Open Group in the U.S and/or other countries. All other product names mentioned herein may be the trademarks of their respective companies.

Confidential computer software. Valid license from Compaq Computer Corporation, a wholly owned subsidiary of Hewlett-Packard Company, required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

None of Compaq, HP, or any of their subsidiaries shall be liable for technical or editorial errors or omissions contained herein. The information is provided "as is" without warranty of any kind and is subject to change without notice. The warranties for HP or Compaq products are set forth in the express limited warranty statements accompanying such products. Nothing herein should be construed as constituting an additional warranty.

Contents

Index

Examples

Figures

Tables

Preface

This is a post-release document that is currently available only on line, in HTML and PDF formats, at the HP Tru64 UNIX web site. This document is not orderable in printed form nor is it included on the Tru64 UNIX documentation CD-ROMs.

If you are using a web browser to read the HTML version of this document, you can click on documentation cross-references to display them. Some cross-references are to different locations in this document and some cross-references are to reference pages not included in this document. External references display in a different window from sections in this document. You can therefore navigate among sections of this document in one window and among external documents in the supplementary window.

Audience

This document is aimed at system administrators and programmers who will be using Tru64 UNIX Version 5.1 and Version 5.1A on GS80, GS160, and GS320 AlphaServer systems or Tru64 UNIX Version 5.1B on GS80, GS160, GS320, ES80, and GS1280 AlphaServer systems.

Conventions

This document uses the following conventions:

%	
$	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.
#	A number sign represents the superuser prompt.
file	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|]	
{ | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside

	brackets or braces indicate that you choose one item from among those listed.
⋮	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

1

NUMA Concepts

On traditional multiprocessor systems, there is one interconnect, either a bus or a switch, that links all system resources. This means that all CPUs in the system are subject to the same latency and bandwidth restrictions with respect to accessing the system's memory and I/O channels. Uniform Memory Access (UMA) is a term sometimes used to describe the system architecture in which all CPUs access memory and I/O by using the same bus or switch. This document refers to systems that use UMA architecture as traditional symmetric multiprocessor (SMP) systems.

The drawback of the architecture of traditional SMP systems is that scaling the system to large numbers of CPUs causes the system bus to become a performance bottleneck. One way to address this bottleneck is to design a system composed of SMP units (each with a limited number of CPUs, memory arrays, and I/O ports) that have access one another's resources through a high speed bus or switch. Non-Uniform Memory Access (NUMA) is the term used to describe this type of system architecture because it results in bandwidth and latency differences, depending on whether a particular CPU accesses memory and I/O resources in its own SMP unit or in one more distant.

Hardware Requirements for NUMA Support

The first NUMA implementations did not support cache coherency between the SMP building blocks, only within them. On these early NUMA implementations, software was responsible for ensuring cache coherency when a CPU accessed memory in any SMP unit other than the one in which the CPU was located.

All AlphaServer NUMA systems are cache-coherent NUMA (CC-NUMA) systems, for which system hardware handles cache coherency between the system's SMP units as well as within them. Because software is relieved of this responsibility, both the operating system and user applications can treat a CC-NUMA system the same way they treat a traditional SMP system and still be programmatically correct.

The GS80, GS160, and GS320 AlphaServer systems were the first Alpha implementations of CC-NUMA systems.

The ES80 and GS1280 AlphaServer systems are the current generation of cache-coherent NUMA systems and are supported by Tru64 UNIX Version 5.1B and higher versions.

Performance Implications of NUMA Support

Although software can treat a CC-NUMA system as a traditional SMP system and still be programmatically correct, obtaining optimal performance from a CC-NUMA system depends on appropriate use of its capabilities. In a network of systems, an application must sometimes run on a remote system rather than one local to the user. However, the application will almost always run faster on a local system that has all the resources needed by the application. This is because the connection between the systems increases response latency. The same principal applies when you consider a CC-NUMA multiprocessor system as a network of SMP units, each of which contains a set of CPUs, memory arrays, and I/O ports. The CPUs in one SMP unit can access memory that is available locally or remotely (in another SMP unit). However, using local memory is fastest; there is some performance penalty for accessing memory in another SMP unit.

Starting with Tru64 UNIX Version 5.1, the operating system includes kernel algorithms, utilities, and programming APIs that are NUMA aware. These algorithms and user interfaces are used to maximize the ratio of local to remote memory accesses and thereby help ensure optimal performance on CC-NUMA hardware.

In most of the software product documentation pertaining to NUMA support, references to a NUMA system assume a CC-NUMA hardware implementation. Therefore, this document also refers to CC-NUMA systems as NUMA systems.

1.1 RADs and Hardware Design

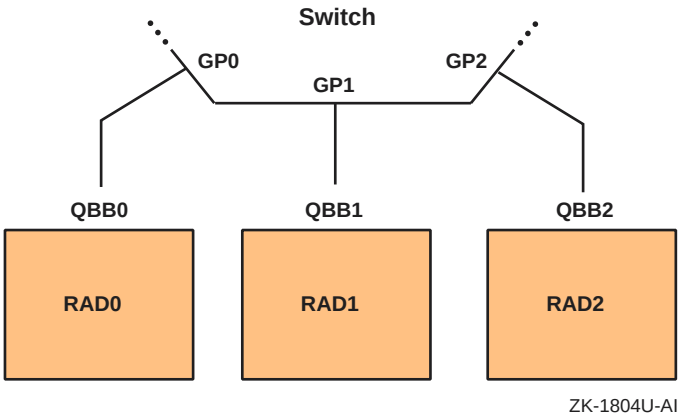
On Tru64 UNIX systems, the building blocks that make up a NUMA system are mapped to structures called Resource Affinity Domains (RADs). A RAD identifies the set of CPUs, memory arrays, and I/O busses that, when used together, allow the system to work most efficiently.

The concept of a resource affinity domain, like most abstract concepts, is easier learned in the context of a concrete example. For that reason, it is worthwhile to consider what a RAD corresponds to in current AlphaServer systems.

1.1.1 GS80, GS160, and GS320 AlphaServer Systems

On the GS80, GS160, and GS320 AlphaServer systems, each SMP unit is called a Quad Building Block (QBB or QUAD). Figure 1–1 shows how a RAD maps to a QBB.

Figure 1–1: RAD/QBB Mapping



Each QBB can contain up to four CPUs, a set of memory arrays, and an I/O processor (IOP) that, through two I/O risers, accommodates two to eight I/O busses. An internal switch in each QBB allows all CPUs equal access to both local memory and the I/O busses connected to the local I/O processor. An application running on a CPU in one QBB accesses the memory in another QBB by routing through the global port (GP) of the local QBB and the global port (GP) of the other QBB. On larger NUMA systems (GS160 and GS320), access is also routed through a Hierarchical Switch (sometimes called the HSwitch or Global Switch) that connects the global ports of all QBBs. Therefore, the remote/local response latency between QBBs is on the order of 2/1 or 3/1, depending on the type of system.

Although the operating system supports transparent resource access through the global ports and the HSwitch, performance is optimized when process operations use memory and I/O channels in the same QBB as the CPU where the process is running. Therefore, the CPUs, memory arrays, and I/O busses in the same QBB are viewed by the operating system as having affinity for one another and are included in the same RAD.

Starting with Tru64 UNIX Version 5.1, the operating system makes a best effort to:

- Schedule all threads of a multithreaded application on CPUs in the same RAD

- Allocate memory for each process or application thread in the same RAD as the CPU where the process or thread is running

The default NUMA-aware algorithms for scheduling and allocating resources to a process or thread work well when the resources in one RAD can accommodate the number of threads and the memory demands in any one application.

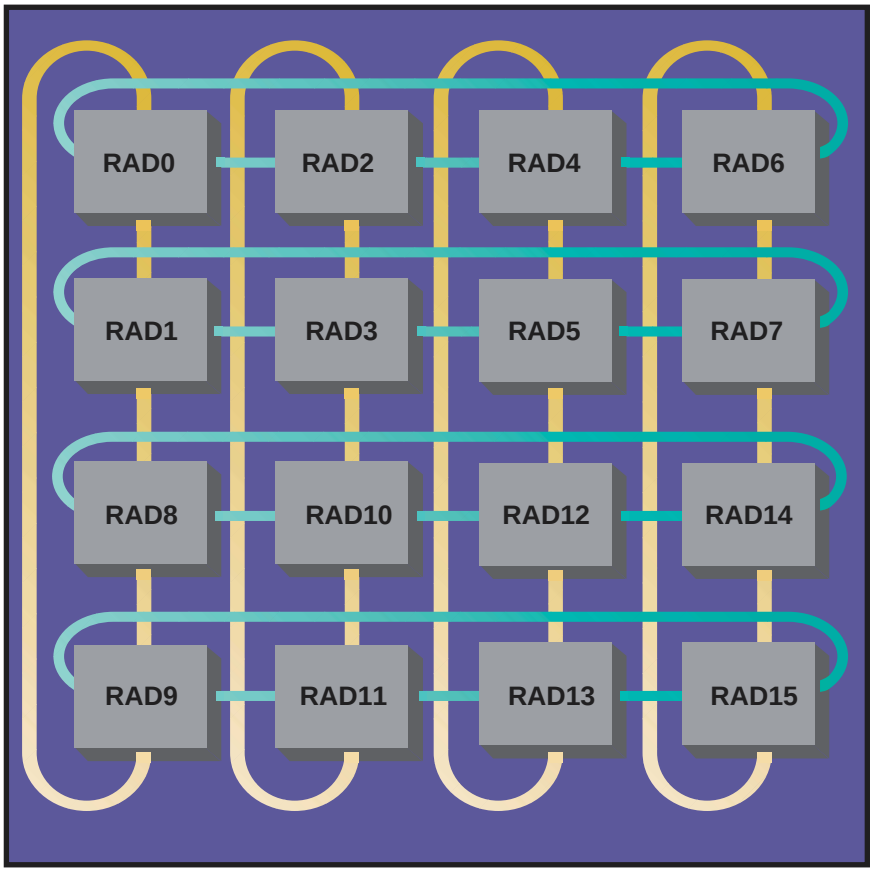
The NUMA application programming interfaces (APIs) allow applications to make scheduling and resource allocation decisions based on advance knowledge of the application's resource needs and behavior. Proper manipulation of system resources and process scheduling through NUMA APIs has the following potential advantages:

- A master application can distribute associated applications among available QBBs in a way that will ensure each the most likelihood of using CPU cycles, memory cache, and I/O channels of the same QBB.
- An application can notify the operating system of relationships between processes and threads that should be scheduled on the same RAD and, if migration to another RAD becomes advantageous, must be moved together.
- A very large and complex application whose resource demands and number of threads exceed the capacity of one QBB can stripe its CPU cycles, I/O load, and the memory that contains program data across QBBs.

1.1.2 ES80 and GS1280 AlphaServer Systems

On the GS1280 AlphaServer systems, each SMP unit is called a RAD. The ES80 and GS1280 AlphaServer platforms use a design wherein one- or two-processor RADs are connected to one another in a switchless mesh configuration, as shown by Figure 1–2.

Figure 1–2: RAD Mapping in a Switchless Mesh Configuration



ZK-1961U-AI

Note how the cables from RADs at the edges of the mesh wrap around to connect to RADs on the opposite side of the mesh. Although not shown in the figure, each RAD has its own memory and I/O channel.

In this NUMA implementation, the inter-RAD distance varies and can be defined in terms of how many "hops" it takes to get from a RAD in one location to a RAD in another location. For example, assuming that a process is running on a CPU in RAD0, access to additional resources is:

- One hop away if the resources are in RADs 1, 2, 6, and 9
- Two hops away if the resources are in RADs 3, 4, 7, 8, 11, and 15
- Three hops away if the resources are in RADs 5, 10, and 13

The advantage of the switchless mesh implementation is that the faster chip speed and extremely fast inter-RAD connections result in a negligible

increase in memory latency for each hop. More specifically, it would take about five hops before the memory latency penalty for remote memory access equals what it would be on a NUMA system using a hierarchical switch implementation, such as in a GS80, GS160, or GS320 AlphaServer. Therefore, the switchless mesh implementation, particularly when single-processor RADs are used, more closely emulates the behavior of a traditional SMP system but is one that permits modular growth. This makes a GS1280 AlphaServer system the best NUMA platform option for large, multithreaded applications that are written for traditional SMP systems but need more CPU resources than a traditional SMP system can provide.

1.1.3 Implications for Portability

It is important to emphasize two points about NUMA platforms:

- A RAD is a more generic concept than what it maps to in hardware.

  Applications can assign themselves to a particular RAD. In addition, system administrators can start applications on a RAD by using the `-r` option of the `runon` command. However, to be portable and maintainable, applications and scripts should not bind themselves to hardware topology. For example, applications and scripts should not assume that a RAD always contains a particular number of CPUs. In addition, applications and system administration scripts should never depend on the existence of a particular RAD identifier, such as 0, 1, or 2.

- In most cases, it is not necessary for programmers to rewrite existing applications or for system administrators to assign applications to specific RADs to obtain good performance on NUMA platforms running a mix of applications. Use of NUMA APIs and RAD-specific scheduling by a system administrator are recommended only for specific cases.

A RAD is used by software to identify and use optimal combinations of run-time resource combinations independently of hardware platform differences. Therefore, RADs support application portability among different NUMA implementations.

This application portability also applies to traditional SMP systems, such as those in the ES and DS families of AlphaServer systems. Starting with Tru64 UNIX Version 5.1, NUMA-aware applications can run on these SMP systems by handling them as single-RAD systems. On single-RAD systems, the only RAD that exists is RAD0, which contains all the CPUs, memory arrays, and I/O channels in the system. There is no performance advantage to running NUMA-aware applications on a single-RAD system (all resources are treated as being equidistant from one another). However, application portability is preserved as long as the application is designed to:

- Query the configuration to get information about available RADs

- Use only the RADs that are currently available

The NUMA structures and functions discussed in Chapter 2 are analogous to those used by the operating system software. Although they can be used in any program, they are recommended for use only in specialized layered software, such as databases, transaction processing products, or high performance technical computing applications, for which dedicated use and control of system resources are appropriate. For system administrators, there are also tuning parameters that can be adjusted to customize memory allocation on a per-RAD basis. However, even RAD-specific tunable parameters are best left to be automatically set by the operating system (or, if reset, be the same value for all RADs) unless all user applications being run on the system are NUMA aware. Therefore, RAD-specific system tuning should be used only when the NUMA system (or one of its hardware partitions) is dedicated to running NUMA-aware applications. (See Section 1.2 for more information about partitioning.)

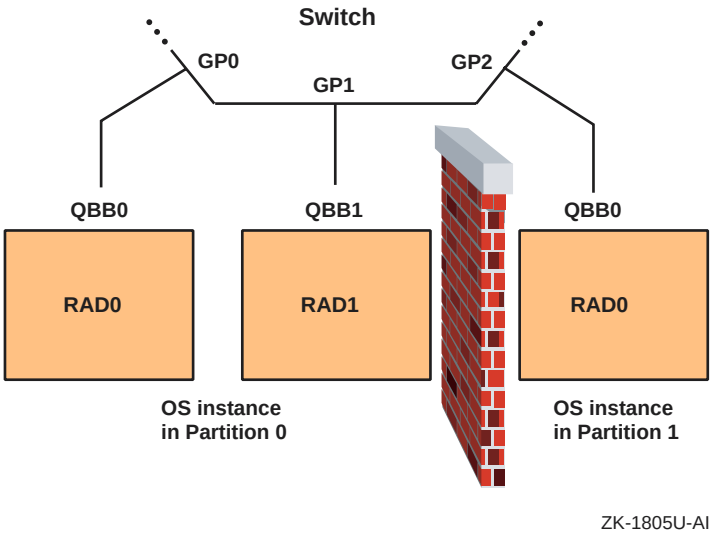
1.2 RADs and Partitioning

Partitioning refers to dividing a system into two or more resource groups, or partitions, each partition containing CPUs, memory, and I/O channels. After a system is partitioned, each partition is used independently of the others. Theoretically, partitioning can be done through hardware (hard partitioning) or operating system software (soft partitioning).

Hard partitioning is supported on all NUMA platforms and partitions all resources (CPUs, memory, and I/O channels). For hard partitioning, the operating system provides resource management features that do not cross partition boundaries.

For example, a hard partition on a GS160 or GS320 AlphaServer system can contain one or more QBBs and is set up through a hardware console facility. When a system is partitioned, an operating system is installed in each partition. Therefore, a GS320 with two partitions can be running two instances of the same version of Tru64 UNIX software, two different versions of Tru64 UNIX software, or two entirely different operating systems. As shown in Figure 1–3, each operating system instance is essentially firewalled from access to resources in any hard partition except the one in which it is running.

Figure 1–3: Partitioned NUMA System



The instance of Tru64 UNIX software that is running in Partition 0 can access two QBBs whereas the instance that is running in Partition 1 can access 1 QBB. In each partition, QBB numbering and RAD numbering starts at 0 and are unique only within the same partition. An operating system does not have access to information about RADs or their associated QBBs in any hard partition save the one in which it is installed. Even firmware upgrades must be installed independently on each hard partition.

This point is important when using NUMA user and programming interfaces provided by the operating system. When the operating system is installed in a partition, queries about the number of RADs or CPU slots available in the system return the number of RADs available in the partition.

Recognition that the system contains three QBBs occurs at the hardware level through the HSwitch, which recognizes each QBB through its unique global port identifier. System operators can access platform-wide information through an external System Management Console (SMC). Operating system instances that are installed on any of the partitions do not have access to the SMC.

Hardware partitioning is also supported on NUMA platforms based on the switchless mesh architecture but the implementation details are different. However, the end result is the same in that operating system software has no access to partitions other than the one in which it is installed.

User-Defined Processor Sets

A partial implementation of soft partitioning, called a user-defined processor set (pset), is supported on both SMP and NUMA platforms. A pset partitions CPUs but does not partition I/O channels. Psets are created and managed by using the `pset_*` commands or the comparable programming interfaces that are included in the operating system product.

On NUMA platforms, system administrators should factor in the RAD locations of CPUs when assigning them to psets. On a NUMA system, it is important to create a pset that contains processors in the same RAD or (if more processors are required than a RAD contains) in the fewest number of RADs that are needed to meet the resource requirements of the applications to be run on the processor set. On ES80 and GS1280 AlphaServer systems, which use the switchless mesh NUMA implementation, a processor set should include CPUs in the same RAD or in a set of RADs adjacent to one another. Creating a pset to contain CPUs that are as close as possible to one another optimizes performance on NUMA systems because it minimizes the performance penalty of accesses to remote memory. System administrators and programmers should apply this principle both to processor sets being defined for new applications and processor sets being defined for applications that previously ran on traditional SMP systems. See `pset_create`(1) for information about creating a pset and for cross-references to other pset-related reference pages.

1.3 RADs, Resource Allocation, and Process Scheduling

NUMA enhancements are available through command-line and programming interfaces.

1.3.1 NUMA Enhancements to System Utilities and Daemons

Starting with Tru64 UNIX Version 5.1, system administrators can use the `-r` option of the `runon` command to execute an application on a specific RAD. This and additional NUMA enhancements for resource monitoring and scheduling are described in Table 1–1.

Table 1–1: NUMA Enhancements to System Utilities and Daemons

Command	Description	Reference Page
`hwmgr view hier`	Displays the RAD location of CPUs and devices. Output from this command identifies the hardware construct that corresponds to a RAD.	`hwmgr_view`(8)

Table 1–1: NUMA Enhancements to System Utilities and Daemons (cont.)

Command	Description	Reference Page
`inetd -r`	Customizes the RAD locations on which to start Internet server child daemons. By default, one child daemon is started on each RAD.	`inetd`(8)
`netstat -R`	Displays the network routing tables for each RAD.	`netstat`(1)
`nfsd -t` and `nfsd -u`	Customizes the number of TCP and UCP server threads, respectively, that are spawned per RAD. This feature allows the NFS server to automatically scale the number of TCP and UCP server threads according to the size of the system.	`nfsd`(8)
`ps -o RAD`	Includes RAD binding in the information displayed about processes running on the system.	`ps`(1)
`ps -O NUMA`	Includes process ID, user, terminal, time, and command information along with processor, pset, RAD, and NSG information for running processes.	
`runon -r`	Executes an application on a specific RAD.	`runon`(1)
`sched_stat`	Displays process scheduling statistics on a per-RAD basis (available starting with Tru64 UNIX Version 5.1B).	`sched_stat`(8)
`vmstat -r`	Displays virtual memory statistics for a specific RAD.	`vmstat`(1)

Note

On Tru64 UNIX Version 5.1 and Version 5.1A systems, system administrators cannot determine the RAD location of a CPU through a command-line or graphical interface. (For programs, the `rad_get_cpus( )` function returns this information.) However, Appendix A contains the source code for a utility that queries the system for RAD and CPU identifiers. Sites can copy, adapt, and build this program for local use. Starting with Tru64

UNIX Version 5.1B, the `sched_stat` utility displays tables that reveal the RAD locations of system CPUs and the RAD distance (number of hops) that separates any two CPUs.

For the GS80, GS160, and GS320 AlphaServer systems, CPU identifiers and RAD identifiers have a fixed relationship, such that CPUs 0 to 3 are in RAD0, CPUs 4 to 7 are in RAD1, and so forth. Therefore, system administrators can assume for use in the `pset_assign_cpu` command that this fixed relationship of CPU numbers to RAD numbers is valid. However, the fixed relationship of these numbers does not apply to the ES80 and GS1280 AlphaServer systems. Therefore, users should not write scripts or programs that assume a fixed relationship of CPU numbers to RAD numbers if they want these scripts and programs to be portable to all types of NUMA AlphaServer systems. In addition, users should not write scripts that assume that the boot CPU always resides in RAD0. RAD0 is typically a default location but is not a required location for the boot CPU on NUMA platforms using current versions of firmware.

1.3.2 NUMA APIs

The NUMA APIs are used to:

- Identify and query the number of existing RADs and the availability of resources in these RADs
- Identify the RADs that are equal to or less than a specified distance (number of hops) from a particular resource
- Schedule processes and threads to run in RADs that offer the appropriate balance of available CPU cycles and memory for what the processes will be doing

See Section 2.2 for a summary of the library routines associated with NUMA-aware resource allocation and process scheduling.

The NUMA APIs are recommended for new versions of applications that currently create and manipulate psets if those applications will run on NUMA AlphaServer systems as well as traditional SMP systems. Existing applications that use the functions `create_pset( )`, `destroy_pset( )`, `assign_cpu_to_pset( )`, `assign_pid_to_pset( )`, and `print_pset_error( )` do not require changes to be able to run on NUMA AlphaServer systems. However, the manner in which existing programs assign CPUs to a processor set does not take into account the recommended practice of minimizing the distance of memory accesses on a NUMA system. If this distance is not as small as possible (given the amount

of local or nearby memory that is available), the application might not achieve optimal performance.

For this reason, new applications designed for use on NUMA systems or on both traditional SMP and NUMA systems should use NUMA APIs. Reference pages for these APIs are listed in Section 2.2.

2

NUMA-Aware Applications

Starting with Tru64 UNIX Version 5.1, Tru64 UNIX software is composed of NUMA-aware programs. Therefore, the majority of user applications do not have to use NUMA APIs to achieve reasonable performance on NUMA systems. However, certain user applications might be optimized through direct use of these APIs. This chapter describes the default NUMA-aware behavior in the operating system, and provides an overview of the NUMA APIs that applications can use directly.

2.1 Default NUMA-Aware Behavior of the Operating System

Starting with Tru64 UNIX Version 5.1, the following defaults are in place to increase the likelihood that NUMA system resources are used efficiently for most types of applications:

- The operating system defines a "home RAD" for each process and all its threads. Default process or thread scheduling and memory allocation are done on the assigned home RAD whenever possible.

  In other words, the operating system attempts to schedule a process and all its threads on CPUs in the home RAD. Furthermore, the operating system attempts to allocate memory for application and kernel data on the home RAD. The cache affinity algorithms previously available for traditional SMP systems are also used. Therefore, if a thread that previously ran on a particular CPU needs to be scheduled, the operating system attempts to schedule that thread on the same CPU.

  The operating system also defines a default overflow set of RADs. When there is insufficient free memory for application and kernel data on the home RAD, the operating system attempts to allocate memory from one or more remote RADs based on the default overflow set.

  Starting with Tru64 UNIX Version 5.1B and for NUMA platforms where RADs are not equidistant from one another in terms of response latency, the operating system first attempts to allocate the needed memory from a RAD that is one hop from the home RAD, then from a RAD that is two hops from the home RAD, and so on; this helps ensure that accesses to memory in remote RADs have minimal impact on performance.

- For data that is globally accessed, the operating system attempts to replicate the data in or stripe it across all RADs where it might be accessed. More specifically, the operating system attempts to:

  - Replicate kernel code and kernel read-only data on all RADs at boot time
  - Replicate other kinds of read-only data, such as shared program and library code, on all RADs where a running process or thread needs to access it

    If there is insufficient free memory on the RAD where the process or thread is running to replicate shared, read-only data, the operating system will utilize a copy on a remote RAD rather than wait for free memory on the local RAD to make the copy.
  - Stripe System V shared memory (which is not read-only) across all RADs

    Striping minimizes the likelihood that certain processes and threads always access System V shared memory locally while others always access it remotely.

- The operating system attempts to balance the load on each RAD so that local CPU cycles and local memory pages are both available to the processes running on the RAD.

  Local availability of memory and CPU cycles influences RAD selection at the time a process is created. The same factors might cause the operating system to migrate a process and associated memory pages from one RAD to another in response to changing resource requirements and access patterns.

2.2 NUMA APIs for User Applications

When a mix of applications with differing resource needs are run on the same system, it is best for user applications to rely on the default behavior of operating system software. However, large and highly specialized user applications might realize additional performance advantages through direct use of NUMA APIs. For example:

- An application for which I/O requests are extremely large might realize significant performance advantages when the CPU cycles and memory pages associated with an I/O request are striped across all available RADs. This optimization strategy works only if the data being read from or written to disk is also striped across controllers that are attached to the I/O ports of different RADs. (If I/O ports on different RADs channel data into the same RAID controllers, device latency will likely offset the bandwidth increase for CPU cycles and memory.)
- Applications with many subprocesses or threads that operate on large but different subsets of the same data might benefit from explicit resource management. In this case, NUMA APIs can help to increase the

ratio of local to remote accesses by changing the default algorithms for replicating or striping program data and System V shared memory.

NUMA APIs are included in the following libraries:

- The NUMA Library (`libnuma`)
- The Standard C Library (`libc`)

  Certain routines required for NUMA-aware programming are included in the `libc` library because they perform operations that are also useful in more generic types of programs.

- The POSIX Threads Library (`libpthread`)

  NUMA routines that are useful only in multithreaded programs are included in the `libpthread` library.

The NUMA data types, structures, and function prototypes are defined by including the `numa.h` header file. These APIs introduce three new constructs:

- RAD set

  A RAD set is a mechanism for passing information about RADs between an application and the operating system or between two applications. For example, an application can use a RAD set to query the operating system about the number of existing RADs. An application can also specify a RAD set to pass information about the number of RADs needed to meet application resource requirements.

  In traditional applications, the identifiers for system components are typically returned as a bit mask that is stored in a word or longword buffer. A fixed-length buffer limits the number of components that can be identified to the number of bits in the buffer (32 or 64 for a word or longword, respectively). However, a RAD set is represented by an opaque data type so that applications do not include a fixed-length buffer for querying or passing information about RADs.

- CPU set

  A CPU set is also a mechanism for passing information about CPUs between an application and the operating system or between two applications. Like a RAD set, a CPU set is represented by an opaque data type.

  A CPU set is different from the processor set (pset) that is created and manipulated by the APIs and commands already in use on traditional SMP systems. A pset reserves specific CPUs for use only by user-specified applications, while a CPU set is simply an information-passing mechanism.

  A NUMA-aware application that requests allocation of system resources uses RAD sets and CPU sets to ensure that CPUs and memory are

evaluated and used in the context of the RADs in which these resources are located. If the application intends to isolate some number of CPUs on the system for exclusive use by one or more key processes, the application first queries the number of RADs on the system and the RAD locations of the available CPUs. In almost all cases, the CPUs selected for a processor set should be from the same RAD or, if any one RAD has an insufficient number of CPUs for the expected workload, from a set of adjacent RADs. If the application runs on a traditional SMP system, all available CPUs are in a single RAD; however, the NUMA-aware logic for evaluating and using information about system processors remains the same.

- NUMA Scheduling Group (NSG)

  A NUMA Scheduling Group is the construct through which a NUMA-aware application ensures that a related set of processes and threads execute on the same RAD.

  An application can attach the identifiers of one RAD and of one or more processes to a NUMA Scheduling Group. By doing this, the application specifies that:

  – All those processes and any of their subprocesses or threads must execute on the same RAD

  – In the event that any of the processes or threads must be moved to a new RAD, all other processes and threads attached to the NUMA Scheduling Group are moved as well

See numa_types for a detailed description of the data types, structures, and macros used with the NUMA functions. See numa_scheduling_groups for a description of a NUMA scheduling group.

NUMA functions can be grouped into categories according to what is being queried or used. The tables referred to in the following list include the name, purpose, library, and reference page for each routine in the category. Some routines are duplicated in two tables because they query or create a relationship that spans two categories:

- RADs and RAD sets: Table 2–1
- CPUs and CPU sets: Table 2–2
- NUMA Scheduling Groups: Table 2–3
- Processes and threads: Table 2–4
- Memory management: Table 2–5

  See Section 2.3 for a summary of policies for NUMA memory management.

Table 2–1: RADs and RAD Sets

Function	Purpose	Library	Reference Page
`nloc( )`	Returns the set of RADs that is local to or a specified distance from a resource.	`libnuma`	nloc
`rad_attach_pid( )`	Attaches a process to a RAD (assigns a home RAD but allows execution on other RADs).	`libnuma`	rad_attach_pid
`rad_bind_pid( )`	Binds a process to a RAD (assigns a home RAD and restricts execution to the home RAD).	`libnuma`	rad_attach_pid
`rad_detach_pid( )`	Detaches a process from a RAD.	`libnuma`	rad_detach_pid
`rad_foreach( )`	Scans a RAD set for members and returns the first member found.	`libnuma`	rad_foreach
`rad_get_current_home( )`	Returns the caller's home RAD.	`libnuma`	rad_get_current_home
`rad_get_cpus( )`	Returns the set of CPUs that are in a RAD.	`libnuma`	rad_get_num
`rad_get_freemem( )`	Returns a snapshot of the free memory pages that are in a RAD.	`libnuma`	rad_get_num
`rad_get_info( )`	Returns information about a RAD, including its state (online or offline) and the number of CPUs and memory pages it contains.	`libnuma`	rad_get_num
`rad_get_max( )`	Returns the number of RADs in the system. [a]	`libnuma`	rad_get_num
`rad_get_num( )`	Returns the number of RAD's in the caller's partition. [a]	`libnuma`	rad_get_num
`rad_get_physmem( )`	Returns the number of memory pages assigned to a RAD.	`libnuma`	rad_get_num
`rad_get_state( )`	Reserved for future use. (Currently, RAD state is always set to ONLINE.)	`libnuma`	rad_get_num
`radaddset( )`	Adds a RAD to a RAD set.	`libnuma`	radsetops

Table 2–1: RADs and RAD Sets (cont.)

Function	Purpose	Library	Reference Page
`radandset( )`	Performs a logical AND operation on two RAD sets, storing the result in a RAD set.	`libnuma`	radsetops
`radcopyset( )`	Copies the contents of one RAD set to another RAD set.	`libnuma`	radsetops
`radcountset( )`	Returns the members of a RAD set.	`libnuma`	radsetops
`raddelset( )`	Removes a RAD from a RAD set.	`libnuma`	radsetops
`raddiffset( )`	Finds the logical difference between two RAD sets, storing the result in another RAD set.	`libnuma`	radsetops
`rademptyset( )`	Initializes a RAD set such that no RADs are included.	`libnuma`	radsetops
`radfillset( )`	Initializes a RAD set such that it includes all RADs.	`libnuma`	radsetops
`radisemptyset( )`	Tests whether a RAD set is empty.	`libnuma`	radsetops
`radismember( )`	Tests whether a RAD belongs to a given RAD set.	`libnuma`	radsetops
`radorset( )`	Performs a logical OR operation on two RAD sets, storing the result in another RAD set.	`libnuma`	radsetops
`radsetcreate( )`	Allocates a RAD set and sets it to empty.	`libnuma`	radsetops
`radsetdestroy( )`	Releases the memory allocated for a RAD set.	`libnuma`	radsetops
`radxorset( )`	Performs a logical XOR operation on two RAD sets, storing the result in another RAD set.	`libnuma`	radsetops

[a] On a partitioned system, the system and the partition are equivalent. In this case, the operating system returns information only for the partition in which it is installed.

Table 2–2: CPUs and CPU Sets

Function	Purpose	Library	Reference Page
`cpu_foreach( )`	Enumerates the members of a CPU set.	`libc`	cpu_foreach
`cpu_get_current( )`	Returns the identifier of the current CPU on which the calling process is running.	`libc`	cpu_get_current
`cpu_get_info( )`	Returns CPU information for the system. [a]	`libc`	cpu_get_info
`cpu_get_max( )`	Returns the number of CPU slots available in the caller's partition. [a]	`libc`	cpu_get_info
`cpu_get_num( )`	Returns the number of available CPUs.	`libc`	cpu_get_info
`cpu_get_rad( )`	Returns the RAD identifier for a CPU.	`libnuma`	cpu_get_rad
`cpuaddset( )`	Adds a CPU to a CPU set.	`libc`	cpusetops
`cpuandset( )`	Performs a logical AND operation on the contents of two CPU sets, storing the result in a third CPU set.	`libc`	cpusetops
`cpucopyset( )`	Copies the contents of one CPU set to another CPU set.	`libc`	cpusetops
`cpucountset( )`	Returns the number of CPUs in a CPU set.	`libc`	cpusetops
`cpudelset( )`	Deletes a CPU from a CPU set.	`libnuma`	cpusetops
`cpudiffset( )`	Finds the logical difference between two CPU sets, storing the result in a third CPU set.	`libnuma`	cpusetops
`cpuemptyset( )`	Initializes a CPU set such that it includes no CPUs.	`libnuma`	cpusetops
`cpufillset( )`	Initializes a CPU set such that it includes all CPUs.	`libnuma`	cpusetops
`cpuisemptyset( )`	Tests whether a CPU set is empty.	`libnuma`	cpusetops
`cpuismember( )`	Tests whether a CPU is a member of a particular CPU set.	`libnuma`	cpusetops
`cpuorset( )`	Performs a logical OR operation on the contents of two CPU sets, storing the result in a third CPU set.	`libnuma`	cpusetops

Table 2–2: CPUs and CPU Sets (cont.)

Function	Purpose	Library	Reference Page
`cpusetcreate( )`	Allocates a CPU set and sets it to empty.	`libnuma`	cpusetops
`cpusetdestroy( )`	Releases the memory allocated to a CPU set.	`libnuma`	cpusetops
`cpuxorset( )`	Performs a logical XOR operation on the contents of two CPU sets, storing the result in a third CPU set.	`libnuma`	cpusetops

[a] On a partitioned system, the system and the partition are equivalent. In this case, the operating system returns information only for the partition in which it is installed.

Table 2–3: NUMA Scheduling Groups

Function	Purpose	Library	Reference Page
`nsg_attach_pid( )`	Attaches a process to a NUMA scheduling group.	`libnuma`	nsg_attach_pid
`nsg_destroy( )`	Removes a NUMA scheduling group and deallocates its structures.	`libnuma`	nsg_destroy
`nsg_detach_pid( )`	Detaches a process from a NUMA scheduling group.	`libnuma`	nsg_attach_pid
`pthread_nsg_attach( )`	Attaches a thread to a NUMA scheduling group.	`libpthread`	pthread_nsg_attach
`pthread_nsg_detach( )`	Detaches a thread from a NUMA scheduling group.	`libpthread`	pthread_nsg_detach
`nsg_get( )`	Returns the status of a NUMA scheduling group.	`libnuma`	nsg_get
`nsg_get_nsgs( )`	Returns a list of NUMA scheduling groups that are active.	`libnuma`	nsg_get_nsgs
`nsg_get_pids( )`	Returns a list of processes attached to a NUMA scheduling group.	`libnuma`	nsg_get_pids
`nsg_init( )`	Looks up (and possibly creates) a NUMA scheduling group.	`libnuma`	nsg_init

Table 2–3: NUMA Scheduling Groups (cont.)

Function	Purpose	Library	Reference Page
nsg_set( )	Sets group ID, user ID, and permissions for a NUMA scheduling group.	libnuma	nsg_set
pthread_nsg_get( )	Returns a list of threads attached to a NUMA scheduling group.	libpthread	pthread_nsg_get

Table 2–4: Processes and Threads

Function	Purpose	Library	Reference Page
nfork( )	Creates a child process that is an exact copy of its parent process. See also the table entry for rad_fork( ).	libnuma	nfork
nmadvise( )	Tells the system what behavior to expect from a process with respect to referencing mapped files and shared memory regions.	libnuma	nmadvise
nsg_attach_pid( )	Attaches a process to a NUMA scheduling group.	libnuma	nsg_attach_pid
nsg_detach_pid( )	Detaches a process from a NUMA scheduling group.	libnuma	nsg_attach_pid
pthread_nsg_attach( )	Attaches a thread to a NUMA scheduling group.	libpthread	pthread_nsg_attach
pthread_nsg_detach( )	Detaches a thread from a NUMA scheduling group.	libpthread	pthread_nsg_detach
pthread_rad_attach( )	Attaches a thread to a RAD set.	libpthread	pthread_rad_attach
pthread_rad_bind( )	Attaches a thread to a RAD set and restricts its execution to the home RAD.	libpthread	pthread_rad_attach
pthread_rad_detach( )	Detaches a thread from a RAD set.	libpthread	pthread_rad_detach

Table 2–4: Processes and Threads (cont.)

Function	Purpose	Library	Reference Page
`rad_attach_pid( )`	Attaches a process to a RAD (assigns a home RAD but allows execution on other RADs).	`libnuma`	rad_attach_pid
`rad_bind_pid( )`	Binds a process to a RAD (assigns a home RAD and restricts execution to the home RAD).	`libnuma`	rad_attach_pid
`rad_detach_pid( )`	Detaches a process from a RAD.	`libnuma`	rad_detach_pid
`rad_fork( )`	Creates a child process on a RAD that optionally does not inherit the RAD assignment of its parent. See also the table entry for `nfork( )`.	`libnuma`	rad_fork

Table 2–5: Memory Management

Function	Purpose	Library	Reference Page
`memalloc_attr( )`	Returns the memory allocation policy for a RAD set specified by its virtual address.	`libnuma`	memalloc_attr
`nacreate( )`	Sets up an arena [a] for memory allocation for use with the `amalloc( )` function.	`libc`	`amalloc`(3)
`nmadvise( )`	Tells the system what behavior to expect from a process with respect to referencing mapped files and shared memory regions.	`libnuma`	nmadvise

Table 2–5: Memory Management (cont.)

Function	Purpose	Library	Reference Page
`nmmap( )`	Maps an open file (or anonymous memory) onto the address space for a process by using a specified memory allocation policy.	`libnuma`	nmmap
`nshmget( )`	Returns or creates the ID for a shared memory region.	`libnuma`	nshmget

[a] An arena is used in multithreaded programs when there is a need for thread-specific heap memory allocation.

2.3 NUMA Memory Management Policies

Starting with Tru64 UNIX Version 5.1, application programmers can choose among different policies to control memory allocation on NUMA systems. The following policies, which are defined in the `numa_types.h` file, can be specified for either a specific memory object or a kernel memory allocation request:

`MPOL_DIRECTED` Allocate memory from a specific RAD (directed allocation)

`MPOL_THREAD` Allocate memory from the current thread's home RAD (directed allocation that operates in the context of a multithreaded application)

`MPOL_STRIPED` Stripe application data across the memory in a specified RAD set

`MPOL_REPLICATED` Replicate application data in the memory of all RADs

These major policies can be refined by associated parameters. For the directed and thread-related memory allocation policies, an application programmer can specify an overflow set of RADs for use when sufficient resources are unavailable in the preferred RAD. For the striped memory allocation policy, a programmer can specify the number of pages for the stripe width (stride). To request that the operating system not migrate already allocated pages to another RAD, a programmer can combine the major policies with the `MPOL_NO_MIGRATE` policy.

When the NUMA memory allocation policy is not set by the application, the operating system applies the following defaults for different parts of an application's address space:

- Memory for private data, such as the heap and stacks for processes and threads, is allocated from the home RADs of the processes and threads.
- Program text and shared libraries are replicated in all RADs
- Shared data, such as System V shared memory, is striped (using a one-page stride) across all RADs

To override these defaults, an application programmer can use the following functions:

- The `nmmap( )` function to override the default policy for a new file object or to map a new range of addresses for a file that is already open
- The `nshmget( )` function to override the default policy for an already mapped address range of shared memory
- The `nmadvise( )` function to override the default policy used for process access to an already mapped range of address space for an open file or a region of shared memory

The `nmmap( )`, `nshmget( )`, and `nmadvise( )` functions include a parameter of type `memalloc_attr_t` to contain the NUMA memory allocation policy and associated attribute values. When this argument is null, the `nmmap( )`, `nshmget( )`, and `nmadvise( )` functions have the same behavior as their traditional counterparts (`mmap( )`, `shmget( )`, and `madvise( )`, respectively).

Note

A memory allocation policy request on any UNIX® platform (NUMA or traditional SMP) is not implemented in an absolute manner. UNIX architecture is designed for efficient sharing of resources among system and user processes rather than dedicated resource assignments, particularly where memory is concerned. This means that the operating system does not allow the policies requested by or for any one application to completely override the minimal memory requirements of other user and system processes that are running at the same time. Therefore, to ensure consistent implementation of the memory allocation policy requested through NUMA APIs, a NUMA-aware application should be run on a system (or system partition) that contains sufficient memory resources for both the application's processes and any other processes that will be running at the same time.

A

The radtool Program

This appendix contains the source code (Example A–1 and Example A–2) and the Makefile (Example A–3) for the `radtool` utility. This utility queries the system for identifiers of available RADs and for identifiers of CPUs in a specified RAD. The source code for this tool illustrates the use of several NUMA APIs that all NUMA-aware programs will need to use. In addition, the `radtool` utility can be built and installed on a customer system, then used by system administrators and site-specific scripts to avoid dependence on static assignments of RAD numbers and of CPU numbers within RADs.

The command-line synopsis for `radtool` is as follows:

path`/radtool [-x] | [ [-v] [-r | -c` *rad-id*`] ]`

Where:

`-x`	Displays the utility's usage message.
`-v`	Displays descriptive headings and comma separators for returned values.
`-r`	Returns identifiers of existing RADs. This is also the behavior when the command is entered without any options.
`-c` *`rad-id`*	Returns identifiers of available CPUs in the specified RAD.

The program header file for this example is a template and is included for possible site enhancements. (The header file does not supply definitions used in this version of the program but is referred to in the Makefile.) For example, the program might be internationalized to support message catalogs for translated messages and also include a header file that is created by the `mkcatdefs` command. In this case, the header file will be renamed `radtool_msg.h` and will define macros for default message strings. See `mkcatdefs`(1) for more information about creating message catalogs and a header file that centralizes maintenance of default message strings.

Example A–1: Source File for the radtool Program

```
/*
 * radtool.c -- NUMA API Example program
 *
 *
 */
#include <sys/types.h>
#include <sys/time.h>
#include <sys/siginfo.h>

#include <errno.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <numa.h>
#include <cpuset.h>
#include <radset.h>
#include "radtool.h"

/*
 * command-line options:
 *
 * -r         = display existing RADs
 * -c <radid> = display CPUs for specified RAD
 * -x         = display eXplanation.
 */
#define OPTIONS "c:rvx"

/*
 * command-line settable parameters and flags:
 */
bool show_rads    = false;   /* display existing RADs */
bool verbose      = false;   /* annotate output */
radid_t parm_rad = RAD_NONE; /* display CPUs for this RAD */

/*
 * usage/help message
 */
char *USAGE = "\nUsage:  %s {[-r] | [-c <radid>]} [-v] | [-x]\n\n\
Where:\n\
\t-r         = Display existing RADs. Same as no arguments.\n\
\t-c <radid> = Display CPUs for specified RAD.\n\
\t-v         = Include formatting text in display.\n\
\t-x         = Display this explanation.\n\
";
char *cmd;

bool  error = false;

/*
 * die() - Emit error message and exit w/ specified return code.
 *    If exit_code < 0, save current errno, and fetch associated
 *    error string. Print error string after app error message.
 *    Then exit with abs(exit_code).
 */
void
die(int exit_code, char *format, ... )
{
 va_list ap;
 char *errstr;
```

Example A–1: Source File for the radtool Program (cont.)

```
 int saverrno;

 va_start(ap, format);

 if (exit_code < 0) {
  saverrno = errno;
  errstr = strerror(errno);
 }

 (void) vfprintf(stderr, format, ap);
 va_end(ap);

 if (exit_code < 0)
  fprintf(stderr,"Error = (%d) %s\n", saverrno, errstr);

 exit(abs(exit_code));
}

void
usage(){
 fprintf(stderr, USAGE, cmd);
 exit(1);
}


/*
 * ============================================================================
 */
/*
 * rad_get_existing() -- return set of currently existing rads, using nloc()
 *
 * NOTE: It is the caller's responsibility to free the returned radset when it
 *       is no longer needed. See the call to radsetdestroy() at the end
 *       of the program.
 */
radset_t
rad_get_existing()
{
 radset_t allrads;
 numa_attr_t nat;

 if(radsetcreate(&allrads) == -1) {
  die(0-errno, "Unable to create radset for allrads\n");
 }

 /*
  * This works for Tru64 UNIX Version 5.1 or higher versions.
  * It returns the set of RADs that are <= RAD_DIST_REMOTE from an
  * empty RAD set. All existing RADs satisfy this relationship.
  */
 nat.nattr_type         = R_RAD;
 nat.nattr_descr.rd_radset = allrads;
 nat.nattr_distance     = RAD_DIST_REMOTE;
 nat.nattr_flags        = 0;

 if(nloc(&nat, allrads) == -1) {
  die(0-errno, "rad_get_existing: failure to get allrads\n");
 }

 return(allrads);
}
```

Example A–1: Source File for the radtool Program (cont.)

```
/*
 * radshowset():  display a RAD set
 *
 * The "note" parameter is for annotating the display with text
 * to indicate what the returned numbers represent.
 * If "note" is NULL, RAD numbers are printed in a single line,
 * separated by whitespace. The latter case is useful for returning
 * values to commands in a shell script, for example:
 *
 * for rad in ‘radtool -r‘; do whatever; done
 *
 */
void
radshowset(const radset_t set, const char *note)
{
 radid_t id;
 rad_cursor_t cursor = SET_CURSOR_INIT;
 int flags = 0;
 int i;

 if (note != NULL)
  printf("\n%s:\n", note);

 if (radisemptyset(set)) {
  if (note != NULL)
   fprintf(stderr, "\tNone");
  return;
 }

 for(i = 0;
   (id = rad_foreach(set, flags, &cursor)) != RAD_NONE;
  ++i) {
  if (note != NULL) {
   /*
    * "pretty print" - 8 to the bar
    */
   if((i % 8) == 0)
    printf("\n");
   else
     printf(", ");
  }
  printf("%3d", id);
 }
 printf("\n");
}
/*
 * cpushowset():  display the CPU set
 *
 * The "note" parameter is for annotating the display with text to
 * indicate what the returned numbers represent.
 * If "note" is NULL, CPU numbers are printed in a single line and
 * separated by whitespace. The latter case is useful for returning
 * values to commands in a shell script, for example:
 *
 * for cpu in ‘radtool -c 2‘; do whatever; done
 */
void
cpushowset(const cpuset_t set, const char *note)
{
```

Example A–1: Source File for the radtool Program (cont.)

```
 cpuid_t id;
 cpu_cursor_t cursor = SET_CURSOR_INIT;
 int flags = 0;
 int i;

 if (note != NULL)
  printf("\n%s:\n", note);

 if (cpuisemptyset(set)) {
  if (note != NULL)
   fprintf(stderr, "\tNone");
  return;
 }

 for(i = 0;
   (id = cpu_foreach(set, flags, &cursor)) != CPU_NONE;
  ++i) {
  if (note != NULL) {
   /*
    * "pretty print" - 8 to the bar
    */
   if((i % 8) == 0)
    printf("\n");
   else
     printf(", ");
  }
  printf("%3d", id);
 }
 printf("\n");
}


/*
 * ============================================================================
 */


void
main(int argc, char *argv[])
{
 extern int optind;
 extern char *optarg;
 char c;

 cmd = argv[0];

 /*
  * process command-line options.
  */
 while ((c = getopt(argc, argv, OPTIONS)) != (char)EOF ) {
  char *next;

  switch (c) {
  case 'c':
   parm_rad = strtoul(optarg, &next, 0);
   if (parm_rad < 0 || *next != '\0') {
    fprintf(stderr,
   "Error:  RAD identifier must be a positive integer\n");
    error = true;
   }
   break;
```

Example A–1: Source File for the radtool Program (cont.)

```
  case 'r':
   show_rads = true;
   break;

  case 'v':
   verbose = true;
   break;

  case 'x':
   usage();
   /* NOT REACHED */

  default:
   error = true;
   break;
  }
 }
done:

 if (error) {
  usage();
 }

 /*
  * If a number was specified, it must be the "-c" argument.
     * Display CPUs on the RAD with that number.
  */
 if (parm_rad != RAD_NONE) {
  cpuset_t cpus_in_rad;
  char note[32]; /* big enough for now */

  cpusetcreate(&cpus_in_rad);

  if (rad_get_cpus(parm_rad, cpus_in_rad) == -1)
   die(-2, "Unable to get CPUs in RAD %d\n", parm_rad);

  if (verbose)
   sprintf(note, "CPUs in RAD %d", parm_rad);

  cpushowset(cpus_in_rad, verbose ? note : NULL);

  cpusetdestroy(&cpus_in_rad);

  exit(0);

 } else {
  show_rads = true; /* show something! */
 }

 /*
  * Show all existing RADs, if requested.
  */
 if (show_rads) {
  radset_t allrads = rad_get_existing();

  radshowset(allrads, verbose ? "Existing RADs" : NULL);

  radsetdestroy(&allrads); /* be a good citizen */
 }
```

Example A–1: Source File for the radtool Program (cont.)

```
 exit(0);
}
```

Example A–2: Header File for the radtool Program

```
/*
 * radtool.h -- local header template for NUMA RAD tool program
 */

/*
 * a useful type definition -- for example:
 */
typedef enum {false=0, true} bool;
```

Example A–3: Makefile for the radtool Program

```
# Makefile template for NUMA sample programs
#
SHELL   = /bin/sh

MACH    =

CMODE = -std0
COPT = $(CMODE) -O2 #-non_shared
DEFS    =
INCLS   =
CFLAGS  = $(COPT) $(DEFS) $(INCLS) $(ECFLAGS)

CXX     = cxx
CXXMODE =
CXXFLAGS = $(CXXMODE) $(DEFS) $(INCLS) $(ECXXFLAGS)

# ASFLAGS for alpha assembler:
ASDEFS  = -DLANGUAGE_ASSEMBLY -D_LANGUAGE_ASSEMBLY -D__alpha
ASPIC =
ASCPP =
ASFLAGS = $(ASPIC) -tune generic $(ASCPP) $(ASDEFS) $(EASFLAGS)

LDOPTS = #-dnon_shared
LDLIBS = -lnuma
LDFLAGS = $(CMODE) $(LDOPTS) $(ELDFLAGS)

# export to environment as needed...
ROOTDIR = /
TMPDIR  = /var/tmp
COMP_HOST_ROOT =
COMP_TARGET_ROOT =

HDRS    = radtool.h

OBJS    = radtool.o
PROGS = radtool

#---------------------------------

all:    $(PROGS)
```

Example A–3: Makefile for the radtool Program (cont.)

```
radtool:  radtool.o
 $(CC) -o $@ $(LDFLAGS) $@.o $(LDLIBS)

$(OBJS):    $(HDRS)

install:
 @echo "Nothing to do..."

clean:
 -rm -f *.o core.[0-9]*

clobber: clean
 -rm -f  $(PROGS)
```

B

Reference Pages for NUMA APIs

NAME

numa_types – Data types used by NUMA application interfaces

SYNOPSIS

```
#include <sys/numa_types.h>
```

DESCRIPTION

This reference page lists and describes the data types, flags, structures, and unions that are defined in the <numa_types.h> header file to support the HP Tru64 UNIX NUMA APIs.

The program must call radsetcreate( ) or cpusetcreate( ) to allocate the RAD set or CPU set associated with any fields defined as type radset_t or cpuset_t, respectively.

Note that numa_types.h is indirectly included by the <numa.h> header file, which is the header file more frequently specified in the SYNOPSIS sections of reference pages for NUMA-related functions.

Definitions

The <numa_types> header defines the following data types, flags, structures, and unions, and associated symbolic values:

iopath_t

Reserved for future use.

memalloc_attr_t

A structure type that defines the policy and associated parameters for memory allocation. The memalloc_attr structures are associated with memory objects and with processes and threads. This structure contains the following members:

memalloc_policy_t mattr_policy

Specifies the memory allocation policy (as described in the entry for memalloc_policy_t). Remaining members of the structure contain parameters used to implement this policy.

`radid_t mattr_rad`

Specifies the primary or preferred RAD (region) from which to allocate memory for the `MPOL_DIRECTED` memory allocation policy.

`int mattr_distance`

Specifies the distance to overflow. This value is not currently used for any memory allocation policy.

`int mattr_stride`

Specifies the stride (in pages) for the `MPOL_STRIPED` memory allocation policy.

`int mattr_pagesz`

Specifies the page size in bytes. This value is not currently used for any memory allocation policy.

`radset_t mattr_radset`

If `mattr_policy` is `MPOL_DIRECTED` or `MPOL_THREAD`, specifies the overflow RAD set.

If `mattr_policy` is `MPOL_STRIPED`, specifies the RAD set across which memory is striped.

`memalloc_policy_t`

An enumeration type that determines, along with associated parameter attributes, how memory will be allocated for a memory object or a kernel memory allocation request. Valid policy values are:

`MPOL_DIRECTED`	Allocate pages from a specified (meaning preferred) RAD with overflow into a specified, possibly NULL, overflow RAD set.
`MPOL_THREAD`	Equivalent to `MPOL_DIRECTED` but having the thread context determine the preferred RAD from which pages are allocated. (In this case, the `mattr_rad` value is ignored.)

MPOL_STRIPED	Allocate pages so that they are striped across a specified RAD set by using a specified (page multiple) stripe as specified by the mattr_stride member of the memalloc_attr_t structure. Starting with a specified RAD, pages will be allocated from RADs in the RAD set in order of increasing RAD number. An mattr_stride number of pages will be allocated from each RAD before pages are allocated from the next RAD. After pages are allocated from the highest numbered RAD in the set, allocation will wrap, which means that pages are next allocated from the lowest numbered RAD in the set.
MPOL_REPLICATED	Replicate pages on the home RAD of the thread that caused the pages to be allocated.

Note

MPOL_REPLICATED is the default and only memory allocation policy supported for shared library text pages and program text pages. Furthermore, the operating system ignores MPOL_REPLICATED when it is specified for other types of memory (stack, heap, and shared data pages). Therefore, specifying MPOL_REPLICATED has no effect in the current implementation.

The following modifier may be combined (by using a logical OR operation) with the preceding policies:

MPOL_NO_MIGRATE	Disable automatic migration of pages by the system.

The following table indicates how different memory policies are supported for the different types of memory in the program address space:

SHPT = shared program text
SHLT = shared library text
SVSHM = System V shared memory
STACK = program or thread stack
DATA = initialized program data
BSS = uninitialized program data and heap
MPRIV = `[n]mmap(MAP_PRIVATE)`
MSHAR = `[n]mmap(MAP_SHARED)`
MANON = `[n]mmap(MAP_ANONYMOUS)`

	MPOL_DIRECTED	MPOL_THREAD	MPOL_STRIPED	MPOL_REPLI-CATED
SHPT	Ignored *	Ignored *	Ignored *	Default
SHLT	Ignored *	Ignored *	Ignored *	Default
SVSHM	As specified.	As specified but no migration. **	Default ***	=MPOL_DIRECTED
STACK	As specified.	Default	As specified.	=MPOL_DIRECTED
DATA	As specified.	Default	As specified	=MPOL_DIRECTED
BSS	As specified.	Default	As specified.	=MPOL_DIRECTED
MPRIV	As specified.	Default	As specified.	=MPOL_DIRECTED
MSHAR	As specified.	Default	As specified.	=MPOL_DIRECTED
MANON	As specified.	Default	As specified.	=MPOL_DIRECTED

* "Ignored" means only that the `MPOL_*` flag is ignored; the `nmadvise( )` call is still checked for errors and any applicable behaviors are performed.

** When thread-local memory allocation is specified for System V shared memory segments, the policy is used only for pages that have not yet been allocated, have been paged out, or have been discarded (by means of the `MADV_DONTNEED` flag on an `nmadvise( )` call).

Existing pages are not migrated according to the new memory allocation policy in order to avoid thrashing; in other words, the MADV_CURRENT flag on a nmadvise( ) call is treated as MADV_DONTNEED with respect to System V shared memory segments.

*** The default memory allocation policy for System V shared memory segments is configurable. If the shm_allocate_striped attribute of the IPC kernel subsystem is set to 1 (the default), SVSHM segments are striped. If this attribute is set to 0, SVSHM segments are allocated by using the thread local policy (MPOL_THREAD).

nmemalloc_range_t

Structures of this type are supported for internal use only.

nsgid_t

Identifier for a NUMA Scheduling Group (NSG).

numa_attr_t

A structure that describes the NUMA topology attributes. This structure is used with the nloc( ) function to query the NUMA system topology or with the nfork( ) function to specify the set of RADs from which a RAD will be selected for a new process. This structure contains the following members:

rsrctype_t nattr_type — The type of resource (as described in the entry for type rsrctype_t).

rsrcdescr_t nattr_descr — The resource descriptor, which identifies the instance of the resource specified by the nattr_type value. Specify this field as a union with the appropriate resource handle as described in the entry for rsrcdescr_t. For

	example, a resource descriptor for a RAD set might be specified as: `nat.nattr_descr.rd_radset` For this descriptor, `nat` is a structure of type `numa_attr_t`, `rd_radset` is the resource handle, and `nat.nattr_type` has been set to `R_RAD`.
`ulong nattr_distance`	The distance to the requested resource. The `nloc( )` function returns a set of RADs that are less than or equal this distance from the specified resource.
`ulong nattr_flags`	Symbolic values used by certain functions. The `nloc`(3) and `nfork`(3) reference pages list and describe these values.

`radid_t`

Identifier for a Resource Affinity Domain (RAD), which is a grouping of basic system resources that form a NUMA building block. NUMA platforms contain multiple RADs, and each RAD can contain zero or more CPUs, memory arrays, and I/O busses. Single-processor and multiprocessor platforms that do not use NUMA architecture are treated by the operating system as single-RAD systems.

The `radid_t` data type is a generic, integral type, for which there is the following symbolic value:

`RAD_NONE`

No valid RAD ID. Functions return `RAD_NONE` when no RAD matches the specified criteria, there are no more RADs in a RAD set, and so forth.

`radset_t`

A set of RADs. This type is used to specify a set of `radid_t` values to the NUMA APIs. A subset of these APIs perform operations on a set of RADs and manage `radset_t` as an opaque type.

`rad_cursor_t`

A opaque type used in an enumeration or an iteration operation on a RAD set. This type stores the current cursor position during a scan of the members in a RAD set. See `rad_foreach`(3) for more information.

`rad_info_t`

A structure returned by the `rad_get_info( )` function to describe the state and resources associated with a RAD. This structure contains the following members:

`int rinfo_version`	The RAD revision number.
`radid_t rinfo_radid`	The RAD identifier.
`rad_state_t rinfo_state`	The RAD state, whose values are listed in the description of the `rad_state_t` type.
`ssize_t rinfo_physmem`	The amount of physical memory present in the RAD.
`ssize_t rinfo_freemem`	The current amount of free memory available in the RAD.
`cpuset_t rinfo_cpus`	The set of CPUs associated with the RAD.

See `rad_get_info`(3) for more information about querying RAD information.

`rad_state_t`

A RAD's software state. The defined states are:

`RAD_ONLINE`	The specified RAD exists and is on line. Processes and threads may be assigned to and memory allocated to the RAD.
`RAD_OFFLINE`	The specified RAD exists but is not currently on line. No processes or threads may be assigned

to, nor memory allocated to, this RAD; however, its resource complement may be queried.

Note

RAD_ONLINE is the only state currently supported.

rsrctype_t

An enumeration type that specifies the kind of resource with which affinity is desired. Functions that perform NUMA topology queries and resource binding pass rsrctype_t arguments along with rsrcdescr_t descriptors to identify resources. The following symbolic values identify the type of resource being specified by a particular descriptor:

R_RAD — A RAD set.

R_FILDES — A file or device referenced by an open file descriptor.

R_PATH — A file or device specified by a pathname.

R_SHM — A System V shared memory segment that is referenced by a shared memory ID.

R_PID — A process that is referenced by a pid_t identifier.

R_MEM — A physical memory mapped region that is referenced by a process virtual address. This resource type is used to find RADs that are equal to or less than a specified distance from a particular memory location. See nloc(3).

R_NSG — A NUMA Scheduling Group that is referenced by an nsgid_t identifier.

rsrcdescr_t

A union of the various resource handles, or descriptors, that are specified by different resource type (rsrctype_t) values. Along with an rsrctype_t argument, an rsrcdescr_t handle is included on the nattr_descr argument to the NUMA APIs that perform NUMA

topology queries and resource binding. The resource handles for different resource types are as follows:

`radset_t rd_radset`	If the `rsrctype_t` value is `R_RAD`, a RAD set is the resource, for which `rd_radset` is the handle.
`int rd_fd`	If the `rsrctype_t` value is `R_FILDES`, an open file or device (referenced by descriptor) is the resource, for which `rd_fd` is the handle.
`char *rd_pathname`	If the `rsrctype_t` value is `R_PATH`, a file or device (referenced by pathname) is the resource, for which `rd_pathname` is the handle.
`int rd_shmid`	If the `rsrctype_t` value is `R_SHM`, a System V shared memory segment is the resource, for which `rd_shmid` is the handle.
`pid_t rd_pid`	If the `rsrctype_t` value is `R_PID`, a process is the resource, for which `rd_pid` is the handle.
`void *rd_addr`	If the `rsrctype_t` value is `R_MEM`, a mapped region of virtual memory is the resource, for which `rd_addr` is the handle.
`nsgid_t rd_nsg`	If the `rsrctype_t` value is `R_NSG`, a NUMA Scheduling Group is the resource, for which `rd_nsg` is the handle.

See `nloc`(3) and `nfork`(3) for information about using `rsrcdescr_t` handles to query and bind resources in the context of NUMA system topology.

`struct_nsgid_ds`

A structure that specifies the access permissions and associated parameters and statistics for a NUMA Scheduling Group (NSG). This structure contains the following members:

`struct ipc_perm nsg_perm`	A subordinate structure that contains the NSG access permissions.
`int nsg_nattach`	The number of processes attached to the NSG.

`struct_nsg_thread`

A structure that specifies a thread (process ID and thread ID) attached to an NSG. This structure contains the following members:

`pid_t nsgth_pid`

The process ID of a thread that is attached to an NSG.

`unsigned int nsgth_thread`

The thread ID (index) of a thread that is attached to an NSG.

SEE ALSO

Functions: `nfork`(3), `nloc`(3), `numa_intro`(3), `rad_foreach`(3), `rad_get_info`(3)

NAME

numa_scheduling_groups – HP Tru64 UNIX NUMA Scheduling Groups description (`libnuma` library)

DESCRIPTION

Normally, the kernel scheduler attempts to distribute the workload evenly over the entire machine. When the system resources are evenly utilized, the machine is considered to be balanced. When balancing the workload, the scheduler operates in a context-free manner; that is, processes may be distributed to various CPUs, or other resources, without regard to their function or relationship to one another. In certain cases, a user may wish to bundle a group of processes together so that they have equal access to the same system resources. For instance, cooperating processes that share the same physical memory may perform better if all of these processes execute on CPUs that are local to that memory.

NUMA Scheduling Groups (NSG) cause the scheduler load-balancing system to treat all members of an NSG as a unit. If one process belonging to an NSG moves from one Resource Affinity Domain (RAD) to another, all other members of the NSG have to move with it.

NSGs and their members have the following characteristics:

- The resource domain of the first process joining an NSG provides the initial resource domain location for that NSG, called the NSG home RAD.
- All other processes joining the NSG (through the `nsg_attach_pid( )` function) will be migrated to the NSG home RAD. If the joining process is not allowed to migrate, the `nsg_attach_pid( )` function will fail.
- To support load balancing, an NSG is allowed to migrate to any RAD on the system if none of its members is bound to a specific resource (such as another RAD, CPU, and so on).
- An NSG member is allowed to attach to or bind to a resource only if no other members are bound to different resources. The entire NSG will migrate to the RAD containing the resource at the time it was successfully bound.
- If one NSG member is bound to a resource, all other members of that NSG are also bound to the RAD containing that resource, because the NSG and, therefore its members, is no longer allowed to migrate.

SEE ALSO

Commands: runon(1)

Functions: numa_intro(3), bind_to_cpu(3), nsg_attach_pid(3), nsg_detach_pid(3), nsg_destroy(3), nsg_get(3), nsg_get_nsgs(3), nsg_get_pids(3), nsg_init(3), nsg_set(3), rad_attach_pid(3), rad_bind_pid(3), rad_detach_pid(3)

NAME

cpu_foreach – enumerate members of a CPU set (`libc` library)

SYNOPSIS

```
#include <cpuset.h>

cpu_cursor_t cursor = SET_CURSOR_INIT;

cpuid_t cpu_foreach(
          cpuset_t cpuset,
          unsigned int flags,
          cpu_cursor_t *cursor);
```

PARAMETERS

cpuset

Specifies a CPU set whose members are to be enumerated.

flags

Control the processing of set members. The *flags* parameter can be one or more (a logical OR operation) of the following flags:

`SET_CURSOR_FIRST`

Initializes the cursor to the first member of the set before scanning.

`SET_CURSOR_WRAP`

Wraps around to the beginning of the set when scanning for members.

`SET_CURSOR_CONSUME`

Consumes the set members; that is, removes the member from the set when found.

As shown in the SYNOPSIS, a *`cursor`* variable may be initialized to the value `SET_CURSOR_INIT`. Initialization of this variable is equivalent to setting the `SET_CURSOR_FIRST` flag on the initial call to `cpu_foreach( )`.

cursor

Points to an opaque type that records the position in a set for subsequent invocations of the `cpu_foreach( )` function.

DESCRIPTION

The `cpu_foreach( )` function scans the specified *cpuset*, starting at the position saved in the *cursor* parameter, for members of the set and returns the first member found. If the `SET_CURSOR_FIRST` flag is set, the cursor is initialized to the beginning of the set before starting the scan. If no members are found, the `cpu_foreach( )` function will return `CPU_NONE`.

If the `SET_CURSOR_WRAP` flag is set, the scan will wrap from the end of the set to the beginning searching for a member to return. Otherwise, a one pass scan is performed, and when the end of the set is reached, the cursor is positioned at the end of the set. From then on, the `cpu_foreach( )` function will continue to return `CPU_NONE` until the cursor is reinitialized (by specifying the `SET_CURSOR_FIRST` or`SET_CURSOR_WRAP` flag).

If the `SET_CURSOR_CONSUME` flag is set, the member returned, if any, will be removed from the set.

NOTES

Although the preceding description discusses the “beginning” and “end” of the set, and wrapping from the end to the beginning, CPU sets are conceptually unordered. Thus, these end points are arbitrary points in the set that exist to ensure that each member is returned only once per pass through the set. Therefore, applications should not depend on a specific numeric order of the returned member IDs.

RETURN VALUES

The `cpu_foreach( )` function returns the next member in the set starting at the position of the cursor. If no more members are found, `CPU_NONE` is returned. This function always completes successfully.

ERRORS

No errors are defined for the `cpu_foreach( )` function.

EXAMPLES

See the EXAMPLES section of `cpusetops`(3) for a sample program that uses the `cpu_foreach( )` function.

SEE ALSO

Functions: `cpusetops`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

cpu_get_current – Return the caller's current CPU ID (`libc` library)

SYNOPSIS

```
#include <cpuset.h>
cpuid_t cpu_get_current( void );
```

PARAMETERS

The `cpu_get_current( )` function has no parameters.

DESCRIPTION

The `cpu_get_current( )` function fetches the ID of the CPU on which the caller is executing.

NOTES

This function is similar to the following call:

```
getsysinfo(GSI_CPU_CURRENT, &cpuid, ...);
```

However, `cpu_get_current( )` returns the CPU identifier directly (as type `cpuid_t`), whereas the `getsysinfo( )` call stores the identifier into a buffer (as type `long`) whose address is passed to the function.

RESTRICTIONS

As is true for many system information queries, the data returned by `cpu_get_current( )` may be stale by the time it is returned to or used by the caller. In other words, a context switch to a different CPU can occur after the "current CPU" has been fetched by the application.

RETURN VALUES

The `cpu_get_current( )` function returns the CPU ID of the processor where the caller is executing. This function always completes successfully.

ERRORS

None.

SEE ALSO

Functions: `getsysinfo`(2), `numa_intro`(3)

Files: `numa_types`(4)

NAME

cpu_get_info, cpu_get_num, cpu_get_max – Query CPU information for the platform (`libc` library)

SYNOPSIS

```
#include <cpuset.h>

int cpu_get_info(
        ncpu_info_t *info);
int cpu_get_num( void );
int cpu_get_max( void );
```

PARAMETERS

info

Points to an `ncpu_info_t` buffer to receive the CPU information for the booted configuration.

DESCRIPTION

The `cpu_get_info( )` function returns the following information about the platform CPU configuration in the buffer pointed to by the *info* parameter:

`int ncpu_version`

Revision number.

`int ncpu_max`

Maximum number of CPUs supported by the machine architecture.

`cpuset_t ncpu_present`

CPU processor set that is physically plugged into the system and recognized by the system console.

`cpuset_t ncpu_running`

Set of online CPUs in the caller's partition; that is, the set of CPUs on which the caller can schedule work.

`cpuset_t ncpu_binding`

Set of CPUs in the partition that have processes bound to them.

`cpuset_t ncpu_ex_binding`

Set of CPUs in the partition whose processor set is marked for exclusive use.

The CPU sets specified in the `ncpu_info_t` structure must have been created by the caller prior to the call. If the caller specifies zero for a CPU set, the function silently ignores filling in data for that set.

The information returned by the `cpu_get_info( )` function is relative to the caller's partition.

The `cpu_get_num( )` function returns the actual number of CPUs available in the caller's partition.

The `cpu_get_max( )` function returns the maximum number of CPUs, including unpopulated CPU slots, that can be configured in the system.

NOTES

A `cpu_get_info( )` call is similar to a `getsysinfo(GSI_CPU_INFO, ...)` call. The principal difference is that the main `ncpu_info_t` structure fields returned by `cpu_get_info( )` are of type `cpuset_t`, whereas the same information returned by `getsysinfo( )`is of type `ulong_t`. Furthermore, a `getsysinfo(GSI_CPU_INFO, ...)` call returns information only about the first *n* CPUs, where *n* is the number of bits in a `ulong_t` field, or `(sizeof(ulong_t)*8)`.

For `cpu_get_info( )`, the `ncpu_version` field of the *info* argument must be set to `NCPU_INFO_VERSION` prior to the call.

RESTRICTIONS

The information returned by these functions is a snapshot of the platform/partition configuration at the time the information is sampled. The data may be stale by the time the caller uses the information.

RETURN VALUES

The `cpu_get_info( )` function returns the following values:

`0` Success.

`-1` Failure. In this case, `errno` is set to indicate the error.

The `cpu_get_num( )` and `cpu_get_max( )` functions return values as stated in the DESCRIPTION. These functions always complete successfully.

ERRORS

The `cpu_get_info( )` function fails, it sets `errno` to one of the following values:

[EFAULT]

The *info* argument or one of the `cpuset_t` elements, points to an invalid address.

[EINVAL]

One or more of the `cpuset_t` elements of the *info* argument points to an invalid CPU set, possibly one that was not created by `cpusetcreate( )`.

[EPERM]

The version number specified in the `ncpu_version` field of the *info* argument is not recognized by the system.

SEE ALSO

Functions: `cpusetops`(3), `numa_intro`(3)

NAME

cpu_get_rad – Queries the RAD associated with a CPU (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

radid_t cpu_get_rad(
         cpuid_t cpu);
```

PARAMETERS

cpu

Identifies the CPU for which the associated RAD is requested.

DESCRIPTION

The `cpu_get_rad( )` function returns the identifier of the Resource Affinity Domain (RAD) associated with the CPU specified by the *cpu* argument.

RETURN VALUES

The `cpu_get_rad( )` function returns the following values:

RAD ID for *cpu*	Success.
`-1`	Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `cpu_get_rad( )` function fails, it sets `errno` to the following value:

[EINVAL]

The CPU identifier specified by *cpu* is invalid.

SEE ALSO

Functions: `numa_intro`(3)

Files: `numa_types`(4)

NAME

cpusetops: cpuaddset, cpuandset, cpucopyset, cpucountset, cpudelset, cpudiffset, cpuemptyset, cpufillset, cpuisemptyset, cpuismember, cpuorset, cpusetcreate, cpusetdestroy, cpuxorset – Perform operations on CPU sets (libc library)

SYNOPSIS

```
#include <cpuset.h>

int cpuaddset(
          cpuset_t set,
          cpuid_t cpuid);
int cpuandset(
          cpuset_t set_src1,
          cpuset_t set_src2,
          cpuset_t set_dst);
int cpucopyset(
          cpuset_t set_src,
          cpuset_t set_dst);
int cpucountset(
          cpuset_t set);
int cpudelset(
          cpuset_t set,
          cpuid_t cpuid);
int cpudiffset(
          cpuset_t set_src1,
          cpuset_t set_src2,
          cpuset_t set_dst);
int cpuemptyset(
          cpuset_t set);
int cpufillset(
          cpuset_t set);
int cpuisemptyset(
          cpuset_t set);
int cpuismember(
          cpuset_t set,
          cpuid_t cpuid);
int cpuorset(
          cpuset_t set_src1,
          cpuset_t set_src2,
          cpuset_t set_dst);
int cpusetcreate(
          cpuset_t *set);
int cpusetdestroy(
          cpuset_t *set);
```

```
int cpuxorset(
          cpuset_t set_src1,
          cpuset_t set_src2,
          cpuset_t set_dst);
```

PARAMETERS

cpuid

Identifies a CPU.

set

Specifies or points to a CPU set.

set_dst

Specifies a CPU set that is being copied to or that is the result of a logical OR, XOR, or AND operation on two other CPU sets.

set_src[n]

Specifies a CPU set that is being copied to another CPU set or that is part of a logical OR, XOR, or AND operation with another CPU set.

DESCRIPTION

The `cpusetops` primitives manipulate sets of CPUs, by operating on data objects (of type `cpuset_t`) that are created by `cpusetcreate( )`.

The `cpusetcreate( )` function allocates, and sets to empty, a CPU set pointed to by *set*.

The `cpusetdestroy( )` function releases the memory that was obtained by `cpusetcreate( )` for the specified CPU set pointed to by *set*.

The `cpucountset( )` function returns the number of members in the CPU set specified by *set*.

The `cpuemptyset( )` function initializes the CPU set specified by *set*, such that no CPUs are included in the set.

The `cpufillset( )` function initializes the CPU set specified by *set*, such that as many CPUs as the system architecture is capable of supporting are included in the set. Note that this platform maximum might be more than the number of CPUs that are available on the system.

The `cpuismember( )` function tests whether the CPU specified by the value of *cpuid* is a member of the CPU set specified by *set*.

The `cpuisemptyset( )` function tests whether the CPU set specified by the *set* is empty.

The `cpucopyset( )` function copies the contents of the CPU set specified by *set_src* to the CPU set specified by *set_dst*.

The `cpuaddset( )` and `cpudelset( )` functions respectively add or delete the individual CPU specified by the value of *cpuid* to or from the CPU set specified by *set*.

The `cpuandset( )`, `cpuorset( )`, and `cpuxorset( )` functions perform a logical AND, OR, or XOR operation, respectively, on the CPU sets specified by *set_src1* and *set_src2*, storing the result in the CPU set specified by *set_dst*.

The `cpudiffset( )` function finds the logical difference between the CPU sets specified by *set_src1* and *set_src2*, storing the result in the CPU set specified by *set_dst*. (The result is made up of members that are included in *set_src1* but not in *set_src2*.)

RETURN VALUES

These functions return the following values:

`0`	Success (returned by all functions). For `cpuisemptyset( )` and `cpuismember( )` only, 0 also means the condition being tested is false; that is, the specified CPU set is not empty or does not contain the specified member.
`1`	Success (returned by `cpuisemptyset( )` and `cpuismember( )`only). This return value also means the condition being tested is true; that is, the specified CPU set is empty or contains the specified member.
`-1`	Failure (returned by all functions). In this case, `errno` is set to indicate the error.

ERRORS

The `cpuaddset( )`, `cpuandset( )`, `cpucopyset( )`, `cpucountset( )`, `cpudelset( )`, `cpudiffset( )`, `cpuemptyset( )`, `cpufillset( )`,

`cpuisemptyset( )`, `cpuismember( )`, `cpuorset( )`, and `cpuxorset( )` functions set `errno` to the following value for the corresponding condition:

[EINVAL]

The value of a *set* or *set_** argument is invalid (possibly is not a CPU set created by`cpusetcreate( )`).

The `cpusetcreate( )` and `cpusetdestroy( )` functions set `errno` to one of the the following values for the corresponding condition:

[EFAULT]

The address of the specified CPU set is invalid.

[ENOMEM]

For `cpusetcreate( )` only, no memory could be allocated for the specified CPU set.

If the `cpuaddset( )`, `cpudelset( )`, and `cpuismember( )` functions fail, they set `errno` to the following value for the reason specified:

[EDOM]

The value of *cpuid* is an invalid or unsupported CPU identifier.

EXAMPLES

The following example demonstrates a variety of CPU set operations:

```
#include <cpuset.h>

int
main()
{
        cpuset_t cpuset, cpuset2;

        /* Create cpusets - initialized as empty */
        cpusetcreate(&cpuset);
        cpusetcreate(&cpuset2);
        /* demonstrate cpuset operations */

        /* add cpu 0 to cpuset */
        if (cpuaddset(cpuset, 0) == -1) {
                perror("cpuaddset");
                return 0;
        }

        /* copy cpuset to cpuset2 */
        if (cpucopyset(cpuset, cpuset2) == -1) {
```

```
                perror("cpucopyset");
                return 0;
        }

        if (cpuaddset(cpuset, 1) == -1) {
                /* add cpu 1 to cpuset */
                perror("cpuaddset");
                return 0;
        }

        /* difference of cpuset and cpuset2, store in cpuset */
        if (cpudiffset(cpuset, cpuset2, cpuset) == -1) {
                perror("cpudiffset");
                return 0;
        }

        /* Enumerate cpuset. */
        while (1) {
                cpuid_t id;
                int flags = SET_CURSOR_CONSUME;
                cpu_cursor_t cpu_cursor = SET_CURSOR_INIT;

                id = cpu_foreach(cpuset, flags, &cpu_cursor);

                if (id == CPU_NONE) {
                        printf("\n");
                        break;
                } else {
                        printf("%3d ", id);
                }
        }

        /* Destroy cpuset */
        cpusetdestroy(&cpuset);
        cpusetdestroy(&cpuset2);
        return 0;
}
```

SEE ALSO

Functions: cpu_foreach(3), numa_intro(3)

Files: numa_types(4)

NAME

memalloc_attr – Query the memory allocation policy and attributes (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int memalloc_attr(
          vm_offset_t va, memalloc_attr_t *attr);
```

PARAMETERS

va

The user virtual address for which the *memory allocation policy* is requested.

attr

Points to a buffer to receive the memory allocation policy and attributes for the page containing the specified virtual address.

DESCRIPTION

The `memalloc_attr( )` function returns the current *memory allocation policy* and associated attributes in the buffer pointed to by *attr* for the address specified by *va*.

If `radset` information about the *memory allocation policy* is desired, a `radset` must be allocated through the `radsetcreate( )` function, and the `mattr_radset` element of the *attr* argument must point to that `radset`. Otherwise, a 0 must be specified for the `mattr_radset`.

EXAMPLE

```
#include <numa.h>
main()
{
    vm_offset_t va;
    memalloc_attr_t attr;
    int id;
    int flags = SET_CURSOR_CONSUME;
    rad_cursor_t cursor = SET_CURSOR_INIT;
    radsetcreate(&attr.mattr_radset);
    va = (vm_offset_t)&attr;
```

```
    /* no policy in effect - return zeroes */
    if (memalloc_attr(va, &attr) == -1) {
        perror("memalloc_attr");
        radsetdestroy(&attr.mattr_radset); return 0;
    }
    printf("mattr_policy = 0x%lx\n", attr.mattr_policy);
    printf("mattr_rad = 0x%lx\n", attr.mattr_rad);
    printf("mattr_stride = 0x%lx\n", attr.mattr_stride);
    printf("mattr_distance = 0x%lx\n", attr.mattr_distance);
    printf("mattr_pagesz = 0x%lx\n\n", attr.mattr_pagesz);

    /* set policy */
    attr.mattr_policy = MPOL_DIRECTED;
    attr.mattr_rad = 0;
    if (nmadvise((void *)va, sizeof(memalloc_attr_t), 0, &attr) == -1)  {
        perror("nmadvise");
        radsetdestroy(&attr.mattr_radset); return 0;
    }

    if (memalloc_attr(va, &attr) == -1) {
        perror("memalloc_attr");
        radsetdestroy(&attr.mattr_radset);
        return 0;
    }
    printf("mattr_policy = 0x%lx\n", attr.mattr_policy);
    printf("mattr_rad = 0x%lx\n", attr.mattr_rad);
    printf("mattr_stride = 0x%lx\n", attr.mattr_stride);
    printf("mattr_distance = 0x%lx\n", attr.mattr_distance);
    printf("mattr_pagesz = 0x%lx\n", attr.mattr_pagesz);

    /* enumerate the mattr_radset */
    printf("\nEnumerating radset members:\n");
    while ((id = rad_foreach(attr.mattr_radset, flags, &cursor)) != RAD_NONE) {
        if ((id % 8) == 0)
            printf("\n");
    printf("%3d, ", id);
    }
    printf("\n");
}
```

RETURN VALUES

`0`	Success. In this case, the function stores the requested memory allocation policy and attributes in the buffer pointed to by *attr*. If no *memory allocation policy* has been set for the specified virtual address (e.g., `madvise( )` or `nmadvise( )`) not called for that address), a zeroed *attr* structure is returned.
`-1`	Failure. In this case, the function sets `errno` to indicate the error.

ERRORS

If the `memalloc_attr( )` function fails, it sets `errno` to one of the following values:

[EFAULT]

The address pointed to by *va*, *attr*, or `mattr_radset` is invalid.

[EINVAL]

The `mattr_radset` element of the *attr* argument points to an invalid RAD set, possibly one that has not been created by a `radsetcreate( )` call.

SEE ALSO

Functions: `numa_intro`(3)

Files: `numa_types`(4)

NAME

nfork – Creates a child process (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

pid_t nfork(
         numa_attr_t *numa_attr);
```

PARAMETERS

numa_attr

Points to a structure of type `numa_attr_t` that contains the following members:

`nattr_type` The type of resource to which the child process will be attached, or near which the child process will be located.

`nattr_descr` The resource descriptor for the resource to which the child process will be attached, or near which the child process will be located.

`nattr_distance` The distance criteria for selecting resources. RADs in the caller's partition that have a distance (from the specified resource) equal to or less than this value will be considered as candidates for the child process's location.

`nattr_flags` A bit mask of options that help control how the system assigns a "home RAD" to the child process.

The following symbolic values are defined for this bit mask:

`RAD_INSIST` The requested RAD assignment is mandatory. The child process will be created on one of the RADs in

	the specified RAD set regardless of the CPU or memory load of the specified RADs.
RAD_NO_INHERIT	The child process might not be assigned to the same home RAD as its parent process. Allows the system to assign a home RAD to the child process depending on available resources. Normally, child processes do inherit the assignments and attributes of the parent process.
RAD_SMALLMEM	The process has "small memory" requirements, so the system should favor (for the child process's home RAD) those RADs with light CPU loads, independent of their available memory.
RAD_LARGEMEM	The process has large memory requirements, so the system should favor (for the child process's home RAD) those RADs with more available memory, independent of CPU loads.

If *numa_attr* is NULL, the function behaves as `fork( )`.

DESCRIPTION

The `nfork( )` function causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

- Environment
- Close-on-exec flag. See `exec`(2).
- Signal-handling settings (in other words, `SIG_DFL`, `SIG_IGN`, `SIG_HOLD`, function address)
- Set-user-ID mode bit
- Set-group-ID mode bit
- Trusted state
- Profiling on/off status
- Nice value. See `nice`(2).
- All attached shared libraries
- Process group ID
- Session ID (tty group ID)
- Foreground process ID. See `exit`(2).
- Current working directory
- Root directory
- File mode creation mask. See `umask`(2).
- File size limit. See `ulimit`(2).
- All attached shared memory segments. See `shmat`(2).
- All attached mapped regions. See `mmap`(2) and `nmmap`(3).
- All mapped regions with the same protection and sharing mode as in the parent process.

The child process differs from the parent process in the following ways:

- The child process has a unique process ID that does not match any active process group ID.
- The parent process ID of the child process matches the process ID of the parent.

- The child process has its own copy of the parent process's file descriptors. Each of the child's file descriptors refers to the same open file description with the corresponding file descriptor of the parent process.
- The child process has its own copy of the parent's open directory streams. Each open directory stream in the child process may share directory stream positioning with the corresponding directory stream of the parent.
- All `semadj` values are cleared.
- Process locks, text locks and data locks are not inherited by the child. See `plock`(2).
- The child process's values of `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` are set to 0.
- Any pending alarms are cleared in the child process.
- Any interval timers enabled by the parent process are reset in the child process.
- Any signals pending for the parent process are cleared for the child process.
- The NUMA scheduling parameters and memory allocation attributes of the child process may be different from those of the parent process.

The `nfork( )` function is used when the caller wishes to specify the location where the child process should be loaded. If the `nattr_descr` field is NULL, the `nfork( )` function behaves identically to the `fork( )` function, and the child process inherits the calling thread's memory allocation policy and attributes. However, the `nattr_flags` field may still affect how the system selects a home RAD for the child process, as described in PARAMETERS.

If the `nattr_descr` field is non-NULL, it, along with the `nattr_type` and `nattr_distance` fields, identifies the acceptable RADs from which to select the child process's home RAD. The memory allocation policy for the child process will be set to `MPOL_THREAD`.

If `nattr_type` is anything other than `R_RAD` or `R_NSG`, `nfork( )` will behave as though `nloc( )` were called to obtain a RAD set that meets the specified criteria, and then `nfork( )` were called with `nattr_type` equal to `R_RAD`, and `nattr_descr` pointing to the RAD set returned by `nloc( )`. This behavior is described below. The `nattr_distance` parameter is ignored for a `nattr_type` of `R_RAD` or `R_NSG`.

If the nattr_descr field is equal to R_RAD, then nattr_descr points to a radset_t that identifies the acceptable RADs from which to select the child process's initial home RAD. The remainder of the RAD set (in other words, the set less the child process's home RAD) becomes the child's overflow set.

A suitable set of RADs can be located according to available resources by nloc( ) and can be manipulated using the operators described for radsetops( ). Unless RAD_INSIST has been set in nattr_flags, the specified RAD set is considered a hint, which may be overridden if all the RADs in the specified set have very high CPU loads or too little available memory. If the RAD_INSIST flag is specified in nattr_flags, the RAD specification is treated as mandatory, and the child process is assigned to one of the specified RADs despite a large CPU load or memory shortage.

When using nfork( ), the caller can further specify an appropriate RAD by setting the RAD_SMALLMEM or RAD_LARGEMEM bits in the nattr_flags field. RAD_SMALLMEM indicates that the child will have very low memory requirements, so can be placed on a RAD having little available memory if that RAD has a particularly light CPU load. Conversely, if RAD_LARGEMEM is set, the process is placed on the RAD with the most available memory even though that RAD may have a high CPU load. RAD_SMALLMEM and RAD_LARGEMEM are also taken into account during any future process migrations.

If the nattr_descr field is equal to R_NSG, then nattr_descr specifies a NUMA Scheduling Group (NSG) as returned by nsg_init( ). The child process will be attached to the NSG and will receive the same home RAD as the other members in the NSG. If the child process is the first process to attach to the NSG, then the home RAD for the child will be inherited from the calling thread, just as for the fork( ) function.

NOTES

The nfork( ) function is supported for multithreaded applications.

If a multithreaded process calls the nfork( ) function, the new process contains a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process should only execute operations that will not cause deadlock until one of the exec functions is called.

The set of valid resources that may be specified is constrained by the caller's partition.

RETURN VALUES

`0`	Success (returned to the child process). In this case, the function also returns the process ID of the child process to the parent process. The child process and all of the related data structures will be allocated on one of the RADs selected by the system scheduler from among those specified by the `nattr_type`, `nattr_descr`, and `nattr_distance` fields. The initial thread of the child process will be scheduled on one of the available CPUs in the selected RAD.
`-1`	Failure (returned to the parent process). In this case, no child process is created, and `errno` is set to indicate the error.

ERRORS

If the `nfork( )` function fails, it sets `errno` to one of the following values for the condition specified:

[EAGAIN]

The limit on the total number of processes executing for a single user would be exceeded. This limit can be exceeded by a process with superuser privilege.

[EFAULT]

The *numa_attr* argument or the `nattr_descr` structure field points to an invalid address.

[EINVAL]

The `nattr_type` field specifies an invalid resource type, the `nattr_descr` field specifies an invalid resource, or the `nattr_flags` field specifies an undefined flag.

[ENOMEM]

There is not enough memory to create the child process.

SEE ALSO

Functions: exec(2), exit(2), fork(2), mmap(2), plock(2), umask(2), nice(3), nloc(3), nmmap(3), nsg_init(3), numa_intro(3), radsetops(3), ulimit(3)

Files: numa_types(4)

NAME

nloc – Queries the NUMA Topology or Resource Affinity Domains (`libnuma` library)

SYNOPSIS

```
#include <numa.h>
int nloc(
          numa_attr_t *numa_attr,
          radset_t radset);
```

PARAMETERS

numa_attr

Points to a structure that specifies the criteria for selecting a set of resource Affinity Domains (RADs). This structure contains the following elements:

`nattr_type`

The type of resource for which the set of "nearby" RADs is requested.

`nattr_descr`

The resource descriptor for which the RAD set is requested.

`nattr_distance`

The distance criteria for selecting resources. RADs in the caller's partition that have a distance <= this value will be included in the *radset* returned by `nloc( )`. See DESCRIPTION for more information about `nattr_distance`.

`nattr_flags`

Flags that influence the selection of RADs. See DESCRIPTION for details.

radset

Specifies a buffer to contain the set of RADs in the caller's partition that satisfy the criteria specified by *numa_attr*.

DESCRIPTION

The `nloc( )` function will return in *radset* the set of RADs that have a distance <= the `nattr_distance` value from the specified resource. The returned *radset* value may be used as an argument to explicit process or thread placement APIs or in the `mattr_radset` member of a memory allocation policy structure for explicit memory placement.

The following symbolic values for `nattr_distance` are defined:

`RAD_DIST_LOCAL` — Represents the distance value for resources that are directly connected to the specified resource.

`RAD_DIST_REMOTE` — Represents the maximum distance value for the system. Generally, all RADs in the partition will be <= this distance.

For NUMA topologies in which RADs are variable distances from one another, `RAD_DIST_LOCAL` can be incremented to specify a specific distance. In other words, `RAD_DIST_LOCAL + 1` represents the distance to the closest RADs, `RAD_DIST_LOCAL + 2` represents the distance to the next closest RADs, and so forth. For example, when `nattr_distance` is set to `RAD_DIST_LOCAL + 2`, `nloc( )` returns in *radset* the set of RADs that have a distance <= (`RAD_DIST_LOCAL + 2`) from the specified resource. For NUMA topologies where all RADs are equidistant from one another, <= (`RAD_DIST_LOCAL + n`), where *n* is a positive integer, is equivalent to <= `RAD_DIST_REMOTE`.

The following symbolic values are defined for the `nattr_flags` field:

`RAD_BOUND` — When specified, only RADs that have processes bound to them will be returned in *radset*.

`RAD_NOBOUND` — When specified, only RADs that do not have processes bound to them will be returned in *radset*.

RETURN VALUES

`0` — Success.

`-1` — Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `nloc( )` function fails, it sets `errno` to one of the following values for the reason specified:

[EFAULT]

The *numa_attr* argument (or its `nattr_descr` field) or the *radset* argument point to an invalid address.

[EINVAL]

One or more of the following conditions are true:

- The *numa_attr* argument contains an undefined type value.
- The `nattr_descr` field contains an invalid resource value for the specified type.
- The `nattr_distance` or `nattr_flags` fields contain an invalid or undefined value.

[ESRCH]

The process specified by `rd_pid` does not exist.

[ELOOP]

There are too many symbolic links in `rd_pathname`.

[ENAMETOOLONG]

The `rd_pathname` length exceeds `MAXPATHLEN`, or a component of `rd_pathname` exceeds `MAXNAMELEN`.

[ENOENT]

The file named by `rd_pathname` does not exist.

[ENOTDIR]

A component of `rd_pathname` is not a directory.

SEE ALSO

Functions: `rad_get_info`(3)

nloc(3)

Files: numa_types(4)

NAME

nmadvise – Advise the system of the expected paging behavior of a process (libnuma)

SYNOPSIS

```
#include <numa.h>
#include <sys/nman.h>

int nmadvise(
          void *addr,
          long len,
          int behav,
          memalloc_attr_t *attr);
```

PARAMETERS

The parameters to the nmadvise( ) function are the same as for madvise( ), with the addition of the *attr* parameter:

addr

Points to the starting address of the range of pages to which the advice refers.

len

Starting at the address specified by the *addr* parameter, specifies the length (in bytes) of the memory range.

behav

Specifies the expected behavior pattern for referencing pages in the specified range. See DESCRIPTION for details.

attr

Points to a structure containing the memory allocation policy and attributes that will be assigned to the specified range. See the entry for memalloc_attr_t in numa_types(4) for a description of this structure.

DESCRIPTION

The nmadvise( ) function permits a process to advise the system about its expected behavior in referencing a particular range of pages in the process

address space. This advice includes reference patterns that the system can use to optimize page fault behavior (as also supported by `madvise( )`), plus NUMA locality information that the system can use to optimize the placement of the pages that are allocated in response to page faults.

The `nmadvise( )` function supports the following flags to be ORed with one of the *behav* values documented in `madvise`(2). The normal practice is to OR one or more of the following flags with the `MADV_NORMAL` behavior to advise the system about page placement without specifying any particular paging behavior:

`MADV_CURRENT`
: Prepare the specified range or object for migration to the memory region specified by the memory allocation policy and associated attributes. Migration means that pages already allocated in the specified range will be copied to new pages that are allocated according to the NUMA policy and attributes as specified by the *attr* parameter. Without this flag, only new allocations in the specified range will be allocated according to the specified policy and attributes.

  See the discussion following this list for information about the effect of ORing this flag with the `MADV_DONTNEED` behavior value.

`MADV_WAIT`
: This flag may be logically ORed with another *behav* flag to indicate that the requested operation be performed before returning from the function call.

  Without this flag, the `nmadvise( )` function will return as soon as the new memory allocation policy and attributes are in place and, if `MADV_DONTNEED` is also specified, the currently allocated pages are discarded. In this case, migration of page contents (if `MADV_CURRENT` is specified and `MADV_DONTNEED` is not specified) or new allocations of zeroed pages in accordance with the specified policy and attributes does not occur until the program touches a page in the specified range.

`MADV_INSIST`
: This flag may be logically ORed with another *behav*flag to request that the program be notified if

the specified operation cannot be performed. This flag is currently ignored.

Except for MADV_DONTNEED, the *behav* flags supported by both madvise( ) and nmadvise( ) are equivalent. In other words, the *behav* information is orthogonal to the additional NUMA information (the memory allocation policy and associated attributes) that the nmadvise( ) function provides. However, MADV_DONTNEED has special significance in the context of memory location changes within the NUMA topology. The nmadvise( ) call uses *behav* flags as follows to specify how currently allocated pages are to be handled when the requested NUMA allocation policy and attributes are applied:

- MADV_DONTNEED tells the system to discard the contents of any pages currently allocated for the process or thread and then perform future allocations according to the specified NUMA policy and attributes.
- MADV_CURRENT (without MADV_DONTNEED) requests that, if the NUMA policy and attributes indicate that page allocations should start in a location different from the location of pages already allocated, the contents of the already allocated pages should be migrated to the new location.

  The MADV_CURRENT is ignored when ORed with MADV_DONTNEED because the specified behavior is to discard currently allocated pages.
- Omitting both MADV_CURRENT and MADV_DONTNEED preserves the contents of already allocated pages at their current location and allows only future page allocations to be made according to the specified NUMA policy and attributes.

Future page allocations that are performed according to the specified NUMA policy and attributes will be initialized to zero unless the memory allocation is performed to map a file from disk, in which case the memory pages are inititalized from disk.

If, in the structure pointed to by *attr*, the mattr_policy member is MPOL_DIRECTED, then the mattr_radset member specifies the Resource Affinity Domain (RAD) from which pages will be allocated for virtual addresses in the specified range (*addr* to *addr+len*). If the mattr_policy member is MPOL_THREAD, then pages for the virtual addresses in the specified thread will be allocated from the faulting thread's home RAD.

If the mattr_policy member is MPOL_DIRECTED or MPOL_THREAD, then the mattr_radset member specifies the overflow behavior when there is no free memory on the preferred RAD. If mattr_radset is NULL (in other words, no RAD set), then the overflow set is taken to be the set of all RADs in the caller's partition. If mattr_radset specifies an empty RAD set, no overflow RAD set is requested and the process or thread will wait for memory to become available on the preferred RAD.

If the *attr* parameter is a NULL pointer, any *behav* flags specific to nmadvise( )are ignored, and the function is equivalent to madvise( ). In this case, any *behav* flags specific to nmadvise( ) (in other words, not supported by the madvise( ) function) are treated as invalid.

NOTES

As with madvise( ), the behaviors specified with nmadvise( ) are considered by the system to be hints, and may in fact, be unimplemented. Unimplemented behaviors will always return success.

Furthermore, the operating system always attempts to replicate program text and shared library text on all RADs, so any request to change the memory allocation policy for these parts of the application's address space is always silently ignored.

RETURN VALUES

0	Success.
-1	Failure. In this case, errno is set to indicate the error.

ERRORS

If the nmadvise( ) function fails, it sets errno to one of the following values for the reason specified:

[EBUSY]

MADV_DONTNEED was specified but pages could not be freed, most likely because the specified memory range includes a wired page.

[EFAULT]

A non-NULL *attr* argument points to an invalid address, or the range of pages (*addr*, *len*) includes a wired page or "hole" in the virtual address space.

[EINVAL]

One of the following conditions is true:

- The value of the *behav* parameter or a member of the *attr* structure (the specified RAD, RAD set, or memory allocation policy) is invalid.
- The *attr* parameter is a NULL pointer (which makes the `nmadvise( )` call equivalent to an `madvise( )` call) and the logical OR operation on *behav* values includes one or more flags supported only by `nmadvise( )`.

[ENOMEM]

The *attr* structure specifies a RAD that has no memory.

[ENOSPC]

The *behav* parameter specifies `MADV_SPACEAVAIL`, and resources cannot be reserved.

SEE ALSO

Functions: `madvise`(2), `nshmget`(3), `nmmap`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

nmmap – Maps an open file into a process's address space (`libnuma` library).

SYNOPSIS

```
#include <numa.h>
#include <sys/mman.h>

void *nmmap(
          void *addr,
          size_t len,
          int prot,
          ulong_t flags,
          int filedes,
          off_t off,
          memalloc_attr_t *attr);
```

PARAMETERS

The parameters for `nmmap( )` are the same as for `mmap( )` with the addition of the following NUMA-specific parameter:

attr

Points to a memory allocation policy and attributes structure that will be assigned to the memory object created by the mapping.

See `mmap`(2) for descriptions of the remaining parameters.

DESCRIPTION

If the *attr* argument is NULL, the `nmmap( )` function behaves identically to the `mmap( )` function. If the *attr* argument is non-NULL, it points to a memory allocation policy and attributes structure that specifies where the pages for the new memory object should be allocated.

If, in the structure pointed to by *attr*, the value of `mattr_policy` is `MPOL_DIRECTED` and the value of `mattr_rad` is `RAD_NONE`, the `mattr_radset` value specifies the set of Resource Affinity Domains (RADs) from which the system will choose the RAD where the pages of the new memory object will be allocated. If `mattr_radset` is set to NULL, the system will select a RAD for the memory object from among all the RADs in the caller's partition. In this case, the memory object's overflow set will also be the set of all RADs in the caller's partition.

RETURN VALUES

addr	Success. A value returned to *addr* indicates success and is the starting address of the region (truncated to the nearest page boundary) where the new memory object has been mapped.
`(void *)-1`	Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `nmmap( )` function fails, it sets `errno` to one of the values described in the ERRORS section of `mmap`(2), or to one of the following values for the reason specified:

[EFAULT]

A non-NULL *attr* argument points to an invalid address.

[EINVAL]

The structure pointed to by the *attr* argument contains an invalid memory allocation policy, an invalid RAD, or an invalid RAD set.

SEE ALSO

Functions: `mmap`(2), `nmadvise`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

nsg_attach_pid, nsg_detach_pid – Attaches a process to, or detaches a process from a NUMA Scheduling Group (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int nsg_attach_pid(
          nsgid_t nsg,
          pid_t pid,
          ulong_t flags);
int nsg_detach_pid(
          pid_t pid);
```

PARAMETERS

nsg

Specifies the NUMA Scheduling Group (NSG) to which the listed process will be attached.

pid

Specifies the process ID to attach to (or detach from) the NSG.

flags

Specifies a bit mask of options that affect the attachment. The following options are defined for the *flags* argument:

`NSG_INSIST` — The requested attachment and any implied reassignment is mandatory, overriding any prior attachment and/or binding of the specified processes.

`NSG_MIGRATE` — Arrange for existing memory of the process that is assigned a new home RAD to be migrated to the new RAD. If this option is omitted, only newly allocated pages will be allocated on the new home RAD. Existing pages will migrate if or when they experience a high rate of remote cache misses. Migration will occur only for

	pages in memory objects that have inherited the process's default memory allocation policy.
NSG_NO_INHERIT	A child process will not inherit the NSG of the parent and, therefore, can be assigned to any eligible RAD on the system.
NSG_WAIT	Wait for the requested memory migration to be completed, if possible. If insufficient resources exist to satisfy the request, the function will return without having completed the migration. If NSG_INSIST is also specified, memory not migrated will be paged out.

See DESCRIPTION for more detail about these options.

DESCRIPTION

The nsg_attach_pid( ) function attaches the process identified by the *pid* argument to an NSG. An NSG is a set of processes and/or threads that will be constrained to reside on the same Resource Affinity Domain (RAD). That is, the "home RAD" for all of the processes or threads in an NSG will be the same, and the entire group will be migrated together, if at all. The process identified by *pid* will be removed from any NSG of which it might currently be a member, before adding it to the specified NSG.

If the *pid* argument is NULL, then the call is self-directed. That is, the function behaves as if the current process ID were specified.

The nsg_detach_pid( ) will remove *pid* from its current NSG, if any, and will not add *pid* to any new NSG. It is equivalent to the nsg_attach_pid( ) function with the *nsg* argument of NSG_NONE.

RESTRICTIONS

The caller must have partition administration privilege and the process identified by *pid* must be in the caller's partition.

RETURN VALUES

`0`	Success. In this case, the `nsg_attach_pid( )` function successfully attached to the NSG specified by *nsg*.
`0`	Success. In this case, the `nsg_detach_pid( )` function successfully detached from its NSG.
`-1`	Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `nsg_attach_pid( )` function fails, it sets `errno` to one of the following values for the condition specified:

[EACCES]

The caller does not have execute permission required to attach processes to the NSG.

[ESRCH]

The process specified by *pid* does not exist.

[EINVAL]

The *nsg* argument does not specify a valid NSG, or one or more options in the *flags* argument are invalid.

[EBUSY]

The specified process is hard attached (`RAD_INSIST`) to RADs or has memory wired (locked) on its current RAD such that the process cannot be migrated to the RAD selected for the NSG.

[ENOMEM]

The `NSG_INSIST` and `NSG_MIGRATE` options were specified and no RAD can be found with sufficient memory to accommodate the resulting group.

[EPERM]

The real or effective user ID of the caller does not match the real or effective user ID of the process specified in *pid*.

If the `nsg_detach_pid( )` function fails, it sets `errno` to one of the following values for the condition specified:

[EACCES]

The caller does not have execute permission, which is required to detach processes from the NSG.

[ESRCH]

The process specified by *pid* does not exist.

[EINVAL]

The process specified by *pid* is not a member of an NSG.

[EBUSY]

The specified process is hard attached (`RAD_INSIST`) to RADs or has memory wired (locked) on its current RAD such that the process cannot be detached.

[EPERM]

The real or effective user ID of the caller does not match the real or effective user ID of the process specified in *pid*.

SEE ALSO

Functions: `nsg_init`(3), `numa_intro`(3), `pthread_nsg_attach`(3)

Files: `numa_types`(4)

NAME

nsg_destroy – Destroys a NUMA Scheduling Group (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int nsg_destroy(
          nsgid_t nsg);
```

PARAMETERS

nsg

Specifies the NUMA Scheduling Group (NSG).

DESCRIPTION

Remove the NSG identified by *nsg* and deallocate associated structures. If the NSG is currently non-empty, existing members are removed before deleting the NSG.

RESTRICTIONS

The effective user ID of the calling process must be equal to the value of `nsg_perm.cuid` or `nsg_perm.uid` in the associated `nsgid_ds` structure, or the calling process must have write permissions to the NSG.

RETURN VALUES

`0`	Success. In this case, the NSG was successfully destroyed.
`-1`	Failure. The NSG was not destroyed and `errno` is set to indicate the error.

ERRORS

If the `nsg_destroy( )` function fails, it sets `errno` to one of the following values for the specified condition:

[EACCES]

The calling process does not have write permission.

[EINVAL]

The *nsg* argument does not specify a valid NSG ID.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_init`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

nsg_get – Query status of a NUMA Scheduling Group (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int nsg_get(
          nsgid_t nsg,
          nsgid_ds_t *result);
```

PARAMETERS

nsg

Specifies the NUMA Scheduling Group (NSG)

result

Points to a structure that returns the result of the status query.

DESCRIPTION

The `nsg_get( )` function queries the status of the NSG by copying its associated `nsgid_ds` structure into a buffer pointed to by *result*.

RESTRICTIONS

The effective user ID of the calling process must be equal to the value of `nsg_perm.cuid` or `nsg_perm.uid` in the associated `nsgid_ds` structure, or the calling process must have read permissions to the NSG.

RETURN VALUES

`0`	Success.
`-1`	Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `nsg_get( )` function fails, it sets `errno` to one of the following values for the specified condition:

[EACCES]

The calling process does not have read permission.

[EFAULT]

The *result* argument specifies an invalid address.

[EINVAL]

The *nsg* argument does not specify a valid NSG ID.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_get`(3), `numa_intro`(3)

Files: `numa_types`(4)

nsg_get_nsgs(3)

NAME

nsg_get_nsgs – Returns a list NUMA Scheduling Groups (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int nsg_get_nsgs(
          nsgid_t *nsgidlist,
          int numnsgs);
```

PARAMETERS

nsgidlist

Points to an array that receives the NUMA Scheduling Group (NSG) identifiers.

numnsgs

Specifies the maximum number of `nsgid_t` entries in *nsgidlist*.

DESCRIPTION

The `nsg_get_nsgs( )` function returns a list of NSGs that are active on the system in the buffer pointed to by *nsgidlist*. The argument *numnsgs* specifies the number of `nsgid_t` entries that can be accommodated in the buffer. The list is terminated by a NULL entry.

The required size of the buffer can be obtained by first calling `nsg_get_nsgs( )` with a *numnsgs* set to zero. In this case, the number of NSGs active on the system will be reported in *`nsgidlist`*`[0]`. As always, on a dynamically changing system, the number of entries may be different by the time it is used for the *numnsgs* argument to the second `nsg_get_nsgs( )` call.

RESTRICTIONS

The effective user ID of the calling process must be equal to the value of `nsg_perm.cuid` or `nsg_perm.uid` in the associated `nsgid_ds` structure; or the calling process must have read permissions to each NSG. If the caller does not have the proper permission, that NSG will not be reported in *nsgidlist*.

RETURN VALUES

`0`	Success. However, if the `errno` is set to `E2BIG`, more NSGs than *numnsgs* were available.
`-1`	Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `nsg_get_nsgs( )` function fails, it sets `errno` to the following value for the specified condition:

[EFAULT]

The *nsgidlist* argument points to an invalid address.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_get`(3), `nsg_get_pids`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

nsg_get_pids – Return a list of a NUMA Scheduling Group's process identifiers (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int nsg_get_pids(
          nsgid_t nsg,
          pid_t *pidlist,
          int numpids);
```

PARAMETERS

nsg

Specifies the NUMA Scheduling Group (NSG).

pidlist

Specifies an array receiving the process identifiers of the specified NSG.

numpids

Specifies the maximum number of `pid_t` entries in *pidlist*.

DESCRIPTION

The `nsg_get_pids( )` function returns a list of process IDs of processes attached to the NSG in the buffer pointed to by *pidlist*. The argument *numpids* specifies the number of process IDs that can be accommodated in the buffer. The list is terminated by a NULL entry. The required size of the buffer can be obtained from the `nsg_nattach` member of the `nsgid_ds` structure returned by the `nsg_get( )` function. As always, on a dynamically changing system, the number of entries may be different by the time `nsg_get_pids( )` is called.

RESTRICTIONS

The effective user ID of the calling process must be equal to the value of `nsg_perm.cuid` or `nsg_perm.uid` in the associated `nsgid_ds` structure; or the calling process must have read permissions to the NSG.

RETURN VALUES

`0`	Success. However, if `errno` is set to `E2BIG` on a successful return, more processes than *numpids* were available.
`-1`	Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `nsg_get_pids( )` function fails, it sets `errno` to one of the following values for the specified condition:

[EACCES]

The calling process does not have read permission.

[EFAULT]

The *pidlist* argument specifies an invalid address.

[EINVAL]

The *NSG* argument does not specify a valid NSG identifier.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_get`(3), `nsg_get_nsgs`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

nsg_init – Look up or create a NUMA Scheduling Group (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

nsgid_t nsg_init(
          key_t key,
          ulong_t flags);
```

PARAMETERS

key

Specifies the key that identifies the NUMA Scheduling Group (NSG). This value may be one of the following:

- An arbitrary binary value other than zero
- If the `NSG_GETBYPID` flag is set, the process ID of a member process of the requested NSG

flags

Specifies lookup or creation flags. The following options are defined for the *flags* argument:

`NSG_CREATE` — Creates the NSG and return its identifier. If `NSG_CREATE` and `NSG_EXCL` are both specified, an error will be returned if the NSG identified by *key* already exists.

`NSG_GETBYPID` — If set, then the *key* parameter is the process ID (`pid_t`) of a process that is currently a member of the requested NSG. Otherwise, *key* is an arbitrary binary value that identifies the requested NSG.

If the NSG identified by *key* does not already exist, an error will be returned unless the `NSG_CREATE` flag is specified. If the `NSG_GETBYPID` flag is set and the process identified by the value of *key* does not exist, an error is returned.

	If both the NSG_CREATE and the NSG_GETBYPID flags are set, and the process identified by the value of *key* exists but is not currently a member of an NSG, a new NSG will be created using the value of *key* (the process's ID) and the process will be attached to the new NSG. This establishes the "home RAD" of the process as the home RAD for the NSG.
NSG_NO_INHERIT	A child process will not inherit the NSG of the parent and, therefore, can be assigned to any eligible RAD on the system. This flag is valid only if both NSG_CREATE and NSG_GETBYPID are also specified.
NSG_CLEANUP	The NSG will be marked for automatic deletion when the nsg_perm.nattach member of the associated nsgid_ds structure transitions from nonzero to zero.

DESCRIPTION

The nsg_init( ) function looks up and possibly creates the NSG identified by the *key* parameter. The *flags* parameter supplies options for the lookup or create operation.

After creating a new NSG, the nsg_init( ) function initializes an associated nsgid_ds structure as follows:

- The nsg_perm.cuid and nsg_perm.uid members are set equal to the effective user ID of the calling process.
- The nsg_perm.cgid and nsg_perm.gid members are set equal to the effective group ID of the calling process.
- The low order nine bits of nsg_perm.mode are set equal to the low order nine bits of *flags*.
- The nsg_perm.nattach member is set to zero (or 1 if NSG_GETBYPID is specified in *flags*).

The nsg_perm.mode permissions control operations on NSGs as follows:

- Write permission is required to destroy the NSG or to set the owner IDs and permissions.
- Read permission is required to query the NSG status or membership roster.
- Execute permission is required to attach processes or threads to an NSG.

RETURN VALUES

NSG identifier of the NUMA Scheduling Group

Success.

-1

Failure. In this case, errno is set to indicate the error.

ERRORS

If the nsg_init( ) function fails, it sets errno to one of the following values for the specified condition:

[EACCES]

An NSG already exists for the specified key, but the caller does not have access based on the NSG's current permissions.

[EEXIST]

An NSG already exists for the specified key, but NSG_CREATE and NSG_EXCL flags were specified.

[ENOENT]

No NSG exists for the specified key, and the NSG_CREATE flag was not specified.

[ENOSPC]

No space exists for the new NSG specified by NSG_CREATE.

[ESRCH]

The NSG_GETBYPID flag was set and the process identified by *key* was not found.

[EINVAL]

One or more illegal values for *flags* was set.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_attach_thread`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

nsg_set – Set NUMA Scheduling Group owner and permissions (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int nsg_set(
          nsgid_t nsg,
          nsgid_ds_t *attrib);
```

PARAMETERS

nsg

Specifies the NUMA Scheduling Group (NSG).

attrib

Points to a structure containing owner and permission attributes.

DESCRIPTION

The `nsg_set( )` function sets the NSG owner IDs (user ID and group ID) and permissions (mode) by using the `nsgid_ds` structure pointed to by *attrib*.

RESTRICTIONS

The effective user ID of the calling process must be equal to the value of `nsg_perm.cuid` or `nsg_perm.uid` in the associated `nsgid_ds` structure, or the calling process must have write permissions to the NSG.

RETURN VALUES

`0`	Success. In this case, `nsg_set( )` set the NSG owner IDs and permissions as specified.
`-1`	Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `nsg_set( )` function fails, it sets `errno` to one of the following values for the specified condition:

[EACCES]

The calling process does not have write permission.

[EFAULT]

The *attrib* argument specifies an invalid address.

[EINVAL]

The *nsg* argument does not specify a valid NSG ID.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_init`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

nshmget – Returns (or creates) the ID for a shared memory region (`libnuma` library)

SYNOPSIS

```
#include <numa.h>
#include <sys/shm.h>

int nshmget(
          key_t key,
          size_t size,
          int shmflg,
          memalloc_attr_t *attr);
```

PARAMETERS

key

Specifies the key that identifies the shared memory region. The value for the *key* parameter can be `IPC_PRIVATE` or a random number other than zero (0). If the value of *key* is `IPC_PRIVATE`, it can be used to assure the return of a new, unused shared memory region.

size

Specifies the minimum number of bytes to allocate for the region.

shmflg

Specifies the creation flags. See `shmget`(2) for a description of these flags.

attr

Points to a memory allocation policy and attributes structure. If the specified *key* does not exist, and a shared memory region is created, these attributes will be assigned to the memory object created to manage the shared memory region.

DESCRIPTION

If the *attr* argument is NULL, the `nshmget( )` function behaves identically to the `shmget( )` function.

If the *attr* argument is non-NULL, it points to a memory allocation policy and attributes structure that specifies where the pages should be allocated for a newly created shared memory region. To change the policy of an existing shared memory region, use the `nmadvise( )` function.

If the `mattr_policy` member of the structure pointed to by *attr* is `MPOL_DIRECTED` and the `mattr_rad` member is `RAD_NONE`, the system will choose the Resource Affinity Domain (RAD) where the pages of the shared memory region will be allocated from among the RADs specified in the `mattr_radset` member of **attr*. If the `mattr_radset` member is the empty set, the system will select a RAD for the memory object from among all of the RADs in the caller's partition, and the overflow set will be the empty set.

RETURN VALUES

ID of a shared memory region

Success.

`-1`

Failure. In this case, `errno` is set to indicate the error.

ERRORS

The `nshmget( )` function returns errors for all the conditions that are documented for the `shmget( )` function. In addition, the `nshmget( )` function sets `errno` for the following:

[EFAULT]

A non-NULL *attr* argument points to an invalid address.

[EINVAL]

The structure pointed to by the *attr* argument contains an invalid memory allocation policy or an invalid RAD number. (The RAD number is less than 0 or greater than *nrads*.) This error can also occur if the memory allocation policy is `MPOL_STRIPED`, but the specified stride (stripe width) is 0 pages.

SEE ALSO

Functions: `shmget`(2), `nmadvise`(3), `numa_intro`(3)

Files: numa_types(4), shmid_ds(4)

NAME

pthread_nsg_attach – Attaches a thread to a NUMA Scheduling Group (`libpthread` library)

SYNOPSIS

```
#include <numa.h>
int pthread_nsg_attach(
          nsgid_t nsg,
          pthread_t thread,
          long flags);
```

PARAMETERS

nsg

Specifies the NUMA Scheduling Group (NSG) to which the thread will belong.

thread

Identifies the thread to attach to the NSG.

flags

Specifies options (bit mask) that affect the attachment. See DESCRIPTION for details.

DESCRIPTION

The `pthread_nsg_attach( )` function attaches the thread specified by the *thread* argument to a NUMA Scheduling Group (NSG) specified by the *nsg* argument. An NSG is a set of processes and/or threads that will be constrained to reside on the same Resource Affinity Domain (RAD). That is, the "home RAD" for all of the processes/threads in an NSG will be the same, and the entire group will be migrated together, if at all. The thread specified by *thread* will be removed from any NSG of which it might currently be a member, before being added to the specified NSG.

If the specified thread resides on a different RAD, the thread will be reassigned to a single RAD. The home RAD for the thread will be selected as follows:

- If the specified NSG already has processes/threads attached, the home RAD for the specified thread will be the home RAD for that NSG.
- If the specified NSG is empty, the home RAD for the thread will be selected based on the setting of the *flags* argument.

The following options are defined for the *flags* argument:

`NSG_INSIST`

The requested attachment and any implied reassignments are mandatory, overriding any prior binding of the specified thread. If this option is not set, and the thread is bound (`RAD_INSIST`) to a different RAD such that the system cannot honor the request, the request will fail.

`NSG_SMALLMEM`

The thread has small memory requirements, so the system should favor (for the home RAD) those RADs with light CPU loads, independent of their available memory. This flag applies only when attaching to an empty NSG.

`NSG_LARGEMEM`

The thread has large memory requirements, so the system should favor (for the home RAD) those RADs with more available memory, independent of their CPU loads. This flag only applies when attaching to an empty NSG.

`NSG_MIGRATE`

Arrange for the existing memory (stack pages) of a thread that is assigned a new home RAD to be migrated to the new RAD. If omitted, only newly allocated pages will be allocated on the new home RAD. Existing pages will migrate if/when they experience a high rate of remote cache misses. Migration will occur only for pages-in-memory objects that have inherited the process's default memory allocation policy.

`NSG_WAIT`

Wait for the requested memory migration to be completed. Effectively, this flag specifies "migrate now!".

RETURN VALUES

`0`	Success.
Integer value	Failure. In this case, the returned integer indicates the type of error. Possible errors include the following:

[EACCES]

The caller does not have execute permission required to attach the thread based on the NSG's permissions.

[EBUSY]

The thread specified by *thread* is hard attached (`RAD_INSIST`) to RADs or has memory wired (locked) on its current RAD such that it cannot be migrated to the common RAD selected for the NSG.

[EINVAL]

The value of the *flags* argument is invalid.

[ENOMEM]

`NSG_INSIST` and `NSG_MIGRATE` were specified and no RAD can be found with sufficient memory to accommodate the resulting group.

[ESRCH]

The thread specified by the *thread* argument does not exist.

ERRORS

None.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_get`(3), `numa_intro`(3)

Files: numa_types(4)

NAME

pthread_nsg_detach – Detaches a thread from a NUMA Scheduling Group (`libpthread` library)

SYNOPSIS

```
#include <numa.h>
int pthread_nsg_detach(
          pthread_t thread);
```

PARAMETERS

thread

Identifies the thread to detach from a NUMA Scheduling Group (NSG).

DESCRIPTION

The `pthread_nsg_detach( )` function detaches the thread specified by the *thread* argument from the NSG to which it is attached. The function does not reassign the thread to a new NSG.

RETURN VALUES

`0`	Success.
Integer value	Failure. In this case, the integer value indicates the type of error. Possible errors include the following:

[EACCES]

Based on the NSG's permissions, the caller does not have execute permission, which is required to detach a thread.

[EBUSY]

The specified thread is hard attached (`RAD_INSIST`) to a RAD or has memory wired (locked) on its current RAD such that it cannot be migrated to a common RAD selected for the NSG.

[EINVAL]

The value of the *flags* argument is invalid.

[ENOMEM]

`NSG_INSIST` and `NSG_MIGRATE` were specified and no RAD can be found with sufficient memory to accommodate the resulting NSG.

[ESRCH]

The *thread* argument specifies a thread that does not exist.

ERRORS

None.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_get`(3), `numa_intro`(3)

Files: `numa_types`(4)

NAME

pthread_nsg_get – Gets the list of threads in a NUMA Scheduling Group (`libpthread` library)

SYNOPSIS

```
#include <numa.h>
int pthread_nsg_get(
          nsgid_t nsg,
          pthread_t *list,
          long size);
```

PARAMETERS

nsg

Specifies the NUMA Scheduling Group (NSG) from which to retrieve the list of threads.

list

Specifies the address of the array of thread identifiers where the list of threads will be written.

size

Specifies the size (in number of thread identifiers) of the array where the list of threads will be written.

DESCRIPTION

The `pthread_nsg_get( )` returns into the buffer pointed to by *list* an array of `nsg_thread` structures containing the process ids and thread indexes for all threads attached to the specified NSG. The *list* argument specifies the number of `nsg_thread` structures that the array must accommodate.

To obtain the value for *size*, the application can first call `pthread_nsg_get( )` with *list* set to null and read in the value of the `nsg_nthread` member in the `nsgid_ds` structure that the call returns. On the second call to `pthread_nsg_get( )`, the application uses the `nsg_nthread` value for *size* and includes the appropriate value for *list*.

RESTRICTIONS

The effective user ID of the calling process must be equal to the value of `nsg_perm.cuid` or `nsg_perm.uid` in the associated `nsgid_ds` structure; or the calling process must have read permissions to the NSG.

RETURN VALUES

`0`

Success.

Integer value

Failure. In this case, the returned integer indicates the type of error. Possible errors are as follows:

[EACCES]

The calling process does not have read permission on the NSG.

[EINVAL]

The *nsg* argument does not specify a valid NSG identifier, or the *nsg* argument is not a valid address.

ERRORS

None.

SEE ALSO

Functions: `nsg_attach_pid`(3), `nsg_get`(3), `numa_intro`(3), `pthread_nsg_attach`(3)

Files: `numa_types`(4)

NAME

pthread_rad_attach, pthread_rad_bind – Attaches or binds a thread to a NUMA Resource Affinity Domain (`libpthread` library)

SYNOPSIS

```
#include <numa.h>
int pthread_rad_attach(
          pthread_t thread,
          radset_t radset,
          ulong_t flags);
int pthread_rad_bind(
          pthread_t thread,
          radset_t radset,
          ulong_t flags);
```

PARAMETERS

thread

Identifies the thread to be attached or bound to the specified set of Resource Affinity Domains (RADs).

radset

Identifies the RAD set to which the thread is to be attached or bound.

DESCRIPTION

The `pthread_rad_attach( )` function attaches the thread specified by the *thread* argument to the RAD set specified by the *radset* argument.

The `pthread_rad_bind( )` function binds the specified thread to the specified RAD set.

While both functions assign a home RAD for the thread, an attach operation allows remote execution on other RADs while a bind operation restricts execution to the home RAD. In the following paragraphs, the term “assign” is used when the description refers equally to both the attach and the bind operations.

The home RAD for the thread will be selected by the system scheduler from among the RADs included in *radset* based on current system load balance

and the *flags* argument. The overflow set (`mattr_radset`) for the thread will be set to *radset*.

The following symbolic values are defined for the *flags* argument:

`RAD_INSIST`

The requested assignments are mandatory. If this option is not set, the system will consider the request to be a "hint" and may take no action for the specified thread.

`RAD_SMALLMEM`

The thread has small memory requirements, so the system should (for the home RAD) favor those RADs with light CPU loads, independent of their available memory.

`RAD_LARGEMEM`

The thread has large memory requirements, so the system should (for the home RAD) favor those RADs with more available memory, independent of their CPU loads.

If the caller does not have partition administration privilege and if the *radset* argument contains RADs that are not in the caller's partition, an error will be returned.

NOTES

The value for the *radset* argument could be obtained from an `nloc( )` call to assign or migrate the process to a RAD close (or closer) to a particular resource. When obtained in this manner, the *radset* value will identify RADs that were in the caller's partition at the time of the `nloc( )` call. The partition configuration could change between the call to `nloc( )` and a subsequent call to `pthread_rad_attach( )`, resulting in an error. An application should be prepared to handle this error, even though it should not be a frequent occurrence.

RETURN VALUES

`0`	Success.
Integer value	Failure. In this case, the integer value indicates the type of error. Possible errors are as follows:

[EBUSY]

The thread is hard attached (RAD_INSIST) to RADs or has memory wired (locked) on its current RAD such that it cannot be migrated to the specified RAD set.

[EINVAL]

One of more of the RADs in the *radset* argument or options in the *flags* argument are invalid.

[ENOMEM]

`RAD_INSIST` and `RAD_MIGRATE` were specified, and the thread cannot be migrated because insufficient memory exists on RADs in the specified RAD set.

[EPERM]

The caller does not have appropriate privileges to assign threads to RADs in the specified RAD set.

[ESRCH]

The thread specified by *thread* does not exist.

ERRORS

None.

SEE ALSO

Functions: `nloc`(3), `pthread_rad_detach`(3)

NAME

pthread_rad_detach – Detach a thread from its Resource Affinity Domain (`libpthread` library)

SYNOPSIS

```
#include <numa.h>
int pthread_rad_detach(
          pthread_t thread);
```

PARAMETERS

thread

Identifies the thread to detach from its current RAD

DESCRIPTION

The `pthread_rad_detach( )` function detaches the thread specified by the *thread* argument from its current Resource Affinity Domain. The thread is free to be scheduled in any RAD available to the process.

RETURN VALUES

`0`

Success. In this case, the specified *thread* is detached from its RAD.

Integer value

In this case, the integer value indicates the type of error. Possible errors are as follows:

[ESRCH]

The thread specified in *thread* does not exist.

ERRORS

None.

SEE ALSO

Functions: `nloc`(3), `pthread_rad_attach`(3), `pthread_rad_bind`(3)

NAME

rad_attach_pid, rad_bind_pid – Attaches or binds a process to a Resource Affinity Domain by process ID (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int rad_attach_pid(
          pid_t pid,
          radset_t radset,
          ulong_t flags);
int rad_bind_pid(
          pid_t pid,
          radset_t radset,
          ulong_t flags);
```

PARAMETERS

pid

Identifies the process to be attached or bound to the specified set of Resource Affinity Domains (RADs).

radset

Specifies the RAD set to which the process will be attached or bound.

flags

Specifies options (a bit mask) that affect the attachment or binding operation. See DESCRIPTION for details.

DESCRIPTION

The `rad_attach_pid( )` function attaches the process specified by *pid* to the set of RADs specified by *radset*.

The `rad_bind_pid( )` function binds the process specified by *pid* to the set of RADs specified by *radset*.

While both functions assign a “home” RAD for the process, an attach operation allows remote execution on other RADs while a bind operation restricts execution to the “home” RAD. For both functions, if the *pid*

argument is NULL, the call is self-directed. That is, the function behaves as if *pid* identified the calling process.

The memory allocation policy for the process will be set to MPOL_THREAD. The home RAD for the process will be selected by the system scheduler from among the RADs included in *radset* and will be based on current system load balance and the *flags* argument. The overflow set (mattr_radset) for the process will be set to *radset*. If the process has multiple threads, then any of those threads that have inherited the process's default memory allocation policy will be attached or bound by using the same new memory allocation policy as used for the process that contains them.

The threads of the specified process will be scheduled on one of the CPUs associated with the selected RAD, except for threads that have been explicitly bound to some other processor. The CPU will be selected by the scheduler from among those CPUs associated with the selected RAD in the process's partition. (This partition might not be the same as the caller's partition if the caller has appropriate privilege.) The selection will be determined by the loading of the CPUs.

The following options are defined for the *flags* argument:

RAD_NO_INHERIT

Any processes later forked by the specified process can be assigned to any RAD on the system, and might not inherit its parent's home RAD assignment; that is, the child processes might not be assigned to the same home RAD as the parent. This allows the system to assign a home RAD to the child process depending on available resources.

Normally, child processes do inherit the assignments and attributes of the parent process.

By default, processes that are later forked by the process specified in a rad_attach_pid( ) or rad_bind_pid( ) call inherit the RAD assignment of their parent.

RAD_INSIST

The requested attachments or bindings are mandatory. If this option is not set, the system will consider the request to be a "hint" and may take no action for the specified process or, if applicable, any child processes that the specified process contains.

RAD_SMALLMEM

The process has small memory requirements, so the system should favor (for the home RAD) those RADs with light CPU loads, independent of their available memory.

RAD_LARGEMEM

The process has large memory requirements, so the system should favor (for the home RAD) those RADs with more available memory, independent of their CPU loads.

RAD_MIGRATE

Arrange for existing memory of the process to be migrated to the new home RAD. If RAD_MIGRATE is omitted, only newly allocated pages will be allocated on the new home RAD. Existing pages will migrate if or when they experience a high rate of remote cache misses. Migration will occur only for pages in memory objects that have inherited the process's default memory allocation policy.

RAD_WAIT

Wait for the requested memory migration to be completed. Effectively, this specifies "migrate now!".

If the caller does not have partition administration privilege and if *pid* is not in the caller's partition, or if the *radset* argument contains RADs that are not in the caller's partition, an error will be returned.

The value for the *radset* argument could be obtained from a prior call to nloc( ) that assigned or migrated the process to a RAD close or closer to a particular resource. When obtained this way, *radset* will contain only the RADs in the caller's partition at the time of the nloc( ) call. The partition configuration could change between a call to nloc( ) and a subsequent call to rad_attach_pid( ) or rad_bind_pid( ), resulting in an error. This error is not likely to occur often, but a robust application should handle it.

RETURN VALUES

0 Success.

`-1` Failure. In this case, the functions set `errno` to indicate the error.

ERRORS

If either of these functions fail, `errno` is set to one of the following values for the condition specified:

[EBUSY]

`RAD_INSIST` and `RAD_MIGRATE` were specified and the specified process cannot be migrated for some reason. For example, memory is wired (locked) on the process's current RAD.

[EFAULT]

The *radset* argument points to an invalid address.

[EINVAL]

One or more of the RADs in the *radset* argument or options in the *flags* argument are invalid.

[ENOMEM]

`RAD_INSIST` and `RAD_MIGRATE` were specified and the specified process cannot be migrated because insufficient memory exists on the specified RAD set.

[EPERM]

The real or effective user ID of the caller does not match the real or effective user ID of the specified process, or the caller does not have appropriate privileges to assign processes to RADs.

[ESRCH]

The process specified by *pid* does not exist.

SEE ALSO

Functions: `nloc`(3), `rad_detach_pid`(3)

NAME

rad_detach_pid – Detach a process from a Resource Affinity Domain by pid (`libnuma` library)

SYNOPSIS

```
#include <numa.h>

int rad_detach_pid(
          pid_t pid);
```

PARAMETERS

pid

Specifies a process identifier (pid) to detach from a RAD set.

DESCRIPTION

The `rad_detach_pid( )` function frees a process that has been bound or attached to a RAD through the functions `rad_bind_pid( )` or `rad_attach_pid( )`, respectively. If the *pid* argument is NULL, the call is self-directed. That is, the function behaves as if the calling process's pid were specified. Calling `rad_detach_pid( )` for a process that is not attached or bound is not considered to be an error.

RETURN VALUES

`0`

Success. In this case, `rad_detach_pid( )` detaches the pid specified by *pid* from the RAD set.

`-1`

Failure. In this case, `errno` is set to indicate the error.

ERRORS

If the `rad_detach_pid( )` function fails, `errno` is set to one of the following values for the reasons specified.

[ESRCH]

The process specified in *pid* does not exist.

[EPERM]

The real or effective user ID of the caller does not match the real or effective user ID of the process pid, or the caller does not have appropriate privileges to free processes from RADs.

SEE ALSO

Functions: `rad_attach_pid`(3), `rad_bind_pid`(3)

NAME

rad_foreach – Enumerates the members of a Resource Affinity Domain (`libnuma` library)

SYNOPSIS

```
#include <numa.h>
rad_cursor_t cursor = SET_CURSOR_INIT;
int rad_foreach(
          radset_t radset,
          unsigned int flags,
          radset_cursor_t *cursor);
```

PARAMETERS

radset

Specifies a set of Resource Affinity Domains (RADs) whose members are to be enumerated.

flags

Specifies one or more flags that control the processing of RAD members in the set. The following symbolic values are defined for *flags*:

`SET_CURSOR_FIRST`

Initialize the cursor to the first member of the set before scanning.

`SET_CURSOR_WRAP`

Wrap around to the beginning of the RAD set when scanning for members.

`SET_CURSOR_CONSUME`

Consume the set members; in other words, remove members from the set as they are found.

cursor

Specifies an opaque type that records the position in a set for subsequent invocations of the `rad_foreach( )` function.

DESCRIPTION

The rad_foreach( ) function scans the specified RAD set, starting at the position saved in *cursor*, for members of the set and returns the first member found. If the SET_CURSOR_FIRST flag is set, the *cursor* is initialized to the beginning of the set before starting the scan. If no members are found, the rad_foreach( ) function will return RAD_NONE.

If the SET_CURSOR_WRAP flag is set, the scan will wrap from the end of the set to the beginning searching for a member to return. Otherwise, a one-pass scan is performed, and when the end of the set is reached, cursor( )is left positioned at the end of the set. From then on, the rad_foreach( ) function will continue to return RAD_NONE until *cursor* is reinitialized, either by specifying the SET_CURSOR_FIRST flag or by specifying the SET_CURSOR_WRAP flag.

If the SET_CURSOR_CONSUME flag is set, the member returned, if any, will be removed from the set.

The *cursor* variable may be initialized to the value SET_CURSOR_INIT. This is equivalent to setting SET_CURSOR_FIRST on the initial call to rad_foreach( ).

NOTES

Although DESCRIPTION discusses the “beginning” and “end” of the set, and wrapping from the end to the beginning, RAD sets are conceptually unordered. Thus, these end points are arbitrary points in the set that exist to ensure that each member is returned only once per pass through the set. Applications should not depend on a numeric ordering of the returned member IDs.

RETURN VALUES

This function returns either the next member in the RAD set, starting at the position in *cursor*, or RAD_NONE (if there is no next member). Execution of this function is always successful.

ERRORS

None.

EXAMPLES

See EXAMPLES in `radsetops`(3) for a sample program that uses the `rad_foreach( )` function.

SEE ALSO

Functions: `numa_intro`(3), `radsetops`(3)

Files: `numa_types`(4)

NAME

rad_fork – Creates a new process on a Resource Affinity Domain (`libnuma` library)

SYNOPSIS

```
#include <numa.h>
pid_t rad_fork(
          radid_t radid,
          ulong_t flags);
```

PARAMETERS

radid

Identifies a Resource Affinity Domain (RAD) on which to allocate data and schedule threads for a new process.

flags

Specifies options (a bit mask) that affect the attachment or binding operation. See DESCRIPTION for details.

DESCRIPTION

The `rad_fork( )` function behaves the same as `nfork( )` when the latter specifies a resource type of `R_RAD` and a resource descriptor that points to a RAD set containing a single RAD identifier. For a description of this behavior, refer to the description of the `R_RAD` resource type in `nfork`(3).

The following option is specified for the *flags* argument:

`RAD_NO_INHERIT` The child process might not be assigned to the same home RAD as its parent process. Allows the system to assign a home RAD to the child process depending on available resources.

Normally, child processes do inherit the assignments and attributes of the parent process.

RETURN VALUES

`0` Success (returned to the child process). In this case, the function also returns the process ID of the child process to the parent process.

The child process and all of its data structures are allocated on the RAD specified by the *radid* argument. In addition, the initial thread of the child process is scheduled on one of the CPUs in the specified RAD.

`-1` Failure (returned to the parent process). In this case, no child process is created and the function sets `errno` to indicate the error.

ERRORS

[EAGAIN]

The limit on the total number of processes executing for a single user would be exceeded. This limit can be exceeded by a process with superuser privilege.

[EINVAL]

The *radid* argument specifies an invalid RAD identifier.

[ENOMEM]

There is insufficient memory to create this process.

SEE ALSO

Functions: `nfork`(3), `nloc`(3), `numa_intro`(3), `radsetops`(3)

Files: `numa_types`(4)

NAME

rad_get_current_home – Returns the caller's home Resource Affinity Domain (`libnuma` library)

SYNOPSIS

```
#include <numa.h>
radid_t rad_get_current_home( void );
```

PARAMETERS

None.

DESCRIPTION

The `rad_get_current_home( )` function returns the home Resource Affinity Domain (RAD) of the caller.

RESTRICTIONS

As is true for many system information queries, the data returned by the `rad_get_current_home( )` function may be stale by the time it is returned to or used by the caller. For example, migration of the process to a different RAD could occur after the "current home RAD" is fetched.

RETURN VALUES

This function returns the caller's home RAD and always completes successfully.

ERRORS

None.

SEE ALSO

Functions: `cpu_get_current`(3), `cpu_get_rad`(3), `nloc`(3), `numa_intro`(3), `rad_attach_pid`(3)

Files: `numa_types`(4)

NAME

rad_get_num, rad_get_cpus, rad_get_freemem, rad_get_info, rad_get_max, rad_get_physmem, rad_get_state – Query resource complements of a Resource Affinity Domain (`libnuma`)

SYNOPSIS

```
#include <numa.h>

int rad_get_cpus(
          radid_t rad,
          cpuset_t cpuset);
ssize_t rad_get_freemem(
          radid_t rad);
int rad_get_info(
          radid_t rad,
          rad_info_t *info);
int rad_get_max( void );
int rad_get_num( void );
ssize_t rad_get_physmem(
          radid_t rad);
ssize_t rad_get_state(
          radid_t rad);
```

PARAMETERS

cpuset	Specifies a buffer to receive the CPU set assigned to the specified Resource Affinity Domain (RAD) in the caller's partition
info	Points to a buffer to receive information about the specified RAD.
rad	Identifies the RAD for which the resource complement is being requested.

DESCRIPTION

A Resource Affinity Domain (RAD) is a collection of resources that are related by the platform hardware topology. The collection of processors and I/O buses connected to a local memory of a NUMA platform, plus the local

memory itself, comprise a RAD. More generally, a RAD may be characterized as a set of resources that are within some "distance" of each other.

The `rad_get_info( )` function stores in the buffer pointed to by *info*, a `rad_info_t` structure containing information about the RAD specified by the *radid* argument. This information includes the state of the RAD, the amount of memory in the RAD, and the CPUs it contains. The remaining functions on this reference page return individual members of the `rad_info_t` structure.

The `rad_get_cpus( )` function stores in the buffer specified by *cpuset* the set of CPUs in the specified RAD that are assigned to the caller's partition.

The `rad_get_freemem( )` function returns a snapshot of the amount of free memory (pages) in the specified RAD in the caller's partition.

The `rad_get_max( )` function returns the maximum number of RADs on the system.

The `rad_get_num( )` function returns the number of RADs in the caller's partition.

The `rad_get_physmem( )` function returns the amount of physical memory (pages) assigned to the specified RAD in the caller's partition.

The `rad_get_state( )` function returns the current state of the RAD specified by the *radid* argument. The possible RAD state values are:

`RAD_ONLINE` — The specified RAD exists and is on line. Processes and threads may be assigned to the RAD and memory may be allocated there.

`RAD_OFFLINE` — The specified RAD exists but is not currently on line. Neither processes nor threads may be assigned to this RAD, and no memory may be allocated there. However, the RAD's resource complement may be queried.

Note

Currently, RAD state is always set to `RAD_ONLINE`; therefore, consider this function as being reserved for future use.

Note that prior to calling any of the rad_get_*( ) functions, the application must set the rinfo_version field in the rad_info_t structure to RAD_INFO_VERSION. The CPU set (*cpuset*) stored in this structure must have been created by the application prior to the call. If zero is specified for *cpuset*, the function does not fill in data for the CPU set.

RESTRICTIONS

As with many queries of system information, the data returned by these functions may be stale by the time it is returned to or used by the calling application.

RETURN VALUES

The rad_get_info( ) and rad_get_cpus( ) functions return the following values:

0 or positive integer	Success. In this case, the integer value is the number of CPUs in the specified RAD.
-1	Failure. In this case, errno is set to indicate the error.

The rad_get_freemem( ) and rad_get_physmem( ) functions return the following:

Number of pages of memory	Success. Depending on the function, this value is the amount of free memory for the specified RAD or the amount of physical memory assigned to the RAD.
(ssize_t)-1	Failure. In this case, errno is set to indicate the error.

The rad_get_num( ) and rad_get_max( ) functions return the number of RADs in the caller's partition or on the system, respectively. There is no value defined to indicate failure for these functions.

The rad_get_state( ) function always returns a state value. There is no value defined to indicate failure for this function.

ERRORS

The `rad_get_cpus( )`, `rad_get_info( )`, `rad_get_freemem( )`, and `rad_get_physmem( )` functions set `errno` to one of the following values for the specified condition:

[EFAULT]	The *cpuset* argument indirectly points to an invalid address, or the specified CPU set does not exist, possibly because it was not created by a call to `cpusetcreate( )`.
[EINVAL]	The *rad* argument specifies a RAD that does not exist.
[EPERM]	The version number specified for the `rinfo_version` field in the *info* argument is not recognized by the system.

EXAMPLES

The following example prints data returned by a call to `rad_get_info( )`:

```
#include <sys/errno.h>
#include <numa.h>

int
print_rad_info(radid_t rad)
{
        rad_info_t radinfo;

        /* Create a cpuset for the radinfo struct. */
        cpusetcreate(&radinfo.rinfo_cpuset);

        radinfo.rinfo_version = RAD_INFO_VERSION;

        /* Fetch the data */
        if (rad_get_info(rad, &radinfo) == -1) {
                perror("rad_get_info");
                return -1;
        }

        /* Simple data types can be printed directly. */
        printf("rinfo_radid = %d\n", radinfo.rinfo_radid);
        printf("rinfo_state = %d\n", radinfo.rinfo_state);
        printf("rinfo_physmem = 0x%lx pages\n", radinfo.rinfo_physmem);
        printf("rinfo_freemem = 0x%lx pages\n", radinfo.rinfo_freemem);
        printf("\ncpuset members: ");

        /* Complex datatypes (cpuset) need to be enumerated. */
```

```
        while (1) {
                cpuid_t id;
                int flags = SET_CURSOR_CONSUME;
                cpu_cursor_t cpu_cursor = SET_CURSOR_INIT;

                id = cpu_foreach(radinfo.rinfo_cpuset, flags, &cpu_cursor);

                if (id == CPU_NONE) {
                        printf("\n");
                        break;
                } else {
                        printf("%3d ", id);
                }
        }

        /* Destroy cpuset */
        cpusetdestroy(&radinfo.rinfo_cpuset);

        return 0;
}
```

SEE ALSO

Functions: cpu_foreach(3), cpusetcreate(3), nloc(3), numa_intro(3)

Files: numa_types(4)

NAME

radsetops: radaddset, radandset, radcopyset, radcountset, raddelset, raddiffset, rademptyset, radfillset, radisemptyset, radismember, radorset, radsetcreate, radsetdestroy, radxorset – Perform operations on a set of Resource Affinity Domains (libnuma library)

SYNOPSIS

```
#include <radset.h>

int radaddset(
          radset_t set,
          radid_t radid);

int radandset(
          radset_t set_src1,
          radset_t set_src2,
          radset_t set_dst);

int radcopyset(
          radset_t set_src,
          radset_t set_dst);

int radcountset(
          radset_t set);

int raddelset(
          radset_t set,
          radid_t radid);

int raddiffset(
          radset_t set_src1,
          radset_t set_src2,
          radset_t set_dst);

int rademptyset(
          radset_t set);

int radfillset(
          radset_t set);

int radisemptyset(
          radset_t set);

int radismember(
          radset_t set,
          radid_t radid);

int radorset(
          radset_t set_src1,
          radset_t set_src2,
          radset_t set_dst);

int radsetcreate(
          radset_t *set);

int radsetdestroy(
          radset_t *set);
```

```
int radxorset(
          radset_t set_src1,
          radset_t set_src2,
          radset_t set_dst);
```

PARAMETERS

radid

Identifies the RAD for which the function is requesting information or on which the function operates.

set

Specifies or points to a set of Resource Affinity Domains (RADs) on which the function operates.

set_src[*n*]

Specifies, depending on the function, one of the following:

- A source RAD set that the function copies to a destination RAD set
- One of two RAD sets for which the function will find a logical difference
- One of two RAD sets on which the function will perform a logical AND, OR, or XOR operation

set_dst

Specifies the destination RAD set where the function stores the results of the logical operation it performs.

DESCRIPTION

The `radsetops` primitives manipulate a specified set of Resource Affinity Domains (RADs). These functions operate on data objects (of type `radset_t`) that are created by calls to `radsetcreate( )`.

The `radsetcreate( )` function allocates, and sets to empty, the specified RAD set.

The `radsetdestroy( )` function releases the RAD set memory allocated by `radsetcreate( )`.

The `radfillset( )` function initializes the specified RAD set, such that all RADs that are currently configured in the caller's partition are included in that set.

The `radcountset( )` function returns the number of members in the specified RAD set.

The `rademptyset( )` function reinitializes the specified RAD set, such that no RADs are included in that set.

The `radisemptyset( )` function tests whether the specified RAD set is empty.

The `radismember( )` function tests whether the specified RAD is a member of the specified RAD set.

The `radaddset( )` and `raddelset( )` functions respectively add or delete the specified RAD from the specified RAD set.

The `raddiffset( )` function finds the logical difference between the RAD sets specified by the arguments *set_src1* and *set_src2* and stores the result in the RAD set specified by *set_dst*. (The result is made up of those members included in *set_src1* but not in *set_src2*.)

The `radandset( )`, `radorset( )`, and `radxorset( )`functions respectively perform a logical AND, OR, or XOR operation on the RAD sets specified by the arguments *set_src1* and *set_src2*, storing the result in the RAD set specified by *set_dst*.

RETURN VALUES

The `radisemptyset( )` and `radismember( )` functions return the following values:

`1` Success (True).

`0` Success (False).

`-1` Failure. In this case, `errno` is set to indicate the type of error.

The remaining functions return the following values:

`0` Success.

`-1` Failure. In this case, `errno` is set to indicate the type of error.

ERRORS

If the `radcountset( )`, `rademptyset( )`, `radfillset( )`, `radisemptyset( )`, `radorset( )`, `radxorset( )`, `radandset( )`, `raddiffset( )`, and `radcopyset( )` functions fail, they set `errno` to the following value for the specified condition specified:

[EINVAL]

The specified RAD set is invalid, possibly not created by `radsetcreate( )`.

If the `radsetcreate( )` and `radsetdestroy( )` functions fail, they set `errno` to one of the following values for the condition specified:

[EFAULT]

The *set* argument points to an invalid address.

[ENOMEM]

(`radsetcreate( )` only) No memory could be allocated for the RAD set.

If the `radaddset( )`, `raddelset( )`, and `radismember( )` functions fail, they set `errno` to one of the following values for the condition specified:

[EINVAL]

The specified RAD set is invalid, possibly not created by `radsetcreate( )`.

[EDOM]

The value of *radid* is an invalid or unsupported RAD identifier.

EXAMPLES

The following example demonstrates various operations on RAD sets:

```
#include <radset.h>

int
main()
{
```

```
    radset_t radset, radset2;

    /* Create radsets - initialized as empty */
    radsetcreate(&radset);
    radsetcreate(&radset2);

    /* demonstrate radset operations */

    /* add rad 0 to radset */
    if (radaddset(radset, 0) == -1) {
            perror("radaddset");
            return 0;
    }


    /* copy radset to radset2 */
    if (radcopyset(radset, radset2) == -1) {
            perror("radcopyset");
            return 0;
    }

    /* add rad 1 to radset */
    if (radaddset(radset, 1) == -1) {
            perror("radaddset");
            return 0;
    }

    /* store the difference of radset and radset2 in radset */
    if (raddiffset(radset, radset2, radset) == -1) {
            perror("raddiffset");
            return 0;
    }

    /* Enumerate radset. */
    while (1) {
            radid_t id;
            int flags = SET_CURSOR_CONSUME;
            rad_cursor_t rad_cursor = SET_CURSOR_INIT;

            id = rad_foreach(radset, flags, &rad_cursor);

            if (id == RAD_NONE) {
                    printf("\n");
                    break;
            } else {
                    printf("%3d ", id);
            }
    }

    /* Destroy radset and radset2*/
    radsetdestroy(&radset);
    radsetdestroy(&radset2);
    return 0;
}
```

SEE ALSO

Functions: `numa_intro`(3), `rad_foreach`(3)

Files: `numa_types`(4)

Index

R

S

T

V