Guide to DECthreads

Order Number: AA–Q2DPD–TK

December 1997

This guide reviews the principles of multithreaded programming, as reflected in the IEEE POSIX 1003.1c-1995 standard, and provides implementation guidelines and reference information for DECthreads, DIGITAL's Multithreading Run-Time Library.

Revision/Update Information: This guide supersedes the *Guide to DECthreads*, printed March 1996.

Product Version: DIGITAL UNIX Version 4.0D

Digital Equipment Corporation
Maynard, Massachusetts

December 1997

The following are trademarks of Digital Equipment Corporation: AlphaServer, AXP, DEC, DEC Ada, DECdirect, Ladebug, DECthreads, DIGITAL, DIGITAL UNIX, OpenVMS, VAX, VAX Ada, VAX MACRO, VMS, ULTRIX, and the DIGITAL logo.

The following are third-party trademarks:

IEEE is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

Microsoft, MS, MS–DOS, Win32, and Windows 95 are registered trademarks, and Windows NT is a trademark of Microsoft Corporation.

Motif, OSF, OSF/1, OSF/Motif, and Open Software Foundation are registered trademarks of the Open Software Foundation, Inc.

POSIX is a registered trademark of the IEEE.

Internet is a registered trademark of Internet, Inc.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company, Ltd.

All other trademarks and registered trademarks are the property of their respective holders.

All other trademarks and registered trademarks are the property of their respective holders.

ZK6493

(This document is available on CD–ROM.)

Contents

Preface xv

Part I DECthreads Overview and Programming Guidelines

1 Introducing DECthreads for Multithreaded Programming

1.1 Advantages of Using Threads 1–1
1.2 Overview of Threads 1–2
1.3 Thread Execution 1–4
1.4 Functional Models for Multithreaded Programming 1–6
1.4.1 Boss/Worker Model 1–6
1.4.2 Work Crew Model 1–6
1.4.3 Pipelining Model 1–7
1.4.4 Combination of Functional Models 1–8
1.5 Potential Issues for Multithreaded Programs 1–8
1.6 DECthreads Libraries and Interfaces 1–9
1.6.1 Multithreading pthread Interface 1–10
1.6.1.1 Optionally Implemented POSIX.1c-1995 Routines 1–20
1.6.2 Thread-Independent Services Interface 1–20
1.6.3 Undocumented and Obsolete DECthreads Interfaces 1–23
1.6.3.1 Undocumented But Supported cma Interface 1–23
1.6.3.2 Obsolete d4 Interfaces 1–24

2 DECthreads Objects and Operations

2.1 Threads, Mutexes, and Condition Variables 2–1
2.2 Attributes Objects 2–1
2.3 Thread Objects and Operations 2–2
2.3.1 Creating and Starting a Thread 2–3

2.3.2 Setting the Attributes of a New Thread 2–4
2.3.2.1 Setting the Inherit Scheduling Attribute 2–5
2.3.2.2 Setting the Scheduling Policy Attribute 2–5
2.3.2.2.1 Techniques for Setting the Scheduling Policy Attribute 2–6
2.3.2.2.2 Comparing Throughput and Real-Time Policies 2–6
2.3.2.2.3 Portability of Scheduling Policy Settings 2–7
2.3.2.3 Setting the Scheduling Parameters Attribute 2–7
2.3.2.4 Setting the Stacksize Attribute 2–8
2.3.2.5 Setting the Stack Address Attribute 2–8
2.3.2.6 Setting the Guardsize Attribute 2–9
2.3.2.7 Setting the Contention Scope Attribute 2–9
2.3.3 Terminating a Thread 2–11
2.3.3.1 Cleanup Handlers 2–14
2.3.4 Detaching and Destroying a Thread 2–14
2.3.5 Joining With a Thread 2–15
2.3.6 Scheduling a Thread 2–16
2.3.6.1 Calculating the Scheduling Priority 2–16
2.3.6.2 Effects of Scheduling Policy 2–17
2.3.7 Canceling a Thread 2–19
2.3.7.1 Thread Cancelation Implemented Using Exceptions 2–19
2.3.7.2 Thread Return Value After Cancelation 2–19
2.3.7.3 Controlling Thread Cancelation 2–20
2.3.7.4 Cancelation Points 2–21
2.3.7.5 Cleanup from Synchronous Cancelation 2–21
2.3.7.6 Cleanup from Asynchronous Cancelation 2–22
2.3.7.7 Example of Thread Cancelation Code 2–23
2.4 Synchronization Objects 2–26
2.4.1 Mutexes 2–26
2.4.1.1 Normal Mutex 2–27
2.4.1.2 Default Mutex 2–27
2.4.1.3 Recursive Mutex 2–27
2.4.1.4 Errorcheck Mutex 2–28
2.4.1.5 Mutex Operations 2–28
2.4.1.6 Mutex Attributes 2–29
2.4.2 Condition Variables 2–29
2.4.3 Condition Variable Attributes 2–34
2.5 Thread-Specific Data 2–34

3 Programming with Threads

3.1 Designing Code for Asynchronous Execution 3–1
3.1.1 Avoid Passing Stack Local Data 3–2
3.1.2 Initialize DECthreads Objects Before Thread Creation 3–2
3.1.3 Don't Use Scheduling As Synchronization 3–3
3.2 Memory Synchronization Between Threads 3–3
3.3 Using Shared Memory 3–4
3.3.1 Using Static Memory 3–4
3.3.2 Using Stack Memory 3–5
3.3.3 Using Dynamic Memory 3–5
3.4 Managing a Thread's Stack 3–6
3.4.1 Sizing the Stack 3–6
3.4.2 Using a Stack Guard Area 3–6
3.4.3 Handling Stack Overflow 3–7
3.5 Scheduling Issues 3–7
3.5.1 Real-Time Scheduling 3–7
3.5.2 Priority Inversion 3–8
3.5.3 Dependencies Among Scheduling Attributes and Contention Scope 3–8
3.6 Using Synchronization Objects 3–9
3.6.1 Distinguishing Proper Usage of Mutexes and Condition Variables 3–9
3.6.2 Avoiding Race Conditions 3–9
3.6.3 Avoiding Deadlocks 3–10
3.6.4 Signaling a Condition Variable 3–10
3.7 One-Time Initialization 3–12
3.8 Managing Dependencies Upon Other Libraries 3–12
3.8.1 Thread Reentrancy 3–13
3.8.2 Thread Safety 3–13
3.8.3 Lacking Thread Safety 3–14
3.8.3.1 Using Mutex Around Call to Unsafe Code 3–14
3.8.3.2 Using or Copying Static Data Before Releasing the Mutex 3–14
3.8.3.3 Using the DECthreads Global Lock 3–14
3.8.4 Use of Multiple Threads Libraries Not Supported 3–15
3.9 Detecting DECthreads Error Conditions 3–15
3.9.1 Contents of a DECthreads Bugcheck Dump File 3–16
3.9.2 Interpreting a DECthreads Bugcheck 3–17

4 Writing Thread-Safe Libraries

4.1 Features of the tis Interface 4–1
4.1.1 Reentrant Code Required 4–2
4.1.2 Performance of tis Interface Routines 4–2
4.1.3 Run-Time Linkage of tis Interface Routines 4–2
4.1.4 Cancelation Points 4–3
4.2 Using Mutexes 4–3
4.3 Using Condition Variables 4–4
4.4 Using Thread-Specific Data 4–4
4.5 Using Read-Write Locks 4–4

5 Using the DECthreads Exception Package

5.1 About the DECthreads Exceptions Package 5–1
5.1.1 Supported Programming Languages 5–1
5.1.2 Relation of Exceptions to Return Codes and Signals 5–2
5.2 Why Use Exceptions 5–2
5.3 Exception Programming 5–3
5.3.1 Declaring and Initializing an Exception 5–4
5.3.2 Raising an Exception 5–4
5.3.3 Catching an Exception 5–4
5.3.4 Reraising an Exception 5–6
5.3.5 Expressing Epilogue Actions 5–7
5.4 Exception Objects 5–7
5.4.1 Declaring and Initializing Exception Objects 5–8
5.4.2 Address Exceptions and Status Exceptions 5–8
5.4.3 How Exceptions Terminate 5–9
5.5 Exception Scopes 5–10
5.6 Raising Exceptions 5–11
5.7 Exception Handling Macros 5–12
5.7.1 Context of the Handler 5–12
5.7.2 Handlers and Macros 5–13
5.7.3 Catching Specific Exceptions 5–13
5.7.4 Catching Unspecified Exceptions 5–14
5.7.5 Reraising the Current Exception 5–15
5.7.6 Defining Epilogue Actions 5–16
5.8 Operations on Exceptions 5–17
5.8.1 Referencing the Caught Exception 5–17
5.8.2 Setting a System-Defined Error Status 5–18
5.8.3 Obtaining a System-Defined Error Status 5–19
5.8.4 Reporting a Caught Exception 5–20
5.8.5 Determining Whether Two Exceptions Match 5–20

5.9 Conventions for Modular Use of Exceptions 5–21
5.9.1 Develop Naming Conventions for Exceptions 5–21
5.9.2 Enclose Appropriate Actions in an Exception Scope 5–22
5.9.3 Raise Exceptions Prior to Performing Side-Effects 5–23
5.9.4 Distinguish Raising Exceptions From Side-Effects 5–23
5.9.5 Declare Variables Within Handler Code as Volatile 5–24
5.9.6 Reraise Caught Exceptions That Are Not Fully Handled 5–26
5.9.7 Declare All Exception Objects as Static 5–26
5.10 Exceptions Defined by DECthreads 5–27
5.11 Interoperability of Language-Specific Exceptions 5–28
5.12 Host Operating System Dependencies 5–29
5.12.1 No DIGITAL UNIX Dependencies 5–29
5.12.2 OpenVMS Conditions and DECthreads Exceptions 5–29

6 DECthreads Examples

6.1 Prime Number Search Example 6–1
6.2 Asynchronous User Interface Example 6–10

Part II POSIX.1c (pthread) Routines Reference

pthread_atfork pthread–3
pthread_attr_destroy pthread–7
pthread_attr_getdetachstate pthread–9
pthread_attr_getguardsize pthread–11
pthread_attr_getguardsize_np pthread–13
pthread_attr_getinheritsched pthread–15
pthread_attr_getname_np pthread–17
pthread_attr_getschedparam pthread–19
pthread_attr_getschedpolicy pthread–21
pthread_attr_getscope pthread–23
pthread_attr_getstackaddr pthread–25
pthread_attr_getstacksize pthread–27
pthread_attr_init pthread–29
pthread_attr_setdetachstate pthread–32
pthread_attr_setguardsize pthread–34
pthread_attr_setguardsize_np pthread–37
pthread_attr_setinheritsched pthread–40
pthread_attr_setname_np pthread–43
pthread_attr_setschedparam pthread–45

pthread_attr_setschedpolicy pthread–48
pthread_attr_setscope pthread–50
pthread_attr_setstackaddr pthread–53
pthread_attr_setstacksize pthread–56
pthread_cancel pthread–58
pthread_cleanup_pop pthread–60
pthread_cleanup_push pthread–62
pthread_condattr_destroy pthread–64
pthread_condattr_init pthread–66
pthread_cond_broadcast pthread–68
pthread_cond_destroy pthread–70
pthread_cond_getname_np pthread–72
pthread_cond_init pthread–74
pthread_cond_setname_np pthread–77
pthread_cond_signal pthread–79
pthread_cond_signal_int_np pthread–81
pthread_cond_timedwait pthread–83
pthread_cond_wait pthread–86
pthread_create pthread–89
pthread_delay_np pthread–94
pthread_detach pthread–96
pthread_equal pthread–98
pthread_exc_get_status_np pthread–100
pthread_exc_matches_np pthread–102
pthread_exc_report_np pthread–104
pthread_exc_set_status_np pthread–106
pthread_exit pthread–108
pthread_getconcurrency pthread–110
pthread_getname_np pthread–112
pthread_getschedparam pthread–114
pthread_getsequence_np pthread–116
pthread_getspecific pthread–118
pthread_get_expiration_np pthread–120
pthread_join pthread–122
pthread_key_create pthread–125
pthread_key_delete pthread–128
pthread_key_getname_np pthread–130

pthread_key_setname_np pthread–132
pthread_kill pthread–134
pthread_lock_global_np pthread–136
pthread_mutexattr_destroy pthread–138
pthread_mutexattr_gettype pthread–140
pthread_mutexattr_gettype_np pthread–142
pthread_mutexattr_init pthread–144
pthread_mutexattr_settype pthread–146
pthread_mutexattr_settype_np pthread–148
pthread_mutex_destroy pthread–150
pthread_mutex_getname_np pthread–152
pthread_mutex_init pthread–154
pthread_mutex_lock pthread–156
pthread_mutex_setname_np pthread–158
pthread_mutex_trylock pthread–160
pthread_mutex_unlock pthread–162
pthread_once pthread–164
pthread_self pthread–167
pthread_setcancelstate pthread–169
pthread_setcanceltype pthread–171
pthread_setconcurrency pthread–174
pthread_setname_np pthread–176
pthread_setschedparam pthread–178
pthread_setspecific pthread–181
pthread_sigmask pthread–183
pthread_testcancel pthread–185
pthread_unlock_global_np pthread–186
sched_get_priority_max pthread–188
sched_get_priority_min pthread–190
sched_yield pthread–192
sigwait pthread–194

Part III DIGITAL-Proprietary Interfaces: tis Routines Reference

tis_cond_broadcast tis–3
tis_cond_destroy tis–5
tis_cond_init tis–7
tis_cond_signal tis–9
tis_cond_wait tis–11
tis_getspecific tis–13
tis_key_create tis–15
tis_key_delete tis–18
tis_lock_global tis–20
tis_mutex_destroy tis–22
tis_mutex_init tis–24
tis_mutex_lock tis–26
tis_mutex_trylock tis–28
tis_mutex_unlock tis–30
tis_once tis–32
tis_read_lock tis–35
tis_read_trylock tis–37
tis_read_unlock tis–39
tis_rwlock_destroy tis–41
tis_rwlock_init tis–43
tis_self tis–45
tis_setcancelstate tis–46
tis_setspecific tis–48
tis_testcancel tis–50
tis_unlock_global tis–51
tis_write_lock tis–53
tis_write_trylock tis–55
tis_write_unlock tis–57

Part IV Appendixes

A Considerations for DIGITAL UNIX Systems

A.1 Overview A–1
A.2 Building DECthreads Applications A–1
A.2.1 Including DECthreads Header Files A–1
A.2.2 Building Multithreaded Applications from DECthreads Libraries A–2
A.2.3 Linking Multithreaded Shared Libraries A–2
A.2.4 Compiling Applications With the tis Interface A–3
A.3 Two-Level Scheduling on DIGITAL UNIX Systems A–3
A.3.1 DECthreads Use of Kernel Threads A–4
A.3.2 Support for Real-Time Scheduling A–4
A.4 Thread Cancelability of System Services A–5
A.4.1 Current Cancelation Points A–6
A.4.2 Future Cancelation Points A–7
A.5 Using Signals A–8
A.5.1 POSIX sigwait Service A–8
A.5.2 Handling Synchronous Signals as Exceptions A–9
A.6 Thread Stack Guard Areas A–10
A.7 Dynamic Activation A–10

B Considerations for OpenVMS Systems

B.1 Overview B–1
B.2 Compiling Under OpenVMS B–2
B.3 Linking OpenVMS Images B–2
B.4 Using DECthreads with AST Routines B–3
B.5 Dynamic Activation B–4
B.6 Declaring an OpenVMS Condition Handler B–4
B.7 Thread Cancelability of System Services B–5
B.8 Using OpenVMS Alpha 64-Bit Addressing B–5
B.9 DECthreads Condition Values B–5
B.10 Two-Level Scheduling on OpenVMS Alpha Systems B–6
B.10.1 Linker Options to Specify Image's Use of Kernel Threads . . . B–8
B.10.2 Setting Kernel Threads Support in Existing Images B–9
B.10.2.1 Examples B–10
B.10.3 Querying and Setting Kernel Threads Features B–11
B.10.4 Creation of Virtual Processors B–11
B.10.5 Delivery of ASTs B–12
B.10.6 Blocking System Services B–13
B.10.7 $HIBER and $WAKE B–14
B.10.8 Event Flags B–14
B.10.9 Interactions with OpenVMS B–15

B.10.10 Image Exit B–16
B.10.11 SYSGEN Parameter MULTITHREAD B–16
B.10.12 Process Control System Services and DCL Commands B–16
B.10.12.1 Process-Level System Services B–17
B.10.12.2 Kernel-Level System Services B–17
B.10.12.3 DCL Commands B–17
B.11 Interoperability with POSIX for OpenVMS B–17

C Considerations for Windows NT Systems

C.1 DECthreads Interfaces on Windows NT Systems C–1
C.1.1 Pthread Interface C–1
C.1.2 Other Interfaces C–2
C.2 Compiling DECthreads Applications C–2
C.3 Linking DECthreads Applications C–3
C.4 Interoperability of Win32 API and DECthreads pthread Routines C–3
C.5 Thread Cancelability of System Services C–4

D Debugging Multithreaded Applications

D.1 Using PTHREAD_CONFIG D–1
D.1.1 Major and Minor Keywords D–1
D.1.2 Specifying Multiple Values D–2
D.2 Running DECthreads in Metered Mode D–2
D.3 Using Ladebug on DIGITAL UNIX D–3
D.4 Debugging Threads on OpenVMS Systems D–3
D.4.1 Display of Stack Trace from Unhandled Exception D–3
D.5 Debugging Threads on Windows NT Systems D–3

E Migrating from the cma Interface

E.1 Overview E–1
E.2 cma Handles E–2
E.3 Interface Routine Mapping E–2
E.4 New pthread Routines E–5

F Migrating from the d4 Interface

F.1	Overview	F–1
F.2	Error Status and Function Returns	F–1
F.3	Replaced or Renamed Routines	F–2
F.4	Routines with No Changes to Syntax	F–2
F.5	Routines with Prototype or Syntax Changes	F–3
F.6	New Routines	F–5

Glossary

Index

Examples

2–1	pthread Cancel	2–23
5–1	Raising an Exception	5–5
5–2	Catching an Exception Using CATCH	5–5
5–3	Catching an Exception Using CATCH and CATCH_ALL	5–6
5–4	Defining Epilogue Actions Using FINALLY	5–7
5–5	Defining an Exception Scope	5–10
5–6	Raising a DECthreads Exception	5–12
5–7	Catching a Specific Exception Using CATCH	5–14
5–8	Catching an Unspecified Exception Using CATCH_ALL	5–15
5–9	Reraising an Exception Using RERAISE	5–16
5–10	Defining Epilogue Actions Using FINALLY	5–17
5–11	Setting an Error Status in an Exception Object	5–18
5–12	Obtaining the Error Status Value from a Status Exception Object	5–19
5–13	Reporting a Caught Exception	5–20
5–14	Comparing Two Exception Objects	5–21
5–15	Incorrect Placement of Statements That Might Raise an Exception	5–22
5–16	Correct Placement of Statements That Might Raise an Exception	5–23
5–17	Correct Placement of Statements That Might Raise an Exception	5–24
6–1	C Program Example (Prime Number Search)	6–4
6–2	C Program Example (Asynchronous User Interface)	6–13

Figures

1–1	Single-Threaded Process	1–3
1–2	Multithreaded Process	1–4
1–3	Thread State Transitions	1–5
1–4	Work Crew Model of Thread Operation	1–7
1–5	Pipelining Model of Thread Operation	1–8
2–1	Flow with FIFO Scheduling	2–18
2–2	Flow with RR Scheduling	2–18
2–3	Flow with Default Scheduling	2–18
2–4	Only One Thread Can Lock a Mutex	2–26
2–5	Thread A Waits on Condition Ready	2–31
2–6	Thread B Signals Condition Ready	2–32
2–7	Thread A Wakes and Proceeds	2–33
4–1	Read-Write Lock Behavior	4–6

Tables

1	Conventions	xix
1–1	DECthreads pthread Routines Summary	1–12
1–2	DECthreads tis Routines Summary	1–21
5–1	Names of Exception Objects Defined by DECthreads	5–27
A–1	DECthreads Header Files	A–1
A–2	DIGITAL UNIX Shared Libraries for Multithreaded Programs	A–2
A–3	Signals Reported as Exceptions	A–9
B–1	DECthreads Header Files	B–2
B–2	DECthreads Images	B–2
B–3	DECthreads Condition Values	B–5
B–4	Results of Keyword Arguments to /THREADS_ENABLE Qualifier	B–9
B–5	Return Values from $GETJPI System Service	B–11
D–1	PTHREAD_CONFIG Settings	D–1
E–1	Corresponding cma and pthread Routines	E–2
F–1	pthread Routines That Replace d4 Routines	F–2
F–2	d4 Routines With Syntax Changes as pthread Routines	F–3
F–3	d4 Routines Whose pthread Counterpart Uses Standard Datatypes	F–5

Preface

This guide describes DECthreads, DIGITAL's Multithreading Run-Time Library. In addition to introducing DECthreads components for building multithreaded applications and libraries to be called from other single-threaded or multithreaded programs, this guide reviews the key principles of multithreaded programming as reflected in the IEEE POSIX 1003.1c-1995 standard.

This guide also presents the concepts behind thread-safe and multithreaded processing environments and provides guidelines for using DECthreads to implement them on various DIGITAL platforms. Finally, this guide describes in detail each routine in the two recommended DECthreads interfaces:

- For building portable, multithreaded applications, DECthreads provides an IEEE POSIX 1003.1c-1995 standard (or **pthread**) interface.
- For building libraries whose routines can be called in either a single-threaded or multithreaded context, DECthreads provides a DIGITAL-proprietary thread-independent services (or **tis**) interface.

The interface you select depends upon your goals and the anticipated environment for your application.

As a complement to this guide, and for a user's guide to multithreaded programming using the IEEE POSIX 1003.1c-1995 standard, we recommend:

> *Programming with POSIX Threads* by David R. Butenhof, published as part of the Addison-Wesley Professional Computing Series (ISBN 0-201-63392-2).

Intended Audience

This guide is for system and application programmers who use DECthreads to create multithreaded applications or to create thread-safe code libraries that can be called from single-threaded or multithreaded applications.

Document Structure

This guide consists of the following:

Part I

- Chapter 1 provides a brief overview of multithreaded programming.
- Chapter 2 discusses the concepts and techniques related to DECthreads.
- Chapter 3 describes thread disciplines and coding issues you may face when writing a multithreaded program.
- Chapter 4 addresses writing thread-safe libraries.
- Chapter 5 introduces and provides conventions for the modular use of the DECthreads exception package.
- Chapter 6 contains an example demonstrating how to call DECthreads routines from a C language program.

Part II

- This part provides detailed reference information on each **pthread** interface routine. Routine descriptions appear in alphabetical order by routine name.

Part III

- This part provides detailed reference information on each **tis** interface routine. Routine descriptions appear in alphabetical order by routine name.

Part IV - Appendixes

- Appendix A discusses DECthreads issues and restrictions specific to DIGITAL UNIX systems.
- Appendix B discusses DECthreads issues and restrictions specific to OpenVMS systems.
- Appendix C discusses DECthreads issues and restrictions specific to the Microsoft Win32 interfaces on Windows NT systems.
- Appendix D discusses debugging issues for a multithreaded program that uses DECthreads.
- Appendix E summarizes the differences between the DIGITAL-proprietary DECthreads CMA (or **cma**) interface and the DECthreads **pthread** interface. Use this appendix to help you migrate your programs and applications to the **pthread** interface.

- Appendix F summarizes the differences between the DECthreads POSIX 1003.4a/Draft 4 (or **d4**) interface and the DECthreads **pthread** interface. Use this appendix to help you migrate your programs and applications to the **pthread** interface.

Glossary

- The Glossary contains definitions of terms used in this guide, listed alphabetically.

Related Documents

See your system's documentation set for more information on that system. DECthreads is available on the following platforms:

- DIGITAL UNIX 4.0
- OpenVMS Alpha Version 7.1
- OpenVMS VAX Version 7.1
- Windows NT Version 3.51 or higher

DIGITAL has changed the name of its UNIX® operating system from DEC OSF/1 to DIGITAL UNIX. The new name reflects DIGITAL's commitment to UNIX and its conformance to UNIX standards.

For a complete list and description of the books in the OpenVMS documentation set, see the *Overview of OpenVMS Documentation*.

The printed version of the DIGITAL UNIX document set is color coded to help specific audiences quickly find the books that meet their needs. This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General users	G	Blue
System and network administrators	S	Red
Programmers	P	Purple
Device driver writers	D	Orange
Reference page users	R	Green

Some books in the OpenVMS or DIGITAL UNIX documentation set help meet the needs of several audiences. For example, the information in some system manager, system administrator, or user books is also used by programmers.

Keep this in mind when searching for information on specific topics. The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the OpenVMS or DIGITAL UNIX documentation set.

Reader's Comments

DIGITAL welcomes your comments on this or any other guide. You can send comments in the following ways:

- Internet electronic mail: `writer@dceidl.enet.dec.com`
- FAX: 603-881-0120
  Attn: Run-Time Libraries Group, ZK02-3/Q18

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book or on its back cover.)
- The type and version of the operating system that you are using. For example, DIGITAL UNIX Version 4.0 or OpenVMS Version 7.0.
- If known, the type of processor that is running the operating system software. For example, AlphaServer 2000.
- The section numbers and page numbers of the information on which you are commenting.

Note: Please address technical questions to your local system vendor or to the approprite DIGITAL technical support office. Information provided with thie software media explains how to send problem reports to DIGITAL.

How To Order Additional Documentation

Use the following table to order additional documentation or information. If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825).

Conventions

In this guide, every use of OpenVMS means the OpenVMS operating system, and every use of UNIX means the DIGITAL UNIX operating system.

Table 1 shows the conventions used in this manual.

Table 1 Conventions

Convention	Description
% $	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne and Korn shells.
#	A number sign represents the superuser prompt.
`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.
Ctrl/*x*	The key combination Ctrl/*x* indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/Y or Ctrl/Z.
`monospaced text`	This typeface indicates the name of a command, routine, service, exception, or file. This typeface is also used in interactive examples and other screen displays.
`monospaced text`	This bolded typeface represents user input in interactive examples in the hardcopy and online versions of this guide.
...	A horizontal ellipsis in a figure or example indicates that not all of the statements are shown.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.

(continued on next page)

Table 1 (Cont.) Conventions

Convention	Description
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term.
italic text	Italic text represents book titles, parameters, arguments, and information that can vary in system messages (for example, Internal error *number*).
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.
mouse	The term *mouse* refers to any pointing device, such as a mouse, a puck, or a stylus.

Part I

DECthreads Overview and Programming Guidelines

Part I contains chapters that provide an overview and concepts of DECthreads as well as defining programming disciplines and guidelines for writing a multithreaded program.

1
Introducing DECthreads for Multithreaded Programming

This chapter introduces the concepts of threads and multithreaded programming. It describes four functional models that can be a basis for constructing multithreaded applications. The concepts and techniques introduced here are described in more detail in Chapter 2 and in this guide's platform-specific appendixes.

This chapter's last section introduces the components of the DECthreads package, in particular the **pthread** and **tis** interfaces, and how those components support building multithreaded applications and thread-safe libraries.

1.1 Advantages of Using Threads

Multithreaded programming means organizing and coding a program so that instances of its routines, called threads, can execute concurrently in the same process. You use threads to improve a program's performance—that is, its throughput, computational speed, responsiveness, or some combination.

Using threads can improve a program's performance on uniprocessor systems by permitting the overlap of input, output, or other slow operations with computational operations. Threads are useful in driving slow devices such as disks, networks, terminals, and printers. A multithreaded program can perform other useful work while waiting for the device to produce its next event, such as the completion of a disk transfer or the receipt of a packet from the network.

Using threads can also be advantageous when constructing an application's user interface. Consider the typical arrangement of a window system. Each time the user invokes an action (for example, by clicking on a mouse button), the program can use a separate thread to implement the action. If the user invokes multiple actions, multiple threads can perform the actions in parallel.

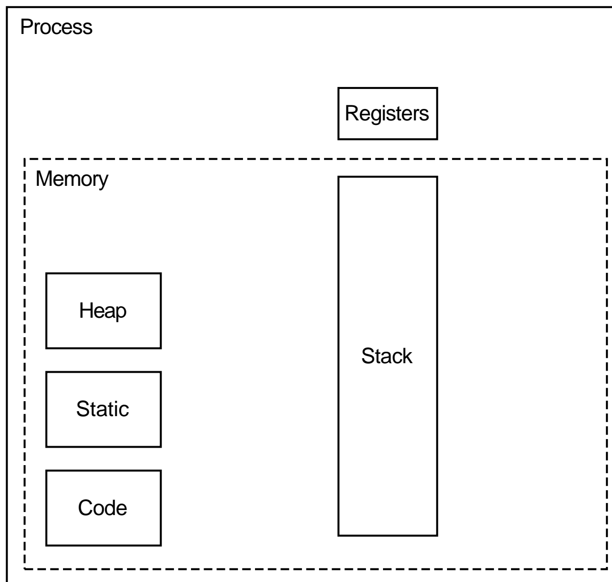
Using threads is especially advantageous when building a distributed system. These systems frequently contain a shared network server, where the server services requests from multiple clients. Using multiple threads allows the server to handle clients' requests in parallel, instead of artificially serializing them or creating (at great expense) one server process per client.

A program with multiple threads can be especially suited to run on a multiprocessor system, where threads run concurrently on separate processors. Threads created using the DECthreads library are capable of utilizing multiprocessors, if the target platform supports parallelism within a process. DIGITAL's DIGITAL UNIX-based platforms and OpenVMS Alpha platforms support parallelism; the OpenVMS VAX platform does not support parallelism.

1.2 Overview of Threads

A **thread** is a single, sequential flow of control within a process. Within each thread there is a single point of execution. Most traditional programs execute as a process with a single thread. Figure 1–1 and Figure 1–2 show the differences between a single-threaded process and a multithreaded process.

Figure 1–1 Single-Threaded Process

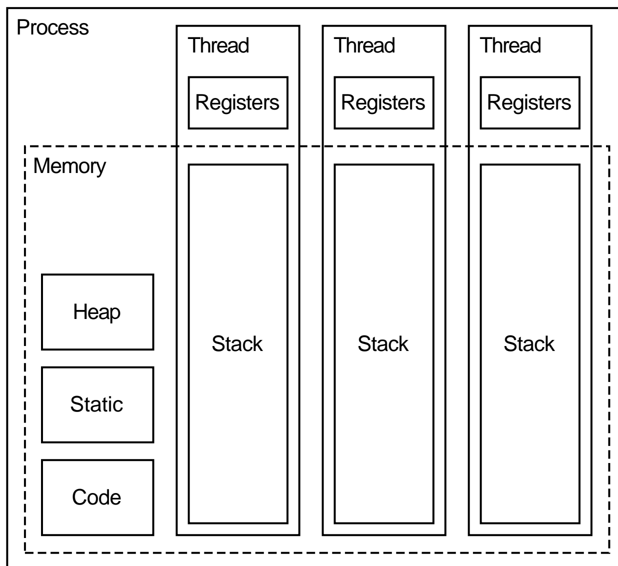


In Figure 1–2, notice that multiple threads share heap storage, static storage, and code but that each thread has its own register set and stack.

Using DECthreads, DIGITAL's multithreading run-time library, a programmer can create several threads within a process. The process's threads execute concurrently. Within a multithreaded program there are at any time multiple points of execution.

Threads execute within (and share) a single address space; therefore, a process's threads can read and write the same memory locations. When the threads access the same memory locations, your program must use synchronization elements, such as mutexes and condition variables, to ensure that the shared memory is accessed correctly. DECthreads provides routines that allow you to use these and other synchronization objects. Section 2.4 describes the synchronization objects that DECthreads offers as well as the operations your program can perform on them.

Figure 1–2 Multithreaded Process



1.3 Thread Execution

You should design and code a multithreaded program with the assumption that its threads execute *simultaneously*. That is, your program cannot make assumptions about the relative start or finish times of its threads or the sequence in which they execute. These are governed by the DECthreads thread scheduler, part of the run-time environment that DECthreads establishes before your program begins running. Nevertheless, your program can influence how DECthreads schedules its threads, by setting each thread's scheduling policy and scheduling priority. (Section 2.3.6 describes how thread scheduling works.)

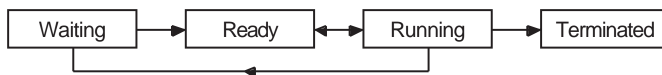
Each thread has its own thread identifier, which distinguishes it from all other threads in the process. In addition to the thread's scheduling policy and scheduling priority, each thread is associated with any thread-specific instances of thread-common data objects and with thread-specific system resources to support a flow of control.

A thread changes its state over the course of its execution. A thread is in one of the following states:

- *Waiting*—The thread is not eligible to execute, because it is synchronizing with another thread or with an external event, such as I/O.
- *Ready*—The thread is eligible to be executed by a processor.
- *Running*—The thread is currently being executed by a processor.
- *Terminated*—The thread has completed all of its work or has been canceled.

Figure 1–3 shows the transitions between states for a typical thread implementation.

Figure 1–3 Thread State Transitions



Note

Building your multithreaded program must produce executable code that is *reentrant*. Therefore, be sure that your compiler generates reentrant code before you design or code your multithreaded program. By default, DIGITAL's C, C++, Ada, Pascal, and BLISS compilers generate reentrant code.

If you cannot build your program so that its executable code is reentrant, it might be impossible to keep the program's threads from interfering with each other. See Section 3.8.1 for more information about thread-reentrant libraries.

In general, when using threads, be aware of language-based programming practices that are inherently not thread safe. ("Thread safety" is explained in Section 3.8.2.) You must address these factors when writing multithreaded applications and thread-safe libraries. For example, Fortran language routines typically rely heavily upon static storage, which can prevent those routines from being thread safe.

1.4 Functional Models for Multithreaded Programming

The following sections describe four functional models of processing information that are especially well suited for implementation in multithreaded programs:

- Boss/worker model
- Work crew model
- Pipelining model
- Combination of models

1.4.1 Boss/Worker Model

In a *boss/worker model*, one thread functions as the "boss" because it assigns tasks for "worker" threads to perform. Each worker performs a distinct task until it has finished, at which point it notifies the boss that it is ready to receive another task. Alternatively, the boss polls workers periodically to see whether any is ready to receive another task.

A variation of the boss/worker model is the *work queue model*. The boss places tasks in a queue, and workers check the queue and take tasks to perform.

An example of the work queue model in an office environment is a secretarial typing pool. The office manager boss puts documents to be typed in a basket, and worker typists take documents from the basket to work on.

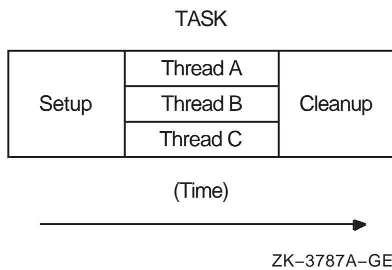
1.4.2 Work Crew Model

In the *work crew model*, multiple threads work together on a single task. The task is divided into pieces that are performed in parallel, and each thread performs one piece.

An example of a work crew is a group of people cleaning a building. Each person cleans certain rooms or performs certain types of work (washing floors, polishing furniture, and so forth), and each works independently.

In a multithreaded program that reflects the work crew model, each thread executes a task that can be performed in parallel. Figure 1–4 shows a task performed by three threads in a work crew model.

Figure 1–4 Work Crew Model of Thread Operation



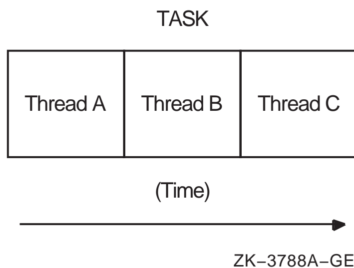
1.4.3 Pipelining Model

In the *pipelining model*, a task is divided into steps. The steps must be performed in sequence to produce a single instance of the desired result, and the work done in each step (except for the first and last) is based on the previous step and is a prerequisite for the work in the next step. However, the goal is to produce multiple instances of the desired result, and the steps are designed to operate in parallel: while one step is performed on one instance of the result, the preceding step can be performed on the next instance of the result.

An example of the pipelining model is an automobile assembly line. Each step or stage in the assembly line is continually busy receiving the product of the previous stage's work, performing its assigned work, and passing the product along to the next stage.

In a multithreaded program that reflects the pipelining model, each thread executes a step in the task. Figure 1–5 shows a task performed by three threads in a pipelining model.

Figure 1–5 Pipelining Model of Thread Operation



1.4.4 Combination of Functional Models

If the task that your program performs is complex, you might find it appropriate to organize it as a combination of the functional models previously described. For example, a program could follow the pipelining model, but with one or more steps performed by a set of threads that follow a work crew model. In addition, threads could be assigned to a work crew by taking a task from a work queue and deciding (based on the task characteristics) which threads are needed for the work crew.

1.5 Potential Issues for Multithreaded Programs

When you design and code a multithreaded program, you must accommodate or eliminate, as appropriate, each of the following issues:

- *Program complexity* is the most significant issue to consider in any multithreaded programming effort. Although using threads can simplify the coding and designing of a program, a certain level of expertise is required to be sure that the design of the synchronization and interplay among threads is appropriate and correctly specified. This level of expertise is higher than that required to design most single-threaded programs.
- *Dependence upon other nonreentrant software* means that your multithreaded program calls a routine or library that is not equipped to deal with threads. Given this dependence, your program must use the DECthreads global lock to prevent conflicts with other threads that use the same nonreentrant routine or library. Section 3.8 presents multithreaded programming techniques for managing dependencies upon other nonreentrant software.

- Due to programming errors, *race conditions* in the program's behavior can cause unpredictable and erroneous program behavior. Similarly, *deadlocks* can cause two or more threads to be blocked from executing indefinitely. Section 3.6.2 discusses race conditions in more detail, and Section 3.6.3 discusses deadlocks.
- *Priority inversion* prevents high-priority threads from executing when interdependencies exist among three or more threads of different priorities. Section 3.5.2 discusses techniques for avoiding priority inversion.

1.6 DECthreads Libraries and Interfaces

As a package, DECthreads is a collection of shared code libraries and C language header files that declare entry points into those libraries. This guide's platform-specific appendixes describe these libraries in more detail and list all other libraries upon which the DECthreads libraries depend.

From the programmer's view the DECthreads libraries offer *interfaces*. Each interface is a distinct set of routines that together provide a well-defined set of related data objects and operations.

This version of DECthreads supports two interfaces that are documented in this guide:

- The **pthread** interface provides multithreading capability in your applications. This interface is based on the IEEE POSIX 1003.1c-1995 standard. Use this interface to build portable, multithreaded applications.

  Section 1.6.1 introduces the **pthread** interface. Chapter 2 and Chapter 3 describe how to use the features and functionality of the **pthread** interface. The reference descriptions in Part II describe in detail each routine in the **pthread** interface.
- The DIGITAL-proprietary **tis** interface offers routines that provide thread-independent services. The routines in this interface enable your software to perform thread-safe processing that requires synchronization, but without requiring the use of threads.

  Section 1.6.2 introduces the **tis** interface. Chapter 4 describes how to use the features and functionality of the **tis** interface. The reference descriptions in Part III describe in detail each routine in the **tis** interface.

This release of DECthreads includes interface definitions for the C programming language only. However, all DECthreads routines are callable from languages other than C. Your application must provide its own declarations for DECthreads routines in a manner appropriate to its

programming language. These definitions should be modeled after the declarations in the C language `pthread.h` header file.

For backward compatibility, this version of DECthreads also supports other interfaces that are not documented in this guide. See Section 1.6.3.

1.6.1 Multithreading pthread Interface

This version of DECthreads offers *one* documented interface for multithreading capability. The **pthread** interface routines implement the IEEE Standard 1003.1c-1995, Portable Operating System Interface (or POSIX) Application Program Interface, also known as **POSIX.1c**. (More specifically, this interface is an extension to the 1003.1 Portable Operating System Interface standard rather than an independent interface specification.)

The DECthreads **pthread** implementation of the POSIX.1c standard is the primary multithreading interface in the DECthreads environment—that is, it is the most portable, efficient, and powerful multithreading interface supported by DECthreads.

Table 1–1 lists and summarizes functionally the DECthreads **pthread** interface routines.

The **pthread** interface contains routines grouped in the following functional categories:

- General threads routines
- Thread attributes object routines
- Thread cancelation routines
- Thread priority, concurrency, and scheduling routines
- Thread-specific data routines
- Mutex routines
- Mutex attributes object routines
- Condition variable routines
- Condition variable attributes object routines

The DECthreads **pthread** interface also provides routines that implement nonportable extensions to the POSIX.1c standard. These routines are grouped in these functional categories:

- Thread execution routines
- DECthreads global mutex routines

- Mutex attributes routines
- Condition variable routines
- Object naming routines
- DECthreads exception object routines

Note

Some routines in the **pthread** interface have a corresponding or similar routine in the **tis** interface. However, you should avoid specifying objects created under one DECthreads interface in calls to a different DECthreads interface.

Among the routines in the **pthread** interface that implement nonportable extensions to the POSIX.1c standard, are the routines in the **DECthreads exception package**. This package consists of a library and C language header file (`pthread_exceptions.h`) that implement a DECthreads-specific exception-handling facility. It is designed specifically for use with the DECthreads **pthread** interface. Chapter 5 describes the DECthreads exception package.

This guide also documents several routines that are not declared entries in the DECthreads **pthread** interface, but that have close affinity with its functionality. Examples are the `sched_yield( )` and `sigwait( )` routines. See the end of Table 1–1 for a list of these routines.

Table 1–1 DECthreads pthread Routines Summary

Routine	Description
General Threads Routines	
pthread_atfork( )	Declares fork handler routines to be called.
pthread_create( )	Creates a thread object and thread.
pthread_detach( )	Marks a thread object for deletion.
pthread_equal( )	Compares one thread identifier to another thread identifier.
pthread_exit( )	Terminates the calling thread.
pthread_join( )	Causes the calling thread to wait for the termination of a specified thread and detach it.
pthread_kill( )	Delivers a signal to a specified thread.
pthread_once( )	Calls an initialization routine to be executed only once.
pthread_self( )	Obtains the identifier of the calling thread.
pthread_sigmask( )	Examines or changes the calling thread's signal mask.

(continued on next page)

Table 1–1 (Cont.) DECthreads pthread Routines Summary

Routine	Description
Thread Attributes Object Routines	
pthread_attr_destroy( )	Destroys a thread attributes object.
pthread_attr_getdetachstate( )	Obtains the detachstate attribute from the specified thread attributes object.
pthread_attr_getguardsize( )	Obtains the guardsize attribute of the specified thread attributes object.
pthread_attr_getinheritsched( )	Obtains the inherit scheduling attribute from the specified thread attributes object.
pthread_attr_getschedparam( )	Obtains the scheduling parameters for an attribute of the specified thread attributes object.
pthread_attr_getschedpolicy( )	Obtains the scheduling policy attribute of the specified thread attributes object.
pthread_attr_getscope( )	Obtains the contention-scope attribute of the specified thread attributes object.

(continued on next page)

Table 1–1 (Cont.) DECthreads pthread Routines Summary

Routine	Description
Thread Attributes Object Routines	
pthread_attr_getstackaddr( )	Obtains the stackaddr attribute of the specified thread attributes object.
pthread_attr_getstacksize( )	Obtains the stacksize attribute of the specified thread attributes object.
pthread_attr_init( )	Initializes a thread attributes object.
pthread_attr_setdetachstate( )	Changes the detachstate attribute in the specified thread attributes object.
pthread_attr_setguardsize( )	Changes the guardsize attribute of the specified thread attributes object.
pthread_attr_setinheritsched( )	Changes the inherit scheduling attribute of the specified thread attributes object.
pthread_attr_setschedparam( )	Changes the values of the parameters associated with the scheduling policy attribute of the specified thread attributes object.
pthread_attr_setschedpolicy( )	Changes the scheduling policy attribute of the specified thread attributes object.
pthread_attr_setscope( )	Changes the contention-scope attribute of the specified thread attributes object.
pthread_attr_setstackaddr( )	Changes the stackaddr attribute in the specified thread attributes object.
pthread_attr_setstacksize( )	Changes the stacksize attribute in the specified thread attributes object.

(continued on next page)

Table 1–1 (Cont.) DECthreads pthread Routines Summary

Routine	Description
Thread Cancelation Routines	
pthread_cancel( )	Allows a thread to request that it, or another thread, terminate execution.
pthread_cleanup_pop( )	Removes a cleanup handler routine from the top of the cleanup stack and optionally executes it.
pthread_cleanup_push( )	Establishes a cleanup handler routine to be executed when the thread exits or is canceled.
pthread_setcancelstate( )	Changes the calling thread's cancelability state.
pthread_setcanceltype( )	Changes the calling thread's cancelability type.
pthread_testcancel( )	Requests delivery of any pending cancelation request to the calling thread.
Thread Priority, Concurrency, and Scheduling Routines	
pthread_getconcurrency( )	Obtains the current concurrency level parameter for the process.
pthread_getschedparam( )	Obtains the current scheduling policy and scheduling parameters of a thread.
pthread_setconcurrency( )	Changes the current concurrency level parameter for the process.
pthread_setschedparam( )	Changes the current scheduling policy and scheduling parameters of a thread.

(continued on next page)

Table 1–1 (Cont.) DECthreads pthread Routines Summary

Routine	Description
Thread-Specific Data Routines	
pthread_getspecific( )	Obtains the thread-specific data associated with the specified key.
pthread_key_create( )	Generates a unique thread-specific data key.
pthread_setspecific( )	Changes the thread-specific data value associated with the specified key for the calling thread.
pthread_key_delete( )	Deletes a thread-specific data key.
Mutex Routines	
pthread_mutex_destroy( )	Destroys a mutex.
pthread_mutex_init( )	Initializes a mutex with attributes specified by the attributes argument.
pthread_mutex_lock( )	Locks an unlocked mutex; if locked, the caller waits for the mutex to become available.
pthread_mutex_trylock( )	Attempts to lock a mutex; returns immediately if mutex is already locked.
pthread_mutex_unlock( )	Unlocks a locked mutex.
Mutex Attributes Object Routines	
pthread_mutexattr_init( )	Initializes a mutex attributes object.
pthread_mutexattr_destroy( )	Destroys a mutex attributes object.
pthread_mutexattr_gettype( )	Obtains the mutex type attribute of a mutex attributes object.
pthread_mutexattr_settype( )	Changes the mutex type attribute of a mutex attributes object.

(continued on next page)

Table 1–1 (Cont.) DECthreads pthread Routines Summary

Routine	Description
Condition Variable Routines	
pthread_cond_broadcast( )	Wakes all threads waiting on a condition variable.
pthread_cond_destroy( )	Destroys a condition variable.
pthread_cond_init( )	Initializes a condition variable.
pthread_cond_signal( )	Wakes at least one thread that is waiting on a condition variable.
pthread_cond_timedwait( )	Causes a thread to wait a specified period of time for a condition variable to be signaled or broadcasted.
pthread_cond_wait( )	Causes a thread to wait for a condition variable to be signaled or broadcasted.
Condition Variable Attributes Object Routines	
pthread_condattr_destroy( )	Destroys a condition variable attributes object.
pthread_condattr_init( )	Initializes a condition variable attributes object.

(continued on next page)

Table 1–1 (Cont.) DECthreads pthread Routines Summary

Routine	Description
Nonportable Extensions	
pthread_delay_np( ) pthread_get_expiration_np( ) pthread_getsequence_np( )	Thread execution routines
pthread_attr_getguardsize_np( ) pthread_attr_setguardsize_np( )	Thread attributes routines
pthread_lock_global_np( ) pthread_unlock_global_np( )	DECthreads global mutex routines
pthread_mutexattr_gettype_np( ) pthread_mutexattr_settype_np( )	Mutex attributes routines
pthread_cond_signal_int_np( )	Condition variable routines
pthread_attr_getname_np( ) pthread_attr_setname_np( ) pthread_cond_getname_np( ) pthread_cond_setname_np( ) pthread_getname_np( ) pthread_key_getname_np( ) pthread_key_setname_np( ) pthread_mutex_getname_np( ) pthread_mutex_setname_np( ) pthread_setname_np( )	Object naming routines
pthread_exc_get_status_np( ) pthread_exc_matches_np( ) pthread_exc_report_np( ) pthread_exc_set_status_np( )	DECthreads exception object routines

(continued on next page)

Table 1–1 (Cont.) DECthreads pthread Routines Summary

Routine	Description
Related Standard Routines	
sched_get_priority_max( )	Returns the maximum priority for the specified scheduling policy.
sched_get_priority_min( )	Returns the minimum priority for the specified scheduling policy.
sched_yield( )	Notifies the scheduler that the calling thread is willing to release its processor to other threads of the same or higher scheduling precedence.
sigwait( )	Suspends a calling thread until a signal arrives.

1.6.1.1 Optionally Implemented POSIX.1c-1995 Routines

In this version of DECthreads, the **pthread** interface does not support the following features that are specified in the POSIX.1c standard:

- Reported by the POSIX.1c `_POSIX_THREAD_PROCESS_SHARED` macro:

  ```
  pthread_condattr_getpshared( )
  pthread_condattr_setpshared( )
  pthread_mutexattr_getpshared( )
  pthread_mutexattr_setpshared( )
  ```

- Reported by the POSIX.1c `_POSIX_THREAD_PRIO_PROTECT` macro:

  ```
  pthread_mutex_getprioceiling( )
  pthread_mutex_setprioceiling( )
  pthread_mutexattr_getprioceiling( )
  pthread_mutexattr_setprioceiling( )
  ```

- Reported by the POSIX.1c `_POSIX_THREAD_PRIO_PROTECT` and `_POSIX_THREAD_PRIO_INHERIT` macros:

  ```
  pthread_mutexattr_getprotocol( )
  pthread_mutexattr_setprotocol( )
  ```

The POSIX.1c standard directs DECthreads to provide the macros named `_POSIX_THREAD_PROCESS_SHARED`, `_POSIX_THREAD_PRIO_PROTECT`, and `_POSIX_THREAD_PRIO_INHERIT` to report whether optionally implemented routines are present.

1.6.2 Thread-Independent Services Interface

The DIGITAL-proprietary **tis** interface offers a set of thread-independent services. Use these routines to build software that performs processing that requires synchronization, but without requiring the use of threads. That is, use **tis** routines to build thread-safe code libraries whose routines can be called from either a single-threaded or multithreaded environment.

In the absence of threads, **tis** routines impose minimal overhead on the calling program. For instance, **tis** routines do not use interlocked instructions and memory barriers.

When threads are present, **tis** routines provide full support for DECthreads synchronization, such as synchronization objects and thread joining. Note that there are no **tis** routines for creating threads or thread objects, because that would have no meaning if called from a single-threaded environment.

The **tis** routines can be classified into these functional categories:

- General routines
- Thread cancelation routines
- Thread-specific data key routines
- Mutex routines
- Condition variable routines
- Read-write lock routines

Note

Some routines in the **pthread** interface have a corresponding or similar routine in the **tis** interface.

Table 1–2 summarizies these groups of **tis** routines.

Table 1–2 DECthreads tis Routines Summary

Routine	Description
General Routines	
`tis_once()`	Calls a one-time initialization routine that can be executed.
`tis_self()`	Obtains the identifier of the calling thread.
Thread Cancelation Routines	
`tis_setcancelstate()`	Changes the calling thread's cancelability state.
`tis_testcancel()`	Creates a cancelation point in the calling thread.

(continued on next page)

Table 1–2 (Cont.) DECthreads tis Routines Summary

Routine	Description
Thread-Specific Data Key Routines	
`tis_getspecific( )`	Obtains the data associated with the specified thread-specific data key.
`tis_key_create( )`	Generates a unique thread-specific data key.
`tis_key_delete( )`	Deletes a thread-specific data key.
`tis_setspecific( )`	Changes the value associated with the specified thread-specific data key.
Mutex Routines	
`tis_lock_global( )`	Locks the DECthreads global mutex.
`tis_mutex_destroy( )`	Destroys the specified tis mutex object.
`tis_mutex_init( )`	Initializes a tis mutex object.
`tis_mutex_lock( )`	Locks the specified tis mutex, if unlocked.
`tis_mutex_trylock( )`	Tries to lock the specified tis mutex.
`tis_mutex_unlock( )`	Unlocks the specified tis mutex.
`tis_unlock_global( )`	Unlocks the DECthreads global mutex.
Condition Variable Routines	
`tis_cond_broadcast( )`	Wakes all threads waiting on the specified condition variable.
`tis_cond_destroy( )`	Destroys the specified condition variable object.
`tis_cond_init( )`	Initializes a condition variable object.
`tis_cond_signal( )`	Wakes at least one thread that is waiting on the specified condition variable.
`tis_cond_wait( )`	Causes the calling thread to wait for the specified condition variable to be signaled or broadcasted.

(continued on next page)

Table 1–2 (Cont.) DECthreads tis Routines Summary

Routine	Description
Read-Write Lock Routines	
`tis_read_lock( )`	Acquires the specified read-write lock for read access.
`tis_read_trylock( )`	Attempts to acquire the specified read-write lock for read access; returns immediately if already locked.
`tis_read_unlock( )`	Unlocks the specified read-write lock already acquired for read access.
`tis_rwlock_destroy( )`	Destroys the specified read-write lock object.
`tis_rwlock_init( )`	Initializes the specified read-write lock object.
`tis_write_lock( )`	Acquires the specified read-write lock for write access.
`tis_write_trylock( )`	Attempts to acquire the specified read-write lock for write access; returns immediately if already locked.
`tis_write_unlock( )`	Unlocks the specified read-write lock already acquired for write access.

1.6.3 Undocumented and Obsolete DECthreads Interfaces

Previous versions of DECthreads offered interfaces that under this DECthreads version are considered to be either undocumented but supported, or obsolete.

1.6.3.1 Undocumented But Supported cma Interface

This version of DECthreads supports the DIGITAL-proprietary CMA (or **cma**) interface. The **cma** interface reports errors by raising exceptions. This interface is layered on top of the DECthreads **pthread** interface. This interface is usually available only on DIGITAL platforms.

DIGITAL will continue to support applications developed using the DECthreads **cma** interface. Binary compatibility will be supported indefinitely. Nonetheless, we recommend that, as soon as possible, you migrate any **cma** code in your existing applications to the latest DECthreads **pthread** interface, to take advantage of its POSIX.1c standard features, portability, and future enhancements.

Routines of the **cma** interface are not documented in this guide, but are documented for previous DECthreads versions. In this guide see Appendix E for information to help you migrate your **cma**-based programs and applications to the latest DECthreads **pthread** interface.

1.6.3.2 Obsolete d4 Interfaces

Note

An obsolete interface will be retired in a future DECthreads release. After retirement, that interface will no longer be enhanced or supported.

For backward compatibility only, this version of DECthreads retains full binary support for the obsolete **d4** interfaces. These interfaces are implementations of the IEEE POSIX 1003.4a/Draft 4 document, and are also known as "DCE threads".

These interfaces include both a "standard" interface that reports errors by setting *errno* and returning a value of -1, and an "exception-returning" interface that, like the **cma** interface, reports errors by raising exceptions.

The DECthreads **d4** interfaces are obsolete and DIGITAL plans to retire them (that is, will not be provided) in the next release of DECthreads. Thus, we recommend that you migrate any **d4** code in your existing applications to the latest DECthreads **pthread** interface, to take advantage of its POSIX.1c standard features, portability, and future enhancements.

Routines of the **d4** interfaces are not documented in this guide, but are documented for previous DECthreads versions. In this guide see Appendix F for information to help you migrate your **d4**-based programs and applications to the latest DECthreads **pthread** interface.

2

DECthreads Objects and Operations

This chapter describes operations that act upon the objects supported in the DECthreads **pthread** interface.

2.1 Threads, Mutexes, and Condition Variables

A multithreaded program typically manipulates three kinds of objects:

- A **thread object** describes a **thread**, which refers to a distinct flow of control within a process. After a thread object is created, DECthreads uses it to maintain information about the thread's state and its associated attributes.
- A **mutex** serves as a lock for a data object that is shared among the program's threads. To access a data object that is guarded by a mutex, a thread must *acquire* the mutex's lock, then access the data object, then *release* the mutex's lock.
- When associated with a shared data object and its mutex, a **condition variable** encapsulates a condition, or predicate, that must be satisfied before a thread can access the data object.

2.2 Attributes Objects

Before creating a thread object, mutex, or condition variable, your program can create and initialize an **attributes object**, which specifies the particular features of that thread, mutex, or condition variable. There are distinct kinds of attributes objects for threads, mutexes, and condition variables.

When your program creates a thread object, mutex, or condition variable, it can accept the default attributes for that object or specify an existing attributes object that contains particular attribute values. For a thread object, you can change some of its attributes after execution of the corresponding thread has begun—for example, you can change the thread's priority.

To create an attributes object, you can use one of the following routines, depending on the type of object to which the attributes apply:

- `pthread_attr_init( )` for thread attributes
- `pthread_condattr_init( )` for condition variable attributes
- `pthread_mutexattr_init( )` for mutex attributes

These routines create an attributes object containing default values for the individual attributes. To modify any attribute values in an attributes object, use one of the "`attr_set`" routines, such as `pthread_attr_setinheritsched( )`, described in later sections.

Creating an attributes object (or changing the values in an attributes object) does not affect the attributes of existing thread objects, mutexes, and condition variables.

To destroy an attributes object, use one of the following routines:

- `pthread_attr_destroy( )` for thread attributes objects
- `pthread_condattr_destroy( )` for condition variable attributes objects
- `pthread_mutexattr_destroy( )` for mutex attributes objects

Deleting an attributes object does not affect the attributes of objects previously created with that attributes object.

2.3 Thread Objects and Operations

Operations on threads take place with respect to a thread object. A thread object is a data structure maintained by DECthreads that contains the attribute information and DECthreads state information about a thread.

The following sections describe these operations on threads:

- Creating and starting a thread
- Setting the attributes of a new thread
- Terminating a thread
- Detaching and destroying a thread
- Joining with another thread
- Scheduling a thread
- Canceling a thread

2.3.1 Creating and Starting a Thread

Creating a thread means directing DECthreads to create a thread object and to assign a unique thread identifier to the thread object. The thread object encapsulates attribute information about the thread and DECthreads' own state information about the thread. After a thread has been created, DECthreads has all the information it requires to start a distinct sequence of execution with this process.

Starting a thread means DECthreads causes a **start routine**, with its argument, to be called on some CPU within this system.

Your program creates a thread using the `pthread_create( )` routine. This routine creates a thread object based on the settings of a specified thread attributes object, which your program must have previously created and initialized. Otherwise, without specifying a thread attributes object, you can create a new thread that has DECthreads default attributes.

DECthreads creates a thread in the *ready* state and prepares the thread to begin executing its start routine, the function passed to the `pthread_create( )` routine. Depending on the presence of other threads and their scheduling and priority attributes, the new thread might start executing immediately. The new thread can also preempt its creator, depending on the two threads' respective scheduling and priority attributes. The caller of `pthread_create( )` can synchronize with the new thread using the `pthread_join( )` routine or using any mutually agreed upon mutexes or condition variables.

For the duration of the new thread's existence, DECthreads maintains and manages the thread object and other thread state overhead. A thread *exists* until it is both *terminated* and *detached*. (See Section 2.3.3 and Section 2.3.4 for more information about terminating and detaching threads.)

DECthreads assigns each new thread a thread identifier, which DECthreads writes into the address specified as the `pthread_create( )` routine's *thread* argument. DECthreads writes the new thread's thread identifier *before* the new thread executes.

By default, the new thread's scheduling policy and priority are inherited from the creating thread—that is, by default, the `pthread_create( )` routine ignores the scheduling policy and priority set in the specified thread attributes object. Thus, to create a thread that is subject to the scheduling policy and priority set in the specified thread attributes object, before calling `pthread_create( )` your program must use the `pthread_attr_setinheritsched( )` routine to set the inherit thread attributes object's scheduling attribute to `PTHREAD_EXPLICIT_SCHED`.

You can create a thread that is detached. To do so, create a thread using a thread attributes object whose detachstate attribute has been set, using the `pthread_attr_setdetach( )` routine, to `PTHREAD_CREATE_DETACHED`. This is useful for creating a thread that your program knows will *not* join with any other thread. That is, when such a thread terminates, DECthreads automatically destroys the thread and its thread object.

For more detailed information about thread creation, see the reference description of the `pthread_create( )` routine in Part II.

2.3.2 Setting the Attributes of a New Thread

When creating a thread, your program can optionally specify the attributes of the new thread using a **thread attributes object**. To do so, your program must:

1. Create a thread attributes object by calling the `pthread_attr_init( )` routine.
2. Set values for the individual attributes of the thread attributes object. (The POSIX.1c standard provides a separate routine for setting each attribute in the thread attributes object.)
3. When ready to create the new thread, pass the address of the thread attributes object as an argument to the `pthread_create( )` routine.

After your program creates a thread attributes object, that object can be reused for each new thread that the program creates. For the details about creating and deleting a thread attributes object, see the descriptions in Part II of the `pthread_attr_create( )` and `pthread_attr_delete( )` routines.

Using the thread attributes object, your program can specify these attributes of a new thread:

- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Stack size
- Stack location
- Stack guard size
- Contention scope

2.3.2.1 Setting the Inherit Scheduling Attribute

The inherit scheduling attribute's value specifies whether the new thread inherits the settings of its scheduling priority attribute and scheduling parameters attribute from the creating thread (the default behavior), or uses the scheduling attributes stored in the attributes object. Inheriting these settings from the creating thread is the default behavior, or you can specify the same by setting the thread attributes object's inherit scheduling attribute to PTHREAD_INHERIT_SCHED. To use the setting in the attributes objects, set the inherit scheduling attribute to PTHREAD_EXPLICIT_SCHED.

Use the pthread_attr_setinheritsched( ) routine to set the thread attributes object's inherit scheduling attribute.

2.3.2.2 Setting the Scheduling Policy Attribute

The scheduling policy attribute describes how DECthreads schedules the new thread for execution relative to the other threads in the process.

A thread has one of the following scheduling policies:

- SCHED_FG_NP (Foreground or "throughput"; also known as SCHED_OTHER)—*This is the default scheduling policy*. All threads are timesliced, and no thread is completely denied execution time. (**Timeslicing** is a mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.) However, higher-priority threads receive more execution time than lower-priority threads.

  Threads with this scheduling policy can be denied execution time by first-in /first-out (FIFO) or round-robin (RR) threads.

- SCHED_BG_NP (Background)—Like the default (SCHED_FG_NP) scheduling policy, this policy ensures that all threads, regardless of priority, receive some scheduling.

  Threads with this scheduling policy can be denied execution time by FIFO or RR threads, and receive less execution time than threads with the throughput scheduling policy.

- SCHED_FIFO (first-in/first-out or FIFO)—The highest-priority thread runs until it blocks. If there is more than one thread with the same priority and that priority is the highest among other threads, the first thread to begin running continues until it blocks. If a thread with this policy becomes ready, and it has a higher priority than the currently running thread, then DECthreads preempts the current thread and immediately begins running the higher priority thread.

- SCHED_RR (round-robin or RR)—The highest-priority thread runs until it blocks; however, threads of equal priority are timesliced. If a thread with this policy becomes ready, and it has a higher priority than the currently running thread, then DECthreads preempts the current thread and immediately begins running the higher-priority thread.

2.3.2.2.1 Techniques for Setting the Scheduling Policy Attribute Use either of two techniques to set a thread attributes object's scheduling policy attribute:

- Set the scheduling policy attribute in the attributes object, which establishes the scheduling policy of a new thread when it is created. To do so, call the pthread_attr_setschedpolicy( ) routine. This allows the creator of a thread to establish the created thread's initial scheduling policy. (Note that this value is used only if the attributes object is set so that the created thread does not inherit its priority from the creating thread. Inheriting priority is the default behavior.)
- Change the scheduling policy of an existing thread (and, at the same time, the scheduling parameters) by calling the pthread_setschedparam( ) routine. This routine allows a thread to change its own scheduling policy and/or scheduling priority, but has no effect on the corresponding settings in the thread attributes object.

Section 2.3.6 describes and shows the effect of the scheduling policy on thread scheduling.

2.3.2.2.2 Comparing Throughput and Real-Time Policies The default throughput scheduling policy is intended to be an "adaptive" policy, giving each thread an opportunity to execute based on its behavior. That is, for a thread that doesn't execute often, DECthreads tends to give it high access to the processor because it isn't greatly affecting other threads. On the other hand, DECthreads tends to schedule with less preference any compute-bound threads with throughput scheduling policy.

This yields a responsive system in which all threads with throughput scheduling policy get a chance to run fairly frequently. It also has the effect of automatically resolving priority inversions, because over time any threads that have received less processing time (among those with throughput scheduling policy) will rise in preference while the running thread drops, and eventually the inversion is reversed.

The FIFO and RR scheduling policies are considered "real-time" policies, because they require DECthreads to schedule such threads strictly by the specified priority. Because threads that use real-time scheduling policies require additional DECthreads overhead, incautious use of the FIFO or RR policies can cause the performance of the application to suffer.

If relative priorities of threads are important to your application—that is, if a compute-bound thread really requires consistently *predictable* execution—then create those threads using either the FIFO or RR scheduling policy. However, use of “real-time” policies can expose the application to unexpected performance problems, such as priority inversions, and therefore their use should be avoided in most applications.

2.3.2.2.3 Portability of Scheduling Policy Settings Only the `SCHED_FIFO` and `SCHED_RR` scheduling policies are portable across POSIX.1c-conformant implementations. The other scheduling policies are DECthreads extensions to the POSIX.1c standard.

Note

The SCHED_OTHER identifier is portable, but the POSIX.1c standard does not specify the behavior that it signifies. For example, on non-DECthreads platforms the `SCHED_OTHER` scheduling policy could be identical to the `SCHED_FIFO` or `SCHED_RR` policy.

2.3.2.3 Setting the Scheduling Parameters Attribute

The scheduling parameters attribute specifies the execution priority of a thread. (Although the terminology and format are designed to allow adding more scheduling parameters in the future, only priority is currently defined.) The priority is an integer value, but each policy can allow only a restricted range of priority values. You can determine the range for any policy by calling the `sched_get_priority_min( )` or `sched_get_priority_max( )` routines. DECthreads also supports a set of nonportable symbols designating the priority range for each policy, as follows:

Low	High
PRI_FIFO_MIN	PRI_FIFO_MAX
PRI_RR_MIN	PRI_RR_MAX
PRI_OTHER_MIN	PRI_OTHER_MAX
PRI_FG_MIN_NP	PRI_FG_MAX_NP
PRI_BG_MIN_NP	PRI_BG_MAX_NP

Section 2.3.6 describes how to specify a priority between the minimum and maximum values, and it also discusses how priority affects thread scheduling.

Use either of two techniques to set a thread attributes object's scheduling parameters attribute:

- Set the scheduling parameters attribute in the thread attributes object, which establishes the execution priority of a new thread when it is created. To do so, call the `pthread_attr_setschedparam( )` routine. This allows the creator of a thread to establish the created thread's initial execution priority. (Note that this value is used only if the thread attributes object is set so that the created thread does not inherit its priority from the creating thread. Inheriting priority is the default behavior.)
- Change the scheduling policy and parameters of an existing thread by calling the `pthread_setschedparam( )` routine. This routine allows a thread to change its own scheduling policy or scheduling priority, *but has no effect on the corresponding settings in the thread attributes object*.

2.3.2.4 Setting the Stacksize Attribute

The stacksize attribute represents the minimum size (in bytes) of the memory required for a thread's stack. To increase or decrease the size of the stack for a new thread, call the `pthread_attr_setstacksize( )` routine and use this thread attributes object when creating the thread and stack. You must specify at least `PTHREAD_STACK_MIN` bytes.

After a thread has been created, your program cannot change the size of the thread's stack. See Section 3.4.1 for more information about sizing a stack.

2.3.2.5 Setting the Stack Address Attribute

The stack address attribute represents the location or address of a region of memory that your program allocates to use as a thread's stack. The value of the stack address attribute represents the origin of the thread's stack. However, please be aware that the actual address you specify, relative to the stack memory you have allocated, is inherently nonportable.

To set the address of the stack origin for a new thread, call the `pthread_attr_setstackaddr( )` routine, specifying an initialized thread attributes object as an argument, and use the thread attributes object when creating the new thread. Use the `pthread_attr_getstackaddr( )` routine to obtain the value of the stack address attribute of an initialized thread attributes object.

After a thread has been created, your program cannot change the address of the thread's stack.

You cannot create two concurrent threads that use the same stack address.

The system uses an unspecified (and varying) amount of the stack to "bootstrap" a newly created thread.

2.3.2.6 Setting the Guardsize Attribute

The guardsize attribute represents the minimum size (in bytes) of the guard area for the stack of a thread. A **guard area** can help a multithreaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that DECthreads allocates at the overflow end of the thread's stack. When the thread attempts to access a memory location within this region, a memory addressing violation occurs.

A new thread can be created using a thread attributes object with a default guardsize attribute value. This value is platform dependent, but will always be at least one "hardware protection unit" (that is, at least one page; non-zero values are rounded up to the next integral page size). For more information, see this guide's platform-specfic appendixes.

DECthreads allows your program to specify the size of a thread stack guard area for two reasons:

- For a thread that allocates large data structures on the stack, a large guard area might be required to detect stack overflow.
- Overflow protection of a thread's stack can potentially waste system resources. An application that creates a large number of threads that will never overflow their stacks can conserve system resources by "turning off" guard areas—that is, by specifying a guardsize attribute of zero for each such thread.

To set the guardsize attribute of a thread attributes object, call the `pthread_attr_setguardsize( )` routine. To obtain the value of the guardsize attribute in a thread attributes object, call the `pthread_attr_getguardsize( )` routine.

The `pthread_attr_setguardsize( )` and `pthread_attr_getguardsize( )` routines replace (and are equivalent to) the `pthread_attr_setguardsize_np( )` and `pthread_attr_getguardsize_np( )` routines, respectively, that were available in previous DECthreads releases. The new routines provide a standardized and portable interface, specified by the Single UNIX Specification, Version 2; however, the older routines remain supported.

2.3.2.7 Setting the Contention Scope Attribute

When creating a thread, you can specify the set of threads with which this thread competes for processing resources. This set of threads is called the thread's **contention scope**.

A thread attributes object includes a contention scope attribute. The contention scope attribute specifies whether the new thread competes for processing resources only with other threads in its own process, called **process contention scope**, or with all threads on the system, called **system contention scope**.

Use the `pthread_attr_setscope( )` routine to set an initialized thread attributes object's contention scope attribute. Use the `pthread_attr_getscope( )` routine to obtain the value of the contention scope attribute of an initialized thread attributes object.

In the thread attributes object, set the contention scope attribute's value to `PTHREAD_SCOPE_PROCESS` to specify process contention scope, or set the value to `PTHREAD_SCOPE_SYSTEM` to specify system contention scope.

DECthreads selects at most one thread to execute on each processor at any point in time. DECthreads resolves the contention based on each thread's scheduling attributes (for example, priority) and scheduling policy (for example, round-robin).

A thread created using a thread attributes object whose contention scope attribute is set to `PTHREAD_SCOPE_PROCESS` contends for processing resources with other threads within its own process that also were created with `PTHREAD_SCOPE_PROCESS`. It is unspecified how such threads are scheduled relative to threads in other processes or threads in the same process that were created with `PTHREAD_SCOPE_SYSTEM` contention scope.

A thread created using a thread attributes object whose contention scope attribute is set to `PTHREAD_SCOPE_SYSTEM` contends for processing resources with other threads in any process that also were created with `PTHREAD_SCOPE_SYSTEM`.

Whether process contention scope and system contention scope are available for your program's threads depends on the host operating system. The following table summarizes DECthreads support for thread contention scope by operating system:

Operating System	Available Thread Contention Scopes	Default Thread Contention Scope
DIGITAL UNIX	Process System	Process
OpenVMS	Process	Process
Windows NT	System	System

Note

On DIGITAL UNIX systems:

When a process contention scope thread creates a system contention scope thread that uses the default inheritance of scheduling attributes, the creation can fail with an [EPERM] error condition. This is because system contention scope threads can exceed "default" priority only if the process is running with root privileges.

2.3.3 Terminating a Thread

Terminating a thread means to cause a thread to end its execution. This can occur for any of the following reasons:

- The thread returns from its start routine. This is the usual case.
- The thread calls the `pthread_exit( )` routine. This routine returns a status value in its *value_ptr* argument. This value indicates the thread's exit status to a thread that joins with this thread.
- The thread is *canceled*, by being specified in a call to the `pthread_cancel( )` routine. This routine requests the thread's termination if the thread permits cancelation. See Section 2.3.7 for more information on canceling threads and on controlling whether or not cancelation is permitted.

When a thread terminates, DECthreads performs these actions:

1. DECthreads writes a return value (if one is available) into the terminated thread's thread object:
   - If the thread has been canceled, DECthreads writes the value `PTHREAD_CANCELED` into the thread's thread object.

- If the thread terminated by returning from its start routine, DECthreads copies the return value from the start routine (if one is available) into the thread's thread object. Alternatively, if the thread explictly called `pthread_exit( )`, DECthreads stores the value received in the *value_ptr* argument (from `pthread_exit( )`) into the thread's thread object.

Another thread can obtain this return value by joining with the terminated thread (using `pthread_join( )`). See Section 2.3.5 for a description of joining with a thread.

Note

If the thread terminated by returning from its start routine normally and the start routine does not provide a return value, the results obtained by joining with that thread are unpredictable.

2. If the termination results from a cancelation or a call to `pthread_exit( )`, DECthreads calls, in turn, each cleanup handler that this thread declared (using `pthread_cleanup_push( )`) and that is not yet removed (using `pthread_cleanup_pop( )`). (DECthreads also transfers control to any appropriate `CATCH`, `CATCH_ALL`, or `FINALLY` blocks, as described in Chapter 5.)

   DECthreads calls the terminated thread's most recently pushed cleanup handler first. See Section 2.3.3.1 for more information about cleanup handlers.

   For C++ programmers: At normal exit from a thread, your program will call the appropriate destructor functions, just as if an exception had been raised.

3. To exit the terminated thread due to a call to `pthread_exit( )`, DECthreads raises the `pthread_exit_e` exception. To exit the terminated thread due to cancelation, DECthreads raises the `pthread_cancel_e` exception.

   Your program can use the DECthreads exception package to operate on the generated exception. (In particular, note that the practice of using `CATCH` handlers in place of `pthread_cleanup_push( )` is not portable.) Chapter 5 describes the DECthreads exception package.

4. For each of the terminated thread's thread-specific data keys that has a non-NULL value:

   - DECthreads sets the thread's value for the corresponding key to NULL.

- In turn, DECthreads calls each thread-specific data destructor function in this multithreaded process's list of destructors.

  DECthreads repeats this step until all thread-specific data values in the thread are NULL, or for up to a number of iterations equal to `PTHREAD_DESTRUCTOR_ITERATIONS`. This destroys all thread-specific data associated with the terminated thread. See Section 2.5 for more information about thread-specific data.

5. DECthreads awakens the thread (if there is one) that is currently waiting to join with the terminated thread. That is, DECthreads awakens the thread that is waiting in a call to `pthread_join( )`.
6. If the thread is already detached, DECthreads destroys its thread object. Otherwise, the thread continues to exist until detached or joined with. Section 2.3.4 describes detaching and destroying a thread.

After a thread terminates, its thread object continues to exist. This means that the thread object data structure remains allocated and contains meaningful information—for instance, the thread identifier is still unique and meaningful. This allows another thread to join with the terminated thread (see Section 2.3.5).

When a terminated thread is no longer needed, your program should detach that thread (see Section 2.3.4).

Note

For DIGITAL UNIX systems:

When the initial thread in a multithreaded process returns from the main routine, the entire process terminates, just as it does when a thread calls `exit( )`.

For OpenVMS systems:

When the initial thread in a multithreaded image returns from the main routine, the entire image terminates, just as it does when a thread calls SYS$EXIT.

2.3.3.1 Cleanup Handlers

A **cleanup handler** is a routine you provide that is associated with a particular lexical scope within your program and that can be invoked under certain circumstances when a thread exits that scope. The cleanup handler's purpose is to restore that portion of the program's state that has been changed within the handler's associated lexical scope. In particular, cleanup handlers allow a thread to react to thread-exit and cancelation requests.

Your program declares a cleanup handler for a thread by calling the `pthread_cleanup_push( )` routine. Your program removes (and optionally invokes) a cleanup handler by calling the `pthread_cleanup_pop( )` routine.

A cleanup handler is invoked when the calling thread exits the handler's associated lexical scope, due to:

- Normal exit of the scope (that is, by calling `pthread_cleanup_pop(TRUE)`)
- Thread termination (that is, via a call to the `pthread_exit( )` routine)
- Thread cancelation
- Raising or reraising a DECthreads exception

For each call to `pthread_cleanup_push( )`, your program must contain a corresponding call to `pthread_cleanup_pop( )`. The two calls form a lexical scope within your program. One pair of calls to `pthread_cleanup_push( )` and `pthread_cleanup_pop( )` cannot overlap the scope of another pair. Pairs of calls can be nested.

Because cleanup handlers are specified by the POSIX.1c standard, they are a portable mechanism. An alternative to using cleanup handlers is to define and/or catch DECthreads exceptions with the DECthreads exception package. Chapter 5 describes how to use the DECthreads exception package. DECthreads considers cleanup handler routines, exception handling clauses (that is, `CATCH`, `CATCH_ALL`, `FINALLY`), and C++ object destructors to be functionally equivalent mechanisms.

2.3.4 Detaching and Destroying a Thread

Detaching a thread means to mark a thread for destruction as soon as it terminates. *Destroying* a thread means to *free*, or make available for reuse, the resources occupied by the thread object (and by DECthreads internal resources) associated with that thread.

If a thread has terminated, then detaching that thread causes DECthreads to destroy it immediately. If a thread is detached before it terminates, then DECthreads frees the thread's resources immediately after it terminates.

A thread can be detached explicitly or implicitly:

- To detach a thread explicitly, use the `pthread_detach( )` routine.
- After a target thread has joined with another thread, DECthreads implicitly detaches the target thread when it terminates.

Your program can create a thread that is detached. See Section 2.3.1 for more information about creating a thread.

It is illegal for your program to attempt to join or detach a detached thread. In general, you cannot perform any operation (for example, cancelation) on a detached thread. This is because the thread ID might have become invalid or might have been assigned to a new thread immediately upon termination of the thread. Unless your program is absolutely certain that the detached thread has not terminated (or is not terminating), using the thread ID can have severe consequences.

2.3.5 Joining With a Thread

Joining with a thread means to suspend this thread's execution until another thread (the target thread) terminates. In addition, DECthreads detaches the target thread after it terminates.

For one thread to join with a functionally related thread is one way to synchronize their execution.

A thread joins with another thread by calling the `pthread_join( )` routine and specifying the thread identifier of the thread. If the target thread has already terminated, then this thread does not wait.

The target thread of a join operation must be created with the detachstate attribute of its thread attributes object set to `PTHREAD_CREATE_JOINABLE`.

Keep in mind these restrictions about joining with a thread:

- If more than one thread calls `pthread_join( )` and specifies the same thread identifier, your program's behavior is undefined. This is because DECthreads detaches the target thread after completing the first join.
- If a thread specifies its own thread identifier when calling `pthread_join( )` routine, the result is a deadlock. See Section 3.6.3 for more information about deadlocks.
- Do not confuse `pthread_join( )` with other routines that cause waits and that are related to the use of a particular DECthreads feature. For example, use the `pthread_cond_wait( )` or `pthread_cond_timedwait( )` routine to wait for a condition variable to be signaled or broadcasted. (See Section 2.4.2 for more information on condition variables.)

2.3.6 Scheduling a Thread

Scheduling means to evaluate and change the states of the process's threads. As your multithreaded program runs, DECthreads detects whether each thread is ready to execute, is waiting for completion of a system call, has terminated, and so on.

Also, for each thread DECthreads regularly checks whether that thread's scheduling priority and scheduling policy, when compared with those of the process's other threads, entail forcing a change in that thread's state. Remember that scheduling priority specifies the "precedence" of a thread in the application. Scheduling policy provides a mechanism to control how DECthreads interprets that priority as your program runs.

To understand this section, you must be familiar with the concepts presented in these sections:

- Section 2.3.2.1 on inheriting of scheduling attributes by created threads
- Section 2.3.2.2 on scheduling policies, including how each policy handles thread scheduling priority
- Section 2.3.2.3 on thread scheduling priorities

2.3.6.1 Calculating the Scheduling Priority

A thread's scheduling priority falls within a range of values, depending on its scheduling policy. To specify the minimum or maximum scheduling priority for a thread, use the `sched_get_priority_min( )` or `sched_get_priority_max( )` routines—or use the appropriate nonportable symbol such as `PRI_OTHER_MIN` or `PRI_OTHER_MAX`. Priority values are integers, so you can specify a value between the minimum and maximum priority using an appropriate arithmetic expression.

For example, to specify a scheduling priority value that is midway between the minimum and maximum for the `SCHED_OTHER` scheduling policy, use the following expression (coded appropriately for your programming language):

```
   .
   .
   .
pri_other_mid = ( sched_get_priority_min(SCHED_OTHER) +
                  sched_get_priority_max(SCHED_OTHER)   ) / 2
```

where *pri_other_mid* represents the priority value you want to set.

Avoid using literal numerical values to specify a scheduling priority setting, because the range of priorities can change from implementation to implementation. Values outside the specified range for each scheduling policy might be invalid.

2.3.6.2 Effects of Scheduling Policy

To demonstrate the results of the different scheduling policies, consider the following example: A program has four threads, A, B, C, and D. For each scheduling policy, three scheduling priorities have been defined: minimum, middle, and maximum. The threads have the following priorities:

A	minimum
B	middle
C	middle
D	maximum

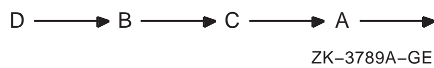
On a uniprocessor system, only one thread can run at any given time. The ordering of execution depends upon the relative scheduling policies and priorities of the threads. Given a set of threads with fixed priorities such as the previous list, their execution behavior is typically predictable. However, in a symmetric multiprocessor (or SMP) system the execution behavior is completely indeterminate. Although the four threads have differing priorities, a multiprocessor system might execute two or more of these threads simultaneously.

When you design a multithreaded application that uses scheduling priorities, it is critical to remember that scheduling is not the same as synchronization. That is, you cannot assume that a higher-priority thread can access shared data without interference from lower-priority threads. For example, if one thread has a FIFO scheduling policy and the highest scheduling priority setting, while another has a background scheduling policy and the lowest scheduling priority setting, DECthreads might allow the two threads to run at the same time. As a corollary, on a four-processor system you also cannot assume that the four highest-priority threads are executing simultaneously at any particular moment.

The following figures demonstrate how DECthreads schedules a set of threads on a uniprocessor based on whether each thread has the FIFO, RR, or throughput setting for its scheduling policy attribute. Assume that all waiting threads are ready to execute when the current thread waits or terminates and that no higher-priority thread is awakened while a thread is executing (that is, executing during the flow shown in each figure).

Figure 2–1 shows a flow with FIFO scheduling.

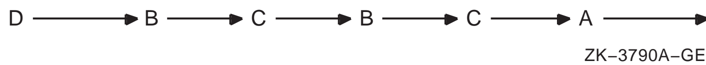
Figure 2–1 Flow with FIFO Scheduling



Thread D executes until it waits or terminates. Next, although thread B and thread C have the same priority, thread B starts because it has been waiting longer than thread C. Thread B executes until it waits or terminates, then thread C executes until it waits or terminates. Finally, thread A executes.

Figure 2–2 shows a flow with RR scheduling.

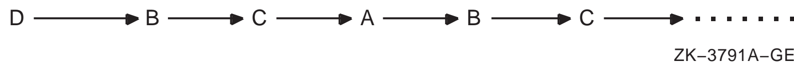
Figure 2–2 Flow with RR Scheduling



Thread D executes until it waits or terminates. Next, thread B and thread C are timesliced, because they both have the same priority. Finally, thread A executes.

Figure 2–3 shows a flow with Default scheduling.

Figure 2–3 Flow with Default Scheduling



Threads D, B, C, and A are timesliced, even though thread A has a lower priority than the others. Thread A receives less execution time than thread D, B, or C if any of those are ready to execute as often as Thread A. However, the default scheduling policy protects thread A against indefinitely being blocked from executing.

Because low-priority threads eventually run, the default scheduling policy protects against occurrences of thread starvation and priority inversion, which are discussed in Section 3.5.2.

2.3.7 Canceling a Thread

Canceling a thread means to request the termination of a target thread as soon as possible. A thread can request the cancelation of another thread or itself.

Thread cancelation is a three-stage operation:

1. A cancelation request is *posted* for the target thread. This occurs when some routine in some thread calls `pthread_cancel( )`.
2. The posted cancelation request is *delivered* to the target thread. This occurs when the target thread invokes a routine that is a cancelation point. (See Section 2.3.7.4 for a discussion of routines that are cancelation points.)

   If the target thread's cancelability state is *disabled*, the target thread does not receive the cancelation request until the next cancelation point after the cancelability state is set to *enabled*. See Section 2.3.7.3 for how to control a thread's cancelability.
3. After the target thread receives the cancelation request, it responds to that request by invoking, in turn, its cleanup handler routines. Previously in its life the target thread might have pushed pointers to cleanup handler routines (using the `pthread_cleanup_push( )` routine) on its handler stack. When the target thread receives the cancelation request, DECthreads automatically calls, in turn, each cleanup handler routine on the handler stack. (A cleanup handler can be removed from the handler stack by calling `pthread_cleanup_pop( )`.)

2.3.7.1 Thread Cancelation Implemented Using Exceptions

The DECthreads **pthread** and **tis** interfaces implement thread cancelation using DECthreads exceptions. Using the DECthreads exception package, it is possible for a thread (to which a cancelation request has been delivered) explicitly to catch the DECthreads-defined thread cancelation exception (`pthread_cancel_e`) and to perform cleanup actions accordingly. After catching this exception, the exception handler code should always reraise the exception, to avoid breaking the "contract" that cancelation leads to thread termination.

Chapter 5 describes the DECthreads exception package.

2.3.7.2 Thread Return Value After Cancelation

When DECthreads terminates a thread due to cancelation, it writes the return value `PTHREAD_CANCELED` into the thread's thread object. This is because cancelation prevents the thread from calling `pthread_exit( )` or returning from its start routine.

2.3.7.3 Controlling Thread Cancelation

Each thread controls whether it can be canceled (that is, whether it receives requests to terminate) and how quickly it terminates after receiving the cancelation request, as follows:

A thread's **cancelability state** determines whether it receives a cancelation request. When created, a thread's cancelability state is *enabled*. If the cancelability state is *disabled*, the thread does not receive cancelation requests.

If the thread's cancelability state is *enabled*, use the `pthread_testcancel()` routine to request the delivery of any pending cancelation request. This routine enables the program to permit cancelation to occur at places where it might not otherwise be permitted, and it is especially useful within very long loops to ensure that cancelation requests are noticed within a reasonable time.

If its cancelability state is *disabled*, the thread cannot be terminated by any cancelation request. This means that a thread could wait indefinitely if it does not come to a normal conclusion; therefore, exercise care.

After a thread has been created, use the `pthread_setcancelstate()` routine to change its cancelability state.

After a thread has been created, use the `pthread_setcanceltype()` routine to change its **cancelability type**, which determines whether it responds to a cancelation request at cancelation points (*synchronous cancelation*), or at any point in its execution (*asynchronous cancelation*).

Initially, a thread's cancelability type is *deferred*, which means that the thread receives a cancelation request only at cancelation points—for example, when a call to the `pthread_cond_wait()` routine is made. If you set a thread's cancelability type to *asynchronous*, the thread can receive a cancelation request at any time.

Note

If the cancelability state is *disabled*, the thread cannot be canceled regardless of the cancelability type. Setting cancelability type to *deferred* or *asynchronous* is relevant only when the thread's cancelability state is *enabled*.

2.3.7.4 Cancelation Points

A **cancelation point** is a routine that delivers a posted cancelation request to that request's target thread. The POSIX.1c standard specifies routines that are cancelation points.

The following routines in the DECthreads **pthread** interface are cancelation points:

```
pthread_cond_timedwait( )
pthread_cond_wait( )
pthread_delay_np( )
pthread_join( )
pthread_setcanceltype( )
```
    (when setting the cancelability type to *asynchronous*)
```
pthread_testcancel( )
```

The following routines in the DECthreads **tis** interface are cancelation points:

```
tis_cond_wait( )
tis_testcancel( )
```

Other routines that are also cancelation points are mentioned in the operating system-specific appendixes of this guide. Refer to the following thread cancelability for system services topics:

- Section A.4 for DIGITAL UNIX
- Section B.7 for OpenVMS
- Section C.5 for Windows NT

2.3.7.5 Cleanup from Synchronous Cancelation

When a cancelation request is delivered to a thread, the thread could be holding some resources, such as locked mutexes or allocated memory. Your program must release these resources before the thread terminates.

DECthreads provides two equivalent mechanisms that can do the cleanup during cancelation, as follows:

- Use the `pthread_cleanup_push( )` and `pthread_cleanup_pop( )` routines to establish and remove cleanup handlers for a section of code that contains a cancelation point. When a cancelation request is delivered, the routine specified in `pthread_cleanup_push( )` is called. This allows the thread to unlock mutexes or otherwise release resources held in the current scope. Each routine can establish one or more cleanup handlers using `pthread_cleanup_push( )`. When the handler is no longer needed it is removed by calling `pthread_cleanup_pop( )`. The *execute* argument to

pthread_cleanup_pop( ) indicates whether the handler routine should be called when it is removed.

Calling the cleanup handler automatically on removal is convenient when the thread is about to leave the scope and you must perform the cleanup actions even though the thread wasn't canceled (for example, releasing the mutex after waking up from a condition variable wait). (For DIGITAL UNIX, see also the pthread(3) reference pages.)

- As described in Chapter 5, use the DECthreads exceptions package TRY/CATCH or TRY/FINALLY macros to clean up during a cancelation request. A cancelation request is sent to the thread by raising a special DECthreads exception. Thus, code that contains a cancelation point can be placed inside a TRY block, and a CATCH or FINALLY block can be used to release the resources the thread is holding when the cancelation request is sent. Note that code should always reraise the cancelation exception; failing to do so will result in the thread not terminating as requested.

2.3.7.6 Cleanup from Asynchronous Cancelation

Because it is impossible to predict exactly when an asynchronous cancelation request will be delivered, it is extremely difficult for a program to recover properly. For this reason, an asynchronous cancelability type should be set only within regions of code that do not need to clean up in any way, such as straight-line code or tight looping code that is compute-bound and that makes no calls and holds no resources.

While a thread's cancelability type is asynchronous, do not call any routine unless it is explicitly documented as "safe for asynchronous cancelation." In particular, you can never use asynchronous cancelability type in code that allocates or frees memory, or that locks or unlocks mutexes—because the cleanup code cannot reliably determine the state of the resource.

Note

None of the general run-time routines are safe for asynchronous cancelation, and likewise for all DECthreads routines except pthread_setcanceltype( ).

For additional information about accomplishing asynchronous cancelation for your platform, see Section A.4, Section B.7, and Section C.5.

2.3.7.7 Example of Thread Cancelation Code

Example 2–1 shows a thread control and cancelation example.

Example 2–1 pthread Cancel

```
/*
 * Pthread Cancel Example
 */
/*
 * Outermost cancelation state
 */
{
 .
 .
 .
int     s, outer_c_s, inner_c_s;
 .
 .
 .
/* Disable cancelation, saving the previous setting.    */
s = pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &outer_c_s);
if(s == EINVAL)
   printf("Invalid Argument!\n");
else if(s == 0)
         .
         .
         .
        /* Now cancelation is disabled.    */
 .
 .
 .
/* Enable cancelation.  */
```

(continued on next page)

Example 2–1 (Cont.) pthread Cancel

```
{
 .
 .
 .
 s = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &inner_c_s);
 if(s == 0)
    .
    .
    .
    /* Now cancelation is enabled.  */
    .
    .
    .
    /* Enable asynchronous cancelation this time.  */
        {
         .
         .
         .
         /* Enable asynchronous cancelation.  */
         int   outerasync_c_s, innerasync_c_s;
         .
         .
         .
         s = pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS,
                                    &outerasync_c_s);
         if(s == 0)
            .
            .
            .
            /* Now asynchronous cancelation is enabled.  */
            .
            .
            .
            /* Now restore the previous cancelation state (by
             * reinstating original asynchronous type cancel).
             */
            s = pthread_setcanceltype (outerasync_c_s,
                                       &innerasync_c_s);
            if(s == 0)
               .
               .
               .
               /* Now asynchronous cancelation is disabled,
                * but synchronous cancelation is still enabled.
                */
```

(continued on next page)

Example 2–1 (Cont.) pthread Cancel

```
                }
        .
        .
        .
       }
 .
 .
 .
/* Restore to original cancelation state.    */

s = pthread_setcancelstate (outer_c_s, &inner_c_s);
if(s == 0)
   .
   .
   .
   /* The original (outermost) cancelation state is now reinstated. */
}
```

2.4 Synchronization Objects

In a multithreaded program, you must use synchronization objects whenever there is a possibility of conflict in accessing shared data. The following sections discuss two kinds of DECthreads synchronization objects: mutexes and condition variables.

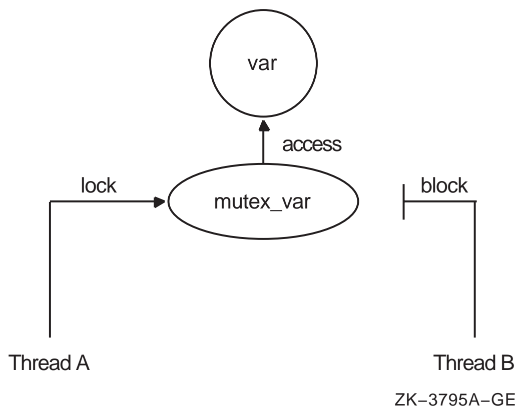
2.4.1 Mutexes

A **mutex** (or *mut*ual *ex*clusion) object is used by multiple threads to ensure the integrity of a shared resource that they access, most commonly shared data, by allowing only one thread to access it at a time.

A mutex has two states, locked and unlocked. A locked mutex has an owner—the thread that locked the mutex. It is illegal to unlock a mutex not owned by the calling thread.

For each piece of shared data, all threads accessing that data must use the same mutex: each thread locks the mutex before it accesses the shared data and unlocks the mutex when it is finished accessing that data. If the mutex is locked by another thread, the thread requesting the lock either waits for the mutex to be unlocked or returns, depending on the lock routine called (see Figure 2–4).

Figure 2–4 Only One Thread Can Lock a Mutex



Each mutex must be initialized before use. DECthreads supports static initialization at compile time, using one of the macros provided in the `pthread.h` header file, as well as dynamic initialization at run time by calling `pthread_mutex_init( )`. This routine allows you to specify an attributes object, which allows you to specify the mutex type. The types of mutexes are described in the following sections.

2.4.1.1 Normal Mutex

A **normal mutex** is locked exactly once by a thread. If a thread tries to lock the mutex again without first unlocking it, the thread waits for itself to release the lock and deadlocks.

This is the most efficient form of mutex. When using interface and function inlining (optional), you can often lock and unlock a normal mutex without a call to DECthreads.

A normal mutex usually does not check thread ownership—that is, a deadlock will result if the owner attempts to "relock" the mutex. The system usually will not report an erroneous attempt to unlock a mutex not owned by the calling thread.

2.4.1.2 Default Mutex

This is the name reserved by the Single UNIX Specification, Version 2, for a vendor's POSIX.1c threads implementation's default mutex type. For DECthreads, "default" is the same as "normal." This might not be true for other implementations of the Single UNIX Specification, Version 2, which could choose errorcheck, recursive, or even some nonportable mutex types as the default.

2.4.1.3 Recursive Mutex

A **recursive mutex** can be locked more than once by a given thread without causing a deadlock. The thread must call the `pthread_mutex_unlock( )` routine the same number of times that it called the `pthread_mutex_lock( )` routine before another thread can lock the mutex.

When a thread first successfully locks a recursive mutex, it owns that mutex and the lock count is set to 1. Any other thread attempting to lock the mutex blocks until the mutex becomes unlocked. If the owner of the mutex attempts to lock the mutex again, the lock count is incremented, and the thread continues running.

When an owner unlocks a recursive mutex, the lock count is decremented. The mutex remains locked and owned until the count reaches zero. It is an error for any thread other than the owner to attempt to unlock the mutex.

A recursive mutex is useful when a thread requires exclusive access to a piece of data, but must call another routine (or itself) that also requires exclusive access to the data. A recursive mutex allows nested attempts to lock the mutex to succeed rather than deadlock.

This type of mutex is called “recursive” because it allows you a capability not permitted by a normal (default) mutex. However, its use requires more careful programming. For instance, if a recursively locked mutex were used with a condition variable, the unlock performed for a `pthread_cond_wait()` or `pthread_cond_timedwait()` would not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate, and the thread would wait indefinitely. See Section 2.4.2 for information on the condition variable wait and timed wait routines.

2.4.1.4 Errorcheck Mutex

An **errorcheck mutex** is locked exactly once by a thread, like a normal mutex. If a thread tries to lock the mutex again without first unlocking it, the thread receives an error. If a thread other than the owner tries to unlock an errorcheck mutex, an error is returned. Thus, errorcheck mutexes are more informative than normal mutexes because normal mutexes deadlock in such a case, leaving you to determine why the thread no longer executes. Errorcheck mutexes are useful during development and debugging. Errorcheck mutexes can be replaced with normal mutexes when the code is put into production use, or left to provide the additional checking.

Errorcheck mutexes are slower than normal mutexes. They cannot be locked without generating a call into DECthreads, and they do more internal tracking.

2.4.1.5 Mutex Operations

To lock a mutex, use one of the following routines, depending on what you want to happen after the mutex is locked:

- `pthread_mutex_lock()`

  If the mutex is locked, the thread waits for the mutex to become available.

- `pthread_mutex_trylock()`

  This routine returns immediately with a status indicating whether or not it was able to lock the mutex. Based on this return value, the calling thread can take the appropriate action.

When a thread is finished accessing a piece of shared data, it unlocks the associated mutex by calling the `pthread_mutex_unlock()` routine. If other threads are waiting on the mutex, one is placed in the ready state. If more than one thread is waiting on the mutex, the scheduling policy (see Section 2.3.2.2) and the scheduling priority (see Section 2.3.2.3) determine which thread is readied, and the next running thread that requests it locks the mutex.

The mutex is not automatically granted to the first waiter. If the unlocking thread attempts to relock the mutex before the first waiter gets a chance to run, the unlocking thread will succeed in relocking the mutex, and the first waiter may be forced to reblock.

You can destroy a mutex—that is, reclaim its storage—by calling the `pthread_mutex_destroy( )` routine. Use this routine only after the mutex is no longer needed by any thread. It is invalid to attempt to destroy a mutex while it is locked.

Warning

DECthreads does not currently detect deadlock conditions involving more than one mutex, but may in the future. *Never write code that depends upon DECthreads* ***not*** *reporting a particular error condition.*

2.4.1.6 Mutex Attributes

A **mutex attributes object** allows you to specify values other than the defaults for mutex attributes when you initialize a mutex with the `pthread_mutex_init( )` routine.

The mutex type attribute specifies whether a mutex is default, normal, recursive, or errorcheck. Use the `pthread_mutexattr_settype( )` routine to set the mutex type attribute in an initialized mutex attributes object. Use the `pthread_mutexattr_gettype( )` routine to obtain the mutex type from an initialized mutex attributes object.

The `pthread_mutexattr_settype( )` and `pthread_mutexattr_gettype( )` routines replace (and are equivalent to) the `pthread_mutexattr_settype_np( )` and `pthread_mutexattr_gettype_np( )` routines, respectively, that were available in previous DECthreads releases. The new routines provide a standardized interface; however, the older routines remain supported.

If you do not use a mutex attributes object to select a mutex type, calling the `pthread_mutex_init( )` routine initializes a normal (default) mutex by default.

2.4.2 Condition Variables

A **condition variable** is a synchronization object used in conjunction with a mutex. It allows a thread to block its own execution until some shared data object reaches a particular state. A mutex controls access to shared data; a condition variable allows threads to wait for that data to enter a defined state.

The state is defined by a predicate in the form of a Boolean expression. A predicate may be a Boolean variable in the shared data or the predicate may be indirect; testing whether a counter has reached a certain value, or whether a queue is empty.

Each predicate should have its own unique condition variable. Sharing a single condition variable between more than one predicate can introduce inefficiency or errors unless you use extreme care.

Cooperating threads test the predicate and wait on the condition variable while the predicate is not in the desired state. For example, one thread in a program produces work-to-do packets and another thread consumes these packets (does the work). If there are no work-to-do packets when the consumer thread checks, that thread waits on a work-to-do condition variable. When the producer thread produces a packet, it signals the work-to-do condition variable.

You must associate a mutex with a condition variable.

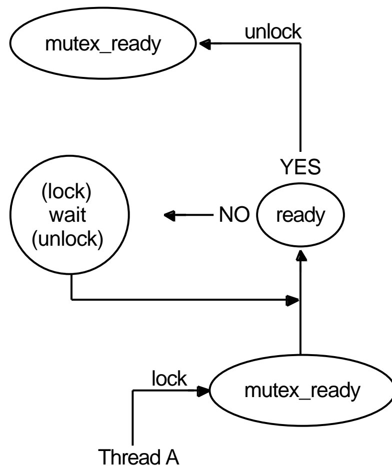
A thread uses a condition variable as follows:

1. A thread locks a mutex for some shared data and then tests the relevant predicate. If it is not in the proper state, the thread waits on a condition variable associated with the predicate. Waiting on the condition variable automatically unlocks the mutex. It is essential that the mutex be unlocked, because another thread needs to acquire the mutex in order to put the data in the state required by the waiting thread.
2. When the thread that acquires the mutex puts the data in the appropriate state, it wakes a waiting thread by signaling the condition variable.
3. One thread comes out of its wait state with the mutex locked (the condition wait relocks the mutex before returning from the thread). Other threads waiting on the condition variable remain blocked.

It is important to wait on the condition variable and evaluate the predicate in a `while` loop. This ensures that the program checks the predicate *after* it returns from the condition wait. This is due to the fact that, because threads execute asynchronously, another thread might consume the state before an awakened thread can run. Also, the test protects against spurious wake-ups and provides clearer program documentation.

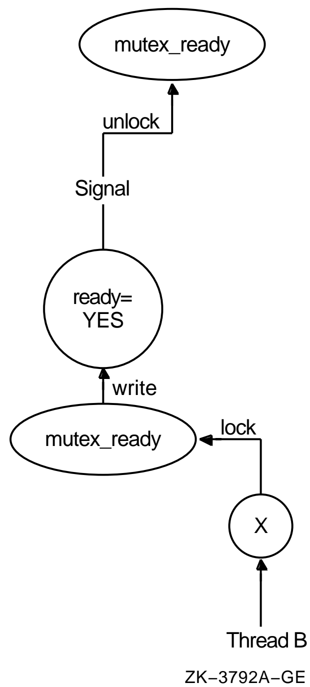
For example, a thread A may need to wait for a thread B to finish a task X before thread A proceeds to execute a task Y. Thread B can tell thread A that it has finished task X by putting a TRUE or FALSE value in a shared variable (the predicate). When thread A is ready to execute task Y, it looks at the shared variable to see if thread B is finished (see Figure 2–5).

Figure 2–5 Thread A Waits on Condition Ready



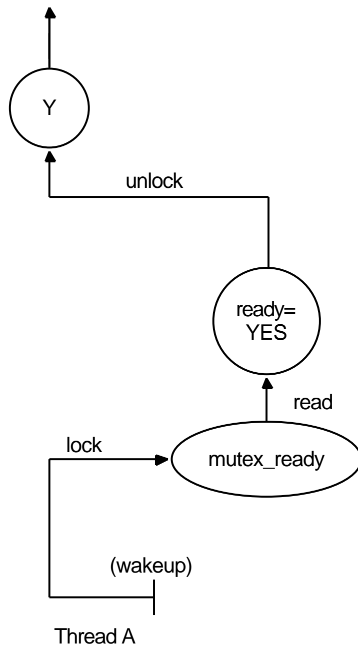
First, thread A locks the mutex named *mutex_ready* that is associated with the shared variable named *ready*. Then it reads the value in *ready*. This test is called the predicate. If the predicate indicates that thread B has finished task X, then thread A can unlock the mutex and proceed with task Y. If the predicate indicates that thread B has not yet finished task X, however, then thread A waits for the predicate to change by calling the `pthread_cond_wait( )` routine. This automatically unlocks the mutex, allowing thread B to lock the mutex when it has finished task X. Thread B updates the shared data (predicate) to the state thread A is waiting for and signals the condition variable by calling the `pthread_cond_signal( )` routine (see Figure 2–6).

Figure 2–6 Thread B Signals Condition Ready



Thread B releases its lock on the shared variable's mutex. As a result of the signal, thread A wakes up, implicitly regaining its lock on the condition variable's mutex. It then verifies that the predicate is in the correct state, and proceeds to execute task Y (see Figure 2–7).

Figure 2–7 Thread A Wakes and Proceeds



Note that although the condition variable is used for communication among threads, the communication is anonymous. Thread B does not necessarily know that thread A is waiting on the condition variable that thread B signals, and thread A does not know that it was thread B that awakened it from its wait on the condition variable.

Use the `pthread_cond_init( )` routine to initialize a condition variable. To create condition variables as part of your program's one-time initialization code, see Section 3.7. You can also statically initialize condition variables using the `PTHREAD_COND_INITIALIZER` macro provided in the `pthread.h` header file.

Use the `pthread_cond_wait( )` routine to cause a thread to wait until the condition is signaled or broadcasted. This routine specifies a condition variable and a mutex that you have locked. If you have not locked the mutex, the results of `pthread_cond_wait( )` are unpredictable.

The `pthread_cond_wait( )` routine automatically unlocks the mutex and causes the calling thread to wait on the condition variable until another thread calls one of the following routines:

- `pthread_cond_signal( )`, to wake one thread that is waiting on the condition variable
- `pthread_cond_broadcast( )`, to wake all threads that are waiting on a condition variable
- `pthread_cond_signal_int_np( )`, to wake a thread from a signal handler (for DIGITAL UNIX) or AST routine (for OpenVMS). There are special restrictions on these functions (see Part II).

If a thread signals or broadcasts on a condition variable and there are no threads waiting at that time, the signal or broadcast has no effect. The next thread to wait on that condition variable blocks until the next signal or broadcast. (Alternatively, the nonportable `pthread_cond_signal_int_np( )` routine creates a pending wake condition, which causes the next wait on the condition variable to complete immediately.)

If you want to limit the time that a thread waits for a condition to be signaled or broadcasted, use the `pthread_cond_timedwait( )` routine. This routine specifies the condition variable, mutex, and absolute time at which the wait should expire if the condition variable has not been signaled or broadcasted.

You can destroy a condition variable and reclaim its storage by calling the `pthread_cond_destroy( )` routine. Use this routine only after the condition variable is no longer needed by any thread. A condition variable cannot be destroyed while one or more threads are waiting on it.

2.4.3 Condition Variable Attributes

Currently, no attributes affecting condition variables are defined. You cannot change any attributes in the condition variable attributes object.

The `pthread_condattr_init( )` and `pthread_condattr_destroy( )` routines are provided for future expandability of the DECthreads **pthread** interface and to conform with the POSIX.1c standard. In this DECthreads release these routines offer no useful function, because there are no DECthreads routines available at this time for setting the attributes of condition variable attributes objects.

2.5 Thread-Specific Data

Each thread can use an area of DECthreads-private memory where DECthreads stores thread-specific data objects. Use this memory to associate arbitrary data with a thread's context. Consider this as the ability to add user-specified fields to the current thread's context or as global variables that have private values in each thread.

A thread-specific data key is shared by all threads within the process—each thread has its own unique value for that shared key.

Use the following routines to create and access thread-specific data:

- Use the `pthread_key_create( )` routine to create a unique key value. One call to `pthread_key_create( )` creates a thread-specific data key shared by all threads. In addition, your program can specify a destructor routine to destroy the context value associated with this key when any thread terminates.

  The process must create each key exactly once—otherwise, subsequent creates will overwrite the first. See Section 3.7 for information about the one-time initialization in a threaded environment.

- Use `pthread_setspecific( )` to associate thread-specific data objects with a key value. Each thread can associate its own private data with the same key.

  For example, each thread might store a pointer to a block of dynamically allocated memory that it has reserved. Although each thread has its own block of memory, your code always uses the same key to get the current thread's block.

- Use `pthread_getspecific( )` to obtain the data associated with a key. This routine obtains the current thread's thread-specific data value associated with a specified key.

3 Programming with Threads

This chapter discusses programming disciplines that you should follow as you use DECthreads routines in your programs. Pertinent examples include programming for asynchronous execution, choosing a synchronization mechanism, avoiding priority scheduling problems, making code thread safe, and working with code that is not thread safe.

3.1 Designing Code for Asynchronous Execution

When programming with threads, always keep in mind that the execution of a thread is inherently asynchronous with respect to other threads running the system (or in the process).

In short, there is no guarantee of when a thread will start. It can start immediately or not for a significant period of time, depending on the priority of the thread in relation to other threads that are currently running. When a thread will start can also depend on the behavior of other processes, as well as on other threaded subsystems within the current process.

You *cannot* depend upon any synchronization between two threads unless you explicitly code that synchronization into your program using one of the following:

- Mutexes
- A properly tested application predicate loop on a condition variable
- A call to join with a thread you expect to terminate
- An equivalent platform-dependent programming construct, such as VAX interlocked instructions or Alpha load locked/store conditional sequences

Some implementations of threads operate by context-switching threads in user mode, within a single operating system process. Context switches between such threads occur only at relatively determinate times, such as when you make a blocking call to the threads library or when a timeslice interrupt occurs. This type of threading library might be termed “slightly asynchronous,” because such a library tolerates many classes of errors in your application.

Systems that support **kernel threads** are less "forgiving" because context switches between threads can occur more frequently and for less deterministic reasons. Systems that allow threads within a single process to run simultaneously on multiple processors are even less forgiving.

The following subsections present examples of programming errors.

3.1.1 Avoid Passing Stack Local Data

Avoid creating a thread with an argument that points to stack local data, or to global or static data that is serially reused for a sequence of threads.

Specifically, the thread started with a pointer to stack local data may not start until the creating thread's routine has returned, and the storage may have been changed by other calls. The thread started with a pointer to global or static data may not start until the storage has been reused to create another thread.

3.1.2 Initialize DECthreads Objects Before Thread Creation

Initialize DECthreads objects (such as mutexes) or global data that a thread uses *before* creating that thread.

On slightly asynchronous systems this is often safe, because the thread will probably not run until the creator blocks. Thus, the error can go undetected initially. On another system (or in a later release of the operating system) that supports kernel threading, the created thread may run immediately, before the data has been initialized. This can lead to failures that are difficult to detect. Note that a thread may run to completion, before the call that created it returns to the creator. The system load may affect the timing as well.

Before your program creates a thread, it should set up all requirements that the new thread needs in order to execute. For example, if your program must set the new thread's scheduling parameters, do so with attributes objects when you create it, rather than trying to use `pthread_setschedparam( )` or other routines afterwards. To set global data for the new thread or to create synchronization objects, do so before you create the thread, else set them in a `pthread_once( )` initialization routine that is called from each thread.

3.1.3 Don’t Use Scheduling As Synchronization

Avoid using scheduling policy and scheduling priority attributes of threads as a synchronization mechanism.

In a uniprocessor system, only one thread can run at a time, and when a higher-priority (real-time policy) thread becomes runnable, it immediately preempts a lower-priority running thread. Therefore, a thread running at higher priority might erroneously be presumed not to need a mutex to access shared data.

On a multiprocessor system, higher- and lower-priority threads are likely to run at the same time. Situations can even arise where higher-priority threads are waiting to run while the threads that are running have a lower priority.

Regardless of whether your code will run only on a uniprocessor implementation, never try to use scheduling as a synchronization mechanism. Even on a uniprocessor system, your SCHED_FIFO thread can become blocked on a mutex (perhaps in a called library routine), on an I/O operation, or even a page fault. Any of these might allow a lower priority thread to run.

3.2 Memory Synchronization Between Threads

Your multithreaded program must ensure that access to data shared between threads is synchronized.

The POSIX.1c standard requires that, when calling the following routines, a thread synchronizes its memory access with respect to other threads:

`fork()`
`pthread_create()`
`pthread_join()`
`pthread_mutex_lock()`
`pthread_mutex_trylock()`
`pthread_mutex_unlock()`
`pthread_cond_wait()`
`pthread_cond_timedwait()`
`pthread_cond_signal()`
`pthread_cond_broadcast()`
`sem_post()`
`sem_trywait()`
`sem_wait()`
`wait()`
`waitpid()`

If a call to one of these routines returns an error, synchronization is not guaranteed. For example, an unsuccessful call to `pthread_mutex_trylock()` does not necessarily provide actual synchronization.

Synchronization is a “protocol” among cooperating threads, not a single operation. That is, unlocking a mutex does not guarantee memory synchronization with all other threads—only with threads that later perform some synchronization operation themselves, such as locking a mutex.

3.3 Using Shared Memory

Most threads do not operate independently. They cooperate to accomplish a task, and cooperation requires communication. There are many ways that threads can communicate, and which method is most appropriate depends on the task.

Threads that cooperate only rarely (for example, a boss thread that only sends off a request for workers to do long tasks) may be satisfied with a relatively slow form of communication. Threads that must cooperate more closely (for example, a set of threads performing a parallelized matrix operation) need fast communication—maybe even to the extent of using machine-specific hardware operations.

Most mechanisms for thread communication involve the use of shared memory, exploiting the fact that all threads within a process share their full address space. Although all addresses are shared, there are three kinds of memory that are characteristically used for communication. The following sections describe the scope (or, the range of locations in the program where code can access the memory) and lifetime (or, the length of time the memory exists) of each of the three types of memory.

3.3.1 Using Static Memory

Static memory is allocated by the language compiler when it translates source code, so the scope is controlled by the rules of the compiler. For example, in the C language, a variable declared as `extern` can be accessed anywhere, and a `static` variable can be referenced within the source module or routine, depending on where it is declared.

In this discussion, static memory is not the same as the C language `static` storage class. Rather, static memory refers to any variable that is permanently allocated at a particular address for the life of the program.

It is appropriate to use static memory in your multithreaded program when you know that only one instance of an object exists throughout the application. For example, if you want to keep a list of active contexts or a mutex to control some shared resource, you would not want individual threads to have their own copies of that data.

The scope of static memory depends on your programming language's scoping rules. The lifetime of static memory is the life of the program.

3.3.2 Using Stack Memory

Stack memory is allocated by code generated by the language compiler at run time, generally when a routine is initially called. When the program returns from the routine, the storage ceases to be valid (although the addresses still exist and might be accessible).

Generally, the storage is valid while the routine runs, and the actual address can be calculated and passed to other threads; however, this depends on programming language rules. If you pass the address of stack memory to another thread, you must ensure that all other threads are finished processing that data before the routine returns; otherwise the stack will be cleared, and values might be altered by subsequent calls. The other threads will not be able to determine that this has happened, and erroneous behavior will result.

The scope of stack memory is the routine or a block within the routine. The lifetime is no longer than the time during which the routine executes.

3.3.3 Using Dynamic Memory

Dynamic memory is allocated by the program as a result of a call to some memory management routine (for example, the C language run-time routine `malloc()` or the OpenVMS common run-time routine LIB$GET_VM).

Dynamic memory is referenced through pointer variables. Although the pointer variables are scoped depending on their declaration, the dynamic memory itself has no intrinsic scope or lifetime. It can be accessed from any routine or thread that is given its address and will exist until explicitly made free. In a language supporting automatic garbage collection, it will exist until the run-time system detects that there are no references to it. (If your language supports garbage collection, be sure the garbage collector is thread safe.)

The scope of dynamic memory is anywhere a pointer containing the address can be referenced. The lifetime is from allocation to deallocation.

Typically dynamic memory is appropriate to manage persistent context. For example, in a thread-reentrant routine that is called multiple times to return a stream of information (such as to list all active connections to a server or to return a list of users), using dynamic memory allows the program to create multiple contexts that are independent of all the program's threads. Thus, multiple threads could share a given context, or a single thread could have more than one context.

3.4 Managing a Thread's Stack

For each thread created by your program, DECthreads sets a default stack size that is acceptable to most applications. You can also set the *stacksize* attribute in a thread attributes object, to specify the stack size needed by the next thread created.

This section discusses the cases in which the stack size is insufficient (resulting in stack overflow) and how to determine the optimal size of the stack.

Most compilers on VAX systems do not probe the stack. Portable code that supports threads should use as little stack memory as practical.

Most compilers on Alpha systems generate code in the procedure prologue that probes the stack, ensuring there is enough space for the procedure to run.

3.4.1 Sizing the Stack

To determine the optimal size of a thread's stack, multiply the largest number of nested subroutine calls by the size of the call frames and local variables. Add to that number an extra amount of memory to accommodate interrupts. Determining this figure is difficult because stack frames vary in size and because it might not be possible to estimate the depth of library routine call frames.

You can also run your program using a profiling tool that measures actual stack use. This is commonly done by "poisoning" the stack before it is used by writing a distinctive pattern, and then checking for that pattern after the thread completes. Remember: Use of profiling monitoring tools typically *increases* the amount of stack memory that your program uses.

3.4.2 Using a Stack Guard Area

By default, at the overflow end of each thread's stack DECthreads allocates a **guard area**, or a region of no-access memory. A guard area can help a multithreaded program detect overflow of a thread's stack. When the thread attempts to access a memory location within this region, a memory addressing violation occurs.

For a thread that allocates large data structures on the stack, create that thread using a thread attributes object in which a large guardsize attribute value has been set. A large stack guard region can help to prevent one thread from overflowing into another thread's stack region.

The low-level memory regions that form a stack guard region are also known as **guard pages**.

3.4.3 Handling Stack Overflow

A process can produce a memory access violation (or bus error or segmentation fault) when it overflows its stack. As a first step in debugging this behavior, it is often necessary to run the program under the control of your system's debugger to determine which routine's stack has overflowed. However, if the debugger shares resources with the target process (as under OpenVMS), perhaps allocating its own data objects on the target process's stack, the debugger might not operate properly when the stack overflows. In this case, you might be required to analyze the target process by means other than the debugger.

To set the stacksize attribute in a thread attributes object, use the `pthread_attr_setstacksize( )` routine. (See Section 2.3.2.4 for more information.)

If a thread receives a memory access exception during a routine call or when accessing a local variable, increase the size of the thread's stack. Of course, not all memory access violations indicate a stack overflow.

For programs that you cannot run under a debugger, determining a stack overflow is more difficult. This is especially true if the program continues to run after receiving a memory access exception. For example, if a stack overflow occurs while a mutex is locked, the mutex might not be released as the thread recovers or terminates. When the program attempts to lock that mutex again, it hangs.

3.5 Scheduling Issues

There are programming issues that are unique to the scheduling attributes of threads.

3.5.1 Real-Time Scheduling

Use care when writing code that uses real-time scheduling to control the priority of threads:

- Review Section 3.1. Scheduling of threads is *not* the same as synchronizing of threads.
- Giving threads higher priority does not necessarily make your code run faster. Real-time priority adds overhead that can slow a program down, especially when interfacing with other libraries. For example, a higher-priority thread that polls for keyboard input may block work being done by other threads.

- Watch for pitfalls like priority inversion. It is best to avoid relying on real-time scheduling, except where necessary to meet design goals. On the other hand, most systems that interact with external devices have some real-time aspect.

3.5.2 Priority Inversion

Priority inversion occurs when the interaction among a group of three or more threads causes that group's highest-priority thread to be blocked from executing. For example, a higher-priority thread waits for a resource locked by a low-priority thread, and the low-priority thread waits while a middle-priority thread executes. The higher-priority thread is made to wait while a thread of lower priority (the middle-priority thread) executes.

You can address the phenomenon of priority inversion as follows:

- To avoid priority inversion, associate a priority (at least as high as the highest-priority thread that will use it) with each resource and force any thread using that object to first increase its priority to that associated with the object.
- To minimize the chance that an occurrence of priority inversion will cause a complete blockage of higher-priority threads, use the (default) throughput scheduling policy. The throughput scheduling policy allows even low-priority threads to execute eventually and to release the resources they hold. The FIFO and RR scheduling policies do not provide for resumption of the low-priority thread if the middle-priority thread executes indefinitely.

3.5.3 Dependencies Among Scheduling Attributes and Contention Scope

On DIGITAL UNIX systems, to use high (real-time) thread scheduling priorities, a thread with system contention scope must run in a process with sufficient real-time scheduling privileges. On the other hand, a thread with process contention scope has access to all levels of priority without requiring special real-time scheduling privileges.

Due to this, for a process that is not privileged, when a thread with a high priority and with process contention scope attempts to create another thread with system contention scope, the creation will fail if the created thread's attributes object specifies to inherit the creating thread's scheduling policy and priority.

3.6 Using Synchronization Objects

The following sections discuss how to determine when to use a mutex versus a condition variable, and how to use mutexes to prevent two erroneous behaviors that are common in multithreaded programs: race conditions and deadlocks.

Also discussed is why you should signal a condition variable with the associated mutex locked.

3.6.1 Distinguishing Proper Usage of Mutexes and Condition Variables

Use a mutex for tasks with fine granularity. Examples of a "fine-grained" task are those that serialize access to shared memory or make simple modifications to shared memory. This typically corresponds to a **critical section** of a few program statements or less.

Mutex waits are not interruptible. Threads waiting to acquire a mutex cannot be alerted or canceled.

Do not use a condition variable to protect access to data. Rather, use it to wait for data to assume a desired state. Always use a condition variable with a mutex that protects the shared data. Condition variable waits are interruptible.

See Section 2.4.1 and Section 2.4.2 for more information about mutexes and condition variables.

3.6.2 Avoiding Race Conditions

A **race condition** occurs when two or more threads perform an operation, and the result of the operation depends on unpredictable timing factors; specifically, when each thread executes and waits and when each thread completes the operation.

For example, if two threads execute routines and each increments the same variable (such as $x = x + 1$), the variable could be incremented twice and one of the threads could use the wrong value. For example:

1. Thread A increments variable *x*.
2. Thread A is interrupted (or blocked, or scheduled off), and thread B is started.
3. Thread B starts and increments variable *x*.
4. Thread B is interrupted (or blocked, or scheduled off), and thread A is started.

5. Thread A checks the value of *x* and performs an action based on that value.

   The value of *x* differs from when thread A incremented it, and the program's behavior is incorrect.

Race conditions result from lack of (or ineffectual) synchronization. To avoid race conditions, ensure that any variable modified by more than one thread has only one mutex associated with it, and ensure that all accesses to the variable are made after acquiring that mutex.

See Section 3.6.4 for another example of a race condition.

3.6.3 Avoiding Deadlocks

A **deadlock** occurs when a thread holding a resource is waiting for a resource held by another thread, while that thread is also waiting for the first thread's resource. Any number of threads can be involved in a deadlock if there is at least one resource per thread. A thread can deadlock on itself. Other threads can also become blocked waiting for resources involved in the deadlock.

Following are two techniques you can use to avoid deadlocks:

- *Use sequence numbers with fast mutexes*. Associate a sequence number with each mutex and acquire mutexes in sequence. Never attempt to acquire a mutex with a sequence number lower than that of a mutex the thread already holds.

  If a thread needs to acquire a mutex with a lower sequence number, it must first release all mutexes with a higher sequence number (after ensuring that the protected data is in a consistent state).

- *Use a recursive mutex*. This technique is useful when a thread needs to acquire the same mutex more than once before releasing it. This technique can help prevent a thread from deadlocking on itself.

3.6.4 Signaling a Condition Variable

When you are signaling a condition variable and that signal might cause the condition variable to be deleted, signal or broadcast the condition variable with the mutex locked.

The following C code fragment is executed by a releasing thread (Thread A) to wake a blocked thread:

```
pthread_mutex_lock (m);

... /* Change shared variables to allow another thread to proceed */

predicate = TRUE;
pthread_mutex_unlock (m);
                                1
pthread_cond_signal (cv);       2
```

The following C code fragment is executed by a potentially blocking thread (thread B):

```
pthread_mutex_lock (m);
while (!predicate )
      pthread_cond_wait (cv, m);

pthread_mutex_unlock (m);
pthread_cond_destroy (cv);
```

1 If thread B is allowed to run while thread A is at this point, it finds the predicate true and continues without waiting on the condition variable. Thread B might then delete the condition variable with the `pthread_cond_destroy( )` routine before thread A resumes execution.

2 When thread A executes this statement, the condition variable does not exist and the program fails.

These code fragments also demonstrate a race condition; that is, the routine, as coded, depends on a sequence of events among multiple threads, but does not enforce the desired sequence. Signaling the condition variable while still holding the associated mutex eliminates the race condition. Doing so prevents thread B from deleting the condition variable until after thread A has signaled it.

This problem can occur when the releasing thread is a worker thread and the waiting thread is a boss thread, and the last worker thread tells the boss thread to delete the variables that are being shared by boss and worker.

Code the signaling of a condition variable with the mutex locked as follows:

```
pthread_mutex_lock (m);
...

/* Change shared variables to allow some other thread to proceed */
pthread_cond_signal (cv);
pthread_mutex_unlock (m);
```

3.7 One-Time Initialization

Your program might have one or more routines that must be executed before any thread executes code in your facility, but that must be executed only once, regardless of the sequence in which threads start executing. For example, your program can initialize mutexes, condition variables, or thread-specific data keys—each of which must be created only once—in a one-time initialization routine.

Use the `pthread_once()` routine to ensure that your program's initialization routine executes only once—that is, by the first thread that attempts to initialize your program's resources. Multiple threads can call the `pthread_once()` routine, and DECthreads ensures that the specified routine is called only once.

On the other hand, rather than use the `pthread_once()` routine, your program can statically initialize a mutex and a flag, then simply lock the mutex and test the flag. In many cases, this technique might be more straightforward to implement.

Finally, you can use implicit (and nonportable) initialization mechanisms, such as OpenVMS LIB$INITIALIZE, DIGITAL UNIX dynamic loader `__init_` code, or Win32 DLL initialization handlers for Windows NT and Windows 95.

3.8 Managing Dependencies Upon Other Libraries

Because multithreaded programming has become common only recently, many existing code libraries are incompatible with multithreaded routines. For example, many of the traditional C run-time library routines maintain state across multiple calls using static storage. This storage can become corrupted if routines are called from multiple threads at the same time. Even if the calls from multiple threads are serialized, code that depends upon a sequence of return values might not work.

For example, the UNIX `getpwent(2)` routine returns the entries in the password file in sequence. If multiple threads call `getpwent(2)` repeatedly, even if the calls are serialized, no thread can obtain all entries in the password file.

Library routines might be compatible with multithreaded programming to different extents. The important distinctions are thread reentrancy and thread safety.

3.8.1 Thread Reentrancy

A routine is **thread reentrant** if it performs correctly despite being called simultaneously or sequentially by different threads. For example, the standard C run-time library routine `strtok()` can be made thread reentrant most efficiently by adding an argument that specifies a context for the sequence of tokens. Thus, multiple threads can simultaneously parse different strings without interfering with each other.

The ideal thread-reentrant routine has no dependency on static data. Because static data must be synchronized using mutexes and condition variables, there is always a performance penalty due to the time required to lock and unlock the mutex and also in the loss of potential parallelism throughout the program. A routine that does not use any data that is shared between threads can proceed without locking.

If you are developing new interfaces, make sure that any persistent context information (like the last-token-returned pointer in `strtok()`) is passed explicitly so that multiple threads can process independent streams of information independently. Return information to the caller through routine values, output parameters (where the caller passes the address and length of a buffer), or by allocating dynamic memory and requiring the caller to free that memory when finished. Try to avoid using *errno* for returning error or diagnostic information; use routine return values instead.

3.8.2 Thread Safety

A routine is **thread safe** if it can be called simultaneously from multiple threads without risk of corruption. Generally this means that it does some simple level of locking (perhaps using the DECthreads global lock) to prevent simultaneously active calls in different threads. See Section 3.8.3.3 for information about the DECthreads global lock.

Thread-safe routines might be inefficient. For example, a UNIX `stdio` package that is thread safe might still block all threads in the process while waiting to read or write data to a file.

Routines such as `localtime(3)` or `strtok()`, which traditionally rely on static storage, can be made thread safe by using thread-specific data instead of static variables. This prevents corruption and avoids the overhead of synchronization. However, using thread-specific data is not without its own cost, and it is not always the best solution. Using an alternate, reentrant version of the routine, such as the POSIX `strtok_r()` interface, is preferable.

3.8.3 Lacking Thread Safety

When your program must call a routine that is not thread safe, your program must ensure serialization and exclusivity of the unsafe routine across all threads in the program.

If a routine is not specifically documented as thread reentrant or thread safe, you are most safe to assume that it is not safe to use as-is with your multithreaded program. Never assume that a routine is fully thread reentrant unless it is expressly documented as such; a routine can use static data in ways that are not obvious from its interface. A routine carefully written to be thread reentrant but that calls some other routine that is not thread safe without proper protection, is itself not thread safe.

3.8.3.1 Using Mutex Around Call to Unsafe Code

Holding a mutex while calling any unsafe code accomplishes this. All threads and libraries using the routine should use the same mutex. Note that even if two libraries carefully lock a mutex around every call to a given routine, if each library uses a different mutex, the routine is not protected against multiple simultaneous calls from different libraries.

Note that your program might be required to protect a *series* of calls, rather than just a single call, to routines that are not thread safe.

3.8.3.2 Using or Copying Static Data Before Releasing the Mutex

In many cases your program must protect more than just the call itself to a routine that is not thread safe. Your program must use or copy any static return values before releasing the mutex that is being held.

3.8.3.3 Using the DECthreads Global Lock

To ensure serialization and exclusivity of the unsafe code, DECthreads provides one **global lock** that can be used by all threads in a program when calling routines or code that is not thread safe. The global lock allows a thread to acquire the lock recursively, so that you do not need to be concerned if you call a routine that also may acquire the global lock.

Acquire the global lock by calling `pthread_lock_global_np( )`; release the global lock by calling `pthread_unlock_global_np( )`.

Because there is only one global lock, you do not need to fully analyze all of the dependencies in unsafe code that your program calls. For example, with private locks to protect unsafe code, one lock might protect calls to the `stdio` routine, while another protects calls to math routines. However, if `stdio` next calls a math routine without acquiring the math routine lock, the call is just as unsafe as if no locks were used.

Use the global lock whenever calling unsafe routines. If you are unsure, assume that a routine is not thread safe unless it is expressly documented otherwise. All DECthreads routines are thread safe.

3.8.4 Use of Multiple Threads Libraries Not Supported

DECthreads performs user-mode execution context-switching within a process (OpenVMS VAX) or virtual processor (DIGITAL UNIX and OpenVMS Alpha) by exchanging register sets, including the program counter and stack pointer. If any other code within the process also performs this sort of context switch, neither DECthreads nor that other code can ever know which context is active at any time. This can result in, at best, unpredictable behavior—and, at worst, severe errors.

For example, under OpenVMS VAX, the VAX Ada run-time library provides its own tasking package that does not use DECthreads scheduling. Therefore, VAX Ada tasking cannot be used within a process that also uses DECthreads. (This restriction does not exist for DEC Ada for DIGITAL UNIX or for OpenVMS Alpha, because it uses DECthreads.)

This potential confusion might not exist for platforms that offer kernel thread-only packages. For example, DECthreads for Windows NT coexists smoothly with that platform's native Win32 threads.

3.9 Detecting DECthreads Error Conditions

DECthreads can detect some of the following types of errors:

- Application programming interface (API) errors can occur when the program specifies an invalid parameter or attempts an inappropriate operation on some DECthreads object.
- Internal errors can occur when DECthreads determines that internal information has become corrupted to the point where it cannot continue operation.

API errors are reported in different ways by the various DECthreads interfaces:

- The DECthreads **pthread** interface returns an integer value indicating the type of error.
- The DECthreads **cma** interface raises exceptions to indicate error conditions.

DECthreads internal errors result in a **bugcheck**. DECthreads writes a message that summarizes the problem to the process's current error device, and (on OpenVMS and Windows NT platforms) writes a file that contains more detailed information.

By default, the file is named `pthread_dump.log` and is created in the process's current (or default) directory. To cause DECthreads to write the bugcheck information into a different file, define `PTHREAD_CONFIG` and set its `dump=` major keyword. (See Section D.1 for more information about using `PTHREAD_CONFIG`.)

If DECthreads cannot create the specified file when it performs the bugcheck, it will try to create the default file. If it cannot create the default file, it will write the detailed information to the error device.

Note

On DIGITAL UNIX systems:

DECthreads no longer creates a dump file, because a core file is sufficient for analysis of the process using the Ladebug debugger.

3.9.1 Contents of a DECthreads Bugcheck Dump File

The header message written to the error device starts with a line reporting that DECthreads has detected an internal problem and that it is terminating execution. It also includes the version of the DECthreads library. The message resembles this:

```
%DECthreads bugcheck (version V3.13-180), terminating execution.
```

The next line states the reason for the failure. On DIGITAL UNIX, this is followed by process termination with SIGABRT (SIGIOT), which causes writing of a core dump file. On other platforms, a final line on the error device specifies the location of the file that contains detailed state information produced by DECthreads, as in the following example:

```
% Dumping to pthread_dump.log
```

The detailed information file contains information that is usually necessary to track down the problem. *If you encounter a DECthreads bugcheck, please contact your DIGITAL support representative and include this information file (or the DIGITAL UNIX core file) along with sample code and output.* Always include the full name and version of the operating system, and any patches that have been installed. If complete version information is lacking, useful core file analysis might not be possible.

3.9.2 Interpreting a DECthreads Bugcheck

The fact that DECthreads terminated the process with a bugcheck can mean that some subtle problem in DECthreads has been uncovered. However, DECthreads does not check for all possible API errors, and there are a number of ways in which incorrect code in your program can lead to a DECthreads bugcheck.

A common example is the use of any mutex operation or of certain condition variable operations from within an interrupt routine (that is, a DIGITAL UNIX signal handler or OpenVMS AST routine). This type of programming error most commonly results in a bugcheck that reports an “krnSpinLockPrm: deadlock detected” message or a “Can’t find null thread” message. To prevent this type of error, avoid using DECthreads routines other than `pthread_cond_signal_int_np( )` from an interrupt routine (or the equivalent routines in other APIs).

In addition, DECthreads maintains a variety of state information in memory which can be overwritten by your own code. Therefore, it is possible for an application to accidentally modify DECthreads state by writing through invalid pointers, which can result in a bugcheck or other undesirable behavior.

4

Writing Thread-Safe Libraries

A **thread-safe library** typically consists of routines that do not themselves create or use threads. However, the routines in a thread-safe library must be coded so that they are safe to be called from applications that use threads. DECthreads provides the thread-independent services (or **tis**) interface to support writing efficient, thread-safe code that does not itself use threads.

When called by a single-threaded program, the **tis** interface provides thread-independent synchronization services that are very efficient. For instance, **tis** routines avoid the use of interlocked instructions and memory barriers.

When called by a multithreaded program, the **tis** routines also provide full support for DECthreads synchronization, such as synchronization objects and thread joining.

The guidelines for using the DECthreads **pthread** interface routines also apply to using the corresponding **tis** interface routine in a multithreaded environment.

4.1 Features of the tis Interface

Among the key features of the DECthreads **tis** interface are:

- Distinct **tis** library, with some unique routines and some routines that correspond to those in the DECthreads **pthread** interface
- Common synchronization objects (such as mutexes and condition variables) with the **pthread** interface
- Unique **tis** synchronization objects (such as the read-write lock)
- Support for thread-specific data objects

Implementation of the DECthreads **tis** interface library varies by DIGITAL operation system. For more information, see this guide's operating system-specific appendixes.

It is not difficult to create thread-safe code using the DECthreads **tis** interface, and with the source code available should be staightforward to convert existing code that is not thread safe to become thread safe.

4.1.1 Reentrant Code Required

Your first consideration is whether the language compiler used in translating the source code produces reentrant code. Most Ada compilers generate inherently reentrant code because Ada supports multithreaded programming. On OpenVMS VAX systems, there are special restrictions on using the VAX Ada compiler to produce code or libraries to be interfaced with DECthreads. See Section 3.8.4.

Although the C, C++, Pascal, and BLISS programming languages do not support multithreaded programming directly, compilers for those languages generally create reentrant code. However, the Fortran and COBOL languages are defined in such a way that compilers can make implicit use of static storage, and such compilers do not generate reentrant code; it is difficult to write reentrant code in a nonreentrant language. The DEC FORTRAN compiler does generate reentrant code.

4.1.2 Performance of tis Interface Routines

Routines in the DECthreads **tis** interface are designed to perform efficiently when called from a single-threaded environment. For example, locking a mutex is essentially just setting a bit, and unlocking the mutex requires clearing the bit.

4.1.3 Run-Time Linkage of tis Interface Routines

All operations of **tis** interface routines require a call into the **tis** library, even when invoked from a multithreaded environment. For a multithreaded program that uses **tis** routines, during program initialization DECthreads automatically revectors the program's run-time linkages to most **tis** routines. This allows subsequent calls to those routines to use the normal DECthreads multithreaded (and SMP-safe) operations.

After the revectoring of run-time linkages has occurred, for example, a call to `tis_mutex_lock( )` operates exactly as if `pthread_mutex_lock( )` had been called. Thus, the transition from **tis** stubs to full DECthreads operation is transparent to library code that uses the **tis** interface. For instance, if DECthreads is dynamically activated while a **tis** mutex is acquired, the mutex can be released normally.

The **tis** interface deliberately provides no way to determine whether DECthreads is active within the process. Thread-safe code should always act as if multiple threads can be active. To do otherwise inevitably results in incorrect program behavior, given that DECthreads can be dynamically activated into the process at any time.

4.1.4 Cancelation Points

The following routines in the DECthreads **tis** interface are cancelation points:

```
tis_cond_wait( )
tis_testcancel( )
```

However, because the **tis** interface has no mechanism for requesting thread cancelation, no cancelation requests are actually delivered in these routines unless threads are present at run-time.

4.2 Using Mutexes

The **tis** interface routines support mutexes, called **tis mutexes**. Like the kinds of mutexes available through the other DECthreads interfaces, tis mutexes provide synchronization between multiple threads that share resources. In fact, you can statically initialize tis mutexes using the POSIX 1003.1c standard `PTHREAD_MUTEX_INITIALIZER` macro (see the DECthreads `pthread.h` header file) or using one of the nonportable DECthreads variants. This means you can create DECthreads recursive or errorcheck mutexes if required, as well as normal mutexes.

You can assign names to your program's tis mutexes by statically initializing them with the `PTHREAD_MUTEX_INITWITHNAME_NP` macro.

Unlike static initialization, dynamic initialization of tis mutexes is limited due to the absence of support for mutex attributes objects among **tis** interface routines. Thus, for example, the `tis_mutex_init( )` routine can create only normal mutexes.

If the DECthreads multithreading run-time environment becomes initialized dynamically, any tis mutexes acquired by your program remain acquired. The ownership of recursive and errorcheck mutexes remains valid.

Operations on the DECthreads global lock are also supported by **tis** interface routines. The DECthreads global lock is a recursive mutex that is reserved by DECthreads but is for use by any thread. Your program can use the global lock without calling the other DECthreads interfaces by calling `tis_lock_global( )` and `tis_unlock_global( )`.

4.3 Using Condition Variables

Tis condition variables behave like condition variables created using the **pthread** interface. You can initialize them statically using the POSIX.1c standard PTHREAD_COND_INITIALIZER macro or using one of the various nonportable DECthreads variants. For example, you can assign names to your tis condition variables by statically initializing them with the PTHREAD_COND_INITWITHNAME_NP macro.

As for tis mutexes, dynamic initialization of tis condition variables is limited due to the absence of support for condition variable attributes objects among **tis** interface routines.

Of course, your program can have more than one thread only if the DECthreads multithreading run-time environment is present. That is, if your program were to wait, there would be no other thread to "awaken" your program. Signaling or broadcasting a tis mutex when called from a single-threaded environment does nothing. This is because your program is not allowed to wait on a tis condition variable when the DECthreads multithreading run-time environment has not been initialized.

For code in a thread-safe library that uses a condition variable, construct its wait predicate so that the code does not actually require a block on the condition variable when called in a single-threaded environment.

4.4 Using Thread-Specific Data

The **tis** interface routines support the use of thread-specific data variables. If the DECthreads multithreading run-time environment is initialized, tis thread-specific data keys are transferred into that environment, so your program can continue to use the same keys.

For a program that uses tis thread-specific data in a multithreaded environment, any thread-specific data values set using a routine in the **tis** interface will be transferred into your program's initial thread.

4.5 Using Read-Write Locks

A **read-write lock** is an object that serializes access to shared information that needs to be read frequently and written only occasionally. Routines that manipulate read-write locks can control access to any shared resource and can be called by either a routine in a thread or by a routine in a thread-safe, single-threaded program.

For example, in a cache of recently accessed information, many threads can simultaneously examine the cache without conflict. When a thread must update the cache, it must have exclusive access.

Only the **tis** interface offers routines that operate on read-write locks. This is not a problem because you can use **tis** routines in a program that uses another DECthreads interface to use its own threads.

Note

In this version of DECthreads, read-write locks are not portable— although the Single UNIX Specification, Version 2, provides a similar set of routines that will be made available in a future version of DECthreads.

Your program can acquire a read-write lock for shared read access or for exclusive write access. An attempt to acquire a read-write lock for read access will block when any thread or program has already acquired that lock for write access. An attempt to acquire a read-write lock for write access will block when another thread has already acquired that lock for either read or write access.

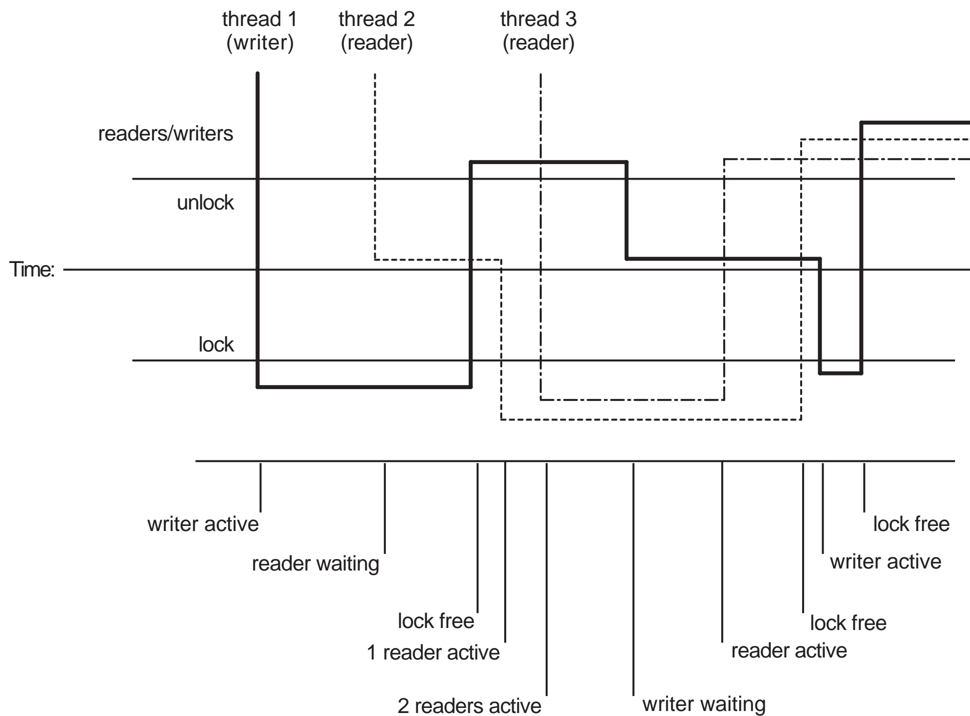
In a multithreaded environment, when both readers and writers are waiting at the same time for access via an already acquired read-write lock, DECthreads gives precedence to the readers when the lock is released. This policy of “read precedence” favors concurrency because it potentially allows many threads to accomplish work simultaneously. Figure 4–1 shows a read-write lock's behavior in response to three threads (one writer and two readers) that must access the same memory object.

The `tis_rwlock_init( )` routine initializes a read-write lock by allocating and initializing a `tis_rwlock_t` structure.

Your program uses the `tis_read_lock( )` or `tis_write_lock( )` routine to acquire a read-write lock when access to a shared resource is required. `tis_read_trylock( )` and `tis_write_trylock( )` can also be called to acquire a read-write lock. Note that if the lock is already acquired by another caller, `tis_read_trylock( )` immediately returns [EBUSY], rather than waiting.

Your program calls the `tis_rwlock_destroy( )` routine when it is finished using a read-write lock. This routine frees the lock's resources for re-use.

Figure 4–1 Read-Write Lock Behavior



Use the following routines to manipulate read-write locks:

Routine	Description
`tis_rwlock_init( )`	Initializes a read-write lock.
`tis_rwlock_destroy( )`	Destroys a read-write lock.
`tis_read_lock( )`	Acquires a read-write lock for read access.
`tis_write_lock( )`	Acquires a read-write lock for write access.
`tis_read_trylock( )`	Attempts to acquire a read-write lock for read access without waiting.
`tis_write_trylock( )`	Attempts to acquire a read-write lock for write access without waiting.
`tis_read_unlock( )`	Unlocks a read-write lock acquired for read access.
`tis_write_unlock( )`	Unlocks a read-write lock acquired for write access.

For more information about each **tis** interface routine that manipulates a read-write lock, see Part III.

5

Using the DECthreads Exception Package

This chapter describes how to use the DECthreads exception package and demonstrates conventions for the modular use of DECthreads exceptions in a multithreaded program.

This chapter:

- Describes the DECthreads exception package.
- Shows how to declare, initialize, and handle a DECthreads exception object in your program.
- Describes the DECthreads exception package's macros that support exception handling.
- Describes the DECthreads exception package's API-level routines that operate on exception objects.
- Lists the names of exception objects that the DECthreads exception package defines for its own use.

5.1 About the DECthreads Exceptions Package

The DECthreads exception package is a dynamic library and C language header file (`pthread_exception.h`) that together provide an interface for defining and handling exceptions. It is designed for use with the DECthreads **pthread** (POSIX.1c-1995) interface.

5.1.1 Supported Programming Languages

The DECthreads exception package is most useful when you are programming in the C language. The DECthreads exception package must be used with the DECthreads **pthread** interface.

You can use the C language exception library to catch DECthreads exceptions, but you cannot use the C++ language exception library to catch DECthreads exceptions. However, you cannot use the C language or C++ language exception library to define or raise DECthreads exceptions.

Because the DEC C++ runtime library and the POSIX thread runtime environment both use your platform's underlying exceptions mechanisms, the DEC C++ runtime library is able to run destructors an any exit from a block, including the exception handler code blocks defined using the DECthreads exception package.

5.1.2 Relation of Exceptions to Return Codes and Signals

Although the DECthreads **pthread** interface reports errors by returning nonzero return codes, DECthreads uses exceptions in the following cases:

- The `pthread_exit()` routine raises the DECthreads-defined exception `pthread_exit_e`.
- Canceling a thread cause DECthreads to raise the DECthreads-defined exception `pthread_cancel_e`.
- On DIGITAL UNIX, synchronous signals (such as SIGSEGV) are converted to exceptions unless a signal action is declared.

5.2 Why Use Exceptions

An **exception** is an object that describes an error condition. Operations on exception objects allow your program to report and handle errors. If an exception can be handled properly, the program can recover from errors. For example, if an exception is raised from a parity error while reading a tape, the recovery action might be to retry reading the tape 100 times before giving up.

You use exception programming to identify a portion of a routine, called an **exception scope**, where a calling thread must respond to particular error conditions or perhaps to any error condition. The thread can respond to each exception in either of two ways:

- *Catch* the exception. This means that the code handles all effects of the error condition and continues normal operation.
- *Finalize* the exception scope. This means that the current routine's context is cleaned up and resources (such as mutexes) are released. The exception is then passed to the next outer exception scope for further processing. The DECthreads exception package supports finalization even when no exception was raised, so that resources are always released without duplication of code.

Finally, you can use the DECthreads exception package to handle thread cancelation and thread exit in a unified and modular manner. Because DECthreads implements both thread cancelation and thread exit by raising DECthreads-reserved exceptions, your code can respond to these “events” in the same modular manner as for error conditions.

5.3 Exception Programming

Each DECthreads exception object is of the `EXCEPTION` type, which is defined in the `pthread_exception.h` header file.

Specifically, you must:

1. Declare one DECthreads exception object for each distinct error condition of interest to your program.
2. Code your program to invoke the DECthreads `RAISE` macro when it detects an error condition.
3. Code an exception scope, using the `TRY` and `ENDTRY` macros, to define the program scope within which an exception might be raised.
4. Associated with each exception scope, use the DECthreads `CATCH` macro to define a block of exception handler code for each exception object that can be raised and to which your program must respond at this point in its work. In this code your program perform activities to repsond to a particular error condition.
5. Associated with each exception scope, use the DECthreads `CATCH_ALL` macro to define an exception handler to catch any exception that your program code does not raise—that is, that can be raised by a facility that your program uses, including the host operating system itself. Because your program code does not raise these exceptions, your handler code must also “reraise” the caught exception so that the next outer exception scope also has the chance to respond to it.
6. Associated with each exception scope, instead of defining an exception handler, use the DECthreads `FINALLY` macro to define **finalization code**, also known as **epilogue code**, that is always executed, regardless of whether the code in the associated exception scope raised an exception. If this code is reached, DECthreads automatically reraises the caught exception and passes it to the next outer exception scope.

When a thread in your program raises an exception, DECthreads determines whether an exception scope has been defined in the current stack frame. If so, DECthreads checks whether there is a specific handler (CATCH code block) for the raised exception or whether there is an unspecified handler (CATCH_ALL code block). If not, DECthreads passes the raised exception to the next outer exception scope that does contain the pertinent code block. Thread execution resumes at that block. Attempting to catch a raised exception can cause a thread's stack to be unwound one or more call frames.

An exception can be caught only by the thread in which it is raised. An exception does not propagate from one thread to another.

5.3.1 Declaring and Initializing an Exception

Before referring to a DECthreads exception object in your code, your program must declare and initialize the object. A DECthreads exception object must be defined (whether explicitly or implicitly) to be of static storage class. In C language terms, the exception object can have local scope, module-wide scope, or global scope.

The next sample code fragment demonstrates declaring and initializing an exception object.

```
static EXCEPTION parity_error;  /* Declare the exception */

EXCEPTION_INIT (parity_error);  /* Initialize the exception */
```

5.3.2 Raising an Exception

Raise a DECthreads exception to indicate that your program has detected an error condition and in response to which the program must take some action. Your program raises the exception by invoking the DECthreads RAISE macro.

Example 5–1 demonstrates how to raise a DECthreads exception.

5.3.3 Catching an Exception

After your program raises a DECthreads exception, it is passed to a location within a block of code called an exception scope. The exception scope defines:

- A TRY code block, a lexical scope within which your program can raise an exception
- (Optionally) A CATCH code block, where your program handles a particular exception that was raised within the scope of this TRY block

Example 5–1 Raising an Exception

```
int  read_tape(void)
{
   int ret;

   if (tape_is_ready) {
      static EXCEPTION parity_error;     /* Declare it */
      EXCEPTION_INIT (parity_error);     /* Initialize it */
      ret = read(tape_device);
      if (ret = BAD_PARITY)
         RAISE (parity_error);           /* Raise it */
   }
}
```

Example 5–2 Catching an Exception Using CATCH

```
TRY {
   read_tape ();
}
CATCH (parity_error) {
   printf ("Oops, parity error, program terminating\n");
   printf ("Try cleaning the heads!\n");
}
ENDTRY
```

- (Optionally) A CATCH_ALL code block, where your program handles any exception raised within the scope of this TRY block that also is not named as an argument in a CATCH block in this TRY block
- (Optionally) A FINALLY code block, where your program performs finalization, or epilogue, actions at the end of the TRY block, whether an exception was raised or not

Example 5–2 shows a TRY code block with a CATCH code block defined to catch the exception object named parity_error when it is raised within the read_tape( ) routine.

Example 5–3 demonstrates how CATCH and CATCH_ALL code blocks work together to handle different raised exceptions within a given TRY code block.

Example 5–3 Catching an Exception Using CATCH and CATCH_ALL

```
int *local_mem;

local_mem = malloc (sizeof (int));
TRY {               /* An exception can be raised within this scope */
   read_tape ();
   free (local_mem);
}
CATCH (parity_error) {
   printf ("Oops, parity error, program terminating\n");
   printf ("Try cleaning the heads!\n");
   free (local_mem);
}
CATCH_ALL {
   free (local_mem);
   RERAISE;
}
ENDTRY
```

5.3.4 Reraising an Exception

Reraising an exception means to pass it to the next outer exception scope for further processing. Your program should take this step for a given exception when it must respond to the error condition but cannot completely recover from it.

As shown in Example 5–3, within a `CATCH` or `CATCH_ALL` code block, your program can invoke the `RERAISE` macro to pass a caught exception to the next outer exception scope in your program. If there is no next outer `TRY` block, the DECthreads default handler for unhandled exceptions receives the exception, produces a default error message that identifies the unhandled exception, then terminates the process.

Reraising is most appropriate for an exception caught in a `CATCH_ALL` block. Because this code block catches exceptions that are not “known” to your program’s code, it is unlikely that your code is able to fully recover from the error condition that the exception represents.

5.3.5 Expressing Epilogue Actions

Example 5–4 demonstrates the use of the optional FINALLY block.

Example 5–4 Defining Epilogue Actions Using FINALLY

```
int *local_mem;

local_mem = malloc (sizeof (int));
TRY {                        /* An exception can be raised within this scope */
   operation (local_mem);
}
FINALLY {
   free (local_mem);
}
ENDTRY
```

A FINALLY block catches an exception and implicitly reraises the exception for the next outer exception scope to handle. The actions defined by a FINALLY block are also performed on normal exit from the TRY block without an exception being raised. This means that those actions need not be duplicated in your code.

Do not combine a FINALLY block with a CATCH block or CATCH_ALL block in the same TRY block.

5.4 Exception Objects

This section describes the attributes of DECthreads exceptions (that is, the EXCEPTION type) and the behavior of the DECthreads exception package's exception handling macros (that is, RAISE and RERAISE, TRY, CATCH and CATCH_ALL, and FINALLY).

An **exception** is a data object that represents an error condition that has occurred in a particular context. The error condition can be detected by the operating system, by the native programming language, by another programmatic facility that your program calls, or by your own program.

The DECthreads exception package supports several operations on exceptions. Operations on exceptions allow a program to report and handle errors. If an exception is handled properly, the program can recover from certain errors. For example, if an exception is raised from a parity error while reading a tape, the recovery action might be to retry 100 times before giving up.

5.4.1 Declaring and Initializing Exception Objects

In the DECthreads exception package, an exception is a statically allocated variable of type EXCEPTION. Declaring and initializing an exception object documents that a program reports or handles a particular error.

The EXCEPTION type is designed to be an opaque type and should only be manipulated by the DECthreads exception package routines. The actual definition of the type may differ from one DECthreads release to another. The EXCEPTION type is defined in the pthread_exception.h header file.

In general, you should declare the type as static or extern. For example:

```
static EXCEPTION an_error;
```

Because on some platforms an exception object may require dynamic initialization, the DECthreads exception package requires a run-time initialization call in addition to the declaration. The initialization routine is a macro named EXCEPTION_INIT. The name of the exception is passed as a parameter.

Following code fragment shows declaring and initializing an exception object:

```
EXCEPTION parity_error;           /* Declare it */
EXCEPTION_INIT (parity_error);    /* Initialize it */
```

5.4.2 Address Exceptions and Status Exceptions

By default, when your program raises an exception object that has been properly initialized, only an address value is passed to the current exception scope. This form of DECthreads exception object is called an **address exception**, because the object's value is the address in your program where an error condition was detected. Your program code that handles address exceptions is fully portable among DECthreads-supported platforms.

Use address exceptions if the error conditions that can occur in your program are not assigned to a range of status codes. Address exceptions are always unique, so using them cannot cause a "collision" with another facility's status codes and possibly lead inadvertently to handling the wrong exception.

Alternatively, after initializing an exception object and before the exception can be raised, your program can assign a status value to it. The status value is typically an operating system-specific status code that represents a particular error condition. That is, your program uses the DECthreads exception package's pthread_exc_set_status_np( ) routine to assign a DIGITAL UNIX *errno* code (or OpenVMS condition code or Win32 status

code) to the exception object. This form of DECthreads exception object is called a **status exception**.

Using status exceptions can make sense if your program's target platform supports a universal definition of error status. That is, a status exception has the advantage of having some global meaning within your program and with respect to other libraries that your program uses. Your program can interpret, handle, and report the values used in status exceptions in a "centralized" manner, regardless of which facility in your program defines the status value.

When a facility called by DECthreads raises a system-level exception, DECthreads and its clients can catch the exception using a DECthreads status exception. Similarly, when a routine in your code raises a DECthreads exception, the calling routine might handle it using facilities provided by the language or platform.

Given two different exception objects that have been set with the same status value, the DECthreads exception package considers the two objects as functionally identical. For example, if one of the two exceptions is raised, it can be caught by specifying another exception object that has been set to the same status value. In contrast, DECthreads never considers two distinct address exception objects to be identical.

5.4.3 How Exceptions Terminate

DECthreads exceptions are *terminating* exceptions. This means that after a thread raises a particular exception, the thread never resumes execution in the code that immediately follows the statement that invokes the RAISE macro.

Instead, raising the exception causes the thread to resume execution at the appropriate block of handler code (that is, program statements in a CATCH or CATCH_ALL block) that is declared in the current exception scope. If the handler in the current exception scope contains a RERAISE statement, control reverts to the appropriate handler in the next outer exception scope.

Propagation of the exception—that is, transfer of control to an outer exception scope after executing the RERAISE statement—continues until entering a CATCH or CATCH_ALL block that does not end with a RERAISE statement; after that block's statements are executed, program execution continues at the first statement after the ENDTRY statement that terminates that exception scope.

When any thread raises an exception, if no exception scope in that exception's stack of scope handles the exception, DECthreads terminates the process, regardless of the state of the process's other threads. Termination prevents the unhandled error from affecting other areas of the process.

5.5 Exception Scopes

An **exception scope** serves two purposes:

- It defines a lexical scope within your program where it can respond to a specific raised exception or to any raised exception.
- It also associates this lexical scope with a set of exception handlers. Each of an exception scope's handlers is a code block enclosed within a DECthreads reserved macro, as described in Section 5.7.

Use the `TRY`/`ENDTRY` pair of macros to define an exception scope. (Throughout the discussion, we refer to this pair of macros simply as the `TRY` macro.) The `TRY` macro defines the beginning of an exception scope, and the `ENDTRY` macro defines the scope's end.

Example 5–5 illustrates defining an exception scope that encloses one operation, a call to the `read_tape( )` routine.

Example 5–5 Defining an Exception Scope

```
int  my_function(void)
{
   TRY {             /* Beginning of exception scope */
      read_tape ();  /* Operation(s) whose execution can raise an exception */
   }
   ENDTRY            /* End of exception scope */
}


int  read_tape(void)
{
   int ret;

   if (tape_is_ready) {
      static EXCEPTION parity_error;     /* Declare it */

      EXCEPTION_INIT (parity_error);     /* Initialize it */
      ret = read(tape_device);
      if (ret = BAD_PARITY)
         RAISE (parity_error);           /* Raise it */
   }
}
```

Defining an exception scope identifies a block of code in which an exception can be raised. That is, the block contains code that invokes, or directly or indirectly calls other code that invokes, the DECthreads exception package's `RAISE` macro when the program detects an error condition. Any exception raised within the block, or within any routines called directly or indirectly within the block, must pass through the control of this scope.

Because your program can detect different error conditions at different points in the code, your program can define more than one exception scope within its routines.

One exception scope cannot span the boundary of another exception scope. That is, it is invalid for one exception scope to contain only the beginning (the invocation of the `TRY` macro) or end (the invocation of the `ENDTRY` macro) of another exception scope.

5.6 Raising Exceptions

After your program declares and initializes an exception object, your program raises that exception when it detects an error condition. Use the DECthreads exception package's `RAISE` macro to raise an exception.

Raising an exception reports an error not by returning a value, but by *propagating* the exception. Propagating an exception takes place in a series of steps, as follows:

1. Searching in the current scope then to the next outer scope, in turn, and so on, for an exception handler, or code that explicitly or implicitly responds to the error.
2. Invoking the handler code that is found.
3. Invoking on optional block of finalization code (or epilogue code), regardless of whether exception handler was found for the raised exception.

If the exception scope within which an exception is raised does not define a handler or finalization block, then DECthreads simply "tears down" the current exception scope as the exception propagates up the DECthreads stack of exception scopes. This is also referred to as "unwinding" the stack.

Example 5–6 illustrates raising a DECthreads exception.

Example 5–6 Raising a DECthreads Exception

```
error = get_data();
if (error) {
   EXCEPTION parity_error;           /* Declare it */

   /* Initialize exception object and
       optionally set its status code */

   EXCEPTION_INIT (parity_error);
   pthread_exc_set_status_np (&parity_error, ENOMEM);
   RAISE (parity_error);             /* Raise it */
}
```

DECthreads exceptions are classified as terminating exceptions because after an exception is raised, the thread does not resume its execution at the point where the error condition was detected. Rather, execution resumes within the innermost exception scope that defines a handler block that explicitly or implicitly “catches” that exception, or that defines an epilogue block for finalization processing. See Section 5.4.3 for further details.

5.7 Exception Handling Macros

The DECthreads exception package allows your program to define an exception scope and to define and associate one or more blocks of code, each called an *exception handler*, with that scope. The purpose of an exception handler is to take appropriate actions, in context, to an error condition. “Appropriate actions” can mean merely cleaning up a routine’s local context and propagating the exception to the next outer exception scope, or can mean fully responding to the error in such a manner that allows the routine to continue its work.

5.7.1 Context of the Handler

An exception handler always runs within the context of the thread that generates the exception. Exceptions are synchronous “events,” like an access violation or segmentation fault, that are tied to a specified thread’s context.

Exception handlers are also closely tied to the execution context of the block that declares the handler. Thus, in the DECthreads exception package, exception handlers are *attached*, which means that the handler code appears within the same routine where the specified exceptions are raised. This allows the programmer to see what actions the program takes when an exception occurs with that exception scope.

5.7.2 Handlers and Macros

Unlike a signal handler routine, an exception handler can call any **pthread** routine.

Exception handler code is invoked when the exception specified in the handler is raised (or reraised) or when any unspecified exception is raised (or reraised) within the lexical scope of the associated exception scope.

Use the DECthreads exception package's `CATCH` macro to define an exception handler code block that is invoked when the macro's specified exception object is raised within the associated exception scope. Use the DECthreads exception package's `CATCH_ALL` macro to define an exception handler code block that is invoked when any non-specified exception object is raised (or reraised) within the associated exception scope.

An exception handler's code can *reraise* an exception. That is, the code can pass an exception object to the next outer exception scope for further processing. Use the DECthreads exception package's `RERAISE` macro to do so.

Related to exception handler code is finalization code, or epilogue code. You can define a block of epilogue code and associate it with an exception scope. After an exception has been raised (or reraised), epilogue code performs cleanup actions within the current exception scope (such as releasing resources) then passes on the raised exception to outer scopes for further processing. Additionally, finalization occurs even if no exception was raised, so that resources are always released without duplication of code.

Use the DECthreads exception package's `FINALLY` macro to define an epilogue code block. Note that, for a given exception scope, you code a `FINALLY` block instead of coding `CATCH` and `CATCH_ALL` blocks.

Each of these macros is discussed in greater detail in the following sections.

5.7.3 Catching Specific Exceptions

The exception scope can express interest in catching a specific exception by naming it as the argument in a statement that invokes the `CATCH` macro. When an exception reaches the exception scope, control is transferred to the first `CATCH` code block that specifies the exception by name. If there is more than one `CATCH` code block that specifies the same exception object by name within a single `TRY/ENDTRY` scope, only the first one gains control.

To catch an address exception, the `CATCH` macro must specify the name of the exception object as used in the invoked `RAISE` macro. In general, your program should raise and catch using the same exception object, even when using status exceptions. However, status exceptions can be caught using any exception

object that has been set to the same status code as the exception that was raised.

Example 5–7 shows an exception scope with one exception handler that uses the CATCH macro to catch a specific exception (parity_error) and to specify a recovery action (produce a message).

Example 5–7 Catching a Specific Exception Using CATCH

```
TRY {
   read_tape ();
}
CATCH (parity_error) {
   printf ("Oops, parity error, program terminating\n");
   printf ("Try cleaning the heads!\n");
   RERAISE;
}
ENDTRY
```

In this example, after catching the exception and executing the recovery action, the handler explicitly reraises the caught exception. This causes the exception object to propagate to the next outer exception scope.

Typically, you code one exception handler for each distinct error condition that can be raised anywhere in the program's call stack that is also within the associated exception scope.

If it is preferable for the caught exception to be propagated to the next higher exception scope, the CATCH code block can use the RERAISE macro to explicitly raise the same exception again.

5.7.4 Catching Unspecified Exceptions

The exception scope can express interest in catching all exceptions by coding an exception handler that uses the CATCH_ALL macro.

There must be only one CATCH_ALL code block within an exception scope. Note that it is invalid for a CATCH macro to follow a CATCH_ALL macro within an exception scope.

Example 5–8 demonstrates using the CATCH_ALL macro to define an exception handler for expressing actions in response to exceptions not explicitly raised your program's code.

Example 5–8 Catching an Unspecified Exception Using CATCH_ALL

```
int *local_mem;

local_mem = malloc (sizeof (int));
TRY {
   operation(local_mem);
   free (local_mem);
}
CATCH (an_error) {
   printf ("Oops; caught one!\n");
   free (local_mem);
}
CATCH_ALL {
   free (local_mem);
   RERAISE;
}
ENDTRY
```

It is best practice for your program to reraise any exception that is caught by a CATCH_ALL code block. (Not doing so is called *absorbing* the exception.) It is inappropriate to absorb a raised exception that your code is not explicitly aware of.

Because you cannot necessarily predict all possible exceptions that your code might encounter, you cannot assume that your code can recover in every possible situation. Therefore, your CATCH_ALL code block should explicitly reraise each caught exception as its final action; this allows an outer exception scope also to catch the same exception and to respond appropriately for its own context.

5.7.5 Reraising the Current Exception

Within an exception scope's CATCH or CATCH_ALL code blocks, you can invoke the RERAISE macro to reraise a caught exception object. This allows the next outer exception scope to handle the exception as it finds appropriate. Invoking the RERAISE macro is valid only with a CATCH or CATCH_ALL code block.

Use the RERAISE macro in a CATCH or CATCH_ALL code block that must restore some permanent program state (for example, releasing resources such as memory or a mutex) but does not have enough context about the detected error condition to attempt to recover fully. Thus, a CATCH_ALL code block should always reraise the caught exception as its last action.

Example 5–9 Reraising an Exception Using RERAISE

```
int *local_mem;

local_mem = malloc (sizeof (int));
TRY {
   operation(local_mem);
   free (local_mem);
}
CATCH (an_error) {
   printf ("Oops; caught one!\n");
   free (local_mem);
}
CATCH_ALL {
   free (local_mem);
   RERAISE;
}
```

Example 5–9 demonstrates invoking the `RERAISE` macro as the last action in a `CATCH_ALL` code block.

5.7.6 Defining Epilogue Actions

Some of your program's `CATCH` or `CATCH_ALL` code blocks catch exceptions only to perform cleanup actions, such as releasing resources. In many cases, these actions are performed whether the `TRY` code block exits normally or after an exception has been caught. Under other exception models, this requires duplicating code in the `CATCH_ALL` code block and following the exception scope (in case an exception does not occur).

The DECthreads exception package's `FINALLY` macro defines a code block that catches an exception and then implicitly reraises that exception for the next outer exception scope to handle. The actions in a `FINALLY` code block are also performed when the scope exits normally (that is, without catching an exception), so that they need not be coded more than once.

Example 5–10 demonstrates the `FINALLY` macro.

In this example, if the thread was canceled while it was waiting, the call to `pthread_cond_wait( )` could raise the `pthread_cancel_e` exception. The operations in the `FINALLY` code block ensure that the shared data associated with the lock is correct for the next thread that acquires the mutex.

Example 5–10 Defining Epilogue Actions Using FINALLY

```
pthread_mutex_lock (&some_object.mutex);
some_object.num_waiters = some_object.num_waiters + 1;
TRY {
    while (! some_object.data_available)
        pthread_cond_wait (&some_object.condition, &some_object.mutex);
    /* The code to act on the data_available goes here */
}
FINALLY {
    some_object.num_waiters = some_object.num_waiters - 1;
    pthread_mutex_unlock (&some_object.mutex);
{
ENDTRY
```

Note

Do not define a FINALLY code block if your exception scope uses a CATCH or CATCH_ALL code block. Doing so results in unpredictable behavior.

5.8 Operations on Exceptions

In addition to raising, catching, and reraising exception objects, the DECthreads exception package supports the following API-level operations on exception objects:

- Determine the current exception.
- Import a system-defined error status.
- Export a system-defined error status.
- Report an exception.
- Determine whether two exceptions match.

The following sections discuss these operations.

5.8.1 Referencing the Caught Exception

Within a CATCH or CATCH_ALL code block the caught exception object can be referenced by using the THIS_CATCH symbol.

The THIS_CATCH definition has a type of EXCEPTION *. This value can be passed to the pthread_exc_get_status_np( ), pthread_exc_report_np( ), or pthread_exc_matches_np( ) routines, as described in Section 5.8.3, Section 5.8.4, and Section 5.8.5.

Note

Because of the way that the DECthreads exception package propagates exception objects, the address contained in `THIS_CATCH` might not be the actual address of a DECthreads address exception. To match `THIS_CATCH` against known exceptions, use the `pthread_exc_matches_np( )` routine, as described in Section 5.8.5.

5.8.2 Setting a System-Defined Error Status

Use the `pthread_exc_set_status_np( )` routine to set a status value in an existing DECthreads address exception object. This transforms the address exception object into a DECthreads status exception object.

This routine's exception object argument must already have been initialized with the DECthreads exception package's `EXCEPTION_INIT` macro, as described in Section 5.3.1.

In a program that uses DECthreads status exceptions, use this routine to associate any system-specific status value with the specified address exception object. Note that any exception objects set to the same status value are considered equivalent by DECthreads.

Example 5–11 demonstrates setting an error status in a DECthreads address exception object.

Example 5–11 Setting an Error Status in an Exception Object

```
void pthread_exc_set_status_np (EXCEPTION *exception,
                                unsigned long code);
static EXCEPTION an_error;

EXCEPTION_INIT (an_error);
unsigned long status_code = ENOMEM;

/* Import status code into an existing, initialized,
    address exception object */

pthread_exc_set_status_np (&an_error, status_code);
```

Note

On OpenVMS systems:

DECthreads exception status values always have a SEVERE severity level. If necessary, calling the `pthread_exc_set_status_np( )` routine modifies the severity level of the status code to SEVERE.

5.8.3 Obtaining a System-Defined Error Status

In a program that uses DECthreads status exceptions, use the `pthread_exc_get_status_np( )` routine to obtain the status value from a DECthreads status exception, such as after an exception is caught. If the routine's *exception* argument is a status exception, it sets the status *code* argument and returns 0 (zero); otherwise, it returns [EINVAL] and does not set the status value argument.

Example 5–12 demonstrates using the `pthread_exc_get_status_np( )` routine to obtain the status value associated with a caught DECthreads status exception:

Example 5–12 Obtaining the Error Status Value from a Status Exception Object

```
int pthread_exc_get_status_np (EXCEPTION *exception, unsigned long code);
.
.
.
TRY {
   operation ();
    }
CATCH_ALL {
   unsigned long  status_code;

   if (pthread_exc_get_status_np (THIS_CATCH, &status_code) == 0
        && status_code == SOME_ERROR)
     fprintf (stderr, "%Exception %s caught from system.\n", SOME_ERROR);
   else
     pthread_exc_report_np (THIS_CATCH);
   }
ENDTRY
```

5.8.4 Reporting a Caught Exception

Use the `pthread_exc_report_np( )` routine to produce a message that reports what a given exception object represents. Your program calls this routine within a `CATCH`, `CATCH_ALL`, or `FINALLY` code block to report on a caught exception.

An exception that has been raised but not caught by a `CATCH` or `CATCH_ALL` anywhere in your program, causes the DECthreads unhandled exception handler to report the exception and immediately terminate the process. However, you might prefer to report a caught exception as part of your program's error recovery.

The `pthread_exc_report_np( )` routine prints a message to `stderr` (on DIGITAL UNIX and Windows NT systems) or SYS$ERROR (on OpenVMS systems) that describes the exception.

Each DECthreads-defined exception has an associated message that describes a given error condition. Typically, when the DECthreads exception package is well-integrated with the host platform's status mechanism, external status values can also be reported. On the other hand, when an address exception is reported, DECthreads can only report the fact that an exception has occurred and the address of the exception object.

Example 5–13 demonstrates using the `pthread_exc_report_np( )` routine to report an error.

Example 5–13 Reporting a Caught Exception

```
void pthread_exc_report_np (EXCEPTION *exception);
.
.
.
pthread_exc_report_np (&illinstr);
```

5.8.5 Determining Whether Two Exceptions Match

The `pthread_exc_matches_np( )` routine compares two exception objects, taking into consideration whether each is an address exception or a status exception. Whenever you must compare two exceptions, use this routine.

Example 5–14 Comparing Two Exception Objects

```
int pthread_exc_matches_np (EXCEPTION *exception1,
                            EXCEPTION *exception2);
.
.
.
EXCEPTION my_status;

EXCEPTION_INIT (&my_status);
pthread_exc_set_status_np (&my_status, status_code);
.
.
.
TRY {
   .
   .
   .
}
.
.
.
CATCH_ALL {
   if (pthread_exc_matches_np (THIS_CATCH, &my_status))
      fprintf (stderr, "This is my exception\n");
   RERAISE;
}
ENDTRY
```

Example 5–14 demonstrates how to use the `pthread_exc_matches_np( )` routine to test for the equivalence of two DECthreads exception objects.

5.9 Conventions for Modular Use of Exceptions

This section presents guidelines for using DECthreads exceptions in a modular way, so that independent software components can be written without requiring knowledge of each other.

5.9.1 Develop Naming Conventions for Exceptions

Develop naming conventions for exception objects. A naming convention ensures that the names for exceptions that are declared `extern` in different modules do not clash. The following convention is recommended:

facility-prefix_error-name_e

Example: `pthread_cancel_e`

5.9.2 Enclose Appropriate Actions in an Exception Scope

In a TRY code block avoid including code that more appropriately belongs outside it (in particular, before it). That is, the TRY macro should guard only operations for which there are appropriate handler operations in the scope's FINALLY, CATCH, or CATCH_ALL code blocks.

A common misuse of a TRY code block is to include code that should be executed before the TRY macro is invoked. Example 5–15 demonstrates this misuse.

Example 5–15 Incorrect Placement of Statements That Might Raise an Exception

```
TRY {
    handle = open_file (file_name);

    /* Statements that might raise an exception here */

}
FINALLY {
    close (handle);
}
ENDTRY
```

In this example the FINALLY code block assumes that no exception is raised by calling the open_file( ) routine. If calling open_file( ) results in raising an exception, the FINALLY code block's close( ) operation will use an invalid identifier.

Thus, the code in Example 5–15 should be rewritten as shown in Example 5–16.

Example 5–16 Correct Placement of Statements That Might Raise an Exception

```
handle = open_file (file_name);
TRY {

    /* Statements that might raise an exception here */

}
FINALLY {
    close (handle);
}
ENDTRY
```

The code that is an opening bracket belongs prior to invoking the TRY macro, and the code that is its matching closing bracket belongs in the FINALLY code block.

5.9.3 Raise Exceptions Prior to Performing Side-Effects

Raise exceptions prior to performing side-effects. That is, write routines that propagate exceptions to their callers, so that the routine does not modify any persistent process state before raising the exception. A matching close( ) call is required only if the open_file( ) operation is successful. (If an exception is raised, the caller cannot access the output parameters of the function, because the compiler may not have copied temporary values back to their home locations from registers.)

If the open_file( ) routine raises an exception, the identifier will not have been written, so this open operation must not require that a corresponding close( ) routine is called when open_file( ) raises an exception. This property is also what allows the call to be moved to open_file( ) prior to invoking the TRY macro.

5.9.4 Distinguish Raising Exceptions From Side-Effects

Do not place a return or goto statement between TRY and ENDTRY. It is invalid to return from, branch from, or leave by other means a TRY, CATCH, CATCH_ALL, or FINALLY block. After a given TRY macro is executed, the DECthreads exception package requires that the corresponding ENDTRY macro is also executed.

5.9.5 Declare Variables Within Handler Code as Volatile

When declaring certain variables that are used within an exception scope, you must use the ANSI C volatile type attribute. The volatile attribute prohibits the compiler from producing certain optimizations with respect to such variables. This ensures that such a variable's value is meaningful after a DECthreads exception object is raised.

Specifically, use the volatile type attribute for a variable whose value is written after the TRY macro is invoked and before the first CATCH/CATCH_ALL /FINALLY macro is invoked *and* whose value must be used after an exception is caught within a CATCH/CATCH_ALL/FINALLY block or (if the exception is caught and not reraised) after the ENDTRY macro is invoked.

Example 5–17 demonstrates the significance of using the volatile type qualifier for variables that are referenced within an exception scope:

Example 5–17 Correct Placement of Statements That Might Raise an Exception

```
void demonstrate_volatile_in_exception_scope (void )
{
   int          updated_before_try;
   int          updated;
   static int   updated_static;
   volatile int updated_volatile;

   updated_before_try = 1;
   updated = 2;
   updated_static = 3;
   updated_volatile = 4;

   TRY {
      updated = 6;
      updated_static = 7;
      updated_volatile = 8;

      something_that_might_result_in_an_exception();
   }
   CATCH (fully_handled_exception) {

      /*  Fully handle the exception here.
          Execute the code after ENDTRY next.  */

   }
   CATCH_ALL {
```

(continued on next page)

Example 5–17 (Cont.) Correct Placement of Statements That Might Raise an Exception

```
      /*  Values of updated_volatile and updated_before_try
           are meaningful.

          Values of updated and updated_static
           are unspecified!                    */

      if (updated > updated_static)
         printf ("%d, %d", updated, updated_before_try);
      if (updated > updated_volatile)
         printf ("%d, %d", updated, updated_before_try);
      RERAISE;
   }
   ENDTRY

   /*  Regardless of the path to this code, the
        values of updated_volatile and updated_before_try
        are meaningful.

       If this code is reached after the ENDTRY macro
        is invoked and no exception has been raised,
        the values of updated and updated_static
        are meaningful.

       If this code is reached after the exception
        fully_handled_exception has been caught,
        the values of updated and updated_static
        are unspecified!

       **The following two statements use invalid
          references to the variables updated and
          updated_static.**                    */

   if (updated > updated_static)
      printf ("%d, %d", updated, updated_before_try);
   if (updated > updated_volatile)
      printf ("%d, %d", updated, updated_before_try);

}      /* end demonstrate_volatile_in_exception_scope() */
```

The code in Example 5–17 demonstrates:

- For variables referenced within exception handler code blocks, it is significant to distinguish between those whose value is set before versus after the `TRY` macro is invoked.

- The requirement to use the volatile type qualifier pertains to a variable regardless of its C storage class—that is, for both automatic and static variables).

This exception scope includes both CATCH and CATCH_ALL code blocks. The variables updated, updated_static, and updated_volatile are set after the TRY macro is invoked. The value of the variable updated_before_try is set once, before the TRY macro is invoked.

The variables updated, updated_static, updated_volatile, and updated_before_try can also be referenced after an exception is caught: within the CATCH_ALL code block or after the ENDTRY macro is executed. Note that the code following the ENDTRY macro can be executed after the exception named fully_handled_exception is caught and its handler executes or if the exception scope is exited without an exception being raised.

Test your program after compiling it with the "optimize" compiler option, to ensure that your compiler produces the appropriate exception handler code.

5.9.6 Reraise Caught Exceptions That Are Not Fully Handled

Reraise exceptions that are not fully handled. That is, reraise any exception that you catch, unless your handler has performed the complete recovery action for the error. This rule permits an unhandled exception to propagate to some final default handler that knows how to recover fully.

A corollary of this rule is that CATCH_ALL handlers must reraise the exceptions they catch because they can catch any exception, including those not explicitly known to your code.

It is important to follow this convention, so that your program does not absorb a thread cancelation exception or thread-exit request exception. DECthreads maps these requests into exceptions, so that exception handler code can have the opportunity to handle all exceptional conditions—from access violations to thread-exit. In some applications it is important to be able to catch these to preserve an external invariant, such as an on-disk database.

5.9.7 Declare All Exception Objects as Static

Ensure that you declare (explicitly or implicitly) all exception objects as static, regardless of their scope in your program.

5.10 Exceptions Defined by DECthreads

Table 5–1 lists the names of exception objects that are defined by DECthreads and the meaning of each exception.

Exception object names that begin with the prefix pthread_ are raised within the DECthreads runtime environment itself and are not meant to be raised by your program code. Names of exception objects that begin with pthread_exc_ are generic and belong to the DECthreads exception package or the underlying system.

Table 5–1 Names of Exception Objects Defined by DECthreads

Exception	Definition
pthread_cancel_e	Thread cancelation in progress
pthread_exc_aritherr_e	Unhandled floating-point exception signal (“arithmetic error”)
pthread_exc_decovf_e	Unhandled decimal overflow trap exception
pthread_exc_excpu_e	“cpu-time limit exceeded”
pthread_exc_exfilsiz_e	“File size limit exceeded”
pthread_exc_exquota_e	Operation failed due to insufficient quota
pthread_exc_fltdiv_e	Unhandled floating-point/decimal divide by zero trap exception
pthread_exc_fltovf_e	Unhandled floating-point overflow trap exception
pthread_exc_fltund_e	Unhandled floating-point underflow trap exception
pthread_exc_illaddr_e	Data or object could not be referenced
pthread_exc_illinstr_e	Unhandled illegal instruction signal (“illegal instruction”)
pthread_exc_insfmem_e	Insufficient virtual memory for requested operation
pthread_exc_intdiv_e	Unhandled integer divide by zero trap exception
pthread_exc_intovf_e	Unhandled integer overflow trap exception
pthread_exc_nopriv_e	Insufficient privilege for requested operation
pthread_exc_privinst_e	Unhandled privileged instruction fault exception
pthread_exc_resaddr_e	Unhandled reserved addressing fault exception
pthread_exc_resoper_e	Unhandled reserved operand fault exception
pthread_exc_SIGABRT_e	Unhandled signal ABRT

(continued on next page)

Table 5–1 (Cont.) Names of Exception Objects Defined by DECthreads

Exception	Definition
pthread_exc_SIGBUS_e	Unhandled bus error signal
pthread_exc_SIGEMT_e	Unhandled EMT signal
pthread_exc_SIGFPE_e	Unhandled floating-point exception signal
pthread_exc_SIGILL_e	Unhandled illegal instruction signal
pthread_exc_SIGIOT_e	Unhandled IOT signal
pthread_exc_SIGPIPE_e	Unhandled broken pipe signal
pthread_exc_SIGSEGV_e	Unhandled segmentation violation signal
pthread_exc_SIGSYS_e	Unhandled bad system call signal
pthread_exc_SIGTRAP_e	Unhandled trace or breakpoint trap signal
pthread_exc_subrng_e	Unhandled subscript out of range exception
pthread_exc_uninitexc_e	Uninitialized exception raised
pthread_exit_e	Thread exited using pthread_exit_e
pthread_stackovf_e	Attempted stack overflow was detected

5.11 Interoperability of Language-Specific Exceptions

In general, the parts of your program that are coded in a given language (C, C++, Ada) can use only that language's own exception objects. This is also true for a program that uses DECthreads.

For example, your program cannot use the DECthreads CATCH to catch a C++ or Ada exception, and cannot use a C++ catch or an Ada except to catch an exception that the program defines using DECthreads.

However, in a program that uses DECthreads, C++ object destructors (as well as the DECthreads FINALLY facility and the equivalent functionality in Ada) will run when an exception from any facility, including DECthreads, reaches that frame. This includes the DECthreads exceptions pthread_cancel_e (cancelation of thread) and pthread_exit_e (thread exit).

Note

On DIGITAL UNIX systems:

Prior to DIGITAL UNIX Version 4.0, DECthreads exceptions could not fully interroperate with the C++ and Ada native language exception-handling facilities. For example, raising a DECthreads exception could not trigger invocation of C++ object destructors, and a C++ exception could not be caught by the DECthreads "last chance" exception handler for the calling thread.

As a result, a C++ `try/catch` block could not catch the DECthreads exceptions that indicate cancelation of a thread or thread exit. (However, once caught, the program's DECthreads exception handler can raise the exception again as a C++ exception, which in turn triggers the proper C++ actions to take place.)

5.12 Host Operating System Dependencies

This section mentions dependences of the DECthreads exception package on the operating system environment.

5.12.1 No DIGITAL UNIX Dependencies

DIGITAL UNIX has an architecturally specified exception model that is used by DECthreads as well as DIGITAL C++, Ada, and other languages that support exceptions. The DEC C compiler has extensions that allow "native" exception handling.

5.12.2 OpenVMS Conditions and DECthreads Exceptions

On OpenVMS, DECthreads propagates exceptions within the context of the OpenVMS Condition Handling Facility (CHF). An exception is typically raised by calling LIB$STOP with one of the condition codes listed in Table B–3.

Like the **pthread** `pthread_cleanup_push( )` routine, the DECthreads exception package's `CATCH` and `FINALLY` macros establish an OpenVMS condition handler that catches conditions of "fatal" or "severe" severity. Conditions with other severity values are passed through and thus cannot be caught using DECthreads exception handler code.

This requirement also pertains to DECthreads status exceptions. Thus, you cannot use the DECthreads exception package's `CATCH`, `CATCH_ALL`, and `FINALLY` macros to handle a status exception that is not of "severe" or "fatal" severity.

When an exception is raised, your program believes that a OpenVMS condition has been signalled. Until the exception is actually caught (that is, before passing through any TRY blocks or DECthreads cleanup handlers), the primary condition code is either CMA$_EXCEPTION (for an address exception) or a status value (for a status exception).

When a status exception is reraised, whether performed explicitly in a CATCH block or implicitly at the end of a FINALLY block or a DECthreads cleanup handler, DECthreads changes the primary condition code to either CMA$_EXCCOP or CMA$_EXCCOPLOS (depending on whether the contents of the exception can be reliably copied) and chains the original status code to the new primary as a secondary condition code. DECthreads propagates the exception by calling LIB$STOP with the new argument array.

When a status exception is reraised, DECthreads changes the primary condition code to indicate, first, that the exception has been reraised and, second, that the state of the program has been altered since the original exception was raised—that is, some number of frames have been unwound from the stack, which makes unavailable the values of any local variables.

This behavior also has these effects:

- The new primary condition code is not available to any CATCH blocks in call frames further into the stack, because those blocks trigger based on the status value in the original DECthreads status exception.
- The status code for the original status exception is available to any "native" OpenVMS condition handler in the argument array as a chained (secondary) OpenVMS condition. You must code such a handler to recognize the CMA$_EXCCOP and CMA$_EXCCOPLOS condition codes and to use the chained condition code when those are encountered as the primary.

For example, output of the following form:

```
%CMA-F-EXCCOP, exception raised; VMS condition code follows
-SYSTEM-F-ACCVIO, access violation, reason mask=00, virtual
address=0000000000000000, PC=000000000002013C, PS=0000001B
```

indicates that some thread incurred an access violation which was propagated as a DECthreads exception without being fully handled, which caused DECthreads to terminate the process. After noticing the location where the access violation occurred, or by running the failing program under the debugger with a breakpoint set on exceptions, you can determine where the exception (in this example, the ACCVIO condition) is originating.

6 DECthreads Examples

This chapter presents two example programs that use routines in the DECthreads **pthread** interface. Example 6–1 utilizes one parent thread and a set of worker threads to perform a prime number search. Example 6–2 implements a simple, text-based, asynchronous user interface that reads and writes commands to the terminal.

Both examples use the **pthread** interface routines and rely upon their default status-returning mechanism to indicate routine completion status. Example 6–1 uses the POSIX cleanup handler mechanism to cleanup from thread cancelation. In contrast, Example 6–2 uses the DECthreads exception package to capture and cleanup from thread cancelation and other synchronous fatal error conditions.

6.1 Prime Number Search Example

Example 6–1 shows the use of DECthreads **pthread** interface routines in a C program that performs a prime number search. The program finds a specified number of prime numbers, then sorts and displays these numbers. Several threads participate in the search: each thread takes a number (the next one to be checked), checks whether it is a prime, records it if it is prime, and then takes another number, and so on.

This program reflects the work crew functional model (see Section 1.4.2.) The worker threads increment the integer variable `current_num` to get their next work assignment. As a whole, the worker threads are responsible for finding a specified number of prime numbers, at which point their work is complete.

The number of worker threads to use and the number of prime numbers to find are defined as constants. A macro checks for an error status from each call to DECthreads and prints a given string and the associated error value. Data that is accessed by all threads (mutexes, condition variables, and so on) are declared as global items.

Each worker thread executes the `prime_search( )` routine, which immediately waits for permission to continue from the parent thread. The worker thread synchronizes with the parent thread using a predicate and a condition variable. Before and after waiting on the condition variable, each worker thread pushes and pops, respectively, a cleanup handler routine (`unlock_cond( )` to allow recovery from cancelation or other unexpected thread exit.

Notice that a predicate loop encloses the condition wait, to prevent the worker thread from continuing if it is wrongly signaled or broadcasted. The lock associated with the condition variable must be held by the thread during the call to condition wait. The lock is released within the call and acquired again upon being signaled or broadcasted. Note that the same mutex must be used for all operations performed on a specific condition variable.

After the parent sets the predicate and broadcasts, each worker thread begins finding prime numbers until canceled by a fellow worker who has found the last requested prime number. Upon each iteration a given worker increments the current number to examine and takes that new value as its next work item. Each worker thread uses a mutex to access the next work item, to ensure that no two threads are working on the same item. This type of locking protocol should be performed on all global data to ensure its integrity.

Next, each worker thread determines whether its current work item is prime by trying to divide numbers into it. If the number proves to be nondivisible, it is put on the list of primes. The worker thread disables its own cancelability while working with the list of primes, better to control any cancelation requests that might occur. The list of primes and its current count are protected by mutexes, which also protect the step of canceling all other worker threads upon finding the last requested prime. While the prime list mutex's remains locked, the worker checks whether it has found the last requested prime, and, if so, unsets a predicate and cancels all other worker threads. Finally, the worker enables its own cancelability.

The canceling thread should fall out of the work loop as a result of the predicate that it unsets.

The parent thread's flow of execution is as follows:

- Set up the environment, which means initialize the program's mutexes and one condition variable.
- Create worker threads. Creation of worker threads is straightforward and uses the default attributes.
- Broadcast to the worker threads that they can start.

- Join each thread as it finishes. As the parent joins each of the returning worker threads, it receives an exit value from each that indicates whether that worker thread exited normally. In this case, the exit values on all but one of the worker threads should be –1, indicating that the thread was canceled.
- Sort and print the list of primes.

The following DECthreads **pthread** interface routines are used in Example 6–1:

```
pthread_cancel( )
pthread_cleanup_pop( )
pthread_cleanup_push( )
pthread_cond_wait( )
pthread_create( )

pthread_join( )

pthread_mutex_lock( )
pthread_mutex_unlock( )

pthread_setcancelstate( )

pthread_testcancel( )
```

Example 6–1 C Program Example (Prime Number Search)

```
/*
 *
 * DECthreads example program conducting a prime number search
 *
 */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * Constants used by the example.
 */
#define     workers     5           /* Threads to perform prime check */
#define     request     110         /* Number of primes to find */

/*
 * Macros
 */

#define check(status,string) if (status != 0) {         \
    errno = status;                                      \
    fprintf (stderr, "%s status %d: %s\n", status, string, strerror (status)); \
}

/*
 * Global data
 */

pthread_mutex_t prime_list = PTHREAD_MUTEX_INITIALIZER; /* Mutex for use in
                                                           accessing the
                                                           prime */
pthread_mutex_t current_mutex = PTHREAD_MUTEX_INITIALIZER; /* Mutex associated
                                                              with current
                                                              number */
pthread_mutex_t cond_mutex = PTHREAD_MUTEX_INITIALIZER; /* Mutex used for
                                                           thread start */
pthread_cond_t  cond_var = PTHREAD_COND_INITIALIZER;    /* Condition variable
                                                           for thread start */
int             current_num= -1;/* Next number to be checked, start odd */
int             thread_hold=1;  /* Number associated with condition state */
int             count=0;        /* Count of prime numbers - index to primes */
int             primes[request];/* Store prime numbers - synchronize access */
pthread_t       threads[workers];   /* Array of worker threads */
```

(continued on next page)

Example 6–1 (Cont.) C Program Example (Prime Number Search)

```
static void
unlock_cond (void* arg)
{
    int     status;                 /* Hold status from pthread calls */

    status = pthread_mutex_unlock (&cond_mutex);
    check (status, "Mutex_unlock");
}

/*
 * Worker thread routine.
 *
 * Worker threads start with this routine, which begins with a condition wait
 * designed to synchronize the workers and the parent. Each worker thread then
 * takes a turn taking a number for which it will determine whether or not it
 * is prime.
 */
void *
prime_search (void* arg)
{
    int     numerator;              /* Used to determine primeness */
    int     denominator;            /* Used to determine primeness */
    int     cut_off;                /* Number being checked div 2 */
    int     notifiee;               /* Used during a cancelation */
    int     prime;                  /* Flag used to indicate primeness */
    int     my_number;              /* Worker thread identifier */
    int     status;                 /* Hold status from pthread calls */
    int     not_done=1;             /* Work loop predicate */
    int     oldstate;               /* Old cancel state */

    my_number = (int)arg;

    /*
     * Synchronize threads and the parent using a condition variable, the
     * predicate of which (thread_hold) will be set by the parent.
     */

    status = pthread_mutex_lock (&cond_mutex);
    check (status, "Mutex_lock");

    pthread_cleanup_push (unlock_cond, NULL);

    while (thread_hold) {
        status = pthread_cond_wait (&cond_var, &cond_mutex);
        check (status, "Cond_wait");
    }

    pthread_cleanup_pop (1);
```

(continued on next page)

Example 6–1 (Cont.) C Program Example (Prime Number Search)

```
/*
 * Perform checks on ever larger integers until the requested
 * number of primes is found.
 */

while (not_done) {

    /* Test for cancellation request */
    pthread_testcancel ();

    /* Get next integer to be checked */
    status = pthread_mutex_lock (&current_mutex);
    check (status, "Mutex_lock");
    current_num = current_num + 2;              /* Skip even numbers */
    numerator = current_num;
    status = pthread_mutex_unlock (&current_mutex);
    check (status, "Mutex_unlock");

    /* Only need to divide in half of number to verify not prime */
    cut_off = numerator/2 + 1;
    prime = 1;

    /* Check for prime; exit if something evenly divides */
    for (denominator = 2;
         ((denominator < cut_off) && (prime));
         denominator++) {
        prime = numerator % denominator;
    }

    if (prime != 0) {

        /* Explicitly turn off all cancels */
        pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &oldstate);

        /*
         * Lock a mutex and add this prime number to the list. Also,
         * if this fulfills the request, cancel all other threads.
         */

        status = pthread_mutex_lock (&prime_list);
        check (status, "Mutex_lock");
```

(continued on next page)

Example 6–1 (Cont.) C Program Example (Prime Number Search)

```
            if (count < request)  {
                primes[count] = numerator;
                count++;
            }
            else if (count == request) {
                not_done = 0;
                count++;
                for (notifiee = 0; notifiee < workers; notifiee++) {
                    if (notifiee != my_number) {
                        status = pthread_cancel (threads[notifiee]);
                        check (status, "Cancel");
                    }
                }
            }

            status = pthread_mutex_unlock (&prime_list);
            check (status, "Mutex_unlock");

            /* Reenable cancellation */
            pthread_setcancelstate (oldstate, &oldstate);
        }

        pthread_testcancel ();
    }

    return arg;
}


main()
{
    int     worker_num;     /* Counter used when indexing workers */
    void    *exit_value;    /* Individual worker's return status */
    int     list;           /* Used to print list of found primes */
    int     status;         /* Hold status from pthread calls */
    int     index1;         /* Used in sorting prime numbers */
    int     index2;         /* Used in sorting prime numbers */
    int     temp;           /* Used in a swap; part of sort */
    int     line_idx;       /* Column alignment for output */

    /*
     * Create the worker threads.
     */
```

(continued on next page)

Example 6–1 (Cont.) C Program Example (Prime Number Search)

```
    for (worker_num = 0; worker_num < workers; worker_num++) {
        status = pthread_create (
            &threads[worker_num],
            NULL,
            prime_search,
            (void*)worker_num);
        check (status, "Pthread_create");
    }

    /*
     * Set the predicate thread_hold to zero, and broadcast on the
     * condition variable that the worker threads may proceed.
     */
    status = pthread_mutex_lock (&cond_mutex);
    check (status, "Mutex_lock");
    thread_hold = 0;
    status = pthread_cond_broadcast (&cond_var);
    check (status, "Cond_broadcast");
    status = pthread_mutex_unlock (&cond_mutex);
    check (status, "Mutex_unlock");

    /*
     * Join each of the worker threads inorder to obtain their
     * summation totals, and to ensure each has completed
     * successfully.
     *
     * Mark thread storage free to be reclaimed upon termination by
     * detaching it.
     */

    for (worker_num = 0; worker_num < workers; worker_num++) {
        status = pthread_join (threads[worker_num], &exit_value);
        check (status, "Pthread_join");

        if (exit_value == (void*)worker_num)
            printf ("Thread %d terminated normally\n", worker_num);
        else if (exit_value == PTHREAD_CANCELED)
            printf ("Thread %d was cancelled\n", worker_num);
        else
            printf ("Thread %d terminated unexpectedly with %#lx\n",
                    worker_num, exit_value);

        /*
         * Upon normal termination the exit_value is equivalent to worker_num.
         */
    }
```

(continued on next page)

Example 6–1 (Cont.) C Program Example (Prime Number Search)

```
    /*
     * Take the list of prime numbers found by the worker threads and
     * sort them from lowest value to highest.  The worker threads work
     * concurrently; there is no guarantee that the prime numbers
     * will be found in order. Therefore, a sort is performed.
     */
    for (index1 = 1; index1 < request; index1++) {
        for (index2 = 0; index2 < index1; index2++) {
            if (primes[index1] < primes[index2]) {
                temp = primes[index2];
                primes[index2] = primes[index1];
                primes[index1] = temp;
            }
        }
    }

    /*
     * Print out the list of prime numbers that the worker threads
     * found.
     */
    printf ("The list of %d primes follows:\n", request);

    for (list = 0, line_idx = 0; list < request; list++, line_idx++) {

        if (line_idx >= 10) {
            printf (",\n");
            line_idx = 0;
        }
        else if (line_idx > 0)
            printf (",\t");

        printf ("%d", primes[list]);
    }
    printf ("\n");
}
```

6.2 Asynchronous User Interface Example

Example 6–2 implements a simple, text-based, asynchronous user interface. It allows you to use the terminal to start multiple commands that run concurrently and that report their results at the terminal when complete. You can monitor the status of, or cancel, commands that are already running.

This C program utilizes DECthreads **pthread** interface routines but also uses the DECthreads exception package to capture and cleanup from thread cancelations (and other synchronous fatal errors) as exceptions.

Asynchronous Commands

The asynchronous commands are `date` and `time`.

The asynchronous commands are as follows:

- `date` *delay_number_of_seconds*

  Waits the specified number of seconds before displaying today's date.

- `time` *delay_number_of_seconds*

  Waits the specified number of seconds before displaying the time of day.

For example, issuing the following command causes the program to wait 10 seconds before reporting the time:

```
Info> time 10
```

Housekeeping Commands

The housekeeping commands are as follows:

- `status` *command_number*

  Displays the state of a command.

- `wait` *command_number*

  Waits for a command to finish.

- `cancel` *command_number*

  Stops a command.

The argument *command_number* is the number of the command that assigned and displayed when the asynchronous command starts.

This program is limited to four outstanding commands.

Here is a sample of the output that the program produces:

```
Info> help
Commands are formed by a verb and an optional numeric argument.
The following commands are available:
        Cancel  <COMMAND>   Cancel running command
        Date    <DELAY>     Print the date
        Help                     Print this text
        Quit                     Quit (same as EOF)
        Status  [<COMMAND>] Report on running command
        Time    <DELAY>     Print the time
        Wait    <COMMAND>   Wait for command to finish

<COMMAND> refers to the command number.
<DELAY> delays the command execution for some number of seconds.
This delay simulates a command task that actually takes some
period of time to execute.  During this delay, commands may be
initiated, queried, and/or canceled.

Info> time 5
This is command #0.
Info> date 15
This is command #1.

(0) At the tone the time will be, 11:19:46.

Info> status 1
Command #1:  "date", 8 seconds remaining.

Info> status 1
Command #1:  "date", 5 seconds remaining.

Info> time 10
This is command #0.

Info> status 0
Command #0:  "time", 8 seconds remaining.

Info> status 1
Command #1:  "date", waiting to print.

(1) Today is Tue, 6 Oct 1992.

Info> time 3
This is command #0.

Info> wait 0
(0) At the tone the time will be, 11:21:26.

Info> date 10
This is command #0.
```

```
Info> cancel 0
(0) Canceled.
Info> quit
```

The following pthread routines are used in Example 6–2:

```
pthread_cancel( )
pthread_cond_signal( )
pthread_cond_wait( )
pthread_create( )

pthread_delay_np( )
pthread_detach( )

pthread_exc_report_np( )

pthread_join( )

pthread_mutex_init( )
pthread_mutex_lock( )
pthread_mutex_unlock( )

pthread_once( )

sched_yield( )
```

In the program source, notice that:

- The `main( )` routine uses `pthread_once( )` to perform one-time initialization.
- The `do_delay( )` routine specifies the preset delay interval. For a `timespec` structure, initializing `tv.sec = 1` and `tv.nsec = 0` results in a delay of one second.
- The `do_cleanup( )` and `find_free_thread( )` routines must lock two mutexes at the same time. To avoid deadlock, each routine in the program must lock the two mutexes in the same order.
- The `find_free_thread( )` routine uses `pthread_detach( )` to detach the free thread found because no other threads will join with it.

Example 6–2 C Program Example (Asynchronous User Interface)

```
/*
 *
 * DECthreads example program featuring an asynchronous user interface
 *
 */

/*
 * Include files
 */
#include <pthread.h>
#include <pthread_exception.h>
#include <stdio.h>
#include <time.h>

#define check(status,string) if (status != 0) {         \
    errno = status;                                     \
    fprintf (stderr, "%s status %d: %s\n", status, string, strerror (status)); \
}

/*
 * Local definitions
 */
#define PROMPT     "Info> "             /* Prompt string */
#define MAXLINSIZ   81                  /* Command line size */
#define THDNUM       5                  /* Number of server threads */

/*
 * Server thread "states"
 */
#define ST_INIT         0               /* "Initial" state (no thread) */
#define ST_FINISHED     1               /* Command completed */
#define ST_CANCELED    2                /* Command was canceled */
#define ST_ERROR        3               /* Command was terminated by an error */
#define ST_RUNNING      4               /* Command is running */

#ifndef FALSE                           /* Just in case these are not defined */
# define FALSE      0
# define TRUE       (!FALSE)
#endif

#ifndef NULL                            /* Just in case this is not defined */
# define NULL       ((void*)0)
#endif
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Global variables
 */
struct THREAD_DATA {
    pthread_t       thread;      /* Server thread handle */
    pthread_mutex_t mutex;       /* Mutex to protect fields below */
    int             time;              /* Amount of delay remaining */
    char            task;              /* Task being performed ('t' or 'd') */
    int             state;             /* State of the server thread */
} thread_data[THDNUM];

pthread_mutex_t     free_thread_mutex = /* Mutex to protect "free_thread" */
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t      free_thread_cv =     /* Condition variable for same */
    PTHREAD_COND_INITIALIZER;
int                 free_thread;       /* Flag indicating a free thread */


/*
 * Local Routines
 */
static void
dispatch_task (void *(*routine)(void*), char task, int time);

static void
do_cancel (int index);

static void
do_cleanup (int index, int final_state);

static void*
do_date (void* arg);

static void
do_delay (int index);

static void
do_status (int index);

static void*
do_time (void* arg);

static void
do_wait (int index);

static int
find_free_thread (int *index);

static char *
get_cmd (char *buffer, int size);
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
static int
get_y_or_n (char *query, char defans);

static void
init_routine (void);

static void
print_help (void);

/*
 * The main program:
 */
main()
{
    int     done = FALSE;               /* Flag indicating user is "done" */
    char    cmdline[MAXLINSIZ];         /* Command line */
    char    cmd_wd[MAXLINSIZ];          /* Command word */
    int     cmd_arg;                    /* Command argument */
    int     cmd_cnt;                    /* Number of items on command line */
    int     status;
    void    *(*routine)(void*);         /* Routine to execute in a thread */
    static pthread_once_t   once_block = PTHREAD_ONCE_INIT;


    /*
     * Perform program initialization.
     */
    status = pthread_once (&once_block, init_routine);
    check (status, "Pthread_once");

    /*
     * Main command loop
     */
    do {
        /*
         * Get and parse a command.  Yield first so that any threads waiting
         * to execute get a chance to before we take out the global lock
         * and block for I/O.
         */
        sched_yield ();
        if (get_cmd(cmdline, sizeof (cmdline))) {
            cmd_cnt = sscanf (cmdline, "%s %d", cmd_wd, &cmd_arg);
            routine = NULL;   /* No routine yet */
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
if ((cmd_cnt == 1) || (cmd_cnt == 2)) {      /* Normal result */
    cmd_wd[0] = tolower(cmd_wd[0]);         /* Map to lower case */
    switch (cmd_wd[0]) {
    case 'h':   /* "Help" */
    case '?':
        print_help();
        break;
    case 'q':   /* "Quit" */
        done = TRUE;
        break;
    case 's':   /* "Status" */
        do_status ((cmd_cnt == 2 ? cmd_arg : -1));
        break;

    /*
     * These commands require an argument
     */
    case 'c':   /* "Cancel" */
    case 'd':   /* "Date" */
    case 't':   /* "Time" */
    case 'w':   /* "Wait" */
        if (cmd_cnt != 2)
            printf ("Missing command argument.\n");
        else {
            switch (cmd_wd[0]) {
            case 'c':       /* "Cancel" */
                do_cancel (cmd_arg);
                break;
            case 'd':        /* "Date" */
                routine = do_date;
                break;
            case 't':        /* "Time" */
                routine = do_time;
                break;
            case 'w':          /* "Wait" */
                do_wait (cmd_arg);
                break;
            }
        }
        break;
    default:
        printf ("Unrecognized command.\n");
        break;
    }
}
else if (cmd_cnt != EOF)       /* Ignore blank command line */
    printf ("Unexpected parse error.\n");
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
            /*
             * If there is a routine to be executed in a server thread,
             * create the thread.
             */
            if (routine) dispatch_task (routine, cmd_wd[0], cmd_arg);
        }
        else
            done = TRUE;
    } while (!done);
}



/*
 * Create a thread to handle the user's request.
 */
static void
dispatch_task (void *(*routine)(void*), char task, int time)
{
    int i;              /* Index of free thread slot */
    int status;

    if (find_free_thread (&i)) {
        /*
         * Record the data for this thread where both the main thread and the
         * server thread can share it.  Lock the mutex to ensure exclusive
         * access to the storage.
         */
        status = pthread_mutex_lock (&thread_data[i].mutex);
        check (status, "Mutex_lock");
        thread_data[i].time = time;
        thread_data[i].task = task;
        thread_data[i].state = ST_RUNNING;
        status = pthread_mutex_unlock (&thread_data[i].mutex);
        check (status, "Mutex_unlock");
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
        /*
         * Create the thread, using the default attributes.  The thread will
         * execute the specified routine and get its data from array slot 'i'.
         */
        status = pthread_create (
            &thread_data[i].thread,
            NULL,
            routine,
            (void*)i);
        check (status, "Pthread_create");
        printf ("This is command #%d.\n\n", i);
    }
}

/*
 * Wait for the completion of the specified command.
 */
static void
do_cancel (int index)
{
    int cancelable;
    int status;

    if ((index < 0) || (index >= THDNUM))
        printf ("Bad command number %d.\n", index);
    else {
        status = pthread_mutex_lock (&thread_data[index].mutex);
        check (status, "Mutex_lock");
        cancelable = (thread_data[index].state == ST_RUNNING);
        status = pthread_mutex_unlock (&thread_data[index].mutex);
        check (status, "Mutex_unlock");

        if (cancelable) {
            status = pthread_cancel (thread_data[index].thread);
            check (status, "Pthread_cancel");
        }
        else
            printf ("Command %d is not active.\n", index);
    }
}
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Post-task clean-up routine.
 */
static void
do_cleanup (int index, int final_state)
{
    int status;

    /*
     * This thread is about to make the change from "running" to "finished",
     * so lock a mutex to prevent a race condition in which the main thread
     * sees this thread as finished before it is actually done cleaning up.
     *
     * Note that when attempting to lock more than one mutex at a time,
     * always lock the mutexes in the same order everywhere in the code.
     * The ordering here is the same as in "find_free_thread".
     */
    status = pthread_mutex_lock (&free_thread_mutex);
    check (status, "Mutex_lock");

    /*
     * Mark the thread as finished with its task.
     */
    status = pthread_mutex_lock (&thread_data[index].mutex);
    check (status, "Mutex_lock");
    thread_data[index].state = final_state;
    status = pthread_mutex_unlock (&thread_data[index].mutex);
    check (status, "Mutex_unlock");

    /*
     * Set the flag indicating that there is a free thread, and signal the
     * main thread, in case it is waiting.
     */
    free_thread = TRUE;
    status = pthread_cond_signal (&free_thread_cv);
    check (status, "Cond_signal");
    status = pthread_mutex_unlock (&free_thread_mutex);
    check (status, "Mutex_unlock");
}
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Thread routine that prints out the date.
 *
 * Synchronize access to ctime as it is not thread-safe (it returns the address
 * of a static string).  Also synchronize access to stdio routines.
 */
static void*
do_date (void* arg)
{
    time_t  clock_time;             /* Julian time */
    char    *date_str;              /* Pointer to string returned from ctime */
    char    day[4], month[4], date[3], year[5]; /* Pieces of ctime string */


    TRY {
        /*
         * Pretend that this task actually takes a long time to perform.
         */
        do_delay ((int)arg);
        clock_time = time ((time_t *)0);
        date_str = ctime (&clock_time);
        sscanf (date_str, "%s %s %s %*s %s", day, month, date, year);
        printf ("%d) Today is %s, %s %s %s.\n\n", arg, day, date, month, year);
    }
    CATCH (pthread_cancel_e) {
        printf ("%d) Canceled.\n", arg);

        /*
         * Perform exit actions
         */
        do_cleanup ((int)arg, ST_CANCELED);
        RERAISE;
    }
    CATCH_ALL {
        printf ("%d) ", arg);
        pthread_exc_report_np (THIS_CATCH);

        /*
         * Perform exit actions
         */
        do_cleanup ((int)arg, ST_ERROR);
        RERAISE;
    }
    ENDTRY;
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    /*
     * Perform exit actions (thread was not canceled).
     */
    do_cleanup ((int)arg, ST_FINISHED);

    /*
     * All thread routines return a value.  This program doesn't check the
     * value, however.
     */
    return arg;
}


/*
 * Delay routine
 *
 * Since the actual tasks that threads do in this program take so little time
 * to perform, execute a delay to make it seem like they are taking a long
 * time. Also, this will give the user something to query the progress of.
 */
static void
do_delay (int index)
{
    static struct timespec interval = {1, 0};
    int done;                                   /* Loop exit condition */
    int status;

    while (TRUE) {
        /*
         * Decrement the global count, so the main thread can see how much
         * progress we've made. Keep decrementing as long as the remaining
         * time is greater than zero.
         *
         * Lock the mutex to ensure no conflict with the main thread that
         * might be reading the time remaining while we're decrementing it.
         */
        status = pthread_mutex_lock (&thread_data[index].mutex);
        check (status, "Mutex_lock");
        done = ((thread_data[index].time--) <= 0);
        status = pthread_mutex_unlock (&thread_data[index].mutex);
        check (status, "Mutex_unlock");

        /*
         * Quit if the time is up.
         */
        if (done) break;
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
        /*
         * Wait for one second.
         */
        pthread_delay_np (&interval);
    }
}

/*
 * Print the status of the specified thread.
 */
static void
do_status (int index)
{
    int start, end;                 /* Range of commands queried */
    int i;                          /* Loop index */
    int output = FALSE;             /* Flag: produced output */
    int status;

    if ((index < -1) || (index >= THDNUM))
        printf ("Bad command number %d.\n", index);
    else {
        if (index == -1)
            start = 0, end = THDNUM;
        else
            start = index, end = start + 1;

        for (i = start; i < end; i++) {
            status = pthread_mutex_lock (&thread_data[i].mutex);
            check (status, "Mutex_lock");

            if (thread_data[i].state != ST_INIT) {
                printf ("Command #%d:  ", i);

                switch (thread_data[i].task) {
                case 't':
                    printf ("\"time\", ");
                    break;
                case 'd':
                    printf ("\"date\", ");
                    break;
                default:
                    printf ("[unknown] ");
                    break;
                }
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
                switch (thread_data[i].state) {
                case ST_FINISHED:
                    printf ("completed");
                    break;
                case ST_CANCELED:
                    printf ("canceled");
                    break;
                case ST_ERROR:
                    printf ("terminated by error");
                    break;
                case ST_RUNNING:
                    if (thread_data[i].time < 0)
                        printf ("waiting to print");
                    else
                        printf (
                            "%d seconds remaining",
                            thread_data[i].time);
                    break;
                default:
                    printf ("Bad thread state.\n");
                    break;
                }

                printf (".\n");
                output = TRUE;
            }
            status = pthread_mutex_unlock (&thread_data[i].mutex);
            check (status, "Mutex_unlock");
        }
        if (!output) printf ("No such command.\n");
        printf ("\n");
    }
}
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Thread routine that prints out the date.
 */
static void*
do_time (void* arg)
{
    time_t  clock_time;             /* Julian time */
    char    *date_str;              /* Pointer to string returned from ctime */
    char    time_str[8];            /* Piece of ctime string */


    TRY {
        /*
         * Pretend that this task actually takes a long time to perform.
         */
        do_delay ((int)arg);
        clock_time = time ((time_t *)0);
        date_str = ctime (&clock_time);
        sscanf (date_str, "%*s %*s %*s %s", time_str);
        printf ("%d) At the tone the time will be, %s.%c\n\n",
                arg,
                time_str,
                '\007');
    }
    CATCH (pthread_cancel_e) {
        printf ("%d) Canceled.\n", arg);
        do_cleanup ((int)arg, ST_CANCELED);
        RERAISE;
    }
    CATCH_ALL {
        printf ("%d) ", arg);
        pthread_exc_report_np (THIS_CATCH);
        do_cleanup ((int)arg, ST_ERROR);
        RERAISE;
    }
    ENDTRY;

    /*
     * Perform exit actions (thread was not canceled).
     */
    do_cleanup ((int)arg, ST_FINISHED);
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    /*
     * All thread routines return a value.  This program doesn't check the
     * value, however.
     */
    return arg;
}

/*
 * Wait for the completion of the specified command.
 */
static void
do_wait (int index)
{
    int status;
    void *value;

    if ((index < 0) || (index >= THDNUM))
        printf ("Bad command number %d.\n", index);
    else {
        status = pthread_join (thread_data[index].thread, &value);
        check (status, "Pthread_join");

        if (value == (void*)index)
            printf ("Command %d terminated successfully.\n", index);
        else if (value == PTHREAD_CANCELED)
            printf ("Command %d was cancelled.\n", index);
        else
            printf ("Command %d terminated with unexpected value %#lx",
                    index, value);

    }

}


/*
 * Find a free server thread to handle the user's request.
 *
 * If a free thread is found, its index is written at the supplied address
 * and the function returns true.
 */
static int
find_free_thread (int *index)
{
    int i;                              /* Loop index */
    int found;                          /* Free thread found */
    int retry = FALSE;                  /* Look again for finished threads */
    int status;
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
do {
    /*
     * We're about to look for a free thread, so prevent the data state
     * from changing while we are looking.
     *
     * Note that when attempting to lock more than one mutex at a time,
     * always lock the mutexes in the same order everywhere in the code.
     * The ordering here is the same as in "do_cleanup".
     */
    status = pthread_mutex_lock (&free_thread_mutex);
    check (status, "Mutex_lock");

    /*
     * Find a slot that doesn't have a running thread in it.
     *
     * Before checking, lock the mutex to prevent conflict with the thread
     * if it is running.
     */
    for (i = 0, found = FALSE; i < THDNUM; i++) {
        status = pthread_mutex_lock (&thread_data[i].mutex);
        check (status, "Mutex_lock");
        found = (thread_data[i].state != ST_RUNNING);
        status = pthread_mutex_unlock (&thread_data[i].mutex);
        check (status, "Mutex_unlock");

        /*
         * Now that the mutex is unlocked, break out of the loop if the
         * thread is free.
         */
        if (found) break;
    }

    if (found)
        retry = FALSE;
    else {
        retry = get_y_or_n (
            "All threads are currently busy, do you want to wait?",
            'Y');

        if (retry) {
            /*
             * All threads were busy when we started looking, so clear
             * the "free thread" flag.
             */
            free_thread = FALSE;
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
                /*
                 * Now wait until some thread finishes and sets the flag
                 */
                while (!free_thread)
                    pthread_cond_wait (&free_thread_cv, &free_thread_mutex);
            }
        }
        pthread_mutex_unlock (&free_thread_mutex);
    } while (retry);

    if (found) {
        /*
         * Request DECthreads reclaim its internal storage for this old thread
         * before we use the handle to create a new one.
         */
        status = pthread_detach (thread_data[i].thread);
        check (status, "Pthread_detach");
        *index = i;
    }

    return (found);
}



/*
 * Get the next user command.
 *
 * Synchronize I/O with other threads to prevent conflicts if the stdio
 * routines are not thread-safe.
 */
static char *
get_cmd (char *buffer, int size)
{
    printf (PROMPT);
    return fgets (buffer, size, stdin);
}
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Get a yes or no answer to a query.  A "blank" answer uses default answer.
 *
 * Returns TRUE for "yes" and FALSE for "no".
 */
static int
get_y_or_n (char *query, char defans)
{
    char    buffer[MAXLINSIZ];          /* User's answer */
    int     answer;                     /* Boolean equivalent */
    int     retry = TRUE;               /* Ask again? */

    do {
        buffer[0] = '\0';               /* Initialize the buffer */
        flockfile (stdout);
        flockfile (stdin);
        printf ("%s [%c] ", query, defans);
        fgets (buffer, sizeof (buffer), stdin);
        funlockfile (stdin);
        funlockfile (stdout);

        if (buffer[0] == '\0') buffer[0] = defans;       /* Apply default */

        switch (buffer[0]) {
        case 'y':
        case 'Y':
            answer = TRUE;
            retry = FALSE;
            break;
        case 'n':
        case 'N':
            answer = FALSE;
            retry = FALSE;
            break;
        default:
            printf ("Please enter \"Y\" or \"N\".\n");
            retry = TRUE;
            break;
        }
    } while (retry);

    return answer;
}
```

(continued on next page)

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Initialization routine;
 *
 * Called as a one-time initialization action.
 */
static void
init_routine (void)
{
    int i;

    for (i = 0; i < THDNUM; i++) {
        pthread_mutex_init (&thread_data[i].mutex, NULL);
        thread_data[i].time = 0;
        thread_data[i].task = '\0';
        thread_data[i].state = ST_INIT;
    }

}

/*
 * Print help text.
 */

static void
print_help (void)
{
    printf ("Commands are formed by a verb and optional numeric argument.\n");
    printf ("The following commands are available:\n");
    printf ("\tCancel\t[command]\tCancel running command\n");
    printf ("\tDate\t[delay]\t\tPrint the date\n");
    printf ("\tHelp\t\t\tPrint this text\n");
    printf ("\tQuit\t\t\tQuit (same as EOF)\n");
    printf ("\tStatus\t[command]\tReport on running command\n");
    printf ("\tTime\t[delay]\t\tPrint the time\n");
    printf ("\tWait\t[command]\tWait for command to finish\n");
    printf ("\n[command] refers to the command number.\n");;
    printf ("[delay] delays command execution for some number of seconds.\n");
    printf ("This delay simulates a command task that actually takes some\n");
    printf ("period of time to execute.  During this delay, commands may be\n");
    printf ("initiated, queried, and/or canceled.\n");
}
```

Part II

POSIX.1c (pthread) Routines Reference

Part II provides detailed descriptions of routines that constitute the DECthreads **pthread** interface. These routines (with the prefix `pthread_`) implement the IEEE POSIX 1003.1c-1995 (or POSIX.1c) standard, subject to the capabilities of the host operating system.

Note

The **pthread** routines described here are based on the final POSIX.1c standard approved by the IEEE.

DECthreads users should be aware that applications that use the obsolete DECthreads **d4** interfaces will require significant modifications to upgrade to the **pthread** interface. (The obsolete DECthreads **d4** interface corresponds to the IEEE POSIX 1003.4a/Draft 4 document.)

The global *errno* variable is not used by the DECthreads **pthread** interface routines. To indicate errors, the **pthread** routines return integer values to indicate the error condition.

Routine names with the `_np` suffix denote that the routine is *not portable*, with respect to the POSIX.1c standard. That is, the routine might not be available in implementations of the POSIX.1c standard other than DECthreads.

DECthreads is beginning to provide capabilities specified by the Open Group's (formerly X/Open) Single UNIX Specification, Version 2—also known as UNIX98. Some of the **pthread** interface routines that UNIX98 specifies are not present the IEEE POSIX 1003.1c-1995 standard; these routines include `pthread_attr_getguardsize( )`, `pthread_attr_setguardsize( )`, `pthread_mutexattr_gettype( )`, and `pthread_mutexattr_settype( )`. DECthreads does *not* designate these routines as nonportable—that is, their names do not use the `_np` suffix naming convention. While portable to other implementations of the Single UNIX Specification, Version 2, these routines are not porable to other implementations of the POSIX.1c standard.

pthread_atfork

Declares fork handler routines to be called when the calling thread's process forks a child process.

This routine is for DIGITAL UNIX systems only.

Syntax

```
pthread_atfork(
        prepare,
        parent,
        child );
```

Argument	Data Type	Access
prepare	Handler	read
parent	Handler	read
child	Handler	read

C Binding

```
#include <pthread.h>

int
pthread_atfork (
        void    (*prepare)(void),
        void    (*parent)(void),
        void    (*child)(void) );
```

Arguments

prepare
Address of a routine that performs the fork preparation handling. This routine is called in the parent process before creating the child process.

parent
Address of a routine that performs the fork parent handling. This routine is called in the parent process after creating the child process and before returning to the caller of `fork(2)`.

child
Address of a routine that performs the fork child handling. This routine is called in the child process before returning to the caller of `fork(2)`.

Description

This routine allows a main program or library to control resources during a DIGITAL UNIX `fork(2)` operation by declaring fork handler routines, as follows:

- The fork handler routine specified in the *prepare* argument is called before `fork(2)` executes.
- The fork handler routine specified in the *parent* argument is called after `fork(2)` executes within the parent process.
- The fork handler routine specified in the *child* argument is called in the new child process after `fork(2)` executes.

Your program (or library) can use fork handlers to ensure that program context in the child process is consistent and meaningful. After `fork(2)` executes, only the calling thread exists in the child process, and the state of all memory in the parent process is replicated in the child process, including the states of any mutexes, condition variables, and so on.

For example, in the new child process there might exist locked mutexes that are copies of mutexes that were locked in the parent process by threads that do not exist in the child process. Therefore, any associated program state might be inconsistent in the child process.

The program can avoid this problem by calling `pthread_atfork( )` to provide routines that acquire and release resources that are critical to the child process. For example, the prepare handler should lock all mutexes that you want to be usable in the child process. The parent handler just unlocks those mutexes. The child handler will also unlock them all—and might also create threads or reset any program state for the child process.

To illustrate, if your library uses the mutex *my_mutex*, you might provide `pthread_atfork( )` handler routines coded as follows:

```
void  my_prepare(void)
      {
      pthread_mutex_lock(&my_mutex);
      }
```

```
void  my_parent(void)
      {
      pthread_mutex_unlock(&my_mutex);
      }

void  my_child(void)
      {
      pthread_mutex_unlock(&my_mutex);
      /* Reinitialize state that doesn't apply...like heap owned */
      /* by other threads         */
      }

   {
      .
      .
      .
    pthread_atfork(my_prepare, my_parent, my_child);
      .
      .
    fork();
   }
```

If no fork handling is desired, you can set any of this routine's arguments to NULL.

Note

It is not legal to call `pthread_atfork( )` from within a fork handler routine. Doing so could cause a deadlock.

Return Values

If an error occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ENOMEM]	Insufficient table space exists to record the fork handler routines' addresses.

pthread_atfork

Associated Routines

`pthread_create( )`

pthread_attr_destroy

Destroys a thread attributes object.

Syntax

```
pthread_attr_destroy(
                    attr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	modify

C Binding

```
#include <pthread.h>

int
pthread_attr_destroy (
                  pthread_attr_t     *attr);
```

Arguments

attr
Thread attributes object to be destroyed.

Description

This routine destroys a thread attributes object. Call this routine when a thread attributes object will no longer be referenced.

Threads that were created using this thread attributes object are not affected by the destruction of the thread attributes object.

The results of calling this routine are unpredictable if the value specified by the *attr* argument refers to a thread attributes object that does not exist.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_attr_init( )
pthread_create( )
```

pthread_attr_getdetachstate

Obtains the detachstate attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getdetachstate(
                    attr,
                    detachstate );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
detachstate	integer	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getdetachstate (
            const pthread_attr_t    *attr,
            int    *detachstate);
```

Arguments

attr
Thread attributes object whose detachstate attribute is obtained.

detachstate
Receives the value of the detachstate attribute.

Description

This routine obtains the detachstate attribute of a thread attributes object. This attribute specifies whether threads created using the specified thread attributes object are created in a detached state.

On successful completion, this routine returns a zero and the detachstate attribute is set in *detachstate*. A value of `PTHREAD_CREATE_JOINABLE` indicates the thread is not detached, and a value of `PTHREAD_CREATE_DETACHED` indicates the thread is detached.

pthread_attr_getdetachstate

See the `pthread_attr_setdetachstate( )` description for information about the detachstate attribute.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* does not refer to an existing thread attributes object.

Associated Routines

```
pthread_attr_init( )
pthread_attr_setdetachstate( )
```

pthread_attr_getguardsize

Obtains the guardsize attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getguardsize(
                          attr,
                          guardsize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
guardsize	size_t	write

C Binding

#include <pthread.h>

```
int
pthread_attr_getguardsize (
            const pthread_attr_t    *attr,
            size_t    *guardsize);
```

Arguments

attr
Address of the thread attributes object whose guardsize attribute is obtained.

guardsize
Receives the value of the guardsize attribute of the thread attributes object specified by *attr*.

Description

This routine obtains the value of the guardsize attribute of the thread attributes object specified in the *attr* argument and stores it in the location specified by the *guardsize* argument. The specified attributes object must already be initialized at the time this routine is called.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The guardsize attribute of a thread attributes object specifies the minimum size (in bytes) of the guard area for the stack of a new thread.

A guard area can help a multithreaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that DECthreads allocates at the overflow end of the thread's stack. When any thread attempts to access a memory location within this region, a memory addressing violation occurs.

Note that the value of the guardsize attribute of a particular thread attributes object does not necessarily correspond to the actual size of the guard area of any existing thread in your multithreaded program.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_attr_init( )
pthread_attr_setguardsize( )
pthread_attr_setstacksize( )
pthread_create( )
```

pthread_attr_getguardsize_np

Obtains the guardsize attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getguardsize_np(
                    attr,
                    guardsize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
guardsize	size_t	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getguardsize_np (
            const pthread_attr_t    *attr,
            size_t    *guardsize);
```

Arguments

attr
Address of the thread attributes object whose guardsize attribute is obtained.

guardsize
Receives the value of the guardsize attribute of the thread attributes object specified by *attr*.

Description

This routine obtains the value of the guardsize attribute of the thread attributes object specified in the *attr* argument and stores it in the location specified by the *guardsize* argument. The specified attributes object must already be initialized at the time this routine called.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The guardsize attribute of a thread attributes object specifies the minimum size (in bytes) of the guard area for the stack of a new thread.

A guard area can help a multithreaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that DECthreads allocates at the overflow end of the thread's stack. When any thread attempts to access a memory location within this region, a memory addressing violation occurs.

Note that the value of the guardsize attribute of a particular thread attributes object does not necessarily correspond to the actual size of the stack guard area of any existing thread in your multithreaded program.

Note

This routine has been superseded by the `pthread_attr_getguardsize()` routine, as specified by the Single UNIX Specification, Version 2.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_attr_init()
pthread_attr_setguardsize_np()
```

pthread_attr_getinheritsched

Obtains the inherit scheduling attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getinheritsched(
                    attr,
                    inheritsched );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
inheritsched	integer	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getinheritsched (
          const pthread_attr_t    *attr,
          int    *inheritsched);
```

Arguments

attr
Thread attributes object whose inherit scheduling attribute is obtained.

inheritsched
Receives the value of the inherit scheduling attribute. Refer to the description of the `pthread_attr_setinheritsched( )` function for valid values.

Description

This routine obtains the value of the inherit scheduling attribute from the specified thread attributes object. The inherit scheduling attribute specifies whether threads created using the attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the attributes object that is passed to `pthread_create( )`.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_attr_init( )
pthread_attr_setinheritsched( )
pthread_create( )
```

pthread_attr_getname_np

Obtains the object name attribute from a thread attributes object.

Syntax

```
pthread_attr_getname_np(
                    attr,
                    name,
                    len,
                    mbz );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
name	char	write
len	opaque size_t	read
mbz	void	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getname_np (
            const pthread_attr_t    *attr,
            char    *name,
            size_t    len,
            void    **mbz);
```

Arguments

attr
Address of the thread attributes object whose object name attribute is to be obtained.

name
Location to store the obtained object name.

len
Length in bytes of buffer at the location specified by *name*.

mbz
(Must be zero) Location for use by DECthreads. On DIGITAL UNIX Alpha and OpenVMS Alpha platforms, the value to which this argument points must be a 64-bit pointer. If compiling with short pointers, ensure that you have allocated a 64-bit value to receive the result.

Description

This routine copies the object name attribute from the thread attributes object specified by the *attr* argument to the buffer at the location specified by the *name* argument. Before calling this routine, your program must allocate the buffer indicated by *name*. A new thread created using the thread attributes object is initialized with the object name that was set in that attributes object.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

If the specified thread attributes object has not been previously set with an object name, this routine copies a C language null string into the buffer at location *name*.

This routine contrasts with `pthread_getname_np( )`, which obtains the object name from the thread object for an existing thread.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_getname_np( )
pthread_attr_setname_np( )
pthread_setname_np( )
```

pthread_attr_getschedparam

Obtains the scheduling parameters for an attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getschedparam(
                          attr,
                          param );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
param	struct sched_param	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getschedparam (
                  const pthread_attr_t    *attr,
                  struct sched_param      *param);
```

Arguments

attr
Thread attributes object of the scheduling policy attribute whose parameters are obtained.

param
Receives the values of scheduling parameters for the scheduling policy attribute of the attributes object specified by the *attr* argument. Refer to the description of the `pthread_attr_setschedparam( )` routine for valid parameters and their values.

Description

This routine obtains the scheduling parameters associated with the scheduling policy attribute of the specified thread attributes object.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_attr_init( )
pthread_attr_setschedparam( )
pthread_create( )
```

pthread_attr_getschedpolicy

Obtains the scheduling policy attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getschedpolicy(
                              attr,
                              policy );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
policy	integer	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getschedpolicy (
                const pthread_attr_t    *attr,
                int    *policy);
```

Arguments

attr
Thread attributes object whose scheduling policy attribute is obtained.

policy
Receives the value of the scheduling policy attribute. Refer to the description of the `pthread_attr_setschedpolicy( )` routine for valid values.

Description

This routine obtains the value of the scheduling policy attribute of the specified thread attributes object. The scheduling policy attribute defines the scheduling policy for threads created using the attributes object.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_attr_init( )
pthread_attr_setschedpolicy( )
pthread_create( )
```

pthread_attr_getscope

Obtains the contention scope attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getscope(
                    attr,
                    scope );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
scope	int	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getscope (
                const pthread_attr_t    *attr,
                int     *scope);
```

Arguments

attr
Address of the thread attributes object whose contention scope attribute is obtained.

scope
Receives the value of the contention scope attribute of the thread attributes object specified by *attr*.

Description

This routine obtains the value of the contention scope attribute of the thread attributes object specified in the *attr* argument and stores it in the location specified by the *scope* argument. The specified attributes object must already be initialized at the time this routine is called.

The contention scope attribute specifies the set of threads with which a thread must compete for processing resources. The contention scope attribute specifies whether the new thread competes for processing resources only with other threads in its own process, called **process contention scope**, or with all threads on the system, called **system contention scope**.

DECthreads selects at most one thread to execute on each processor at any point in time. DECthreads resolves the contention based on each thread's scheduling attributes (for example, priority) and scheduling policy (for example, round-robin).

A thread created using a thread attributes object whose contention scope attribute is set to `PTHREAD_SCOPE_PROCESS` contends for processing resources with other threads within its own process that also were created with `PTHREAD_SCOPE_PROCESS`. It is unspecified how such threads are scheduled relative to threads in other processes or threads in the same process that were created with `PTHREAD_SCOPE_SYSTEM` contention scope.

A thread created using a thread attributes object whose contention scope attribute is set to `PTHREAD_SCOPE_SYSTEM` contends for processing resources with other threads in any process that also were created with `PTHREAD_SCOPE_SYSTEM`.

Note that the value of the contention scope attribute of a particular thread attributes object does not necessarily correspond to the actual scheduling contention scope of any existing thread in your multithreaded program.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ENOSYS]	This routine is not supported by the implementation.

Associated Routines

```
pthread_attr_init()
pthread_attr_setscope()
```

pthread_attr_getstackaddr

Obtains the stack address attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getstackaddr(
                          attr,
                          stackaddr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
stackaddr	void	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getstackaddr (
                const pthread_attr_t    *attr,
                void    **stackaddr);
```

Arguments

attr
Address of the thread attributes object whose stack address attribute is obtained.

stackaddr
Receives the value of the stack address attribute of the thread attributes object specified by *attr*.

Description

This routine obtains the value of the stack address attribute of the thread attributes object specified in the *attr* argument and stores it in the location specified by the *stackaddr* argument. The specified attributes object must already be initialized at the time this routine is called.

The stack address attribute of a thread attributes object points to the origin of the stack for a new thread.

Note that the value of the stack address attribute of a particular thread attributes object does not necessarily correspond to the actual stack origin of any existing thread in your multithreaded program.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.

Associated Routines

```
pthread_attr_getguardsize( )
pthread_attr_getstacksize( )
pthread_attr_init( )
pthread_attr_setguardsize( )
pthread_attr_setstackaddr( )
pthread_attr_setstacksize( )
pthread_create( )
```

pthread_attr_getstacksize

Obtains the stacksize attribute of the specified thread attributes object.

Syntax

```
pthread_attr_getstacksize(
                          attr,
                          stacksize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
stacksize	size_t	write

C Binding

```
#include <pthread.h>

int
pthread_attr_getstacksize (
            const pthread_attr_t    *attr,
            size_t    *stacksize);
```

Arguments

attr
Thread attributes object whose stacksize attribute is obtained.

stacksize
Receives the value for the stacksize attribute of the thread attributes object specified by the *attr* argument.

Description

This routine obtains the stacksize attribute of the thread attributes object specified in the *attr* argument.

Return Values

On successful completion, this routine returns a zero (0) and the stacksize value in the location specified in the *stacksize* argument.

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
EINVAL	The value specified by *attr* is invalid.

Associated Routines

```
pthread_attr_init( )
pthread_attr_setstacksize( )
pthread_create( )
```

pthread_attr_init

Initializes a thread attributes object.

Syntax

```
pthread_attr_init(
                  attr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write

C Binding

```
#include <pthread.h>

int
pthread_attr_init (
            pthread_attr_t     *attr);
```

Arguments

attr
Address of a thread attributes object to be initialized.

Description

This routine initializes the thread attributes object specified by the *attr* argument with a set of default attribute values. A thread attributes object is used to specify the attributes of one or more threads when they are created. The attributes object created by this routine is used only in calls to the `pthread_create( )` routine.

The following routines change individual attributes of an initialized thread attributes object:

```
pthread_attr_setdetachstate( )
pthread_attr_setguardsize( )
pthread_attr_setinheritsched( )
pthread_attr_setschedparam( )
pthread_attr_setschedpolicy( )
pthread_attr_setscope( )
```

```
pthread_attr_setstackaddr( )
pthread_attr_setstacksize( )
```

The attributes of the thread attributes object are initialized to default values. The default value of each attribute is discussed in the reference description for each routine listed above.

When a thread attributes object is used to create a thread, the object's attribute values determine the characteristics of the new thread. Thus, attributes objects act as additional arguments to thread creation. Changing the attributes of a thread attributes object does not affect any threads that were previously created using that attributes object.

You can use the same thread attributes object in successive calls to `pthread_create( )`, from any thread. (However, you *cannot* use the same value of the stack address attribute to create multiple threads that might run concurrently; threads cannot share a stack.) If more than one thread might change the attributes in a shared attributes object, your program must use a mutex to protect the integrity of the attributes object's contents.

When you set the scheduling policy or scheduling parameters, or both, in a thread attributes object, you must disable scheduling inheritance if you want the scheduling attributes you set to be used at thread creation. To disable scheduling inheritance, before creating the new thread use the `pthread_attr_setinheritsched( )` routine to specify the value `PTHREAD_EXPLICIT_SCHED` for the *inherit* argument.

Return Values

If an error condition occurs, the thread attributes object cannot be used, and this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.
[ENOMEM]	Insufficient memory exists to initialize the thread attributes object.

Associated Routines

```
pthread_attr_destroy( )
pthread_attr_setdetachstate( )
pthread_attr_setguardsize( )
pthread_attr_setinheritsched( )
pthread_attr_setschedparam( )
pthread_attr_setschedpolicy( )
pthread_attr_setscope( )
pthread_attr_setstackaddr( )
pthread_attr_setstacksize( )
pthread_create( )
```

pthread_attr_setdetachstate

Changes the detachstate attribute in the specified thread attributes object.

Syntax

```
pthread_attr_setdetachstate(
                            attr,
                            detachstate);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
detachstate	integer	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setdetachstate (
            pthread_attr_t   *attr,
            int   detachstate);
```

Arguments

attr
Thread attributes object to be modified.

detachstate
New value for the detachstate attribute. Valid values are as follows:

PTHREAD_CREATE_JOINABLE	This is the default value. Threads are created in "undetached" state.
PTHREAD_CREATE_DETACHED	The created thread is detached immediately, before it begins running.

Description

This routine changes the detachstate attribute in the thread attributes object specified by the *attr* argument. The detachstate attribute specifies whether the thread created using the specified thread attributes object is created in a detached state or not. A value of `PTHREAD_CREATE_JOINABLE` indicates the thread is not detached, and a value of `THREAD_CREATE_DETACHED` indicates the thread is detached. `PTHREAD_CREATE_JOINABLE` is the default value.

Your program cannot use the thread handle (the value of type `pthread_t` returned by the `pthread_create( )` routine) of a detached thread because the thread might terminate asynchronously, and a detached thread ID is not valid after termination. In particular, it is an error to attempt to detach or join with a detached thread.

When a thread that has not been detached completes execution, DECthreads retains the state of that thread to allow another thread to join with it. If the thread is detached before it completes execution, DECthreads is free to immediately reclaim the thread's storage and resources. Failing to detach threads that have completed execution can result in wasting resources, so threads should be detached as soon as the program is done with them. If there is no need to use the thread's handle after creation, create the thread initially detached.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by the *detachstate* argument is invalid.

Associated Routines

```
pthread_attr_init( )
pthread_attr_getdetachstate( )
pthread_create( )
pthread_join( )
```

pthread_attr_setguardsize

Changes the guardsize attribute of the specified thread attributes object.

Syntax

```
pthread_attr_setguardsize(
                          attr,
                          guardsize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
guardsize	size_t	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setguardsize (
            pthread_attr_t    *attr,
            size_t    guardsize);
```

Arguments

attr
Address of the thread attributes object whose guardsize attribute is to be modified.

guardsize
New value for the guardsize attribute of the thread attributes object specified by *attr*.

Description

This routine uses the value specified in the *guardsize* argument to set the guardsize attribute of the thread attributes object specified in the *attr* argument.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The guardsize attribute of a thread attributes object specifies the minimum size (in bytes) of the guard area for the stack of a new thread.

A guard area can help a multithreaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that DECthreads allocates at the overflow end of the thread's stack. When any thread attempts to access a memory location within this region, a memory addressing violation occurs.

A new thread can be created with a default guardsize attribute value. This value is platform dependent, but will always be at least one "hardware protection unit" (that is, at least one page). For more information, see this guide's platform-specific appendixes.

After this routine is called, due to platform-specific factors DECthreads might reserve a larger guard area for the new thread than was specified in the *guardsize* argument. See this guide's platform-specific appendixes for more information.

DECthreads allows your program to specify the size of a thread stack's guard area for two reasons:

- When a thread allocates large data structures on its stack, a guard area with a size greater than the default size might be required to detect stack overflow.
- Overflow protection of a thread's stack can potentially waste system resources, such as for an application that creates a large number of threads that will never overflow their stacks. Your multithreaded program can conserve system resources by "turning off" a thread's stack guard area—that is, by specifying a guardsize attribute of zero.

If a thread is created using a thread attributes object whose stackaddr attribute is set (using the `pthread_attr_setstackaddr()` routine), this routine ignores the object's guardsize attribute and provides no thread stack guard area for the new thread.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The argument *attr* is invalid, or the argument *guardsize* contains an invalid value.

Associated Routines

```
pthread_attr_init( )
pthread_attr_getguardsize( )
pthread_attr_setstacksize( )
pthread_create( )
```

pthread_attr_setguardsize_np

Changes the guardsize attribute of the specified thread attributes object.

Syntax

```
pthread_attr_setguardsize_np(
                    attr,
                    guardsize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
guardsize	size_t	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setguardsize_np (
          pthread_attr_t    *attr,
          size_t    guardsize);
```

Arguments

attr
Address of the thread attributes object whose guardsize attribute is to be modified.

guardsize
New value for the guardsize attribute of the thread attributes object specified by *attr*.

Description

This routine uses the value specified in the *guardsize* argument to set the guardsize attribute of the thread attributes object specified in the *attr* argument.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The guardsize attribute of a thread attributes object specifies the minimum size (in bytes) of the guard area for the stack of a new thread.

A guard area can help a multithreaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that DECthreads allocates at the overflow end of the thread's stack. When any thread attempts to access a memory location within this region, a memory addressing violation occurs.

A new thread can be created with a default guardsize attribute value. This value is platform dependent, but will always be at least one "hardware protection unit" (that is, at least one page). For more information, see this guide's platform-specific appendixes.

After this routine is called, due to platform-specific factors DECthreads might reserve a larger guard area for the new thread than was specified in the *guardsize* argument. See this guide's platform-specific appendixes for more information.

DECthreads allows your program to specify the size of a thread stack's guard area for two reasons:

- When a thread allocates large data structures on its stack, a guard area with a size greater than the default size might be required to detect stack overflow.
- Overflow protection of a thread's stack can potentially waste system resources, such as for an application that creates a large number of threads that will never overflow their stacks. Your multithreaded program can conserve system resources by "turning off" a thread's stack guard area—that is, by specifying a guardsize attribute of zero.

If a thread is created using a thread attributes object whose stackaddr attribute is set (using the `pthread_attr_setstackaddr( )` routine), this routine ignores the object's guardsize attribute and provides no thread stack guard area for the new thread.

Note

This routine has been superseded by the `pthread_attr_setguardsize( )` routine, as specified by the Single UNIX Specification, Version 2.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The argument *attr* is invalid, or the argument *guardsize* contains an invalid value.

Associated Routines

```
pthread_attr_init( )
pthread_attr_getguardsize_np( )
pthread_attr_setstacksize( )
pthread_create( )
```

pthread_attr_setinheritsched

Changes the inherit scheduling attribute of the specified thread attributes object.

Syntax

```
pthread_attr_setinheritsched(
                         attr,
                         inheritsched );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
inheritsched	integer	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setinheritsched (
         pthread_attr_t    *attr,
         int    inheritsched);
```

Arguments

attr
Thread attributes object whose inherit scheduling attribute is to be modified.

inheritsched
New value for the inherit scheduling attribute. Valid values are as follows:

PTHREAD_INHERIT_SCHED	The created thread inherits the scheduling policy and associated scheduling attributes of the thread calling pthread_create( ). Any scheduling attributes in the attributes object specified by the pthread_create( ) *attr* argument are ignored during thread creation. This is the default value.
PTHREAD_EXPLICIT_SCHED	The scheduling policy and associated scheduling attributes of the created thread are set to the corresponding values from the attribute object specified by the pthread_create( ) *attr* argument.

Description

This routine changes the inherit scheduling attribute of the thread attributes object specified by the *attr* argument. The inherit scheduling attribute specifies whether a thread created using the specified attributes object inherits the scheduling attributes of the creating thread, or uses the scheduling attributes stored in the attributes object specified by the pthread_create( ) *attr* argument.

The first thread in an application has a scheduling policy of SCHED_OTHER. See the pthread_attr_setschedparam( ) and pthread_attr_setschedpolicy( ) routines for more information on valid priority values and valid scheduling policy values, respectively.

Inheriting scheduling attributes (instead of using the scheduling attributes stored in the attributes object) is useful when a thread is creating several helper threads—that is, threads that are intended to work closely with the creating thread to cooperatively solve the same problem. For example, inherited scheduling attributes ensure that helper threads created in a sort routine execute with the same priority as the calling thread.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	One or both of the values specified by *inherit* and *attr* are invalid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

Associated Routines

pthread_attr_init( )
pthread_attr_getinheritsched( )
pthread_attr_setschedpolicy( )
pthread_attr_setschedparam( )
pthread_create( )

pthread_attr_setname_np

Changes the object name attribute in a thread attributes object.

Syntax

pthread_attr_setname_np(
attr,
name,
mbz);

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
name	char	read
mbz	void	read

C Binding

#include <pthread.h>

```
int
pthread_attr_setname_np (
        pthread_attr_t    *attr,
        const char    *name,
        void    *mbz);
```

Arguments

attr
Address of the thread attributes object whose object name attribute is to be changed.

name
Object name value to copy into the thread attributes object's object name attribute.

mbz
(Must be zero) Argument for use by DECthreads.

pthread_attr_setname_np

Description

This routine changes the object name attribute in the thread attributes object specified by the *attr* argument to the value specified by the *name* argument. A new thread created using the thread attributes object is initialized with the object name that was set in that attributes object.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

This routine contrasts with `pthread_setname_np( )`, which changes the object name in the thread object for an existing thread.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid, or the length in characters of *name* exceeds 31.
[ENOMEM]	Insufficient memory exists to create a copy of the object name string.

Associated Routines

```
pthread_attr_getname_np( )
pthread_getname_np( )
pthread_setname_np( )
```

pthread_attr_setschedparam

Changes the values of the parameters associated with a scheduling policy of the specified thread attributes object.

Syntax

```
pthread_attr_setschedparam(
                    attr,
                    param );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
param	struct sched_param	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setschedparam (
          pthread_attr_t    *attr,
          const struct sched_param    *param);
```

Arguments

attr
Thread attributes object for the scheduling policy attribute whose parameters are to be set.

param
A structure containing new values for scheduling parameters associated with the scheduling policy attribute of the specified thread attributes object.

Note

DECthreads provides only the `sched_priority` scheduling parameter. It allows specifying the scheduling priority. See below for information about this scheduling parameter.

Description

This routine sets the scheduling parameters associated with the scheduling policy attribute of the thread attributes object specified by the *attr* argument.

Scheduling Priority

Use the `sched_priority` field of a `sched_param` structure to set a thread's execution priority. The effect of the scheduling priority you assign depends on the scheduling policy specified for the attributes object specified by the *attr* argument.

By default, a created thread inherits the priority of the thread calling `pthread_create( )`. To specify a priority using this routine, scheduling inheritance must be disabled at the time the thread is created. Before calling `pthread_create( )`, call `pthread_attr_setinheritsched( )` and specify the value `PTHREAD_EXPLICIT_SCHED` for the *inherit* argument.

An application specifies priority only to express the urgency of executing the thread relative to other threads. *Do not use priority to control mutual exclusion when accessing shared data*. With a sufficient number of processors present, all ready threads, regardless of priority, execute simultaneously.

Valid values of the `sched_priority` scheduling parameter depend on the chosen scheduling policy. Use the POSIX routines `sched_get_priority_min( )` or `sched_get_priority_max( )` to determine the low and high limits of each policy.

Additionally, DECthreads provides *nonportable* priority range constants, as follows:

Policy	Low	High
SCHED_FIFO	PRI_FIFO_MIN	PRI_FIFO_MAX
SCHED_RR	PRI_RR_MIN	PRI_RR_MAX
SCHED_OTHER	PRI_OTHER_MIN	PRI_OTHER_MAX
SCHED_FG_NP	PRI_FG_MIN_NP	PRI_FG_MAX_NP
SCHED_BG_NP	PRI_BG_MIN_NP	PRI_BG_MAX_NP

The default priority varies by DECthreads platform. On DIGITAL UNIX, the default is 19 (that is, the POSIX priority of a normal timeshare process). On other platforms, the default priority is the midpoint between `PRI_FG_MIN_NP` and `PRI_FG_MAX_NP`. (Section 2.3.6 describes how to specify priorities between the minimum and maximum values.)

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *param* is invalid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

Associated Routines

```
pthread_attr_init( )
pthread_attr_getschedparam( )
pthread_attr_setinheritsched( )
pthread_attr_setschedpolicy( )
pthread_create( )
sched_yield( )
```

pthread_attr_setschedpolicy

Changes the scheduling policy attribute of the specified thread attributes object.

Syntax

```
pthread_attr_setschedpolicy(
                    attr,
                    policy );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
policy	integer	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setschedpolicy (
          pthread_attr_t    *attr,
          int    policy);
```

Arguments

attr
Thread attributes object to be modified.

policy
New value for the scheduling policy attribute. Valid values are as follows:

SCHED_BG_NP
SCHED_FG_NP (also known as SCHED_OTHER)
SCHED_FIFO
SCHED_RR

SCHED_OTHER is the default value. See Section 2.3.2.2 for a description of the scheduling policies.

Description

This routine sets the scheduling policy of a thread that is created using the attributes object specified by the *attr* argument. The default value of the scheduling attribute is `SCHED_OTHER`.

By default, a created thread inherits the policy of the thread calling `pthread_create( )`. To specify a policy using this routine, scheduling inheritance must be disabled at the time the thread is created. Before calling `pthread_create( )`, call `pthread_attr_setinheritsched( )` and specify the value `PTHREAD_EXPLICIT_SCHED` for the *inherit* argument.

Never attempt to use scheduling as a mechanism for synchronization. (Refer to Chapter 1 and Chapter 2.)

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *policy* is invalid.

Associated Routines

```
pthread_attr_init( )
pthread_attr_getschedpolicy( )
pthread_attr_setinheritsched( )
pthread_attr_setschedparam( )
pthread_create( )
```

pthread_attr_setscope

Sets the contention scope attribute of the specified thread attributes object.

Syntax

```
pthread_attr_setscope(
                attr,
                scope );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
scope	int	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setscope (
            pthread_attr_t    *attr,
            int    scope);
```

Arguments

attr
Address of the thread attributes object whose contentions scope attribute is to be modified.

scope
New value for the contention scope attribute of the thread attributes object specified by *attr*.

Description

This routine uses the value specified in the *scope* argument to set the contention scope attribute of the thread attributes object specified in the *attr* argument. The specified attributes object must already be initialized at the time this routine is called.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The contention scope attribute specifies the set of threads with which a thread must compete for processing resources. The contention scope attribute specifies whether the new thread competes for processing resources only with other threads in its own process, called **process contention scope**, or with all threads on the system, called **system contention scope**.

Note

On DIGITAL UNIX, DECthreads supports both process contention scope and system contention scope threads. On OpenVMS, DECthreads supports only process contention scope threads. On Windows NT, DECthreads supports only system contention scope threads.

DECthreads selects at most one thread to execute on each processor at any point in time. DECthreads resolves the contention based on each thread's scheduling attributes (for example, priority) and scheduling policy (for example, round-robin).

A thread created using a thread attributes object whose contention scope attribute is set to `PTHREAD_SCOPE_PROCESS` contends for processing resources with other threads within its own process that also were created with `PTHREAD_SCOPE_PROCESS`. It is unspecified how such threads are scheduled relative to threads in other processes or threads in the same process that were created with `PTHREAD_SCOPE_SYSTEM` contention scope.

A thread created using a thread attributes object whose contention scope attribute is set to `PTHREAD_SCOPE_SYSTEM` contends for processing resources with other threads in any process that also were created with `PTHREAD_SCOPE_SYSTEM`.

Note that the value of the contention scope attribute of a particular thread attributes object does not necessarily correspond to the actual scheduling contention scope of any existing thread in your multithreaded program.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ENOSYS]	This routine is not supported by the implementation.
[EINVAL]	The value of the attribute being set is not valid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

Associated Routines

```
pthread_attr_destroy( )
pthread_attr_init( )
pthread_attr_getscope( )
pthread_create( )
```

pthread_attr_setstackaddr

Changes the stack address attribute of the specified thread attributes object.

Syntax

```
pthread_attr_setstackaddr(
                          attr,
                          stackaddr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
stackaddr	void	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setstackaddr (
            pthread_attr_t    *attr,
            void    *stackaddr);
```

Arguments

attr
Address of the thread attributes object whose stack address attribute is to be modified.

stackaddr
New value for the stack address attribute of the thread attributes object specified by *attr*.

Description

This routine uses the value specified in the *stackaddr* argument to set the stack address attribute of the thread attributes object specified in the *attr* argument.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The stack address attribute of a thread attributes object points to the origin of the stack for a new thread.

The default value for the stack address attribute of an initialized thread attributes object is NULL.

Note

Correct use of this routine depends upon details of the target platform's stack architecture. Thus, this routine cannot be used in a portable manner.

The size of the stack must be at least `PTHREAD_STACK_MIN` bytes (see the `pthread.h` header file). However, because DECthreads must use a portion of this stack memory to begin thread execution and to maintain thread state, your program's "user thread code" cannot rely on using all of the stack memory allocated.

For your program to calculate a value for the stackaddr attribute, note that:

- Your program must allocate the memory that will be used for the new thread's stack.
- On DIGITAL UNIX, to create a new thread using a thread attributes object, the stackaddr attribute must be an address that points to the high-memory end of the memory region allocated for the stack. This address must point to the highest even-boundary quadword in the allocated memory region.

Also note that:

- If you use the `pthread_attr_setstackaddr( )` routine to set a thread attributes object's stack address attribute and use that attributes object to create a new thread, DECthreads ignores the attributes object's guardsize attribute and provides no thread stack guard area for the new thread.
- If you use the same thread attributes object to create more than one thread and each created thread uses a nondefault stack address, you must use the `pthread_attr_setstackaddr( )` routine to set a unique stack address attribute value for each new thread created using that attributes object.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.

Associated Routines

```
pthread_attr_getguardsize( )
pthread_attr_getstackaddr( )
pthread_attr_getstacksize( )
pthread_attr_init( )
pthread_attr_setguardsize( )
pthread_attr_setstacksize( )
pthread_create( )
```

pthread_attr_setstacksize

Changes the stacksize attribute in the specified thread attributes object.

Syntax

```
pthread_attr_setstacksize(
                          attr,
                          stacksize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
stacksize	size_t	read

C Binding

```
#include <pthread.h>

int
pthread_attr_setstacksize (
            pthread_attr_t    *attr,
            size_t    stacksize);
```

Arguments

attr
Threads attributes object to be modified.

stacksize
New value for the stacksize attribute of the thread attributes object specified by the *attr* argument. The *stacksize* argument must be greater than or equal to `PTHREAD_STACK_MIN`. `PTHREAD_STACK_MIN` specifies the minimum size (in bytes) of stack needed for a thread.

Description

This routine sets the stacksize attribute in the thread attributes object specified by the *attr* argument. Use this routine to adjust the size of the writable area of the stack for a new thread.

The size of a thread's stack is fixed at the time of thread creation. Only the initial thread can dynamically extend its stack.

Many compilers do not check for stack overflow. Ensure that the new thread's stack is sufficient for the resources required by routines that are called from the thread.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid, or the value specified by *stacksize* is less than `PTHREAD_STACK_MIN` or exceeds a DECthreads-imposed limit.

Associated Routines

```
pthread_attr_init( )
pthread_attr_getstacksize( )
pthread_create( )
```

pthread_cancel

Allows a thread to request a thread to terminate execution.

Syntax

```
pthread_cancel(
               thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

C Binding

```
#include <pthread.h>

int
pthread_cancel (
            pthread_t     thread);
```

Arguments

thread
Thread that receives a cancelation request.

Description

This routine sends a cancelation request to the specified target *thread*. A cancelation request is a mechanism by which a calling thread requests the target thread to terminate as quickly as possible. Issuing a cancelation request does not guarantee that the target thread will receive or handle the request.

When the cancelation request is acted on, all active cleanup handler routines for the target thread are called. When the last cleanup handler returns, the thread-specific data destructor routines are called for each thread-specific data key with a destructor and for which the target thread has a non-NULL value. Finally, the target thread is terminated.

Note that cancelation of the target thread runs asynchronously with respect to the calling thread's returning from `pthread_cancel( )`. The target thread's cancelability state and type determine when or if the cancelation takes place, as follows:

1. The target thread can delay cancelation during critical operations by setting its cancelability state to `PTHREAD_CANCEL_DISABLE`.
2. Because of communication delays, the calling thread can only rely on the fact that a cancelation request will eventually become pending in the target thread (provided that the target thread does not terminate beforehand).
3. The calling thread has no guarantee that a pending cancelation request will be delivered because delivery is controlled by the target thread.

When a cancelation request is delivered to a thread, termination processing is similar to that for `pthread_exit( )`. For more information about thread termination, see the Thread Termination section of `pthread_create( )`.

This routine is preferred in implementing an Ada `abort` statement and any other language- or software-defined construct for requesting thread cancelation.

The results of this routine are unpredictable, if the value specified in *thread* refers to a thread that does not currently exist.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified *thread* is invalid.
[ESRCH]	The *thread* argument does not specify an existing thread.

Associated Routines

```
pthread_cleanup_pop( )
pthread_cleanup_push( )
pthread_create( )
pthread_exit( )
pthread_join( )
pthread_setcancelstate( )
pthread_setcanceltype( )
pthread_testcancel( )
```

pthread_cleanup_pop

(Macro) Removes the cleanup handler routine from the calling thread's cleanup handler stack and optionally executes it.

Syntax

```
pthread_cleanup_pop(
          execute );
```

Argument	Data Type	Access
execute	integer	read

C Binding

```
#include <pthread.h>

void
pthread_cleanup_pop(
          int     execute);
```

Arguments

execute
Integer that specifies whether the cleanup handler routine specified in the matching call to `pthread_cleanup_push( )` is executed. A nonzero value causes the cleanup handler routine to be executed.

Description

This routine removes the cleanup handler routine established by the matching call to `pthread_cleanup_push( )` from the calling thread's cleanup handler stack, then executes it if the value specified in this routine's *execute* argument is nonzero.

A cleanup handler routine can be used to clean up from a block of code whether exited by normal completion, cancelation, or the raising (or reraising) of an exception. The routine is popped from the calling thread's cleanup handler stack and is executed with the *arg* argument when any of the following actions occur:

- The thread calls `pthread_cleanup_pop( )` and specifies a nonzero value for the *execute* argument.

- The thread calls `pthread_exit( )`.
- The thread is canceled.
- An exception is raised and is caught when DECthreads unwinds the calling thread's stack to the lexical scope of the `pthread_cleanup_push( )` and `pthread_cleanup_pop( )` pair.

This routine and `pthread_cleanup_push( )` are implemented as macros and must appear as statements and in pairs within the same lexical scope. You can think of the `pthread_cleanup_push( )` macro as expanding to a string whose first character is a left brace ({) and `pthread_cleanup_pop( )` as expanding to a string containing the corresponding right brace (}).

Return Values

None

Associated Routines

```
pthread_cancel( )
pthread_cleanup_push( )
pthread_create( )
pthread_exit( )
```

pthread_cleanup_push

(Macro) Establishes a cleanup handler routine to be executed when the thread exits or is canceled.

Syntax

```
pthread_cleanup_push(
                    routine,
                    arg );
```

Argument	Data Type	Access
routine	procedure	read
arg	user_arg	read

C Binding

```
#include &lt;phtread.h&gt;

void
pthread_cleanup_push(
          void    (*routine)(void *),
          void    *arg);
```

Arguments

routine
Routine executed as the cleanup handler.

arg
Argument pass to the cleanup handler routine.

Description

This routine pushes the specified routine onto the calling thread's cleanup handler stack. The cleanup handler routine is popped from the stack and executed with the *arg* argument when any of the following actions occur:

- The thread calls `pthread_cleanup_pop( )` and specifies a nonzero value for the *execute* argument.
- The thread calls `pthread_exit( )`.
- The thread is canceled.

- An exception is raised and is caught when DECthreads unwinds the calling thread's stack to the lexical scope of the `pthread_cleanup_push( )` and `pthread_cleanup_pop( )` pair.

This routine and `pthread_cleanup_pop( )` are implemented as macros and must appear as statements and in pairs within the same lexical scope. You can think of the `pthread_cleanup_push( )` macro as expanding to a string whose first character is a left brace ({) and `pthread_cleanup_pop( )` as expanding to a string containing the corresponding right brace (}).

Return Values

None

Associated Routines

```
pthread_cancel( )
pthread_cleanup_pop( )
pthread_create( )
pthread_exit( )
pthread_testcancel( )
```

pthread_condattr_destroy

Destroys a condition variable attributes object.

Syntax

```
pthread_condattr_destroy(
                         attr );
```

Argument	Data Type	Access
attr	opaque pthread_condattr_t	write

C Binding

```
#include <pthread.h>

int
pthread_condattr_destroy (
            pthread_condattr_t    *attr);
```

Arguments

attr
Condition variable attributes object to be destroyed.

Description

This routine destroys the specified condition variable attributes object—that is, the object becomes uninitialized.

Condition variables that were created using this attributes object are not affected by the deletion of the condition variable attributes object.

After calling this routine, the results of using *attr* in a call to any routine (other than `pthread_condattr_init( )`) are unpredictable.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The attributes object specified by *attr* is invalid.

Associated Routines

`pthread_condattr_init( )`

pthread_condattr_init

Initializes a condition variable attributes object.

Syntax

pthread_condattr_init(
    attr);

Argument	Data Type	Access
attr	opaque pthread_condattr_t	write

C Binding

```
#include <pthread.h>

int
pthread_condattr_init (
            pthread_condattr_t    *attr);
```

Arguments

attr
Address of the condition variable attributes object to be initialized.

Description

This routine initializes the condition variable attributes object specified by the *attr* argument with a set of default attribute values.

When an attributes object is used to create a condition variable, the values of the individual attributes determine the characteristics of the new condition variable. Attributes objects act as additional arguments to condition variable creation. Changing individual attributes in an attributes object does not affect any condition variables that were previously created using that attributes object.

You can use the same condition variable attributes object in successive calls to `pthread_condattr_init( )`, from any thread. If multiple threads can change attributes in a shared attributes object, your program must use a mutex to protect the integrity of that attributes object.

Results are undefined if this routine is called and the *attr* argument specifies a condition variable attributes object that is already initialized.

Currently, no attributes affecting condition variables are defined. You cannot change any attributes in the condition variable attributes object.

The `pthread_condattr_init( )` and `pthread_condattr_destroy( )` routines are provided for future expandability of the DECthreads **pthread** interface and to conform with the POSIX.1c standard. These routines serve no useful function, because there are no `pthread_condattr_set*( )` type routines available at this time.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ENOMEM]	Insufficient memory exists to initialize the condition variable attributes object.

Associated Routines

```
pthread_condattr_destroy( )
pthread_cond_init( )
```

pthread_cond_broadcast

Wakes all threads that are waiting on the specified condition variable.

Syntax

```
pthread_cond_broadcast(
                    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

C Binding

```
#include <pthread.h>

int
pthread_cond_broadcast (
            pthread_cond_t     *cond);
```

Arguments

cond
Condition variable upon which the threads (to be awakened) are waiting.

Description

This routine unblocks all threads waiting on the specified condition variable *cond*. Calling this routine implies that data guarded by the associated mutex has changed, so that it might be possible for one or more waiting threads to proceed. The threads that are unblocked shall contend for the mutex according to their respective scheduling policies (if applicable).

If only one of the threads waiting on a condition variable may be able to proceed, but any single thread can proceed, then use `pthread_cond_signal()` instead.

Whether the associated mutex is locked or unlocked, you can still call this routine. However, if predictable scheduling behavior is required, that mutex should then be locked by the thread calling the `pthread_cond_broadcast()` routine.

If no threads are waiting on the specified condition variable, this routine takes no action. The broadcast does not propagate to the next condition variable wait.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid.

Associated Routines

```
pthread_cond_destroy( )
pthread_cond_init( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
```

pthread_cond_destroy

Destroys a condition variable.

Syntax

```
pthread_cond_destroy(
                         cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write

C Binding

```
#include <pthread.h>

int
pthread_cond_destroy (
              pthread_cond_t     *cond);
```

Arguments

cond
Condition variable to be destroyed.

Description

This routine destroys the condition variable specified by *cond*. This effectively uninitializes the condition variable. Call this routine when a condition variable will no longer be referenced. Destroying a condition variable allows DECthreads to reclaim internal memory associated with the condition variable.

It is safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are blocked results in unpredictable behavior.

The results of this routine are unpredictable, if the condition variable specified in *cond* does not exist or is not initialized.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid.
[EBUSY]	The object being referenced by *cond* is being referenced by another thread that is currently executing `pthread_cond_wait( )` or `pthread_cond_timedwait( )` on the condition variable specified in *cond*.

Associated Routines

```
pthread_cond_broadcast( )
pthread_cond_init( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
```

pthread_cond_getname_np

Obtains the object name from a condition variable object.

Syntax

```
pthread_cond_getname_np(
                        cond,
                        name,
                        len );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read
name	char	write
len	opaque size_t	read

C Binding

```
#include <pthread.h>

int
pthread_cond_getname_np (
            pthread_cond_t    *cond,
            char    *name,
            size_t    len);
```

Arguments

cond
Address of the condition variable object whose object name is to be obtained.

name
Location to store the obtained object name.

len
Length in bytes of buffer at the location specified by *name*.

Description

This routine copies the object name from the condition variable object specified by the *cond* argument to the buffer at the location specified by the *name* argument. Before calling this routine, your program must allocate the buffer indicated by *name*.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

If the specified condition variable object has not been previously set with an object name, this routine copies a C language null string into the buffer at location *name*.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid.

Associated Routines

`pthread_cond_setname_np( )`

pthread_cond_init

Initializes a condition variable.

Syntax

```
pthread_cond_init(
                cond,
                attr );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write
attr	opaque pthread_condattr_t	read

C Binding

```
#include <pthread.h>

int
pthread_cond_init (
            pthread_cond_t    *cond,
            const pthread_condattr_t    *attr);
```

Arguments

cond
Condition variable to be initialized.

attr
Condition variable attributes object that defines the characteristics of the condition variable to be initialized.

Description

This routine initializes the condition variable *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used.

A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to data that is shared among threads; a condition variable allows threads to wait for that data to enter a defined state.

Condition variables are not owned by a particular thread. Any associated storage is not automatically deallocated when the creating thread terminates.

Use the DECthreads macro `PTHREAD_COND_INITIALIZER` to initialize statically allocated condition variables to the default condition variable attributes. To call this macro, enter:

`pthread_cond_t` *condition* = `PTHREAD_COND_INITIALIZER`

When statically initialized, a condition variable should not also be initialized using `pthread_cond_init( )`. Also, a statically initialized condition variable need not be destroyed using `pthread_cond_destroy( )`.

Under certain circumstances it might be impossible to wait upon a statically initialized condition variable when the process virtual address space (or some other memory limit) is nearly exhausted. In such a case `pthread_cond_wait( )` or `pthread_cond_timedwait( )` can return [ENOMEM]. To avoid this possibility, initialize critical condition variables using `pthread_cond_init( )`.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error, the condition variable is not initialized, and the contents of *cond* are undefined. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize another condition variable, or The system-imposed limit on the total number of condition variables under execution by a single user is exceeded.
[EBUSY]	The implementation has detected an attempt to reinitialize the object referenced by *cond*, a previously initialized, but not yet destroyed condition variable.
[EINVAL]	The value specified by *attr* is invalid.
[ENOMEM]	Insufficient memory exists to initialize the condition variable.

Associated Routines

```
pthread_cond_broadcast( )
pthread_cond_destroy( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
```

pthread_cond_setname_np

Changes the object name in a condition variable object.

Syntax

```
pthread_cond_setname_np(
                        cond,
                        name,
                        mbz );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write
name	char	read
mbz	void	read

C Binding

```
#include <pthread.h>

int
pthread_cond_setname_np (
            pthread_cond_t    *cond,
            const char    *name,
            void    *mbz);
```

Arguments

cond
Address of the condition variable object whose object name is to be changed.

name
Object name value to copy into the condition variable object.

mbz
(Must be zero) Argument for use by DECthreads.

pthread_cond_setname_np

Description

This routine changes the object name in the condition variable object specified by the *cond* argument to the value specified by the *name* argument. To set a new condition variable object's object name, call this routine immediately after initializing the condition variable object.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid, or the length in characters of *name* exceeds 31.
[ENOMEM]	Insufficient memory exists to create a copy of the object name string.

Associated Routines

`pthread_cond_getname_np()`

pthread_cond_signal

Wakes at least one thread that is waiting on the specified condition variable.

Syntax

```
pthread_cond_signal(
                    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

C Binding

```
#include <pthread.h>

int
pthread_cond_signal (
            pthread_cond_t     *cond);
```

Arguments

cond
Condition variable to be signaled.

Description

This routine unblocks at least one thread waiting on the specified condition variable *cond*. Calling this routine implies that data guarded by the associated mutex has changed, thus it might be possible for one of the waiting threads to proceed. In general, only one thread will be released.

If no threads are waiting on the specified condition variable, this routine takes no action. The signal does not propagate to the next condition variable wait.

This routine should be called when any thread waiting on the specified condition variable might find its predicate true, but only one thread should proceed. If more than one thread can proceed, or if any thread would not be able to proceed, then you must use `pthread_cond_broadcast( )`.

The scheduling policy determines which thread is awakened. For policies `SCHED_FIFO` and `SCHED_RR`, a blocked thread is chosen in priority order, using first-in/first-out (FIFO) within priorities.

You can call this routine even when the associated mutex is locked. However, if predictable scheduling behavior is required, then that mutex should be locked by the thread calling `pthread_cond_signal( )`.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid.

Associated Routines

```
pthread_cond_broadcast( )
pthread_cond_destroy( )
pthread_cond_init( )
pthread_cond_timedwait( )
pthread_cond_wait( )
```

pthread_cond_signal_int_np

Wakes one thread that is waiting on the specified condition variable (called from interrupt level only).

Syntax

```
pthread_cond_signal_int_np(
                           cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

C Binding

```
#include <pthread.h>

int
pthread_cond_signal_int_np(
            pthread_cond_t    *cond);
```

Arguments

cond
Condition variable to be signaled.

Description

This routine wakes one thread waiting on the specified condition variable. It can only be called from a software interrupt handler routine (such as from a DIGITAL UNIX signal handler or OpenVMS AST). Calling this routine implies that it might be possible for a single waiting thread to proceed.

The scheduling policies of the waiting threads determine which thread is awakened. For policies SCHED_FIFO and SCHED_RR, a blocked thread is chosen in priority order, using first-in/first-out (FIFO) within priorities.

This routine does not cause a thread blocked on a condition variable to resume execution immediately. A thread resumes execution at some time after the interrupt handler routine returns.

You can call this routine regardless of whether the associated mutex is locked (by some other thread). Never lock a mutex from an interrupt handler routine.

Note

This routine allows you to signal a thread from a software interrupt handler. Do not call this routine from noninterrupt code. To signal a thread from the normal noninterrupt level, use `pthread_cond_signal( )`.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid.

Associated Routines

```
pthread_cond_broadcast( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
```

pthread_cond_timedwait

Causes a thread to wait for the specified condition variable to be signaled or broadcasted, such that it will awake after a specified period of time.

Syntax

```
pthread_cond_timedwait(
                        cond,
                        mutex,
                        abstime );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify
mutex	opaque pthread_mutex_t	modify
abstime	structure timespec	read

C Binding

```
#include <pthread.h>

int
pthread_cond_timedwait (
            pthread_cond_t          *cond,
            pthread_mutex_t         *mutex,
            const struct timespec   *abstime);
```

Arguments

cond
Condition variable that the calling thread waits on.

mutex
Mutex associated with the condition variable specified in *cond*.

abstime
Absolute time at which the wait expires, if the condition has not been signaled or broadcasted. See the `pthread_get_expiration_np( )` routine, which is used to obtain a value for this argument.

The *abstime* argument is specified in Universal Coordinated Time (UTC). In the UTC-based model, time is represented as seconds since the Epoch. The Epoch is defined as the time 0 hours, 0 minutes, 0 seconds, January 1st, 1970 UTC. Seconds since the Epoch is a value interpreted as the number of seconds between a specified time and the Epoch.

Description

This routine causes a thread to wait until one of the following occurs:

- The specified condition variable is signaled or broadcasted.
- The current system clock time is greater than or equal to the time specified by the *abstime* argument.

This routine is identical to `pthread_cond_wait( )`, except that this routine can return before a condition variable is signaled or broadcasted; specifically, when the specified time expires. For more information, see the `pthread_cond_wait( )` description.

This routine atomically releases the mutex and causes the calling thread to wait on the condition. The atomicity is important, because it means the thread cannot miss a wakeup while the mutex is unlocked. When the timer expires or when the wait is satisfied as a result of some thread calling `pthread_cond_signal( )` or `pthread_cond_broadcast( )`, the mutex is reacquired before returning to the caller.

If the current time equals or exceeds the expiration time, this routine returns immediately, releasing and reacquiring the mutex. It might cause the calling thread to yield (see the `sched_yield( ) description`. Your code should check the return status whenever this routine returns and take the appropriate action. Otherwise, waiting on the condition variable can become a nonblocking loop.

Call this routine after you lock the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond*, *mutex*, or *abstime* is invalid, or:
	Different mutexes are supplied for concurrent `pthread_cond_timedwait( )` operations or `pthread_cond_wait( )` operations on the same condition variable, or:
	The mutex was not owned by the calling thread at the time of the call.
[ETIMEDOUT]	The time specified by *abstime* expired.
[ENOMEM]	DECthreads cannot acquire memory needed to block using a statically initialized condition variable.

Associated Routines

```
pthread_cond_broadcast( )
pthread_cond_destroy( )
pthread_cond_init( )
pthread_cond_signal( )
pthread_cond_wait( )
pthread_get_expiration_np( )
```

pthread_cond_wait

Causes a thread to wait for the specified condition variable to be signaled or broadcasted.

Syntax

```
pthread_cond_wait(
                  cond,
                  mutex );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify
mutex	opaque pthread_mutex_t	modify

C Binding

```
#include <pthread.h>

int
pthread_cond_wait (
            pthread_cond_t    *cond,
            pthread_mutex_t    *mutex);
```

Arguments

cond
Condition variable that the calling thread waits on.

mutex
Mutex associated with the condition variable specified in *cond*.

Description

This routine causes a thread to wait for the specified condition variable to be signaled or broadcasted. Each condition corresponds to one or more Boolean relations, called a predicate, based on shared data. The calling thread waits for the data to reach a particular state for the predicate to become true. However, the return from this routine does not imply anything about the value of the predicate and it should be reevaluated upon return. Condition variables are discussed in Chapter 2 and Chapter 3.

Call this routine after you have locked the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

This routine atomically releases the mutex and causes the calling thread to wait on the condition. The atomicity is important, because it means the thread cannot miss a wakeup while the mutex is unlocked. When the wait is satisfied as a result of some thread calling `pthread_cond_signal( )` or `pthread_cond_broadcast( )`, the mutex is reacquired before returning to the caller.

A thread that changes the state of storage protected by the mutex in such a way that a predicate associated with a condition variable might now be true, must call either `pthread_cond_signal( )` or `pthread_cond_broadcast( )` for that condition variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This routine might (with low probability) return when the condition variable has not been signaled or broadcasted. When this occurs, the mutex is reacquired before the routine returns. To handle this type of situation, enclose each call to this routine in a loop that checks the predicate. The loop provides documentation of your intent and protects against these spurious wakeups, while also allowing correct behavior even if another thread consumes the desired state before the awakened thread runs.

It is illegal for threads to wait on the same condition variable by specifying different mutexes.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* or *mutex* is invalid, or: Different mutexes are supplied for concurrent `pthread_cond_wait( )` or `pthread_cond_timedwait( )` operations on the same condition variable, or: The mutex was not owned by the calling thread at the time of the call.

Return	Description
[ENOMEM]	DECthreads cannot acquire memory needed to block using a statically initialized condition variable.

Associated Routines

```
pthread_cond_broadcast( )
pthread_cond_destroy( )
pthread_cond_init( )
pthread_cond_signal( )
pthread_cond_timedwait( )
```

pthread_create

Creates a thread.

Syntax

```
pthread_create(
            thread,
            attr,
            start_routine,
            arg );
```

Argument	Data Type	Access
thread	opaque pthread_t	write
attr	opaque pthread_attr_t	read
start_routine	procedure	read
arg	user_arg	read

C Binding

```
#include <pthread.h>

int
pthread_create (
            pthread_t    *thread,
            const pthread_attr_t    *attr,
            void *    (*start_routine) (void *),
            void    *arg);
```

Arguments

thread
Location for thread object to be created.

attr
Thread attributes object that defines the characteristics of the thread being created. If you specify NULL, default attributes are used.

start_routine
Function executed as the new thread's start routine.

arg
Address value copied and passed to the thread's start routine.

Description

This routine creates a thread. A thread is a single, sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations.

Successful execution of this routine includes the following actions:

- DECthreads creates a thread object to describe and control the thread. The thread object includes a **thread environment block** (TEB) that programs can use, with care. (See the `<sys/types.h>` header file on DIGITAL UNIX, or the `pthread.h` header file on other DECthreads platforms.)
- The *thread* argument receives an identifier for the new thread.
- An executable thread is created with attributes specified by the *attr* argument (or with default attributes if NULL is specified).

Thread Creation

DECthreads creates a thread in the *ready* state and prepares the thread to begin executing its start routine, the function passed to `pthread_create( )` as the *start_routine* argument. Depending on the presence of other threads and their scheduling and priority attributes, the new thread might start executing immediately. The new thread can also preempt its creator, depending on the two threads' respective scheduling and priority attributes. The caller of `pthread_create( )` can synchronize with the new thread using the `pthread_join( )` routine or using any mutually agreed upon mutexes or condition variables.

For the duration of the new thread's existence, DECthreads maintains and manages the thread object and other thread state overhead. A thread *exists* until it is both *terminated* and *detached*. A thread is detached when created if the detachstate attribute of its thread object is set to `PTHREAD_CREATE_DETACHED`. It is also detached after any thread returns successfully from calling `pthread_detach( )` or `pthread_join( )` for the thread. Termination is explained in the next section (see Thread Termination).

DECthreads assigns each new thread a thread identifier, which DECthreads writes into the address specified as the `pthread_create( )` routine's *thread* argument. DECthreads writes the new thread's thread identifier *before* the new thread executes.

By default, the new thread's scheduling policy and priority are inherited from the creating thread—that is, by default, the `pthread_create( )` routine ignores the scheduling policy and priority set in the specified thread attributes object. Thus, to create a thread that is subject to the scheduling policy and priority set in the specified thread attributes object, before calling `pthread_create( )` your program must use the `pthread_attr_setinheritsched( )` routine to set the inherit thread attributes object's scheduling attribute to `PTHREAD_EXPLICIT_SCHED`.

On DIGITAL UNIX, the signal state of the new thread is initialized as follows:

1. The signal mask is inherited from the creating thread.
2. The set of signals pending for the new thread is empty.

If `pthread_create( )` fails, no new thread is created, and the contents of the location referenced by *thread* are undefined.

Thread Termination

A thread terminates when one of the following events occurs:

- The thread returns from its start routine.
- The thread calls the `pthread_exit( )` routine.
- The thread is canceled.

When a thread terminates, DECthreads performs these actions:

1. DECthreads writes a return value (if one is available) into the terminated thread's thread object, as follows:
   - If the thread has been canceled, DECthreads writes the value `PTHREAD_CANCELED` into the thread's thread object.
   - If the thread terminated by returning from its start routine, DECthreads copies the return value from the start routine (if one is available) into the thread's thread object. Alternatively, if the thread explictly called `pthread_exit( )`, DECthreads stores the value received in the *value_ptr* argument (from `pthread_exit( )`) into the thread's thread object.

   Another thread can obtain this return value by joining with the terminated thread (using `pthread_join( )`). See Section 2.3.5 for a description of joining with a thread.

Note

If the thread terminated by returning from its start routine normally and the start routine does not provide a return value, the results obtained by joining with that thread are unpredictable.

---

2. If the termination results from a cancelation request or a call to `pthread_exit( )`, DECthreads calls, in turn, each cleanup handler that this thread declared (using `pthread_cleanup_push( )`) and that is not yet removed (using `pthread_cleanup_pop( )`). (DECthreads also transfers control to any appropriate `CATCH`, `CATCH_ALL`, or `FINALLY` blocks , as described in Chapter 5 .)

   DECthreads calls the terminated thread's most recently pushed cleanup handler first. See Section 2.3.3.1 for more information about cleanup handlers.

   For C++ programmers: At normal exit from a thread, your program will call the appropriate destructor functions, just as if an exception had been raised.

3. To exit the terminated thread due to a call to `pthread_exit( )`, DECthreads raises the `pthread_exit_e` exception. To exit the terminated thread due to cancelation, DECthreads raises the `pthread_cancel_e` exception.

   Your program can use the DECthreads exception package to operate on the generated exception. (In particular, note that the practice of using `CATCH` handlers in place of `pthread_cleanup_push( )` is not portable.) Chapter 5 describes the DECthreads exception package.

4. For each of the terminated thread's thread-specific data keys that has a non-NULL value:

   - DECthreads sets the thread's value for the corresponding key to NULL.
   - In turn, DECthreads calls each thread-specific data destructor function in this multithreaded process's list of destructors.

   DECthreads repeats this step until all thread-specific data values in the thread are NULL, or for up to a number of iterations equal to `PTHREAD_DESTRUCTOR_ITERATIONS`. This destroys all thread-specific data associated with the terminated thread. See Section 2.5 for more information about thread-specific data.

5. DECthreads awakens the thread (if there is one) that is currently waiting to join with the terminated thread. That is, DECthreads awakens the thread that is waiting in a call to `pthread_join( )`.

6. If the thread is already detached, DECthreads destroys its thread object. Otherwise, the thread continues to exist until detached or joined with. Section 2.3.4 describes detaching and destroying a thread.

Return Values

If an error condition occurs, no thread is created, the contents of *thread* are undefined, and this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to create another thread, or the system-imposed limit on the total number of threads under execution by a single user is exceeded.
[EINVAL]	The value specified by *attr* is invalid.
[ENOMEM]	Insufficient memory exists to create a thread.
[EPERM]	The caller does not have the appropriate permission to create a thread with the specified attributes.

Associated Routines

```
pthread_atfork( )
pthread_attr_destroy( )
pthread_attr_init( )
pthread_attr_setdetachstate( )
pthread_attr_setinheritsched( )
pthread_attr_setschedparam( )
pthread_attr_setschedpolicy( )
pthread_attr_setstacksize( )
pthread_cancel( )
pthread_detach( )
pthread_exit( )
pthread_join( )
```

pthread_delay_np

Delays a thread's execution.

Syntax

```
pthread_delay_np(
                interval );
```

Argument	Data Type	Access
interval	struct timespec	read

C Binding

```
#include <pthread.h>

int
pthread_delay_np (
            const struct timespec    *interval);
```

Arguments

interval
Number of seconds and nanoseconds to delay execution. The value specified for each must be greater than or equal to zero.

Description

This routine causes a thread to delay execution for a specific interval of time. This interval ends at the current time plus the specified interval. The routine will not return before the end of the interval is reached, but may return an arbitrary amount of time after the end of the interval is reached. This can be due to system load, thread priorities, and system timer granularity.

Specifying an interval of zero (0) seconds and zero (0) nanoseconds is allowed and can be used to force the thread to give up the processor or to deliver a pending cancelation request.

The `timespec` structure contains the following two fields:

- `tv_sec` is an integral number of seconds.
- `tv_nsec` is an integral number of nanoseconds.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *interval* is invalid.

pthread_detach

Marks a thread object for deletion.

Syntax

```
pthread_detach(
          thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

C Binding

```
#include <pthread.h>

int
pthread_detach (
          pthread_t     thread);
```

Arguments

thread
Thread object being marked for deletion.

Description

This routine marks the specified thread object to indicate that storage for the corresponding thread can be reclaimed when the thread terminates. This includes storage for the *thread* argument's return value, as well as the thread object. If *thread* has not terminated when this routine is called, this routine does not cause it to terminate.

When a thread object is no longer referenced, call this routine.

The results of this routine are unpredictable if the value of *thread* refers to a thread object that does not exist.

A thread can be created already detached by setting its thread object's detachstate attribute.

The `pthread_join( )` routine also detaches the target thread after `pthread_join( )` returns successfully.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *thread* does not refer to a joinable thread.
[ESRCH]	The value specified by *thread* cannot be found.

Associated Routines

```
pthread_cancel( )
pthread_create( )
pthread_exit( )
pthread_join( )
```

pthread_equal

Compares one thread identifier to another thread identifier.

Syntax

```
pthread_equal(
          t1,
          t2);
```

Argument	Data Type	Access
t1	opaque pthread_t	read
t2	opaque pthread_t	read

C Binding

```
#include <pthread.h>

int
pthread_equal (
          pthread_t    t1,
          pthread_t    t2);
```

Arguments

t1
The first thread identifier to be compared.

t2
The second thread identifier to be compared.

Description

This routine compares one thread identifier to another thread identifier.

If either *t1* or *t2* are not valid thread identifiers, this routine's behavior is undefined.

Return Values

Possible return values are as follows:

Return	Description
0	Values of *t1* and *t2* do not designate the same object.
Non-zero	Values of *t1* and *t2* designate the same object.

pthread_exc_get_status_np

(Macro) Obtains a system-defined error status from a DECthreads status exception object.

Syntax

```
pthread_exc_get_status_np(
                          exception,
                          code);
```

Argument	Data Type	Access
exception	EXCEPTION	read
code	unsigned long	write

C Binding

```
#include <pthread_exception.h>

int
pthread_exc_get_status_np (
           EXCEPTION      *exception,
           unsigned long   *code);
```

Arguments

exception
DECthreads status exception object whose status code is obtained.

code
Receives the system-specific status code associated with the specified DECthreads status exception object.

Description

This routine obtains and returns the system-specific status value from the DECthreads status exception object specified in the *exception* argument. This value must have already been associated with the exception object using the `pthread_exc_set_status_np( )` routine.

In a program that uses DECthreads status exceptions, use this routine within a `CATCH`, `CATCH_ALL`, or `FINALLY` code block to obtain the status code value associated with a caught exception. Note that any exception objects set to the same status value are considered equivalent by DECthreads.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. If the routine's exception object argument is a DECthreads status exception, it sets the *code* argument and returns zero (0). Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The *exception* argument is not a valid DECthreads status exception object.

Associated Routines

`pthread_exc_set_status_np( )`

pthread_exc_matches_np

(Macro) Determines whether two DECthreads exception objects are identical.

Syntax

```
pthread_exc_matches_np(
                       exception1,
                       exception2);
```

Argument	Data Type	Access
exception1	EXCEPTION	read
exception2	EXCEPTION	read

C Binding

```
#include <pthread_exception.h>

int
pthread_exc_matches_np (
            EXCEPTION    *exception1,
            EXCEPTION    *exception2);
```

Arguments

exception1
DECthreads exception object.

exception2
DECthreads exception object.

Description

This routine compares two DECthreads exception objects, taking into consideration whether each is an address exception or status exception.

This routine returns either the C language value `TRUE` or the C language value `FALSE`, indicating whether the two DECthreads exception objects specified in the arguments *exception1* and *exception2* are identical.

Return Values

The C language value TRUE if the exception objects are identical, or the C language value FALSE if not.

Associated Routines

```
pthread_exc_get_status_np( )
pthread_exc_report_np( )
pthread_exc_set_status_np( )
```

pthread_exc_report_np

Produces a message that reports what a specified DECthreads status exception object represents.

Syntax

```
pthread_exc_report_np(
                    exception);
```

Argument	Data Type	Access
exception	EXCEPTION	read

C Binding

```
#include <pthread_exception.h>

void
pthread_exc_report_np (
            EXCEPTION     *exception);
```

Arguments

exception
DECthreads exception object that has been set with a status value.

Description

This routine produces a text message on the `stderr` device (Digital UNIX and Windows NT systems) or SYS$ERROR device (OpenVMS systems) that describes the exception whose exception object is specified in the *exception* argument.

In a program that uses DECthreads status exceptions, use this routine within a `CATCH`, `CATCH_ALL`, or `FINALLY` code block to obtain the status code value associated with a caught exception. Note that any exception objects set to the same status value are considered equivalent by DECthreads.

Return Values

None

Associated Routines

```
pthread_exc_get_status_np( )
pthread_exc_set_status_np( )
```

pthread_exc_set_status_np

(Macro) Imports a system-defined error status into a DECthreads address exception object.

Syntax

```
pthread_exc_set_status_np(
                          exception,
                          code);
```

Argument	Data Type	Access
exception	EXCEPTION	write
code	unsigned long	read

C Binding

```
#include <pthread_exception.h>

void
pthread_exc_set_status_np (
            EXCEPTION      *exception,
            unsigned long   code);
```

Arguments

exception
DECthreads address exception object into which the specified status code is imported.

code
System-specific status code to be imported.

Description

This routine associates a system-specific status value with the specified DECthreads address exception object. This transforms the address exception object into a DECthreads status exception object.

The *exception* argument must already have been initialized with the DECthreads exception package's `EXCEPTION_INIT` macro.

Use this routine to associate any system-specific status value with the specified DECthreads address exception object. Note that any exception objects set to the same status value are considered equivalent by DECthreads.

Return Values

None

Associated Routines

pthread_exc_get_status_np( )

pthread_exit

Terminates the calling thread.

Syntax

```
pthread_exit(
        value_ptr );
```

Argument	Data Type	Access
value_ptr	void *	read

C Binding

```
#include <pthread.h>

void
pthread_exit (
        void     *value_ptr);
```

Arguments

value_ptr
Value copied and returned to the caller of `pthread_join( )`. Note that `void *` is used as a universal datatype, not as a pointer. DECthreads treats the *value_ptr* as a value and stores it to be returned by `pthread_join( )`.

Description

This routine terminates the calling thread and makes a status value (*value_ptr*) available to any thread that calls `pthread_join( )` and specifies the terminating thread.

Any cleanup handlers that have been pushed and not yet popped from the stack, are popped in the reverse order that they were pushed and then executed. After all cleanup handlers have been executed, appropriate destructor functions shall be called in an unspecified order if the thread has any thread-specific data. Thread termination does *not* release any application-visible process resources, including, but not limited to mutexes and file descriptors, nor does it perform any process-level cleanup actions, including, but not limited to calling any `atexit( )` routine that may exist.

An implicit call to `pthread_exit( )` is issued when a thread returns from the start routine that was used to create it. DECthreads writes the function's return value as the return value in the thread's thread object. The process exits when the last running thread calls `pthread_exit( )`.

After a thread has terminated, the result of access to local (that is, explicitly or implicitly declared `auto`) variables of the thread is undefined. So, references to local variables of the existing thread should *not* be used for the *value_ptr* argument of the `pthread_exit( )` routine.

Return Values

None

Associated Routines

```
pthread_cancel( )
pthread_create( )
pthread_detach( )
pthread_join( )
```

pthread_getconcurrency

Obtains the value of the concurrency level global variable for this process.

Syntax

```
pthread_getconcurrency(
                    );
```

C Binding

```
#include <pthread.h>

int
pthread_getconcurrency (
                void);
```

Description

This routine obtains and returns the value of the “concurrency level” global setting for the calling thread’s process. Because DECthreads automatically manages the concurrency of all threads in a multithreaded process, DECthreads ignores this concurrency level value.

The concurrency level value has no effect on the behavior of a multithreaded program that uses DECthreads. This routine is provided for Single UNIX Specification, Version 2, source code compatibility and has no other effect when called.

The initial concurrency level is zero (0), indicating that DECthreads controls the concurrency level.

The concurrency level can be set using the `pthread_setconcurrency( )` routine.

Return Values

This routine always returns the value of this process’s concurrency level global variable. If this process has never called the `pthread_setconcurrency( )` routine, this routine returns zero (3).

Associated Routines

`pthread_setconcurrency( )`

pthread_getname_np

Obtains the object name from the thread object for an existing thread.

Syntax

```
pthread_getname_np(
                    thread,
                    name,
                    len );
```

Argument	Data Type	Access
thread	opaque pthread_thread_t	read
name	char	write
len	opaque size_t	read

C Binding

```
#include <pthread.h>

int
pthread_getname_np (
             pthread_thread_t     thread,
             char     *name,
             size_t     len);
```

Arguments

thread
Thread object whose object name is to be obtained.

name
Location to store the obtained object name.

len
Length in bytes of buffer at the location specified by *name*.

Description

This routine copies the object name from the thread object specified by the *thread* argument to the buffer at the location specified by the *name* argument. Before calling this routine, your program must allocate the buffer indicated by *name*.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

If the specified thread object has not been previously set with an object name, this routine copies a C language null string into the buffer at location *name*.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ESRCH]	The thread specified by *thread* does not exist.

Associated Routines

`pthread_setname_np( )`

pthread_getschedparam

Obtains the current scheduling policy and scheduling parameters of a thread.

Syntax

```
pthread_getschedparam(
                      thread,
                      policy,
                      param );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
policy	integer	write
param	struct sched_param	write

C Binding

```
#include <pthread.h>

int
pthread_getschedparam (
            pthread_t    thread,
            int    *policy,
            struct sched_param    *param);
```

Arguments

thread
Thread whose scheduling policy and parameters are obtained.

policy
Receives the value of the scheduling policy for the thread specified in *thread*. Refer to the description of the `pthread_setschedparam( )` routine for valid parameters and their values.

param
Receives the value of the scheduling parameters for the thread specified in *thread*. Refer to the description of the `pthread_setschedparam( )` routine for valid values.

Description

This routine obtains both the current scheduling policy and associated scheduling parameters of the thread specified by the *thread* argument.

The priority value returned in the *param* structure is the value specified in the *attr* argument passed to `pthread_create( )` or by the most recent call to `pthread_setschedparam( )` that affects the target thread.

This routine differs from `pthread_attr_getschedpolicy( )` and `pthread_attr_getschedparam( )`, in that those routines get the scheduling policy and parameter attributes that are used to establish the priority and scheduling policy of a new thread when it is created. This routine, however, obtains the scheduling policy and parameters of an existing thread.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ESRCH]	The value specified by *thread* does not refer to an existing thread.

Associated Routines

```
pthread_attr_getschedparam( )
pthread_attr_getschedpolicy( )
pthread_create( )
pthread_self( )
pthread_setschedparam( )
```

pthread_getsequence_np

Obtains the unique identifier for the specified thread.

Syntax

```
pthread_getsequence_np(
                    thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

C Binding

```
#include <pthread.h>

unsigned long
pthread_getsequence_np (
            pthread_t     thread);
```

Arguments

thread
Thread whose sequence number is to be obtained.

Description

This routine obtains and returns the DECthreads thread sequence number for the thread identified by the thread object specified in the *thread* argument.

The thread sequence number provides a unique identifier for each existing thread. A thread's thread sequence number is never reused while the thread exists, but can be reused after the thread terminates. The debugger interfaces use this sequence number to identify each thread in commands and in display output.

The result of calling this routine is undefined if the *thread* argument does not specify a valid thread object.

Return Values

No errors are returned. This routine returns the DECthreads thread sequence number for the thread identified by the thread object specified in the *thread* argument. The result of calling this routine is undefined if the *thread* argument does not specify a valid thread.

Associated Routines

```
pthread_create( )
pthread_self( )
```

pthread_getspecific

Obtains the thread-specific data associated with the specified key.

Syntax

pthread_getspecific(
    key);

Argument	Data Type	Access
key	opaque pthread_key_t	read

C Binding

```
#include <pthread.h>

void
*pthread_getspecific (
            pthread_key_t     key);
```

Arguments

key
The context *key* identifies the thread-specific data to be obtained. Obtain this key by calling the `pthread_key_create( )` routine.

Description

This routine obtains the thread-specific data associated with the specified *key* for the current thread. This routine returns the value currently bound to the specified *key* on behalf of the calling thread.

This routine may be called from a thread-specific data destructor function.

Return Values

No errors are returned. This routine returns the thread-specific data value associated with the specified *key* argument. If no thread-specific data value is associated with *key*, or if *key* is not defined, then this routine returns a NULL value.

Associated Routines

```
pthread_key_create( )
pthread_setspecific( )
```

pthread_get_expiration_np

Obtains a value representing a desired expiration time.

Syntax

```
pthread_get_expiration_np(
                          delta,
                          abstime );
```

Argument	Data Type	Access
delta	struct timespec	read
abstime	struct timespec	write

C Binding

```
#include <pthread.h>

int
pthread_get_expiration_np (
            const struct timespec   *delta,
            struct timespec   *abstime);
```

Arguments

delta

Number of seconds and nanoseconds to add to the current system time. (The result is the time in the future.) This result will be placed in *abstime*.

abstime

Value representing the absolute expiration time. The absolute expiration time is obtained by adding *delta* to the current system time. The resulting *abstime* is in Universal Coordinated Time (UTC). This value should be passed to the `pthread_cond_timedwait( )` routine.

Description

This routine adds a specified interval to the current absolute system time and returns a new absolute time. This new absolute time is used as the expiration time in a call to `pthread_cond_timedwait( )`.

The `timespec` structure contains the following two fields:

- `tv_sec` is an integral number of seconds.
- `tv_nsec` is an integral number of nanoseconds.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *delta* is invalid.

Associated Routines

`pthread_cond_timedwait( )`

pthread_join

pthread_join32( ), pthread_join64( )

The pthread_join32( ) *and* pthread_join64( ) *forms are only valid in 64-pointer environments for OpenVMS Alpha.* For information regarding 32- and 64-bit pointers, see Appendix B. Ensure that your compiler provides 64-bit support prior to using pthread_join64( ).

Causes the calling thread to wait for the termination of a specified thread.

Syntax

```
pthread_join(
        thread,
        value_ptr );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
value_ptr	void *	write

C Binding

```
#include <pthread.h>

int
pthread_join (
        pthread_t    thread,
        void     **value_ptr);
```

Arguments

thread
Thread whose termination is awaited by the calling routine.

value_ptr
Return value of the terminating thread (when that thread either calls pthread_exit( ) or returns.)

Description

This routine suspends execution of the calling thread until the specified target thread *thread* terminates.

On return from a successful pthread_join( ) call with a non-NULL *value_ptr* argument, the value passed to pthread_exit( ) is returned in the location referenced by *value_ptr*, and the terminating thread is detached.

If more than one thread attempts to join with the same thread, the results are unpredictable.

A call to pthread_join( ) returns after the target thread terminates. The pthread_join( ) routine is a deferred cancelation point: the target thread will not be detached if the thread blocked in pthread_join( ) is canceled.

If a thread calls this routine and specifies its own pthread_t, a deadlock can result.

The pthread_join( ) (or pthread_detach( )) routine should eventually be called for every thread that is created with the detachstate attribute of its thread object set to PTHREAD_CREATE_JOINABLE, so that storage associated with the thread can be reclaimed.

For OpenVMS Alpha systems only, you can call pthread_join32( ) or pthread_join64( ) instead of pthread_join( ). The pthread_join32( ) form returns a 32-bit void * value in the address to which *value_ptr* points. The pthread_join64( ) form returns a 64-bit void * value. You can call either, or you can call pthread_join( ). The pthread_join( ) routine is defined to pthread_join64( ) if you compile using /pointer_size=long. If you do not specify /pointer_size, or if you specify /pointer_size=short, then pthread_join( ) is defined to be pthread_join32( ). Note that if you call pthread_join32( ) and the thread with which you join returns a 64-bit value, the high 32 bits of which are not 0 (zero), DECthreads discards those high bits with no warning.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *thread* does not refer to a joinable thread.
[ESRCH]	The value specified by *thread* does not refer to an existing thread ID.
[EDEADLK]	A deadlock was detected, or *thread* specifies the calling thread.

Associated Routines

```
pthread_cancel( )
pthread_create( )
pthread_detach( )
pthread_exit( )
```

pthread_key_create

Generates a unique thread-specific data key.

Syntax

```
pthread_key_create(
                key,
                destructor );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write
destructor	procedure	read

C Binding

```
#include <pthread.h>

int
pthread_key_create (
            pthread_key_t    *key,
            void    (*destructor)(void *));
```

Arguments

key
The new thread-specific data key.

destructor
Procedure called to destroy a thread-specific data value associated with the created key when the thread terminates. Note that the argument to the destructor for the user-specified routine is the non-NULL value associated with a key.

Description

This routine generates a unique, thread-specific data key that is visible to all threads in the process. The variable *key* provided by this routine is an opaque object used to locate thread-specific data. Although the same key value can be used by different threads, the values bound to the key by `pthread_setspecific()` are maintained on a per-thread basis and persist for the life of the calling thread.

DECthreads imposes a maximum number of thread-specific data keys, equal to the symbolic constant `PTHREAD_KEYS_MAX`.

Thread-specific data allows client software to associate "static" information with the current thread. For example, where a routine declares a variable `static` in a single-threaded program, a multithreaded version of the program might create a thread-specific data key to store the same variable.

This routine generates and returns a new key value. The key reserves a cell within each thread. Each call to this routine creates a new cell that is unique within an application invocation. Keys must be generated from initialization code that is guaranteed to be called only once within each process. (See the `pthread_once( )` description for more information.)

When a thread terminates, its thread-specific data is automatically destroyed; however, the key remains unless destroyed by a call to `pthread_key_delete( )`. An optional destructor function can be associated with each key. At thread exit, if a key has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the destructor function is called with the current associated value as its sole argument. *The order in which thread-specific data destructors are called at thread termination is undefined.*

Before each destructor is called, the thread's value for the corresponding key is set to NULL. After the destructors have been called for all non-NULL values with associated destructors, if there are still some non-NULL values with associated destructors, then this sequence of actions is repeated. If there are still non-NULL values for any key with a destructor after four repetitions of this sequence, DECthreads terminates the thread. At this point, any key values that represent allocated heap will be lost. Note that this occurs only when a destructor performs some action that creates a new value for some key. Your program's destructor code should attempt to avoid this sort of circularity.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacked the necessary resources to create another thread-specific data key, or the limit on the total number of keys per process (`PTHREAD_KEYS_MAX`) has been exceeded.
[ENOMEM]	Insufficient memory exists to create the key.

Associated Routines

```
pthread_getspecific( )
pthread_key_delete( )
pthread_once( )
pthread_setspecific( )
```

pthread_key_delete

Deletes a thread-specific data key.

Syntax

pthread_key_delete(
        key);

Argument	Data Type	Access
key	opaque pthread_key_t	write

C Binding

#include <pthread.h>

int
pthread_key_delete (
        pthread_key_t   *key*);

Arguments

key
Context key to be deleted.

Description

This routine deletes the thread-specific data key specified by the *key* argument, which must have been previously returned by `pthread_key_create( )`.

The thread-specific data values associated with *key* need not be NULL at the time this routine is called. The application must free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads. This cleanup can be done either before or after this routine is called.

Do not attempt to use the key after calling this routine; this results in unpredictable behavior.

No destructor functions are invoked by this routine. Any destructor functions that may have been associated with *key* shall no longer be called upon thread exit. `pthread_key_delete( )` can be called from within destructor functions.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The key value is an invalid argument.

Associated Routines

```
pthread_exit( )
pthread_getspecific( )
pthread_key_create( )
```

pthread_key_getname_np

Obtains the object name from a thread-specific data key object.

Syntax

```
pthread_key_getname_np(
                    key,
                    name,
                    len );
```

Argument	Data Type	Access
key	opaque pthread_key_t	read
name	char	write
len	opaque size_t	read

C Binding

```
#include <pthread.h>

int
pthread_key_getname_np (
            pthread_key_t    *key,
            char    *name,
            size_t    len);
```

Arguments

key
Address of the thread-specific data key object whose object name is to be obtained.

name
Location to store the obtained object name.

len
Length in bytes of buffer at the location specified by *name*.

Description

This routine copies the object name from the thread-specific data key object specified by the *key* argument to the buffer at the location specified by the *name* argument. Before calling this routine, your program must allocate the buffer indicated by *name*.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

If the specified thread-specific data key object has not been previously set with an object name, this routine copies a C language null string into the buffer at location *name*.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *key* is invalid.

Associated Routines

`pthread_key_setname_np( )`

pthread_key_setname_np

Changes the object name in a thread-specific data key object.

Syntax

```
pthread_key_setname_np(
                        key,
                        name,
                        mbz );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write
name	char	read
mbz	void	read

C Binding

```
#include <pthread.h>

int
pthread_key_setname_np (
            pthread_key_t    *cond,
            const char    *name,
            void    *mbz);
```

Arguments

key
Address of the thread-specific data key object whose object name is to be changed.

name
Object name value to copy into the condition variable object.

mbz
(Must be zero) Argument for use by DECthreads.

Description

This routine changes the object name in the thread-specific data key object specified by the *key* argument to the value specified by the *name* argument. To set a new thread-specific data key object's object name, call this routine immediately after initializing the key object.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *key* is invalid, or the length in characters of *name* exceeds 31.
[ENOMEM]	Insufficient memory exists to create a copy of the object name string.

Associated Routines

`pthread_key_getname_np( )`

pthread_kill

Delivers a signal to a specified target thread.

This routine is for DIGITAL UNIX systems only.

Syntax

```
pthread_kill(
        thread,
        sig );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
sig	integer	read

C Binding

```
#include <pthread.h>

int
pthread_kill (
        pthread_t    thread,
        int    sig);
```

Arguments

thread
Thread to receive a signal request.

sig
A signal request. If *sig* is zero, error checking is performed, but no signal is sent.

Description

This routine sends a signal to the specified target thread *thread*. Any signal defined to stop, continue, or terminate will stop or terminate the process, even though it can be handled by the target thread. For example, SIGTERM terminates *all* threads in the process, even though it can be handled by the target thread.

Specifying a *sig* argument of 0 (zero) causes this routine to validate the *thread* argument but not to deliver any signal.

The name of the “kill” routine is sometimes misleading, because many signals do not terminate a thread.

The various signals are as follows:

SIGHUP	SIGPIPE	SIGTTIN
SIGINT	SIGALRM	SIGTTOU
SIGQUIT	SIGTERM	SIGIO
SIGTRAP	SIGUSR1	SIGXCPU
SIGABRT	SIGSYS	SIGXFSZ
SIGEMT	SIGURG	SIGVTALRM
SIGFPE	SIGSTOP	SIGPROF
SIGKILL	SIGTSTP	SIGINFO
SIGBUS	SIGCONT	SIGUSR1
SIGSEGV	SIGCHLD	SIGUSR2

If this routine does not execute successfully, no signal is sent.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value of *sig* is invalid or an unsupported signal value.
[ESRCH]	The value of *thread* does not specify an existing thread.

pthread_lock_global_np

Locks the DECthreads global mutex if the global mutex is unlocked. If the global mutex is locked by another thread, causes the thread to wait for the global mutex to become available.

Syntax

```
pthread_lock_global_np( );
```

C Binding

#include <pthread.h>

int
pthread_lock_global_np (void);

Arguments

None

Description

This routine locks the DECthreads global mutex. If the global mutex is currently held by another thread when a thread calls this routine, the thread waits for the global mutex to become available.

The thread that has locked the global mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the global mutex in the locked state and with the current thread as the global mutex's current owner.

Use the DECthreads global mutex when calling a library package that is not designed to run in a multithreaded environment. Unless the documentation for a library function specifically states that it is thread safe, assume that it is not compatible; in other words, assume it is nonreentrant.

The global mutex is one lock. Any code that calls any function that is not known to be reentrant uses the same lock. This prevents problems resulting from dependencies among threads that call library functions and those functions' calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. The locking thread must call `pthread_unlock_global_np( )` as many times as it called this routine, to allow another thread to lock the global mutex.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.

Associated Routines

`pthread_unlock_global_np( )`

pthread_mutexattr_destroy

Destroys the specified mutex attributes object.

Syntax

```
pthread_mutexattr_destroy(
                          attr );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write

C Binding

```
#include <pthread.h>

int
pthread_mutexattr_destroy (
            pthread_mutexattr_t     *attr);
```

Arguments

attr
Mutex attributes object to be destroyed.

Description

This routine destroys a mutex attributes object—that is, the object becomes uninitialized. Call this routine when your program no longer needs the specified mutex attributes object.

After this routine is called, DECthreads may reclaim the storage used by the mutex attributes object. Mutexes that were created using this attributes object are not affected by the destruction of the mutex attributes object.

The results of calling this routine are unpredictable, if the attributes object specified in the *attr* argument does not exist.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

`pthread_mutexattr_init( )`

pthread_mutexattr_gettype

Obtains the mutex type attribute in the specified mutex attribute object.

Syntax

```
pthread_mutexattr_gettype(
                          attr,
                          type );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read
type	integer	write

C Binding

```
#include <pthread.h>

int
pthread_mutexattr_gettype (
            const pthread_mutexattr_t    *attr,
            int    *type);
```

Arguments

attr
Mutex attributes object whose mutex type attribute is obtained.

type
Receives the value of the mutex type attribute. The *type* argument specifies the type of mutex that can be created. Valid values are:

`PTHREAD_MUTEX_NORMAL`
`PTHREAD_MUTEX_DEFAULT` (default)
`PTHREAD_MUTEX_RECURSIVE`
`PTHREAD_MUTEX_ERRORCHECK`

Description

This routine obtains the value of the mutex type attribute in the mutex attributes object specified by the *attr* argument and stores it in the location specified by the *type* argument. See the `pthread_mutexattr_settype( )` description for information about mutex types.

Return Values

On successful completion, this routine returns the mutex type in the location specified by the *type* argument.

If an error condition occurs, this routine returns an integer value indicating the type of the error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_mutexattr_init( )
pthread_mutexattr_settype( )
pthread_mutex_init( )
```

pthread_mutexattr_gettype_np

Obtains the mutex type attribute in the specified mutex attribute object.

Syntax

```
pthread_mutexattr_gettype_np(
                              attr,
                              type );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read
type	integer	write

C Binding

```
#include <pthread.h>

int
pthread_mutexattr_gettype_np (
                const pthread_mutexattr_t    *attr,
                int    *type);
```

Arguments

attr
Mutex attributes object whose mutex type attribute is obtained.

type
Receives the value of the mutex type attribute. The *type* argument specifies the type of mutex that can be created. Valid values are:

`PTHREAD_MUTEX_NORMAL`
`PTHREAD_MUTEX_DEFAULT` (default)
`PTHREAD_MUTEX_RECURSIVE`
`PTHREAD_MUTEX_ERRORCHECK`

Description

This routine obtains the value of the mutex type attribute in the mutex attributes object specified by the *attr* argument and stores it in the location specified by the *type* argument. See the `pthread_mutexattr_settype( )` description for information about mutex types.

Note

This routine has been superseded by the `pthread_mutexattr_gettype( )` routine, as specified by the Single UNIX Specification, Version 2.

Return Values

On successful completion, this routine returns the mutex type attribute in the location specified by the *type* argument.

If an error condition occurs, this routine returns an integer value indicating the type of the error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* is invalid.

Associated Routines

```
pthread_mutexattr_init( )
pthread_mutexattr_settype_np( )
pthread_mutex_init( )
```

pthread_mutexattr_init

Initializes a mutex attributes object.

Syntax

pthread_mutexattr_init(
    attr);

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write

C Binding

```
#include <pthread.h>

int
pthread_mutexattr_init (
            pthread_mutexattr_t    *attr);
```

Arguments

attr
Address of the mutex attributes object to be initialized.

Description

This routine initializes the mutex attributes object specified by the *attr* argument with a set of default values. A mutex attributes object is used to specify the attributes of one or more mutexes when they are created. The attributes object created by this routine is used only in calls to the `pthread_mutex_init( )` routine.

When a mutex attributes object is used to create a mutex, the values of the individual attributes determine the characteristics of the new mutex. Thus, attributes objects act as additional arguments to mutex creation. Changing individual attributes in an attributes object does not affect any mutexes that were previously created using that attributes object.

You can use the same mutex attributes object in successive calls to `pthread_mutex_init( )`, from any thread. If multiple threads can change attributes in a shared mutex attributes object, your program must use a mutex to protect the integrity of the attributes object's contents.

Results are undefined if this routine is called and the *attr* argument specifies a mutex attributes object that is already initialized.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ENOMEM]	Insufficient memory to create the mutex attributes object.

Associated Routines

```
pthread_mutexattr_destroy( )
pthread_mutexattr_gettype( )
pthread_mutexattr_settype( )
pthread_mutex_init( )
```

pthread_mutexattr_settype

Specifies the mutex type attribute that is used when a mutex is created.

Syntax

```
pthread_mutexattr_settype(
                         attr,
                         type );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write
type	integer	read

C Binding

```
#include <pthread.h>

int
pthread_mutexattr_settype (
          pthread_mutexattr_t    *attr,
          int    type);
```

Arguments

attr
Mutex attributes object whose mutex type attribute is to be modified.

type
New value for the mutex type attribute. The *type* argument specifies the type of mutex that will be created. Valid values are:

PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_DEFAULT (default)
PTHREAD_MUTEX_RECURSIVE
PTHREAD_MUTEX_ERRORCHECK

Description

This routine sets the mutex type attribute that is used to determine which type of mutex is created based on a subsequent call to `pthread_mutex_init()`. See Section 2.4.1 for information on the types of mutexes.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* or *type* is invalid.
[ESRCH]	The value specified by *attr* does not refer to an existing mutex attributes object.

Associated Routines

```
pthread_mutexattr_init()
pthread_mutexattr_gettype()
pthread_mutex_init()
```

pthread_mutexattr_settype_np

Specifies the mutex type attribute that is used when a mutex is created.

Syntax

```
pthread_mutexattr_settype_np(
                              attr,
                              type );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write
type	integer	read

C Binding

```
#include <pthread.h>

int
pthread_mutexattr_settype_np (
            pthread_mutexattr_t    *attr,
            int    type);
```

Arguments

attr
Mutex attributes object whose mutex type attribute is to be modified.

type
New value for the mutex type attribute. The *type* argument specifies the type of mutex that will be created. Valid values are:

`PTHREAD_MUTEX_NORMAL`
`PTHREAD_MUTEX_DEFAULT` (default)
`PTHREAD_MUTEX_RECURSIVE`
`PTHREAD_MUTEX_ERRORCHECK`

Description

This routine sets the mutex type attribute that is used to determine which type of mutex is created based on a subsequent call to `pthread_mutex_init( )`. See Section 2.4.1 for information on the types of mutexes.

Note

This routine has been superseded by the `pthread_mutexattr_settype( )` routine, as specified by the Single UNIX Specification, Version 2.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *attr* or *type* is invalid.
[ESRCH]	The value specified by *attr* does not refer to an existing mutex attributes object.

Associated Routines

```
pthread_mutexattr_init( )
pthread_mutexattr_gettype_np( )
pthread_mutex_init( )
```

pthread_mutex_destroy

Destroys a mutex.

Syntax

```
pthread_mutex_destroy(
                    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write

C Binding

```
#include <pthread.h>

int
pthread_mutex_destroy (
            pthread_mutex_t     *mutex);
```

Arguments

mutex
The mutex to be destroyed.

Description

This routine destroys the specified mutex by uninitializing it, and should be called when a mutex object is no longer referenced. After this routine is called, DECthreads may reclaim internal storage used by the specified mutex.

It is safe to destroy an initialized mutex that is unlocked. However, it is illegal to destroy a locked mutex.

The results of this routine are unpredictable if the mutex object specified in the *mutex* argument does not currently exist, or is not initialized.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	An attempt was made to destroy the object referenced by *mutex* while it is locked or referenced.
[EINVAL]	The value specified by *mutex* is invalid.

Associated Routines

```
pthread_mutex_init( )
pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
```

pthread_mutex_getname_np

Obtains the object name from a mutex object.

Syntax

```
pthread_mutex_getname_np(
                         mutex,
                         name,
                         len );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read
name	char	write
len	opaque size_t	read

C Binding

```
#include <pthread.h>

int
pthread_mutex_getname_np (
          pthread_mutex_t    *mutex,
          char    *name,
          size_t    len);
```

Arguments

mutex
Address of the mutex object whose object name is to be obtained.

name
Location to store the obtained object name.

len
Length in bytes of buffer at the location specified by *name*.

Description

This routine copies the object name from the mutex object specified by the *mutex* argument to the buffer at the location specified by the *name* argument. Before calling this routine, your program must allocate the buffer indicated by *name*.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

If the specified condition variable object has not been previously set with an object name, this routine copies a C language null string into the buffer at location *name*.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *mutex* is invalid.

Associated Routines

pthread_mutex_setname_np( )

pthread_mutex_init

Initializes a mutex with attributes specified by the *attr* argument.

Syntax

```
pthread_mutex_init(
            mutex,
            attr );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write
attr	opaque pthread_mutexattr_t	read

C Binding

```
#include <pthread.h>

int
pthread_mutex_init (
        pthread_mutex_t    *mutex,
        const pthread_mutexattr_t    *attr);
```

Arguments

mutex
Mutex created.

attr
Mutex attributes object to be used to initialize the characteristics of the created *mutex*.

Description

This routine initializes a mutex with the attributes specified by the mutex attributes object specified in the *attr* argument. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

The mutex is initialized and set to the unlocked state. If *attr* is set to NULL, the default mutex attributes are used. The `pthread_mutexattr_settype( )` routine can be used to specify the type of mutex that is created (normal, recursive, or errorcheck).

See Chapter 2 for more information about mutex usage.

A mutex is a resource of the process, not part of any particular thread. A mutex is neither destroyed nor unlocked automatically when any thread exits. Because mutexes are shared, they may be allocated in heap or static memory, but not on a stack.

Use the `PTHREAD_MUTEX_INITIALIZER` macro to statically initialize a mutex without calling this routine. Statically initialized mutexes need not be destroyed using `pthread_mutex_destroy( )`. Use this macro as follows:

`pthread_mutex_t` *mutex* = `PTHREAD_MUTEX_INITIALIZER`

Only normal mutexes can be statically initialized.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error, the mutex is not initialized, and the contents of *mutex* are undefined. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize the mutex.
[ENOMEM]	Insufficient memory exists to initialize the mutex.
[EBUSY]	The implementation has detected an attempt to reinitialize the *mutex* (a previously initialized, but not yet destroyed mutex).
[EINVAL]	The value specified by *mutex* is invalid.
[EPERM]	The caller does not have privileges to perform this operation.

Associated Routines

```
pthread_mutexattr_init( )
pthread_mutexattr_gettype( )
pthread_mutexattr_settype( )
pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
```

pthread_mutex_lock

Locks an unlocked mutex. If the mutex is already locked, the calling thread blocks until the mutex becomes available.

Syntax

```
pthread_mutex_lock(
                    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

```
#include <pthread.h>

int
pthread_mutex_lock (
            pthread_mutex_t     *mutex);
```

Arguments

mutex
Mutex to be locked.

Description

This routine locks a mutex with behavior that depends upon the type of mutex, as follows:

- If a normal or default mutex is specified, a deadlock can result if the current owner of the mutex calls this routine in an attempt to lock the mutex a second time. (The deadlock is not detected or reported.)
- If a recursive mutex is specified, the current owner of the mutex can relock the same mutex without blocking. The lock count is incremented for each recursive lock within the thread.
- If an errorcheck mutex is specified and the current owner tries to lock the mutex a second time, this routine reports the [EDEADLK] error. If the mutex is locked by another thread, the calling thread waits for the mutex to become available.

Use the pthread_mutexattr_settype( ) routine to set the type of the mutex to normal, default, recursive, or errorcheck. For more information about mutexes, see Chapter 2.

The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the mutex in the locked state and with the calling thread as the mutex's current owner.

A recursive or errorcheck mutex records the identity of the thread that locks it, allowing debuggers to display this information. In most cases, normal and default mutexes do not record the owning thread's identity.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *mutex* is invalid, or The *mutex* was created with the protocol attribute set to PTHREAD_PRIO_PROTECT and the calling thread's priority set higher than the mutex's current priority ceiling.
[EDEADLK]	A deadlock condition is detected.

Associated Routines

```
pthread_mutexattr_settype( )
pthread_mutex_destroy( )
pthread_mutex_init( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
```

pthread_mutex_setname_np

Changes the object name in a mutex object.

Syntax

```
pthread_mutex_setname_np(
                        mutex,
                        name,
                        mbz );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write
name	char	read
mbz	void	read

C Binding

```
#include <pthread.h>

int
pthread_mutex_setname_np (
            pthread_mutex_t    *mutex,
            const char    *name,
            void    *mbz);
```

Arguments

mutex
Address of the mutex object whose object name is to be changed.

name
Object name value to copy into the mutex object.

mbz
(Must be zero) Argument for use by DECthreads.

Description

This routine changes the object name in the mutex object specified by the *mutex* argument to the value specified by the *name* argument. To set a new mutex object's object name, call this routine immediately after initializing the mutex object.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *mutex* is invalid, or the length in characters of *name* exceeds 31.
[ENOMEM]	Insufficient memory exists to create a copy of the object name string.

Associated Routines

`pthread_mutex_getname_np( )`

pthread_mutex_trylock

Attempts to lock the specified mutex. If the mutex is already locked, the calling thread does not wait for the mutex to become available.

Syntax

```
pthread_mutex_trylock(
                    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

```
#include <pthread.h>

int
pthread_mutex_trylock (
            pthread_mutex_t     *mutex);
```

Arguments

mutex
Mutex to be locked.

Description

This routine attempts to lock the mutex specified in the *mutex* argument. When a thread calls this routine, an attempt is made to immediately lock the mutex. If the mutex is successfully locked, this routine returns zero (0) and the calling thread becomes the mutex's current owner. If the specified mutex is locked when a thread calls this routine, the calling thread does not wait for the mutex to become available.

The behavior of this routine is as follows:

- For a normal, default, or errorcheck mutex: if the mutex is locked by any thread (including the calling thread) when this routine is called, this routine returns [EBUSY] and the calling thread does not wait to acquire the lock.
- For a normal or errorcheck mutex: if the mutex is not owned, this routine returns zero (0) and the mutex becomes locked.

- For a recursive mutex: if the mutex is owned by the current thread, this routine returns zero (0) and the mutex lock count is incremented. (To unlock a recursive mutex, each call to `pthread_mutex_trylock( )` must be matched by a call to `pthread_mutex_unlock( )`.)

Use the `pthread_mutexattr_settype( )` routine to set the mutex *type* attribute (normal, default, recursive, or errorcheck). For information about mutex types and their usage, see Chapter 2.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	The mutex is already locked; therefore, it was not acquired.
[EINVAL]	The value specified by *mutex* is invalid, or
	The *mutex* was created with the protocol attribute set to `PTHREAD_PRIO_PROTECT` and the calling thread's priority set higher than the mutex's current priority ceiling.

Associated Routines

```
pthread_mutexattr_settype( )
pthread_mutex_destroy( )
pthread_mutex_init( )
pthread_mutex_lock( )
pthread_mutex_unlock( )
```

pthread_mutex_unlock

Unlocks the specified mutex.

Syntax

```
pthread_mutex_unlock(
                    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

```
#include <pthread.h>

int
pthread_mutex_unlock (
            pthread_mutex_t     *mutex);
```

Arguments

mutex
Mutex to be unlocked.

Description

This routine unlocks the mutex specified by the *mutex* argument.

This routine behaves as follows, based on the type of the specified mutex:

- For a normal, default, or errorcheck mutex: if the mutex is owned by the calling thread, it is unlocked with no current owner. Further, for a normal or default mutex: if the mutex is not locked or is locked by another thread, this routine can also return [EPERM], but this is not guaranteed. For an errorcheck mutex: if the mutex is not locked or is locked by another thread, this routine returns [EPERM].
- For a recursive mutex: if the mutex is owned by the calling thread, the lock count is decremented. The mutex remains locked and owned until the lock count reaches zero (0). When the lock count reaches zero, the mutex becomes unlocked with no current owner.

If one or more threads are waiting to lock the specified mutex, and the mutex becomes unlocked, this routine causes one thread to unblock and to try to acquire the mutex. The scheduling policy is used to determine which thread to unblock. For the `SCHED_FIFO` and `SCHED_RR` policies, a blocked thread is chosen in priority order, using first-in/first-out within priorities. Note that the mutex might not be acquired by the awakened thread, if any other running thread attempts to lock the mutex first.

On DIGITAL UNIX, if a signal is delivered to a thread waiting for a mutex, upon return from the signal handler, the thread resumes waiting for the mutex as if it was not interrupted.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified for *mutex* is invalid.
[EPERM]	The calling thread does not own the mutex.

Associated Routines

```
pthread_mutexattr_settype( )
pthread_mutex_destroy( )
pthread_mutex_init( )
pthread_mutex_lock( )
pthread_mutex_trylock( )
```

pthread_once

Calls an initialization routine that is executed by a single thread, once.

Syntax

```
pthread_once(
        once_control,
        init_routine );
```

Argument	Data Type	Access
once_control	opaque pthread_once_t	modify
init_routine	procedure	read

C Binding

```
#include <pthread.h>

int
pthread_once (
        pthread_once_t    *once_control,
        void    (*init_routine) (void));
```

Arguments

once_control

Address of a record that defines the one-time initialization code. Each one-time initialization routine must have its own unique `pthread_once_t` record.

init_routine

Address of a procedure that performs the initialization. This routine is called only once, regardless of the number of times it and its associated *once_control* are passed to `pthread_once( )`.

Description

The first call to this routine by any thread in a process with a given *once_control* will call the specified *init_routine* with no arguments. Subsequent calls to `pthread_once( )` with the same *once_control* will not call the *init_routine*. On return from `pthread_once( )`, it is guaranteed that the initialization routine has completed.

For example, a mutex or a per-thread context key must be created exactly once. Calling pthread_once( ) ensures that the initialization is serialized across multiple threads. Other threads that reach the same point in the code would be delayed until the first thread is finished.

Note

If you specify an *init_routine* that directly or indirectly results in a recursive call to pthread_once( ) and that specifies the same *init_routine* argument, the recursive call can result in a deadlock.

To initialize the *once_control* record, your program can zero out the entire structure, or you can use the PTHREAD_ONCE_INIT macro, which is defined in the pthread.h header file, to statically initialize that structure. If using PTHREAD_ONCE_INIT, declare the *once_control* record as follows:

```
pthread_once_t  once_control = PTHREAD_ONCE_INIT;
```

Note that it is often easier to simply lock a statically initialized mutex, check a control flag, and perform necessary initialization (in-line) rather than using pthread_once( ). For example, you can code an initialization routine that begins with the following basic logic:

```
init()
{
 static pthread_mutex_t    mutex = PTHREAD_MUTEX_INITIALIZER;
 static int                flag = FALSE;

 pthread_mutex_lock(&mutex);
 if(!flag)
   {
    flag = TRUE;
    /* initialize code */
   }
 pthread_mutex_unlock(&mutex);
}
```

Return Values

If an error occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	Invalid argument.

pthread_self

Obtains the identifier of the calling thread.

Syntax

pthread_self( );

C Binding

#include <pthread.h>

pthread_t
pthread_self (void);

Arguments

None

Description

This routine returns the address of the calling thread's own thread identifier. For example, you can use this thread object to obtain the calling thread's own sequence number. To do so, pass the return value from this routine in a call to the `pthread_getsequence_np( )` routine, as follows:

```
.
.
.
unsigned long     this_thread_nbr;
.
.
.
this_thread_nbr = pthread_getsequence_np( pthread_self( ) );
.
.
.
```

The return value from the `pthread_self( )` routine becomes meaningless after the calling thread is destroyed.

Return Values

Returns the address of the calling thread's own thread object.

Associated Routines

```
pthread_cancel( )
pthread_create( )
pthread_detach( )
pthread_exit( )
pthread_getsequence_np( )
pthread_join( )
pthread_kill( )
pthread_sigmask( )
```

pthread_setcancelstate

Sets the calling thread's cancelability state.

Syntax

pthread_setcancelstate(
state,
oldstate);

Argument	Data Type	Access
state	integer	read
oldstate	integer	write

C Binding

```
#include <pthread.h>

int
pthread_setcancelstate (
            int    state,
            int    *oldstate );
```

Arguments

state
State of general cancelability to set for the calling thread. The following are valid cancel state values:

```
PTHREAD_CANCEL_ENABLE
PTHREAD_CANCEL_DISABLE
```

oldstate
Previous cancelability state for the calling thread.

Description

This routine sets the calling thread's cancelability state and returns the calling thread's previous cancelability state in *oldstate*.

When cancelability state is set to PTHREAD_CANCEL_DISABLE, a cancelation request cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability type is *enabled*.

When a thread is created, its default cancelability state is PTHREAD_CANCEL_ENABLE.

Possible Problems When Disabling Cancelability

The most important use of thread cancelation is to ensure that indefinite wait operations are terminated. For example, a thread that waits on some network connection, which can possibly take days to respond (or might never respond), should be made cancelable.

When a thread's cancelability is disabled, no routine in that thread is cancelable. As a result, the user is unable to cancel the operation performed by that thread. When disabling cancelability, be sure that no long waits can occur or that it is necessary for other reasons to defer cancelation requests around that particular region of code.

Return Values

On successful completion, this routine returns the calling thread's previous cancelability state in the location specified by the *oldstate* argument.

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

Associated Routines

```
pthread_cancel( )
pthread_setcanceltype( )
pthread_testcancel( )
```

pthread_setcanceltype

Sets the calling thread's cancelability type.

Syntax

pthread_setcanceltype(
    type,
    oldtype);

Argument	Data Type	Access
type	integer	read
oldtype	integer	write

C Binding

```
#include <pthread.h>

int
pthread_setcanceltype (
            int    type,
            int    *oldtype);
```

Arguments

type
The cancelability type to set for the calling thread. The following are valid values:

```
PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_ASYNCHRONOUS
```

oldtype
Returns the previous cancelability type.

Description

This routine sets the cancelability type and returns the previous type in location *oldtype*.

When a thread's cancelability state is set to `PTHREAD_CANCEL_DISABLE`, (see `pthread_setcancelstate( )`), a cancelation request cannot be delivered to that thread, even if a cancelable routine is called or asynchronous cancelability type is enabled.

When the cancelability state is set to `PTHREAD_CANCEL_ENABLE`, cancelability depends on the thread's cancelability type, as follows:

- If the thread's cancelability state is `PTHREAD_CANCEL_ENABLE` and the thread's cancelability type is set to `PTHREAD_CANCEL_DEFERRED`, the thread can only receive a cancelation request at a cancelation point (including condition waits, thread joins, and calls to `pthread_testcancel( )`).
- If the thread's cancelability state is `PTHREAD_CANCEL_ENABLE` and its cancelability type is `PTHREAD_CANCEL_ASYNCHRONOUS`, the thread can be canceled at any point in its execution.

When a thread is created, the default cancelability type is `PTHREAD_CANCEL_DEFERRED`.

Warning

If the asynchronous cancelability type is set, do not call any routine unless it is explicitly documented as "safe for asynchronous cancelation." Note that none of the general run-time libraries and none of the DECthreads libraries are safe for asynchronous cancelation except for `pthread_setcanceltype( )` and `pthread_setcancelstate( )`.

Use asynchronous cancelability only when you have a compute-bound section of code that carries no state and makes no routine calls.

Return Values

On successful completion, this routine returns the previous cancelability type in *oldtype*.

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_AYNCHRONOUS.

Associated Routines

pthread_cancel( )
pthread_setcancelstate( )
pthread_testcancel( )

pthread_setconcurrency

Changes the value of the concurrency level global variable for this process.

Syntax

```
pthread_setconcurrency(
                       level );
```

Argument	Data Type	Access
level	int	read

C Binding

```
#include <pthread.h>

int
pthread_setconcurrency (
                int     level);
```

Arguments

level
New value for the concurrency level for this process.

Description

This routine stores the value specified in the *level* argument in the "concurrency level" global setting for the calling thread's process. Because DECthreads automatically manages the concurrency of all threads in a multithreaded process, DECthreads ignores this concurrency level value.

The concurrency level value has no effect on the behavior of a multithreaded program that uses DECthreads. This routine is provided for Single UNIX Specification, Version 2 source code compatibility and has no other effect when called.

After calling this routine, subsequent calls to the `pthread_getconcurrency( )` routine return the same value, until another call to `pthread_setconcurrency( )` changes that value.

The initial concurrency level is zero (0), indicating that DECthreads manages the concurrency level. To indicate in a portable manner that the implementation is to resume control of concurrency level, call this routine with a *level* argument of zero (0).

The concurrency level value can be obtained using the `pthread_getconcurrency( )` routine.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *new_level* is negative.
[EAGAIN]	The value specified by *new_level* would cause a system resource to be exceeded.

Associated Routines

`pthread_getconcurrency( )`

pthread_setname_np

Changes the object name in the thread object for an existing thread.

Syntax

```
pthread_setname_np(
                    thread,
                    name,
                    mbz );
```

Argument	Data Type	Access
thread	opaque pthread_thread_t	write
name	char	read
mbz	void	read

C Binding

```
#include <pthread.h>

int
pthread_setname_np (
        pthread_thread_t   thread,
        const char   *name,
        void   *mbz);
```

Arguments

thread
Thread object whose object name is to be changed.

name
Object name value to copy into the thread object.

mbz
(Must be zero) Argument for use by DECthreads.

Description

This routine changes the object name in the thread object for the thread specified by the *thread* argument to the value specified by the *name* argument. To set an existing thread's object name, call this routine after creating the thread. However, with this approach your program must account for the possibility that the target thread has already exited or has been canceled before this routine is called.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a DECthreads-based multithreaded application. The maximum number of characters in the object name is 31.

This routine contrasts with `pthread_attr_setname_np( )`, which changes the object name attribute in a thread attributes object that is used to create a new thread.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ESRCH]	The thread specified by *thread* does not exist.
[EINVAL]	The length in characters of *name* exceeds 31.
[ENOMEM]	Insufficient memory exists to create a copy of the object name string.

Associated Routines

```
pthread_attr_getname_np( )
pthread_attr_setname_np( )
pthread_getname_np( )
```

pthread_setschedparam

Changes a thread's scheduling policy and scheduling parameters.

Syntax

```
pthread_setschedparam(
                  thread,
                  policy,
                  param );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
policy	integer	read
param	struct sched_param	read

C Binding

```
#include <pthread.h>

int
pthread_setschedparam (
        pthread_t    thread,
        int    policy,
        const struct sched_param    *param);
```

Arguments

thread
Thread whose scheduling policy and parameters are to be changed.

policy
New scheduling policy value for the thread specified in *thread*. The following are valid values:

```
SCHED_BG_NP
SCHED_FG_NP
SCHED_FIFO
SCHED_OTHER
SCHED_RR
```

See Section 2.3.2.2 for a description of thread scheduling policies.

param
New values of the scheduling parameters associated with the scheduling policy for the thread specified in *thread*. Valid values for the `sched_priority` field of a `sched_param` structure depend on the chosen scheduling policy. Use the POSIX routines `sched_get_priority_min( )` or `sched_get_priority_max( )` to determine the low and high limits of each policy.

Additionally, DECthreads provides *nonportable* priority range constants, as follows:

Low	High
PRI_FIFO_MIN	PRI_FIFO_MAX
PRI_RR_MIN	PRI_RR_MAX
PRI_OTHER_MIN	PRI_OTHER_MAX
PRI_FG_MIN_NP	PRI_FG_MAX_NP
PRI_BG_MIN_NP	PRI_BG_MAX_NP

The default priority varies by DECthreads platform. On DIGITAL UNIX, the default is 19 (that is, the POSIX priority of a normal timeshare process). On other platforms the default priority is the midpoint between `PRI_FG_MIN_NP` and `PRI_FG_MAX_NP`. (Section 2.3.6 describes how to specify priorities between the minimum and maximum values.)

Description

This routine changes both the current scheduling policy and associated scheduling parameters of the thread specified by *thread* to the policy and associated parameters provided in *policy* and *param*, respectively.

All currently implemented DECthreads scheduling policies have one scheduling parameter called `sched_priority`. For the policy you choose, you must specify an appropriate value in the `sched_priority` field of the `sched_param` structure.

Changing the scheduling policy or priority, or both, of a thread can cause it to start executing or to be preempted by another thread. A thread changes its own scheduling policy and priority by using the handle returned by the `pthread_self( )` routine.

This routine differs from `pthread_attr_setschedpolicy( )` and `pthread_attr_setschedparam( )`, in that those routines set the scheduling policy and parameter attributes that are used to establish the scheduling priority and scheduling policy of a new thread when it is created. However, this routine changes the scheduling policy and parameters of an existing thread.

Return Values

If an error condition occurs, no scheduling policy or parameters are changed for the target thread, and this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *policy* or *param* is invalid.
[ENOTSUP]	An attempt was made to set the scheduling policy or a parameter to an unsupported value.
[EPERM]	The caller does not have the appropriate privileges to set the scheduling policy or parameters of the specified thread.
[ESRCH]	The value specified by *thread* does not refer to an existing thread.

Associated Routines

```
pthread_attr_setschedparam( )
pthread_attr_setschedpolicy( )
pthread_create( )
pthread_self( )
sched_yield( )
```

pthread_setspecific

Sets the thread-specific data value associated with the specified key for the calling thread.

Syntax

```
pthread_setspecific(
                    key,
                    value );
```

Argument	Data Type	Access
key	opaque pthread_key_t	read
value	void *	read

C Binding

```
#include <pthread.h>

int
pthread_setspecific (
            pthread_key_t    key,
            const void       *value);
```

Arguments

key
Thread-specific key that identifies the thread-specific data to receive *value*. This key value must be obtained from `pthread_key_create( )`.

value
New thread-specific data value to associate with the specified key for the calling thread.

Description

This routine sets the thread-specific data value associated with the specified *key* for the current thread. If a value is defined for the key in this thread (the current value is not NULL), the new value is substituted for it. The key is obtained by a previous call to `pthread_key_create( )`.

Different threads can bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that are reserved for use by the calling thread.

Do not call this routine from a thread-specific data destructor function.

Note that although the type for *value* `(void *)` implies that it represents an address, the type is being used as a "universal scalar type." DECthreads simply stores *value* for later retrieval.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified *key* is invalid.
[ENOMEM]	Insufficient memory exists to associate the value with the key.

Associated Routines

```
pthread_getspecific( )
pthread_key_create( )
pthread_key_delete( )
```

pthread_sigmask

Examine or change the calling thread's signal mask.

This routine is for DIGITAL UNIX systems only.

Syntax

```
pthread_sigmask(
        how,
        set,
        oset );
```

Argument	Data Type	Access
how	integer	read
set	sigset_t	read
oset	sigset_t	write

C Binding

#include <pthread.h>

```
int
pthread_sigmask (
        int    how,
        const sigset_t    *set,
        sigset_t    *oset);
```

Arguments

how

Indicates the manner in which the set of masked signals is changed. The optional values are as follows:

`SIG_BLOCK`	The resulting set is the union of the current set and the signal set pointed to by the *set* argument.
`SIG_UNBLOCK`	The resulting set is the intersection of the current set and the complement of the signal set pointed to by the *set* argument.

SIG_SETMASK	The resulting set is the signal set pointed to by the *set* argument.

set
Specifies the signal set by pointing to a set of signals used to change the blocked set. If this *set* value is NULL, the *how* argument is ignored and the process signal mask is unchanged.

oset
Receives the value of the current signal mask (unless this value is NULL).

Description

This routine examines or changes the calling thread's signal mask. Typically, you use the SIG_BLOCK option for the *how* value to block signals during a critical section of code, and then use this routine's SIG_SETMASK option to restore the mask to the previous value returned by the previous call to the pthread_sigmask( ) routine.

If there are any unblocked signals pending after a call to this routine, at least one of those signals is before this routine returns.

This routine does not allow the SIGKILL or SIGSTOP signals to be blocked. If a program attempts to block one of these signals, pthread_sigmask( ) gives no indication of the error.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified for *how* is invalid.

pthread_testcancel

Requests delivery of a pending cancelation request to the calling thread.

Syntax

pthread_testcancel( );

C Binding

#include <pthread.h>

void
pthread_testcancel (*void*);

Arguments

None

Description

This routine requests delivery of a pending cancelation request to the calling thread. Thus, calling this routine creates a cancelation point within the calling thread.

The cancelation request is delivered only if a request is pending for the calling thread and the calling thread's cancelability state is *enabled*. (A thread disables delivery of cancelation requests to itself by calling `pthread_setcancelstate( )`.)

When called within very long loops, this routine ensures that a pending cancelation request is noticed by the calling thread within a reasonable amount of time.

Return Values

None

Associated Routines

`pthread_setcancelstate( )`

pthread_unlock_global_np

Unlocks the DECthreads global mutex.

Syntax

```
pthread_unlock_global_np( );
```

C Binding

```
#include <pthread.h>

int
pthread_unlock_global_np (void);
```

Arguments

None

Description

This routine unlocks the DECthreads global mutex. Because the global mutex is recursive, the unlock occurs when each call to `pthread_lock_global_np( )` has been matched by a call to this routine. For example, if you called `pthread_lock_global_np( )` three times, `pthread_unlock_global_np( )` unlocks the global mutex when you call it the third time.

If no threads are waiting for the DECthreads global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, this routine causes one thread to unblock and try to acquire the global mutex. The scheduling policy is used to determine which thread is awakened. For the policies `SCHED_FIFO` and `SCHED_RR`, a blocked thread is chosen in priority order, using first-in/first-out (FIFO) within priorities.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EPERM]	The mutex is unlocked or owned by another thread.

Associated Routines

`pthread_lock_global_np( )`

sched_get_priority_max

Returns the maximum priority for the specified scheduling policy.

Syntax

sched_get_priority_max(*policy*);

C Binding

```
#include <sched.h>

int
sched_get_priority_max (
                int     policy);
```

Arguments

policy
One of the scheduling policies, as defined in `sched.h`.

Description

This routine returns the maximum priority for the scheduling policy specified in the *policy* argument. The argument value must be one of the scheduling policies (`SCHED_FIFO`, `SCHED_RR`, or `SCHED_OTHER`), as defined in the `sched.h` header file.

No special privileges are required to use this routine.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value of the *policy* argument does not represent a defined scheduling policy.

Associated Routines

```
sched_getparam( )
sched_getscheduler( )
sched_get_priority_min( )
sched_setparam( )
sched_setscheduler( )
```

sched_get_priority_min

Returns the minimum priority for the specified scheduling policy.

Syntax

sched_get_priority_min(*policy*);

C Binding

```
 include <sched.h>

int
sched_get_priority_min (
                int     policy);
```

Arguments

policy
One of the scheduling policies, as defined in `sched.h`.

Description

This routine returns the minimum priority for the scheduling policy specified in the *policy* argument. The argument value must be one of the scheduling policies (`SCHED_FIFO`, `SCHED_RR`, or `SCHED_OTHER`), as defined in the `sched.h` header file.

No special privileges are required to use this routine.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value of the *policy* argument does not represent a defined scheduling policy.

Associated Routines

```
sched_getparam( )
sched_getscheduler( )
sched_get_priority_max( )
sched_setparam( )
sched_setscheduler( )
```

sched_yield

Yields execution to another thread.

Syntax

sched_yield( );

C Binding

```
include <sched.h>
include <unistd.h>
```

```
int sched_yield (
void);
```

Arguments

None

Description

In conformance with the IEEE POSIX.1b-1995 standard, the `sched_yield( )` function causes the calling thread to yield execution to another thread. It is useful when a thread running under the `SCHED_FIFO` scheduling policy must allow another thread at the same priority to run. The thread that is interrupted by `sched_yield( )` goes to the end of the queue for its priority.

If no other thread is runnable at the priority of the calling thread, the calling thread continues to run.

Threads with higher priority are allowed to preempt the calling thread, so the `sched_yield( )` function has no effect on the scheduling of higher- or lower-priority threads.

The `sched_yield( )` routine takes no arguments. No special privileges are needed to use the `sched_yield( )` function.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
-1	Unsuccessful completion—*errno* is set to indicate that an error occurred.
[ENOSYS]	The routine `sched_yield( )` is not supported by this implementation.

Associated Routines

`pthread_attr_setschedparam( )`
`pthread_setschedparam( )`
`pthread_getscheduler( )`

sigwait

Suspends a calling thread until a signal arrives.

Syntax

```
sigwait(set,
        signal);
```

C Binding

```
 include <signal.h>

int
sigwait (
            sigset_t   *set,
            int    *signal);
```

Arguments

set
Set of signals to wait for.

signal
Signal number obtained for the selected signal.

Description

This routine suspends the calling thread until at least one of the signals in the *set* argument is in the caller's set of pending signals. When this happens, one of those signals is automatically selected and removed from the set of pending signals. The signal number identifying that signal is then returned.

This routine stores the signal number obtained in the address specified in the *signal* argument.

The effect of calling this routine is unspecified if any signals in the *set* argument are not blocked at the time of the call.

The *set* signal set object is created using the set manipulation routines `sigemptyset( )`, `sigfillset( )`, `sigaddset( )`, and `sigdelset( )`.

If, while this routine is waiting, a signal occurs that is eligible for delivery (that is, not blocked by the signal mask), that signal is handled asynchronously and the wait is interrupted.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value of the *set* argument contains an invalid or unsupported signal number.
[EINTR]	The wait was interrupted by an unblocked, caught signal.

Associated Routines

```
sigaddset( )
sigdelset( )
sigemptyset( )
sigfillset( )
sigaction( )
sigpending( )
sigsuspend( )
```

Part III

DIGITAL-Proprietary Interfaces: tis Routines Reference

Part III provides detailed descriptions of the DIGITAL-proprietary DECthreads thread-independent services (or **tis**) interface routines.

These routines are designed to provide efficient tools for thread safety in libraries whose routines do not themselves use threads. The **tis** interface provides functions identical to several **pthread** functions. In a program that creates or uses threads, the **tis** functions provide full thread synchronization and coherence of memory access. But, in a program that does not use threads, the same **tis** calls provide low-overhead "stub" implementations of **pthread** features.

The objects created using **tis** interface routines are the same as **pthread** interface objects.

The variable *errno* is not used by the **tis** routines. Like the **pthread** routines, the **tis** routines return integer values indicating the type of error.

Note

In a nonthreaded environment, never code **tis** routines to use condition variables to block operations. For example, the single-threaded implementation of `tis_cond_wait( )` cannot block. If it did, no other thread would be running to "awaken" the program.

When threads are present, the guidelines for using **pthread** routines apply to using the corresponding **tis** routines.

tis_cond_broadcast

Wakes all threads that are waiting on a condition variable.

Syntax

tis_cond_broadcast(
cond);

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

C Binding

#include <tis.h>

int
tis_cond_broadcast (
pthread_cond_t **cond*);

Arguments

cond
Address of the condition variable (passed by reference) on which to broadcast.

Description

When threads are not present, this routine performs no actions.

When threads are present, this routine unblocks all threads waiting on the specified condition variable *cond*.

For further information about actions when threads are present, refer to the `pthread_cond_broadcast( )` description.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid.

Associated Routines

```
tis_cond_destroy( )
tis_cond_init( )
tis_cond_signal( )
tis_cond_wait( )
```

tis_cond_destroy

Destroys the specified condition variable.

Syntax

tis_cond_destroy(
    cond);

Argument	Data Type	Access
cond	opaque pthread_cond_t	write

C Binding

#include <tis.h>

```
int
tis_cond_destroy (
            pthread_cond_t     *cond);
```

Arguments

cond
Address of the condition variable (passed by reference) to be destroyed.

Description

This routine destroys the condition variable specified by *cond*. After this routine is called, DECthreads may reclaim internal storage used by the condition variable object. Call this routine when a condition variable will no longer be referenced.

The results of this routine are unpredictable, if the condition variable specified in *cond* does not exist or is not initialized.

For more information about actions when threads are present, refer to the `pthread_cond_destroy( )` description.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid.
[EBUSY]	The object being referenced by *cond* is being referenced by another thread that is currently executing a `tis_cond_wait( )` on the condition variable specified in *cond*. (This error can only occur when threads are present.)

Associated Routines

```
tis_cond_broadcast( )
tis_cond_init( )
tis_cond_signal( )
tis_cond_wait( )
```

tis_cond_init

Initializes a condition variable.

Syntax

```
tis_cond_init(
            cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write

C Binding

```
#include <tis.h>

int
tis_cond_init (
            pthread_cond_t    *cond);
```

Arguments

cond
Address of the condition variable (passed by reference) to be initialized.

Description

This routine initializes a condition variable (*cond*) with the DECthreads default condition variable attributes.

A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data. When threads are present, a condition variable allows threads to wait for data to enter a defined state.

For more information about actions taken when threads are present, refer to the `pthread_cond_init( )` description.

Your program can use the macro `PTHREAD_COND_INITIALIZER` to initialize statically allocated condition variables to the DECthreads default condition variable attributes. Use this macro as follows:

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```

When statically initialized, a condition variable should not also be initialized using `tis_cond_init( )`. Also, a statically initialized condition variable need not be destroyed using `tis_cond_destroy( )`.

Return Values

If there is an error condition, the following occurs:

- The routine returns an integer value indicating the type of error.
- The condition variable is not initialized.
- The contents of conditon variable *cond* are undefined.

The possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize another condition variable, or: The system-imposed limit on the total number of condition variables under execution by a single user is exceeded.
[EBUSY]	The implementation has detected an attempt to reinitialize the object referenced by *cond*, a previously initialized, but not yet destroyed condition variable.
[EINVAL]	The value specified by *attr* is invalid.
[ENOMEM]	Insufficient memory exists to initialize the condition variable.

Associated Routines

```
tis_cond_broadcast( )
tis_cond_destroy( )
tis_cond_signal( )
tis_cond_wait( )
```

tis_cond_signal

Wakes at least one thread that is waiting on the specified condition variable.

Syntax

```
tis_cond_signal(
            cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

C Binding

#include <tis.h>

```
int
tis_cond_signal (
            pthread_cond_t    *cond);
```

Arguments

cond
Address of the condition variable (passed by reference) on which to signal.

Description

When threads are present, this routine unblocks at least one thread that is waiting on the specified condition variable *cond*.

For more information about actions taken when threads are present, refer to the `pthread_cond_signal( )` description.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* is invalid.

Associated Routines

```
tis_cond_broadcast( )
tis_cond_destroy( )
tis_cond_init( )
tis_cond_wait( )
```

tis_cond_wait

Causes a thread to wait for the specified condition variable to be signaled or broadcasted.

Syntax

```
tis_cond_wait(
        cond,
        mutex );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify
mutex	opaque pthread_mutex_t	modify

C Binding

```
#include <tis.h>

int
tis_cond_wait (
        pthread_cond_t    *cond,
        pthread_mutex_t   *mutex);
```

Arguments

cond
Address of the condition variable (passed by reference) on which to wait.

mutex
Address of the mutex (passed by reference) that is associated with the condition variable specified in *cond*.

Description

When threads are present, this routine causes a thread to wait for the specified condition variable *cond* to be signaled or broadcasted.

Calling this routine in a single-threaded environment is a coding error. Because no thread can execute in parallel to issue a call to `tis_cond_signal( )` or `tis_cond_broadcast( )`, using this routine in a single-threaded environment forces the program to exit.

For further information about actions taken when threads are present, refer to the `pthread_cond_wait( )` description.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *cond* or *mutex* is invalid, or: Different mutexes are supplied for concurrent `tis_cond_wait( )` operations on the same condition variable, or: The mutex was not owned by the calling thread at the time of the call.

Associated Routines

```
tis_cond_broadcast( )
tis_cond_destroy( )
tis_cond_init( )
tis_cond_signal( )
```

tis_getspecific

Obtains the data associated with the specified thread-specific data key.

Syntax

tis_getspecific(
    key);

Argument	Data Type	Access
key	opaque pthread_key_t	read

C Binding

#include <tis.h>

```
void *
tis_getspecific (
                pthread_key_t    key);
```

Arguments

key
Identifies a value returned by a call to `tis_key_create( )`. This routine returns the data value associated with the thread-specific data key.

Description

This routine returns the value currently bound to the specified thread-specific data *key*.

This routine can be called from a data destructor function.

When threads are present, the data and keys are thread specific; they enable a library to maintain context on a per-thread basis.

Return Values

No errors are returned. This routine returns the data value associated with the specified thread-specific data key *key*. If no data value is associated with *key*, or if *key* is not defined, then a NULL value is returned.

Associated Routines

```
tis_key_create( )
tis_key_delete( )
tis_setspecific( )
```

tis_key_create

Generates a unique thread-specific data key.

Syntax

```
tis_key_create(
            key,
            destructor );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write
destructor	procedure	read

C Binding

```
#include <tis.h>

int
tis_key_create (
    pthread_key_t     *key,
    void    (*destructor)(void *));
```

Arguments

key
Address of a variable that receives the key value. This value is used in calls to `tis_getspecific( )` and `tis_setspecific( )` to get and set the value associated with this key.

destructor
Address of a routine that is called to destroy the context value when a thread terminates with a non-NULL value for the key. Note that this argument is used only when threads are present.

tis_key_create

Description

This routine generates a unique thread-specific data key. The *key* argument points to an opaque object used to locate data.

This routine generates and returns a new key value. The key reserves a cell. Each call to this routine creates a new cell that is unique within an application invocation. Keys must be generated from initialization code that is guaranteed to be called only once within each process. (See the `tis_once()` description for more information.)

Your program can associate an optional destructor function with each key. At thread exit, if a key has a non-NULL destructor function pointer, and the thread has a non-NULL value associated with that key, the function pointed to is called with the current associated value as its sole argument. The order in which data destructors are called at thread termination is undefined.

When threads are present, keys and any corresponding data are thread specific; they enable the context to be maintained on a per-thread basis. For more information about the use of `tis_key_create()` in a threaded environment, refer to the `pthread_key_create()` description.

DECthreads imposes a maximum number of thread-specific data keys, equal to the symbolic constant `PTHREAD_KEYS_MAX`.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacked the necessary resources to create another thread-specific data key, or the limit on the total number of keys per process (`PTHREAD_KEYS_MAX`) has been exceeded.
[ENOMEM]	Insufficient memory exists to create the key.
[EINVAL]	Invalid argument.

Associated Routines

```
tis_getspecific( )
tis_key_delete( )
tis_setspecific( )
tis_once( )
```

tis_key_delete

Deletes the specified thread-specific data key.

Syntax

```
tis_key_delete(
        key );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write

C Binding

```
#include <tis.h>

int
tis_key_delete (
        pthread_key_t    key);
```

Arguments

key
Thread-specific data key to be deleted.

Description

This routine deletes a thread-specific data key *key* previously returned by a call to the `tis_key_create( )` routine. The data values associated with *key* need not be NULL at the time this routine is called. The application must free any application storage or perform any cleanup actions for data structures related to the deleted key or associated data. This cleanup can be done before or after this routine is called.

Attempting to use the thread-specific data key *key* after calling this routine results in unpredictable behavior.

No destructor functions are invoked by this routine. Any destructor functions that may have been associated with *key* will no longer be called upon thread exit.

This routine can be called from destructor functions.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value for *key* is invalid.

Associated Routines

```
tis_getspecific( )
tis_key_create( )
tis_setspecific( )
```

tis_lock_global

Locks the DECthreads global mutex.

Syntax

tis_lock_global( );

C Binding

#include <tis.h>

int
tis_lock_global (void);

Arguments

None

Description

This routine locks the DECthreads global mutex. The global mutex is recursive. For example, if you called `tis_lock_global( )` three times, `tis_unlock_global( )` unlocks the global mutex when you call it the third time.

For more information about actions taken when threads are present, refer to the `pthread_lock_global_np( )` description.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.

Associated Routines

`tis_unlock_global( )`

tis_mutex_destroy

Destroys the specified mutex object.

Syntax

```
tis_mutex_destroy(
            mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write

C Binding

```
#include <tis.h>

int
tis_mutex_destroy (
            pthread_mutex_t     *mutex);
```

Arguments

mutex
Address of the mutex object (passed by reference) to be destroyed.

Description

This routine destroys a mutex object by uninitializing it, and should be called when a mutex object is no longer referenced. After this routine is called, DECthreads can reclaim internal storage used by the mutex object.

It is safe to destroy an initialized mutex object that is unlocked. However, it is illegal to destroy a locked mutex object.

The results of this routine are unpredictable if the mutex object specified in the *mutex* argument does not currently exist or is not initialized.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	An attempt was made to destroy the object referenced by *mutex* while it is locked or referenced.
[EINVAL]	The value specified by *mutex* is invalid.
[EPERM]	The caller does not have privileges to perform the operation.

Associated Routines

```
tis_mutex_init( )
tis_mutex_lock( )
tis_mutex_trylock( )
tis_mutex_unlock( )
```

tis_mutex_init

Initializes the specified mutex object.

Syntax

```
tis_mutex_init(
        mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write

C Binding

```
#include <tis.h>

int
tis_mutex_init (
        pthread_mutex_t     *mutex );
```

Arguments

mutex
Pointer to a mutex object (passed by reference) to be initialized.

Description

This routine initializes a mutex object with the DECthreads default mutex attributes. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

The mutex object is initialized and set to the unlocked state. Mutexes can be allocated in heap or static memory, but not on a stack.

Your program can use the `PTHREAD_MUTEX_INITIALIZER` macro to statically initialize a mutex object without calling this routine. Statically initialized mutexes need not be destroyed using `tis_mutex_destroy( )`. Use this macro as follows:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error, the mutex is not initialized, and the contents of *mutex* are undefined. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize a mutex.
[ENOMEM]	Insufficient memory exists to initialize the mutex.
[EBUSY]	The implementation has detected an attempt to reinitialize *mutex* (a previously initialized, but not yet destroyed, mutex).
[EINVAL]	The value specified by *mutex* is invalid.
[EPERM]	The caller does not have privileges to perform this operation.

Associated Routines

```
tis_mutex_destroy( )
tis_mutex_lock( )
tis_mutex_trylock( )
tis_mutex_unlock( )
```

tis_mutex_lock

Locks an unlocked mutex.

Syntax

```
tis_mutex_lock(
                mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

```
#include <tis.h>

int
tis_mutex_lock (
                pthread_mutex_t      *mutex);
```

Arguments

mutex
Address of the mutex (passed by reference) to be locked.

Description

This routine locks the specified mutex *mutex*. A deadlock can result if the owner of a mutex calls this routine in an attempt to lock the same mutex a second time. (DECthreads does not detect or report the deadlock.)

In a threaded environment, the thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the mutex in the locked state and with the current thread as the mutex's current owner.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *mutex* is invalid.
[EDEADLK]	A deadlock condition is detected.

Associated Routines

```
tis_mutex_destroy( )
tis_mutex_init( )
tis_mutex_trylock( )
tis_mutex_unlock( )
```

tis_mutex_trylock

Attempts to lock the specified mutex.

Syntax

```
tis_mutex_trylock(
                mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

```
#include <tis.h>

int
tis_mutex_trylock (
                pthread_mutex_t    *mutex);
```

Arguments

mutex
Address of the mutex (passed by reference) to be locked.

Description

This routine attempts to lock the specified mutex *mutex*. When this routine is called, an attempt is made immediately to lock the mutex. If the mutex is successfully locked, zero (0) is returned.

If the specified mutex is already locked when this routine is called, the caller does not wait for the mutex to become available. [EBUSY] is returned, and the thread does not wait to acquire the lock.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	The mutex is already locked; therefore, it was not acquired.
[EINVAL]	The value specified by *mutex* is invalid.

Associated Routines

```
tis_mutex_destroy( )
tis_mutex_init( )
tis_mutex_lock( )
tis_mutex_unlock( )
```

tis_mutex_unlock

Unlocks the specified mutex.

Syntax

```
tis_mutex_unlock(
                mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

```
#include <tis.h>

int
tis_mutex_unlock (
                pthread_mutex_t     *mutex);
```

Arguments

mutex
Address of the mutex (passed by reference) to be unlocked.

Description

This routine unlocks the specified mutex *mutex*.

For more information about actions taken when threads are present, refer to the `pthread_mutex_unlock( )` description.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *mutex* is invalid.
[EPERM]	The caller does not own the mutex.

Associated Routines

```
tis_mutex_destroy( )
tis_mutex_init( )
tis_mutex_lock( )
tis_mutex_trylock( )
```

tis_once

Calls a one-time initialization routine that can be executed by only one thread, once.

Syntax

tis_once(
once_control,
init_routine);

Argument	Data Type	Access
once_control	opaque pthread_once_t	modify
init_routine	procedure	read

C Binding

```
#include <tis.h>

int
tis_once (
        pthread_once_t   *once_control,
        void   (*init_routine) (void));
```

Arguments

once_control
Address of a record (control block) that defines the one-time initialization code. Each one-time initialization routine in static storage must have its own unique `pthread_once_t` record.

init_routine
Address of a procedure that performs the initialization. This routine is called only once, regardless of the number of times it and its associated *once_control* are passed to `tis_once( )`.

Description

The first call to this routine by a process with a given *once_control* calls the *init_routine* with no arguments. Thereafter, subsequent calls to `tis_once( )` with the same *once_control* do not call the *init_routine*. On return from `tis_once( )`, it is guaranteed that the initialization routine has completed.

For example, a mutex or a thread-specific data key must be created exactly once. In a threaded environment, calling `tis_once( )` ensures that the initialization is serialized across multiple threads.

The *once_control* argument must be statically initialized using the `PTHREAD_ONCE_INIT` macro or by zeroing out the entire structure.

Note

If you specify an *init_routine* that directly or indirectly results in a recursive call to `tis_once( )` and that specifies the same *init_block* argument, the recursive call results in a deadlock.

The `PTHREAD_ONCE_INIT` macro, defined in the `pthread.h` header file, must be used to initialize a *once_control* record. Thus, your program must declare a *once_control* record as follows:

```
pthread_once_t  once_control = PTHREAD_ONCE_INIT;
```

Note that it is often easier to simply lock a statically initialized mutex, check a control flag, and perform necessary initialization (in-line) rather than using `tis_once( )`. For example, you can code an "init" routine that begins with the following basic logic:

```
init()
{
 static pthread_mutex_t    mutex = PTHREAD_MUTEX_INIT;
 static int                flag = FALSE;

 tis_mutex_lock(&mutex);
 if(!flag)
   {
    flag = TRUE;
    /* initialize code */
   }
 tis_mutex_unlock(&mutex);
}
```

Return Values

If an error occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	Invalid argument.

tis_read_lock

Acquires a read-write lock for read access.

Syntax

```
tis_read_lock(
            lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

C Binding

```
#include <tis.h>

int
tis_read_lock (
            tis_rwlock_t    *lock);
```

Arguments

lock
Address of the read-write lock.

Description

This routine acquires a read-write lock for read access. This routine waits for any existing lock holder for write access to relinquish its lock before granting the lock for read access. This routine returns when the lock is acquired. If the lock is already held for read access, the lock is granted.

For each call to `tis_read_lock( )` that successfully acquires the lock for read access, a corresponding call to `tis_read_unlock( )` must be issued.

Note that the type `tis_rwlock_p` is a pointer to type `tis_rwlock_t`.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *lock* is invalid.

Associated Routines

```
tis_read_trylock( )
tis_read_unlock( )
tis_rwlock_destroy( )
tis_rwlock_init( )
tis_write_lock( )
tis_write_trylock( )
tis_write_unlock( )
```

tis_read_trylock

Attempts to acquire a read-write lock for read access. Does not wait if the lock cannot be immediately granted.

Syntax

```
tis_read_trylock(
                 lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

C Binding

```
#include <tis.h>

int
tis_read_trylock (
        tis_rwlock_t     *lock);
```

Arguments

lock
Address of the read-write lock to be acquired.

Description

This routine attempts to acquire a read-write lock for read access. If the lock cannot be granted, the routine returns without waiting.

When a thread calls this routine, an attempt is made to immediately acquire the lock for read access. If the lock is acquired, zero (0) is returned. If a holder of the lock for write access exists, [EBUSY] is returned.

If the lock cannot be acquired for read access immediately, the calling program does not wait for the lock to be released.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion; the lock was acquired.
[EBUSY]	The lock is being held for write access. The lock for read access was not acquired.

Associated Routines

```
tis_read_lock( )
tis_read_unlock( )
tis_rwlock_destroy( )
tis_rwlock_init( )
tis_write_lock( )
tis_write_trylock( )
tis_write_unlock( )
```

tis_read_unlock

Unlocks a read-write lock that was acquired for read access.

Syntax

tis_read_unlock(
 lock);

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

C Binding

```
#include <tis.h>

int
tis_read_unlock (
        tis_rwlock_t    *lock);
```

Arguments

lock
Address of the read-write lock to be unlocked.

Description

This routine unlocks a read-write lock that was acquired for read access. If there are no other holders of the lock for read access and another thread is waiting to acquire the lock for write access, that lock acquisition is granted.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *lock* is invalid.

Associated Routines

```
tis_read_lock( )
tis_read_trylock( )
tis_rwlock_destroy( )
tis_rwlock_init( )
tis_write_lock( )
tis_write_trylock( )
tis_write_unlock( )
```

tis_rwlock_destroy

Destroys the specified read-write lock object.

Syntax

tis_rwlock_destroy(
    lock);

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

C Binding

```
#include <tis.h>

int
tis_rwlock_destroy (
            tis_rwlock_t    *lock);
```

Arguments

lock
Address of the read-write lock object to be destroyed.

Description

This routine destroys the specified read-write lock object. Prior to calling this routine, ensure that there are no locks granted to the specified read-write lock and that there are no threads waiting for pending lock acquisitions on the specified read-write lock.

This routine should be called only after all reader threads (and perhaps one writer thread) have finished using the specified read-write lock.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	The lock is in use.

Associated Routines

```
tis_read_lock( )
tis_read_trylock( )
tis_read_unlock( )
tis_rwlock_init( )
tis_write_lock( )
tis_write_trylock( )
tis_write_unlock( )
```

tis_rwlock_init

Initializes a read-write lock object.

Syntax

```
tis_rwlock_init(
        lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

C Binding

```
#include <tis.h>

int
tis_rwlock_init (
        tis_rwlock_t     *lock);
```

Arguments

lock
Address of a read-write lock object.

Description

This routine initializes a read-write lock object. The routine initializes the `tis_rwlock_t` structure that holds the object's lock states.

To destroy a read-write lock object, call the `tis_rwlock_destroy( )` routine.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *lock* is invalid.
[ENOMEM]	Insufficient memory exists to initialize *lock*.

Associated Routines

```
tis_read_lock( )
tis_read_trylock( )
tis_read_unlock( )
tis_rwlock_destroy( )
tis_write_lock( )
tis_write_trylock( )
tis_write_unlock( )
```

tis_self

Returns the identifier of the calling thread.

Syntax

```
tis_self(
        void);
```

C Binding

#include <tis.h>

pthread_t
tis_self (void);

Arguments

None

Description

This routine allows a thread to obtain its own thread identifier.

This value becomes meaningless when the thread is destroyed.

Note that the initial thread in a process can "change identity" when thread system initialization completes—that is, when the DECthreads multithreading run-time environment is loaded.

Return Values

Returns the thread identifier of the calling thread.

Associated Routines

`pthread_create( )`

tis_setcancelstate

Changes the calling thread's cancelability state.

Syntax

```
tis_setcancelstate(
                   state,
                   oldstate );
```

Argument	Data Type	Access
state	integer	read
oldstate	integer	write

C Binding

```
#include <tis.h>

int
tis_setcancelstate (
                int    state,
                int    *oldstate );
```

Arguments

state
State of general cancelability to set for the calling thread. Valid state values are as follows:

```
PTHREAD_CANCEL_ENABLE
PTHREAD_CANCEL_DISABLE
```

oldstate
Receives the value of the calling thread's previous cancelability state.

Description

This routine sets the calling thread's cancelability state to the value specified in the *state* argument and returns the calling thread's previous cancelability state in the location referenced by the *oldstate* argument.

When the a thread's cancelability state is set to PTHREAD_CANCEL_DISABLE, a cancelation request cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability is *enabled*.

When a thread is created, its default cancelability state is PTHREAD_CANCEL_ENABLE. When this routine is called prior to loading threads, the cancelability state propagates to the initial thread in the executing program.

Possible Problems When Disabling Cancelability

The most important use of a cancelation request is to ensure that indefinite wait operations are terminated. For example, a thread waiting on some network connection, which might take days to respond (or might never respond), should be made cancelable.

When a thread's cancelability state is *disabled*, no routine called within that thread is cancelable. As a result, the user is unable to cancel the operation. When disabling cancelability, be sure that no long waits can occur or that it is necessary for other reasons to defer cancelation requests around that particular region of code.

Return Values

On successful completion, this routine returns the calling thread's previous cancelability state in the *oldstate* argument.

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

Associated Routines

tis_testcancel( )

tis_setspecific

Changes the value associated with the specified thread-specific data key.

Syntax

```
tis_setspecific(
        key,
        value );
```

Argument	Data Type	Access
key	opaque pthread_key_t	read
value	void *	read

C Binding

```
#include <tis.h>

int
tis_setspecific (
            pthread_key_t    key,
            const void      *value);
```

Arguments

key
Thread-specific data key that identifies the data to receive *value*. Must be obtained from a call to `tis_key_create( )`.

value
New value to associate with the specified key. Once set, this value can be retrieved using the same key in a call to `tis_getspecific( )`.

Description

This routine sets the value associated with the specified thread-specific data key. If a value is defined for the key (that is, the current value is not NULL), the new value is substituted for it. The key is obtained by a previous call to `tis_key_create( )`.

Do not call this routine from a data destructor function.

Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The key value is invalid.
[ENOMEM]	Insufficient memory exists to associate the value with the key.

Associated Routines

```
tis_getspecific( )
tis_key_create( )
tis_key_delete( )
```

tis_testcancel

Creates a cancelation point in the calling thread.

Syntax

tis_testcancel();

C Binding

#include <tis.h>

void
tis_testcancel (*void*);

Arguments

None

Description

This routine requests delivery of a pending cancelation request to the calling thread. Thus, this routine creates a cancelation point in the calling thread. The cancelation request is delivered only if a request is pending for the calling thread and the calling thread's cancelability state is *enabled*. (A thread disables delivery of cancelation requests to itself by calling `tis_setcancelstate( )`.)

This routine, when called within very long loops, ensures that a pending cancelation request is noticed within a reasonable amount of time.

Return Values

None

Associated Routines

`tis_setcancelstate( )`

tis_unlock_global

Unlocks the DECthreads global mutex.

Syntax

tis_unlock_global( );

C Binding

#include <tis.h>

int
tis_unlock_global (void);

Arguments

None

Description

This routine unlocks the DECthreads global mutex. Because the global mutex is recursive, the unlock occurs when each call to `tis_lock_global( )` has been matched by a call to this routine. For example, if your program called `tis_lock_global( )` three times, `tis_unlock_global( )` unlocks the global mutex when you call it the third time.

For more information about actions taken when threads are present, refer to the `pthread_unlock_global_np( )` description.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EPERM]	The DECthreads global mutex is unlocked or locked by another thread.

Associated Routines

`tis_lock_global()`

tis_write_lock

Acquires a read-write lock for write access.

Syntax

```
tis_write_lock(
             lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

C Binding

```
#include <tis.h>

int
tis_rwlock (
             tis_rwlock_t      *lock);
```

Arguments

lock
Address of the read-write lock to be acquired for write access.

Description

This routine acquires a read-write lock for write access. This routine waits for any other active locks (for either read or write access) to be unlocked before this acquisition request is granted.

This routine returns when the specified read-write lock is acquired for write access.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *lock* is invalid.

Associated Routines

```
tis_read_lock( )
tis_read_trylock( )
tis_read_unlock( )
tis_rwlock_destroy( )
tis_rwlock_init( )
tis_write_trylock( )
tis_write_unlock( )
```

tis_write_trylock

Attempts to acquire a read-write lock for write access.

Syntax

```
tis_write_trylock(
                lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

C Binding

#include <tis.h>

```
int
tis_write_trylock (
        tis_rwlock_t    *lock);
```

Arguments

lock
Address of the read-write lock to be acquired for write access.

Description

This routine attempts to acquire a read-write lock for write access. The routine attempts to immediately acquire the lock. If the lock is acquired, zero (0) is returned. If the lock is held by another thread (for either read or write access), [EBUSY] is returned and the calling thread does not wait for the write-access lock to be acquired.

Note that it is a coding error to attempt to acquire the lock for write access if the lock is already held by the calling thread. (However, this routine returns [EBUSY] anyway, because no ownership error-checking takes place.)

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion; the lock is acquired for write access.
[EBUSY]	The lock was not acquired for write access, as it is already held by another thread.

Associated Routines

```
tis_read_lock( )
tis_read_trylock( )
tis_read_unlock( )
tis_rwlock_destroy( )
tis_rwlock_init( )
tis_write_lock( )
tis_write_unlock( )
```

tis_write_unlock

Unlocks a read-write lock that was acquired for write access.

Syntax

```
tis_write_unlock(
        lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

C Binding

```
#include <tis.h>

int
tis_write_unlock (
        tis_rwlock_t    *lock);
```

Arguments

lock
Address of the read-write lock to be unlocked.

Description

This routine unlocks a read-write lock that was acquired for write access.

Upon completion of this routine, any thread waiting to acquire the lock for read access will have those acquisitions granted. If no threads are waiting to acquire the lock for read access, then a thread waiting to acquire it for write access will have that acquisition granted.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type error. Possible return values are as follows:

tis_write_unlock

Return	Description
0	Successful completion.
[EINVAL]	The value specified by *lock* is invalid.

Associated Routines

```
tis_read_lock( )
tis_read_trylock( )
tis_read_unlock( )
tis_rwlock_init( )
tis_rwlock_destroy( )
tis_write_lock( )
tis_write_trylock( )
```

Part IV

Appendixes

Part IV contains appendixes that provide supporting information about DECthreads, such as operating system-specific information, debugging information, and additional reference information.

A

Considerations for DIGITAL UNIX Systems

This appendix discusses DECthreads issues specific to DIGITAL UNIX systems.

A.1 Overview

The DIGITAL UNIX operating system supports multiple concurrent "execution contexts" within a process. DECthreads uses these kernel execution contexts to implement user threads. One important benefit of this is that user threads can run simultaneously on separate processors in a multiprocessor system. Review Section 3.1 for tips for ensuring that your application will work correctly with kernel threads and multiprocessing.

A.2 Building DECthreads Applications

The following sections discuss points to consider when building using DECthreads.

A.2.1 Including DECthreads Header Files

Include one of the DECthreads header files shown in Table A–1 in your program to use the appropriate DECthreads library.

Table A–1 DECthreads Header Files

Header File	Interface
`pthread.h`	POSIX.1c routines
`tis.h`	Thread-independent services routines

Do not include more than one of these header files in your module.

A.2.2 Building Multithreaded Applications from DECthreads Libraries

Multithreaded applications are built using shared libraries. For a description of shared libraries, see the *DIGITAL UNIX Programmer's Guide*.

Table A–2 contains the libraries supported for multithreaded programming.

Table A–2 DIGITAL UNIX Shared Libraries for Multithreaded Programs

`libmach.so`	Shared version of threads support library. Direct use of mach interfaces is not supported.
`libpthread.so`	Shared version of the base pthreads package. Requires `libmach.so`, `libexc.so`, and `libc.so`
`libexc.so`	Shared version of DIGITAL UNIX exception support package.
`libpthreads.so`	Shared version of DECthreads "legacy" package, implementing the DIGITAL-proprietary CMA (or **cma**) and POSIX 1003.4a/Draft 4 (or **d4**) interfaces.
`libc.so`	Shared version of the C language run-time library (`libc.so`).

Build a multithreaded application using shared versions of `libexc`, `libmach`, `libpthread`, and `libc` using this command:

```
% cc -o myprog myprog.c -pthread
```

If you use a compiler front-end or other (not C) language environment that does not support the `-pthread` compilation switch, you must use the `-D_REENTRANT` compilation switch.

A.2.3 Linking Multithreaded Shared Libraries

The `ld` command does not support the `-pthread` or `-threads` switch. Instead, you must list the individual libraries in the proper order.

For libraries that use only the **pthread** interface, use the following:

```
ld <...> -lpthread -lexc
```

If using the **cma** or **d4** interfaces, use the following:

```
ld <...> -lpthreads -lpthread -lexc
```

Note

If you build software (whether applications or libraries) that links against the static version of a DECthreads library, you must not require developers who use your software to link against any library that dynamically loads any DECthreads shared library, such as `libpthread.so`.

A.2.4 Compiling Applications With the tis Interface

Applications that use the DIGITAL-proprietary thread-independent services (or **tis**) interface should include the `tis.h` header file and link against the shared C run-time library (`libc.so`).

A.3 Two-Level Scheduling on DIGITAL UNIX Systems

Under DIGITAL UNIX Version 4.0 and later, DECthreads implements a new scheduling model, referred to as **two-level scheduling**. DECthreads schedules “user threads” onto kernel execution contexts (often known as “kernel threads” or “virtual processors”), just as DIGITAL UNIX schedules processes onto the processors of a multiprocessing machine.

A user thread is executed on a kernel thread until it blocks or exhausts its timeslice quantum. Then, DECthreads schedules a new user thread to run. While DECthreads is scheduling user threads onto kernel threads, the DIGITAL UNIX kernel is independently scheduling those kernel threads to run on physical processors. The term two-level scheduling refers to this relationship.

This division allows most thread scheduling to take place completely in user mode, without the intervention of the kernel. Since a thread context switch does not involve any privileged information, it can be done much more efficiently in user mode.

The key to making the two-level scheduling model work is efficient two-way communication between DECthreads and the DIGITAL UNIX kernel. When a thread blocks in the kernel, the DECthreads scheduler is notified so that it can schedule another thread to take advantage of the idle kernel thread. This mechanism, sometimes referred to as an **upcall**, is inspired by original research on scheduler activations at the University of Washington. (See *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism* by Anderson, Bershad, Lazowska, and Levy; ACM Operating Systems Review Volume 25, Number 5, *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, October 13-16, 1991*).

A.3.1 DECthreads Use of Kernel Threads

DIGITAL UNIX kernel threads are created as they are needed by the application. The number of kernel threads that DECthreads creates is limited by normal DIGITAL UNIX configuration limits regarding user and system thread creation. Normally, however, DECthreads creates one kernel thread for each actual processor on the system and the kernel creates an additional kernel thread on behalf of the process for bookkeeping operations.

DECthreads does not delete these kernel threads or let them terminate. Kernel threads not currently needed are retained in an idle state until they are needed again. When the process terminates, all kernel threads in the process are reclaimed by the kernel.

The DECthreads scheduler can schedule any user thread onto any kernel thread. Therefore, a user thread can run on different kernel threads at different times. Normally, this should pose no problem. However, for example, the kernel thread ID as reported by the dbx or Ladebug debuggers can change at any time.

A.3.2 Support for Real-Time Scheduling

DECthreads supports DIGITAL UNIX real-time scheduling. This allows you to set the scheduling policy and priority of threads. By default, threads are created using process contention scope. This means that the full range of POSIX.1c scheduling policy and priority is available. However, threads running in process contention scope do not preempt lower-priority threads in another process. For example, a thread in process contention scope with `SCHED_FIFO` policy and `PRI_FIFO_MAX` priority will not preempt a thread in another process running with `SCHED_FIFO` and `PRI_FIFO_MIN`.

In contrast, system contention scope means that each thread created by the program has a direct and unique binding to one kernel execution context. A system contention scope thread competes against all threads in the system and will preempt any thread with lower priority. For this reason, the priority range of threads in system contention scope is restricted unless running with root privilege.

Specifically, a thread with `SCHED_FIFO` policy cannot run at a priority higher than 18 without privilege, since doing so could lock out all other users on the system until the thread blocked. Threads at any other scheduling policy (including `SCHED_RR`) can run at priority 19 because they are subject to periodic timeslicing by the system. For more information, see the *DIGITAL UNIX Realtime Programming Guide*.

If your program lacks necessary privileges, attempting to call the following routines for a thread in system contention scope returns the error value [EPERM]:

`pthread_attr_setschedpolicy( )`	(*Error returned by* `pthread_create( )` *at thread creation*)
`pthread_attr_setschedparam( )`	(*Error returned by* `pthread_create( )` *at thread creation*)
`pthread_setschedparam( )`	

Prior to DIGITAL UNIX Version 4.0, all threads used only system contention scope. In DIGITAL UNIX Version 4.0, all threads created using the **pthread** interface, by default, have process contention scope.

A.4 Thread Cancelability of System Services

DIGITAL UNIX supports the required system cancelation points specified by the POSIX.1c standard. In addition, critical non-POSIX functions supported by DIGITAL UNIX (such as `select( )`) have also been defined as cancelation points.

For legacy multithreaded applications, note that threads created using the **cma** or **d4** interfaces will not be cancelable at any system call. (Here “system call” means any function without the `pthread_` prefix.) If system call cancelation is required, you must write code using the DECthreads **pthread** interface. None of the system calls should be called with asynchronous cancelation enabled.

For more information, see Section 2.3.7.

A.4.1 Current Cancelation Points

The following functions are cancelation points:

```
accept()
aio_suspend()
close()
connect()
creat()
fcntl()
fsync()
mq_receive()
q_send()
msync()
nanosleep()
open()
pause()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_delay_np()
pthread_join()
pthread_testcancel()
read()
readv()
recv()
recvfrom()
recvmsg()
select()
sem_wait()
send()
sendmsg()
sendto()
shutdown()
sigwaitinfo()
sigsuspend()
sigtimedwait()
sigwait()
sleep()
system()
tcdrain()
wait()
waitpid()
write()
writev()
```

A.4.2 Future Cancelation Points

The following list contains POSIX functions that are *not* cancelation points in this release of DECthreads but might be in a future release (as specified by the POSIX.1c standard). Please code your programs accordingly.

```
closedir()
ctermid()
fclose()
fflush()
fgetc()
fgets()
fopen()
fprintf()
fputc()
fputs()
fread()
freopen()
fscanf()
fseek()
ftell()
fwrite()
getc()
getc_unlocked()
getchar()
getcwd()
getgrgid()
getgrgid_r()
getgrnam()
getgrnam_r()
getlogin()
getlogin_r()
getpwnam()
getpwnam_r()
gets()
lseek()
opendir()
perror()
printf()
putc()
putc_unlocked()
putchar()
putchar_unlocked()
puts()
readdir()
remove()
rename()
rewind()
rewinddir()
scanf()
tmpfile()
tmpname()
ttyname()
ttyname_r()
ungetc()
unlink()
```

Note that appropriate non-POSIX functions that do not appear in the preceding list might become cancelation points in the future. DIGITAL UNIX will also implement new cancelation points, as specified by future revisions of the relevant formal or consortium standard bodies.

A.5 Using Signals

This section discusses signal handling based on the POSIX.1c standard.

DIGITAL UNIX Version 4.0 introduces the full POSIX.1c signal model. In previous versions, "synchronous" signals (those resulting from execution errors, such as SIGSEGV and SIGILL) could have different signal actions for each thread. Prior to DIGITAL UNIX Version 3.2, all threads shared a common, processwide signal mask, which meant one thread could not receive a signal while another had the signal blocked.

Under DIGITAL UNIX Version 4.0 and later, all signal actions are processwide. That is, when any thread uses *sigaction* or equivalent to set a signal handler, or to modify the signal action (for example, to ignore a signal), that action will affect all threads. Each thread has a private signal mask so that it can block signals without affecting the behavior of other threads.

Prior to DIGITAL UNIX Version 4.0, asynchronous signals were processed only in the main thread. In DIGITAL UNIX Version 4.0, any thread that doesn't have the signal masked can process the signal. To support binary compatibility, for a thread created by a DECthreads **cma** or **d4** interface routine, the thread starts with all asynchronous signals blocked.

A.5.1 POSIX sigwait Service

The POSIX 1003.1c `sigwait( )` service allows any thread to block until one of a specified set of signals is delivered. A thread can wait for any of the asynchronous signals except for SIGKILL and SIGSTOP.

For example, you can create a thread that blocks on a `sigwait( )` routine for SIGINT, rather than handling a Ctrl/C in the normal way. This thread could then cancel other threads to cause the program to shut down the current activities.

Following are two reasons for avoiding signals:

- Signals cannot be used in a modular way in a multithreaded program.
- Signals, used as an asynchronous programming technique, are unnecessary in a multithreaded program.

In a multithreaded program, signal handlers cannot be used in a modular way because there is only one signal handler routine for all of the threads in an application. If two threads install different signal handlers for the signal, all threads will dispatch to the last handler when they receive the signal.

Most applications should avoid using asynchronous programming techniques in conjunction with threads. For example, techniques that rely on timer and I/O signals are usually more complicated and errorprone than simply waiting synchronously within a thread. Furthermore, most of the threads services are not supported for use in signal handlers, and most run-time library functions cannot be used reliably inside a signal handler.

Some I/O intensive code may benefit from asynchronous I/O, but these programs will generally be more difficult to write and maintain than “pure” threaded code.

A thread should not wait for a synchronous signal. This is because synchronous signals are the result of an error during the execution of a thread, and if the thread is waiting for a signal, then it is not executing. Therefore, a synchronous signal cannot occur for a particular thread while it is waiting, and the thread will wait forever.

The POSIX.1c standard requires that the thread block the signals for which it will wait before calling `sigwait( )`.

A.5.2 Handling Synchronous Signals as Exceptions

For the signals traditionally representing synchronous errors in the program, DECthreads catches the signal and converts it into an equivalent exception. This exception is then propagated up the call stack in the current thread and can be caught and handled using the normal exception catching mechanisms.

Table A–3 lists DIGITAL UNIX signals that are reported as DECthreads exceptions by default. If any thread declares an action for one of these signals (using `sigaction(2)` or equivalent), no thread in the process can receive the exception.

Table A–3 Signals Reported as Exceptions

Signal	Exception
SIGILL	`pthread_exc_illinstr_e`
SIGIOT	`pthread_exc_SIGIOT_e`
SIGEMT	`pthread_exc_SIGEMT_e`
SIGFPE	`pthread_exc_aritherr_e`
SIGBUS	`pthread_exc_illaddr_e`

(continued on next page)

Table A–3 (Cont.) Signals Reported as Exceptions

Signal	Exception
SIGSEGV	pthread_exc_illaddr_e
SIGSYS	pthread_exc_SIGSYS_e
SIGPIPE	pthread_exc_SIGPIPE_e

A.6 Thread Stack Guard Areas

When creating a thread based on a thread attributes object, DECthreads potentially rounds up the value specified in the object's guardsize attribute. DECthreads does so based on the value of the configurable system variable PAGESIZE (see <sys/mman.h>. The default value of the guardsize attribute in a thread attributes object is a number of bytes equal to setting of PAGESIZE.

A.7 Dynamic Activation

Dynamic activation of the DECthreads run-time environment, or of code that depends on DECthreads, is currently not supported.

B

Considerations for OpenVMS Systems

This appendix discusses DECthreads issues and restrictions specific to the OpenVMS operating system.

B.1 Overview

Under OpenVMS, DECthreads offers these application programming interfaces:

- IEEE POSIX 1003.1c-1995 (or POSIX.1c) style interface

  Under OpenVMS Version 7.0 and later, DECthreads offers a POSIX.1c standard style (or **pthread**) interface. The DECthreads **pthread** interface is the most portable, efficient, and powerful interface that OpenVMS offers for building multithreaded programs and applications.

- Thread-independent services interface

  Under OpenVMS Version 7.0 and later, DECthreads includes the thread-independent services (or **tis**) interface. The **tis** interface provides services that support development of thread-safe libraries and APIs, whose routines do not use threads.

  Thread synchronization can involve significant run-time cost, which is undesirable in the absence of threads. The **tis** interface enables you both to build thread-safe libraries that are efficient in a single-threaded environment and to provide the necessary thread synchronization and coherence of memory access in a multithreaded environment.

  When DECthreads is not active within the process, the **tis** routines execute only the minimum steps necessary. That is, code running in a single-threaded environment is not burdened by the run-time synchronization required when the same code is run in a multithreaded environment.

B.2 Compiling Under OpenVMS

The DECthreads C language header files shown in Table B–1 provide interface definitions for the DECthreads **pthread** and **tis** interfaces.

Table B–1 DECthreads Header Files

Header File	Interface
`pthread.h`	POSIX.1c style routines
`tis.h`	DIGITAL-proprietary thread-independent services routines

Include only *one* of these header files in your module.

Special compiler definitions are not required when compiling threaded applications that use the **pthread** interface or the **tis** interface.

B.3 Linking OpenVMS Images

DECthreads is supplied only as shareable images. It is not supplied as object libraries.

When you link an image that calls DECthreads routines, you must link against the appropriate images listed in Table B–2.

Table B–2 DECthreads Images

Image	Routine Library
PTHREAD$RTL.EXE	POSIX.1c style interface
CMA$TIS_SHR.EXE	Thread-independent services

The image files PTHREAD$RTL.EXE, CMA$TIS_SHR.EXE, CMA$RTL.EXE, and CMA$LIB_SHR.EXE are included in the IMAGELIB library, making it unnecessary to specify those images (unless you are using the /NOSYSLIB switch with the linker) in a Linker options file.

When you link an image that utilizes the CMA$OPEN_LIB_SHR.EXE and CMA$OPEN_RTL.EXE images, they must be specified in a Linker options file.

Note

While this version of DECthreads for OpenVMS supports upward compatibility of source and binaries for the DECthreads **d4** interface, DECthreads does not support upward compability for object files.

For instance, under OpenVMS V7.0 and higher, to link object files that were compiled under OpenVMS V6.2, follow these steps:

1. Copy CMA$OPEN_RTL.EXE from SYS$SHARE for OpenVMS V6.2 into the directory with your object files compiled under the current OpenVMS version. During linking, it provides the locations of the transfer vector entries (OpenVMS VAX) or symbol vector entries (OpenVMS Alpha) in CMA$OPEN_RTL.EXE for the the older OpenVMS version.
2. Instead of specifying `SYS$SHARE:CMA$OPEN_RTL/SHARE` in your link options files, specify `CMA$OPEN_RTL/SHARE`. Be careful about the placement of this option in the options file—it should perhaps be placed at the beginning, or close to it, if you are including other images that link against PTHREAD$RTL.
3. Link your program.
4. Delete CMA$OPEN_RTL.EXE from your object directory for the current OpenVMS version.

B.4 Using DECthreads with AST Routines

An asynchronous system trap, or AST, is an OpenVMS mechanism for reporting an asynchronous event to a process. The following are restrictions concerning the use of ASTs with DECthreads:

- Avoid blocking ASTs using any mechanism other than $SETAST.
- Be aware that blocking ASTs in one thread may prevent delivery of ASTs that are actually intended for other threads. Therefore, it is best to avoid blocking ASTs for an extended period of time. Also, it is best to avoid calling DECthreads functions that may block the thread while it has disabled ASTs.
- Do not call DECthreads routines, except those that have the `_int` (interrupt) suffix in their names, from within an AST routine. Calling any other DECthreads routines from code running in an AST can be unreliable or cause unexpected behavior.

- For OpenVMS Alpha, ASTs are handed off to DECthreads by the operating system. This allows ASTs to be delivered in the context of the appropriate thread. On a multiprocessor machine it may be possible to have a thread executing an AST routine in parallel with another thread's execution. When a thread disables ASTs, not only does it block out its own ASTs, but it prevents delivery of any ASTs that do not specifically belong to a particular thread as well.

B.5 Dynamic Activation

Dynamic activation of DECthreads (or images that depend on DECthreads) is currently not supported.

B.6 Declaring an OpenVMS Condition Handler

This section discusses a restriction on declaring an OpenVMS condition handler while using DECthreads exceptions and DECthreads behavior when a condition is signaled.

The following are three ways to declare an OpenVMS condition handler:

- Calling VAXC$ESTABLISH (from a program written in C)
- Calling LIB$ESTABLISH
- Placing the address of the condition handler directly into the stack frame (from a program written in VAX MACRO or VAX BLISS)

Do not declare an OpenVMS condition handler within a DECthreads `TRY/ENDTRY` exception block. Doing so deletes without notification any handler that exists for the current procedure. If your code declares a condition handler within the `TRY/ENDTRY` block, DECthreads exceptions will not be handled correctly until the next `TRY` statement is executed. The `TRY` statement restores the DECthreads condition handler.

On OpenVMS VAX, you can declare a condition handler outside of a `TRY/ENDTRY` block with no restrictions. If a condition handler has already been declared when you execute a `TRY` statement, DECthreads saves the previous handler address. When DECthreads receives a condition it does not handle (including SS$_UNWIND, SS$_DEBUG, or a condition code that does not have a SEVERE severity), DECthreads invokes the saved condition handler. The condition handler will be reestablished when the `TRY` block exits.

B.7 Thread Cancelability of System Services

On OpenVMS Alpha, system calls are now cancelation points for threads created using the POSIX 1003.1c style interface. System calls are *not* cancelation points for threads in legacy multithreaded applications that were created using the DIGITAL-proprietary CMA (or **cma**) or POSIX 1003.4a /Draft 4 (or **d4**) interfaces. None of the system calls should be called with asynchronous cancelation enabled. For more information, see Section 2.3.7.

B.8 Using OpenVMS Alpha 64-Bit Addressing

On OpenVMS Alpha, DECthreads supports the use of 64-bit addressing in the **pthread** interface only. When compiling with the following command, the `pthread_join( )` function returns a 64-bit `void *` value as the result:

```
$ CC/POINTER_SIZE=LONG
```

You can also use `pthread_join64( )` or `pthread_join32( )` to specify the length in bits of the return value.

Note that no other DECthreads functions have special 64-bit versions because the OpenVMS Alpha calling standard always supports 64-bit arguments and return values.

B.9 DECthreads Condition Values

Table B–3 lists the DECthreads condition values for OpenVMS systems and provides an explanation and user action.

Table B–3 DECthreads Condition Values

Condition Value	Explanation and User Action
CMA$_EXCCOP	Exception raised; OpenVMS condition code follows. **Explanation**: One of the DECthreads exception commands (RAISE or RERAISE) raised or reraised an exception condition originating outside the DECthreads library. The secondary condition code in the signal vector will be the original code. **User Action**: See the documentation for the software that your program is calling to determine the reason for this exception.

(continued on next page)

Table B–3 (Cont.) DECthreads Condition Values

Condition Value	Explanation and User Action
CMA$_EXCCOPLOS	Exception raised; some information lost. **Explanation**: CMA$_EXCCOPLOS is nearly the same as CMA$_EXCCOP except that DECthreads determined that the copied signal vector may contain address arguments. However, the address arguments may not be valid when the stack is unwound and the condition is resignaled. Therefore, DECthreads clears the condition codes' arguments in the resignaled vector. In most cases, DECthreads knows that SS$_ code arguments are "safe" and will not clear them. Most other codes with arguments will result in CMA$_EXCCOPLOS. **User Action**: See the documentation for the software that your program is calling to determine the reason for this exception.
CMA$_EXCEPTION	Exception raised; address of exception object is *object-address*. **Explanation**: This condition is used as the primary condition to `RAISE` an address-type DECthreads exception. The condition is signaled with a single argument containing the address of the `EXCEPTION` structure. There is no support for interpreting this value. It is only meaningful to the facility that defined the `EXCEPTION`. It is not good programming practice to let an address exception propagate outside the facility that raised it. There is no support for getting message text, and it cannot be interpreted by other facilities. **User Action**: None.

B.10 Two-Level Scheduling on OpenVMS Alpha Systems

This section applies to OpenVMS Alpha systems only.

Under OpenVMS Alpha Version 7.0 and later, DECthreads implements a new scheduling model, referred to as two-level scheduling. This model is based on the concept of **virtual processors**. Virtual processors are implemented as a result of using kernel thread technology in the OpenVMS Alpha operating system.

DECthreads schedules threads onto virtual processors similar to the way that OpenVMS schedules processes onto the processors of a multiprocessing machine. Thus, to the DECthreads runtime environment, a scheduled thread is executed on a virtual processor until it blocks or until it exhausts its timeslice quantum; then DECthreads schedules a new thread to run.

While DECthreads schedules threads onto virtual processors, the OpenVMS scheduler also schedules virtual processors to run on physical processors. The term two-level scheduling derives from this relationship.

The two-level scheduling model provides these advantages:

- It allows most thread scheduling to take place completely in user mode–that is, without the intervention of the OpenVMS scheduler. Because a thread context switch does not involve any privileged information (rather, only a swapping of registers), it can be done much more efficiently in user mode than a context switch involving the operating system.
- It allows the OpenVMS scheduler to schedule virtual processors onto separate processors of a multiprocessing machine. This allows a process using DECthreads to take advantage of the full resources of a multiprocessor machine.

The key to making the two-level scheduling model work is the upcall mechanism. An upcall is a communication between the OpenVMS scheduler and the DECthreads scheduler. When an event occurs that affects the scheduling of a thread, such as blocking for a system service, the OpenVMS scheduler calls “up” to the DECthreads scheduler to notify it of the change in the thread’s status.

This upcall gives DECthreads the opportunity to schedule another thread to run on the virtual processor in place of the blocking thread, rather than to allow the virtual processor itself to block, which would deny that resource to other threads in the process.

Upcalls are typically arranged in pairs, with an “unblock” upcall corresponding to each “block” upcall. The unblock upcall notifies DECthreads that a previously blocked thread is now eligible to run again. DECthreads schedules that thread to run when it is appropriate, given the thread’s scheduling policy and priority.

B.10.1 Linker Options to Specify Image's Use of Kernel Threads

In OpenVMS Alpha Version 7.1 and later, the linker supports the /THREADS_ENABLE (or /NOTHREADS_ENABLE) qualifier for specifying the role of kernel threads in the resulting image. Use this qualifier to specify that the image controls whether the process can create multiple kernel threads and whether the OpenVMS Alpha kernel's support for DECthreads upcalls is enabled. If this qualifier is not specified, the default linker setting is /NOTHREADS_ENABLE, which results in an image that behaves as under OpenVMS Alpha Version 6.

The /THREADS_ENABLE qualifier takes two keyword arguments, MULTIPLE_KERNEL_THREADS and UPCALLS. Table B–4 summarizes the allowable combinations of these keywords and their effects.

Table B–4 Results of Keyword Arguments to /THREADS_ENABLE Qualifier

Keywords Specified	Result
/NOTHREADS_ENABLE	No kernel threads support for DECthreads
/THREADS_ENABLE or: /THREADS_ENABLE=(MULTIPLE_KERNEL_THREADS,UPCALLS)	Full kernel threads support for DECthreads, including the ability to run multiple use threads simultaneously on different CPUs on a multiprocessor machine
/THREADS_ENABLE=MULTIPLE_KERNEL_THREADS	Same behavior as if /NOTHREADS_ENABLE is specified (without support for upcalls, DECthreads cannot reliably use multiple kernel threads)
/THREADS_ENABLE=UPCALLS	Upcall support for DECthreads (such as making system calls thread-synchronous), but restricts the process's threads to one CPU on a multiprocessor machine

Note

Under no circumstances should a process explicitly create kernel threads. DECthreads creates them as needed when allowed to do so. Explicit creation of kernel threads by an application disrupts the operation of the DECthreads runtime environment and causes incorrect and/or unreliable application behavior.

B.10.2 Setting Kernel Threads Support in Existing Images

Under OpenVMS Alpha only, use the THREADCP tool to set or show the kernel threads features described above for an existing image. The tool provides the ability to enable, disable, and show the state of the thread control bits in an image's header.

The THREADCP command verb is not part of the normal set of DCL commands. To use the tool, you must define the command verb before invoking it.

In a THREADCP command, an image file name is a required parameter for use with all supported qualifiers. THREADCP supports abbreviations to the first character for all qualifiers and parameters. When the SHOW qualifier is used alone with the THREADCP command, the file name can contain wildcard characters.

After you define the THREADCP command verb, an image's thread control bits can be set or cleared using the /ENABLE and /DISABLE qualifiers, respectively. To do so, specify the name of each thread control bit to be enabled, disabled, or shown. One or both thread control bits can be specified. The user must have write access to the image file.

If no thread control bit is specified, the THREADCP default is to operate on both bits. If the image is currently being executed or is installed, it cannot be modified.

B.10.2.1 Examples

This command defines the THREADCP command verb:

```
$ SET COMMAND SYS$UPDATE:THREADCP.CLD
```

This command displays the current settings of both thread control bits for the image TEST.EXE:

```
$ THREADCP/SHOW TEST.EXE
```

This command displays the current settings of both thread control bits for all SYS$SYSTEM images:

```
$ THREADCP/SHOW SYS$SYSTEM:*
```

This command sets both thread control bits explicitly for the image TEST.EXE:

```
$ THREADCP/ENABLE=(MULTIPLE_KERNEL_THREADS, UPCALLS) TEST.EXE
```

This command clears both thread control bits explicitly for the image TEST.EXE:

```
$ THREADCP/DISABLE=(MULTIPLE_KERNEL_THREADS, UPCALLS) TEST.EXE
```

B.10.3 Querying and Setting Kernel Threads Features

On OpenVMS Alpha systems, a program can call the $GETJPI system service and specify the appropriate MULTITHREAD item code to determine whether kernel threads are in use. The return values have the same meanings as are defined for the MULTITHREAD system parameter, as summarized in Table B–5.

Table B–5 Return Values from $GETJPI System Service

Value	Description
0	Both Thread Manager upcalls and the creation of multiple kernel threads are disabled.
1	Thread Manager upcalls are enabled; the creation of multiple kernel threads is disabled.
2 through 16	Both Thread Manager upcalls and the creation of multiple kernel threads are enabled. The number specified represents the maximum number of kernel threads that can be created for a single process.

B.10.4 Creation of Virtual Processors

Virtual processors are created as they are needed by the application. For a multithreaded application, the number of virtual processors that DECthreads creates is limited by the SYSGEN parameter MULTITHREAD. This parameter is typically set to the number of processors present in the system.

In general, there is no reason to create more virtual processors than there are physical processors; that is, the virtual processors would contend with each other for the physical processors and cause unnecessary overhead. Regardless of the value of the MULTITHREAD parameter, DECthreads creates no more virtual processors than there are user threads (excluding DECthreads internal threads).

DECthreads does not delete virtual processors or let them terminate. They are retained in the HIB idle state until they are needed again. During image rundown, they are deleted by OpenVMS.

The DECthreads scheduler can schedule any user thread onto any virtual processor. Therefore, a user thread can run on different kernel threads at different times. Normally, this should pose no problem; however, for example, a user thread's PID (as retrieved by querying the system) can change from time to time.

B.10.5 Delivery of ASTs

When a user mode AST becomes deliverable to a DECthreads process, the OpenVMS scheduler makes an upcall to DECthreads, passing the information that is required to deliver the AST (service routine address, argument, and target user thread ID). DECthreads stores this information and queues the AST to be delivered to the appropriate user thread. That thread is made runnable (if it is not already), and executes the AST routine the next time it is scheduled to run. This means the following:

- A per-thread AST will interrupt the user thread that requested it, regardless of on which virtual processor the thread is running.
- The AST will be run at the priority of the target thread, so that low-priority threads' ASTs do not preempt or interfere with the execution of high-priority threads.
- The AST routine executes in the context of the target thread, so that the danger of surprise stack overflows is diminished, and stack-walks and exception propagation work as they should.

In addition to per-thread ASTs, there are also user mode ASTs that are directed to the process as a whole, or to no thread in particular, or to a thread that has since terminated. These "process" ASTs are queued to the initial thread, making the thread runnable in a fashion similar to per-thread ASTs. They are executed in the context of the initial thread, for the following reasons:

- The initial thread has an expandable stack, unlike the other threads, which minimizes the danger of stack space problems.
- Any code that is making assumptions about specific characteristics of AST delivery is most likely running in the initial thread, so delivering the AST to the initial thread is least likely to cause problems.
- To ensure that the process ASTs are executed promptly, the initial thread gets a boost to the top scheduling priority . Because these ASTs cannot be associated with a particular thread, their priority cannot be assessed, so it is important that they be delivered promptly in the event that a high-priority thread is waiting to be signaled by one of them.

Note

In OpenVMS Version 7.0 and later, all ASTs are directed to the process as a whole. In future releases, AST delivery will be made per thread as individual services are updated.

The following implications must be considered for application development:

- If an application makes heavy use of ASTs, it can starve the initial thread to a degree, because only that thread executes the ASTs that are directed to the entire process. (This is in contrast with the behavior prior to OpenVMS Version 7.0 of starving all threads equally).
- There are also implications for controlling AST delivery. $SETAST generates an upcall similar to the one for AST delivery. This allows DECthreads to note the request by a thread to block (or unblock) AST delivery. When a thread has requested that ASTs be blocked, it will not receive delivery of any per-thread ASTs; nor will the process receive delivery of any process ASTs. This is, in effect, the behavior of prior to OpenVMS Version 7.0, except that a second thread cannot undo a block requested by a previous thread. Avoid using any mechanism other than $SETAST to block ASTs; it will interfere with the process as a whole and may produce undesirable results.
- Another implication is that it is possible for a thread to be executing on one virtual processor at the same time that an AST is executing on another virtual processor. In general, this should not pose a significant problem for multithreaded applications. Such applications should have already minimized their AST use, since ASTs and threads can be difficult to use together reliably.

  In addition, AST routines should already be performing only atomic operations, since thread synchronization is not available to code executing at AST level. Any “legacy” code (such as a nonthreaded application using threaded libraries) is executed in the initial thread, where the normal assumptions about AST delivery are maintained. If a piece of code cannot tolerate concurrent execution with an AST routine, it should disable AST delivery during its execution.

B.10.6 Blocking System Services

All blocking system services are thread synchronous in OpenVMS Alpha Version 7.0 and later. That is, they block only the calling thread. When the thread is to be blocked by the system service, the OpenVMS scheduler makes an upcall to allow DECthreads to schedule another user thread to execute. Therefore, only the calling thread is blocked, all other threads are unaffected, and the process continues running. When the service completes, the thread is awakened by means of another upcall, and DECthreads schedules it to run again at the thread’s next opportunity.

This applies to all "W" forms of system services. For example, $QIOW, $END_TRANSW, and $GETJPIW. Additionally, this applies to the following event flag services: $WAITFR, $WFLAND, and $WFLOR.

B.10.7 $HIBER and $WAKE

$HIBER and $WAKE result in upcalls to DECthreads. When a user thread calls $HIBER, only that thread is blocked; all other threads continue running. The blocking thread is immediately unscheduled and another thread is scheduled to run instead. When a thread (or another process) calls $WAKE, all hibernating threads are awakened.

Prior to OpenVMS Version 7.0, a thread that called a $HIBER (or called a library routine that eventually resulted in a call to $HIBER) would cause the whole process to hibernate for a brief period whenever that thread was scheduled to "run." Also, with multiple threads in calls to $HIBER simultaneously, there was no reliable way to wake the threads (or a specific thread); the next hibernating thread to be scheduled would awaken, and any other threads would continue to sleep.

In OpenVMS Alpha Version 7.0 and later, these problems have been resolved. However, this new behavior has some other effects. For instance, hibernation-based services, such as LIB$WAIT and the C RTL `sleep( )` routine, may be prone to premature completion. If the service does not validate its wakeup (that is, ensure that enough time has passed or that there is some other reason for it to return), then it will be prone to this problem, as are the above services, since they do not perform such wake-up validation.

B.10.8 Event Flags

All event flags are shared by all threads in the process. Therefore, it is possible for different threads' use of the same event flag to cause interference. That is, if two threads use the same event flag in calls to different system services, whichever service completes first will cause both threads to awaken, even though the other service has not completed. This situation can be resolved by specifying an I/O status block (IOSB) for those system services that use them. When an IOSB is present, the blocked thread will not be awakened when the event flag is set, unless the IOSB has also been written.

Note that a DECthreads process is rarely in LEF state. In general, instead of blocking for an event flag wait, DECthreads schedules another thread to be run. If no threads are available, DECthreads schedules a "null" thread, which places the virtual processor in HIB state until it is needed to execute a thread.

Note that no upcall is made for waits on a common event flag. If a thread waits on a common event flag, the virtual processor blocks until the wait is satisfied. (On a uniprocessor, this is most likely to block all threads.)

B.10.9 Interactions with OpenVMS

There are several interactions with the OpenVMS operating system that should be noted:

- Like system service calls, pagefault waits are thread synchronous. When a thread incurs a "hard" pagefault (reading the page from disk), an upcall to DECthreads takes place, and DECthreads places the thread in a blocked state. DECthreads schedules another thread to run in its place.

  When the pagefault resolution is complete, another upcall occurs, and DECthreads schedules it to run at its next opportunity. It is possible for multiple threads to take faults on the same page at approximately the same time. Each thread is blocked, in turn, and becomes unblocked when the page becomes valid.

- Most OpenVMS system services cannot themselves support being called by multiple threads concurrently. Therefore, calls to OpenVMS system services are serialized using a mechanism called the inner-mode semaphore. If one thread attempts to call a system service while another thread is in the middle of calling a system service, the second thread is blocked by an upcall until the first thread completes its service call.

- DECthreads timeslicing changed slightly for OpenVMS Alpha Version 7.0 and later. Prior to Version 7.0, the DECthreads timeslicer was implemented using an OpenVMS timer. This caused a DECthreads scheduler AST to be delivered to the process at regular wall-clock time intervals. While running on wall-clock time was a necessary evil (to support the interruption of system service blocks), this timeslice mechanism had several drawbacks.

  In OpenVMS Version 7.0 and later, timeslicing is implemented as an upcall to DECthreads that is delivered after the thread has consumed a sufficient amount of CPU time. Thus, when no threads are running, no timeslicing takes place.

B.10.10 Image Exit

In multithreaded processes, image exit occurs as follows: $EXIT does not immediately invoke exit handler routines. Instead, it results in an upcall that causes DECthreads to schedule a special thread to execute the exit-handler routines. $EXIT then calls `pthread_exit( )` to terminate the calling thread. This allows the calling thread to release any resources that it might be holding.

To avoid possible deadlocks, the exit-handler routines are executed in a separate thread. For example, if a thread calls $EXIT while holding a mutex that is required by an exit-handler routine, then that routine causes the thread to block forever, as it waits for a mutex that it already holds. Because the exit-handler routine executes in a separate thread, it can block while the thread holding the mutex cleans up.

$FORCEX works in an analogous fashion. Instead of invoking $EXIT directly, it causes an upcall that allows DECthreads to release the exit-handler thread.

DCL Ctrl/Y continues to work as it always has on multithreaded applications. However, typing EXIT or issuing any other command that invokes a new image causes the $FORCEX upcall. While this is an improvement in many cases over the behavior prior to OpenVMS Version 7.0, it does not guarantee that the multithreaded application will exit.

For example, if the application is deadlocked, holding a resource required by one of the exit handler's routines, the application will continue to hang, even after typing Ctrl/Y and EXIT. In these cases, type Ctrl/Y and STOP to terminate the application without running exit handlers. *Note that doing so causes the application to be unable to clean up, and it may leave data files and the terminal in an inconsistent state.*

B.10.11 SYSGEN Parameter MULTITHREAD

The SYSGEN parameter MULTITHREAD limits the maximum number of kernel threads per process. It is set by AUTOGEN to the number of CPUs on the system. If MULTITHREAD is set to zero (0), two-level scheduling support is disabled, and DECthreads reverts to its behavior prior to OpenVMS Version 7.0—that is, no upcalls can occur, and DECthreads does not use all processors in multiprocessor systems.

B.10.12 Process Control System Services and DCL Commands

OpenVMS system services and DCL commands are either process based or operate on a per-thread basis. This section identifies several system services on this basis.

B.10.12.1 Process-Level System Services

The following system services continue to be process based: $SUSPEND, $RESUME, and $DELPRC. These services will operate on an entire process; they are not thread based. For example, $SUSPEND issued by a thread will suspend all of the virtual processors in process, not just the calling thread.

Under OpenVMS Version 7.0 or later, it is possible to see all but one of your kernel threads in SUSP state, such as when at a breakpoint in the debugger. This effect is a part of the debugging support and is not the result of calling $SUSPND.

B.10.12.2 Kernel-Level System Services

The following system services now operate on a per-thread basis: $HIBER, $SCHDWK, and $SYNCH. These services will not operate on an entire process; they are thread based. For example, $HIBER will cause the calling thread to become inactive but will not affect other threads in the process.

B.10.12.3 DCL Commands

The following DCL commands operate as indicated:

- STOP/IDENTIFICATION—This command continues to work on a process basis.
- SET PROCESS—This command continues to work on a process basis except for SET PROCESS/PRIORITY.
- SET PROCESS/PRIORITY—This command now sets the priority of a kernel thread. Avoid setting different priorities for kernel threads in the same process. Refer to Section B.10.4 for more information.

B.11 Interoperability with POSIX for OpenVMS

Previous releases of the POSIX for OpenVMS layered product had very limited interoperability with DECthreads. Under OpenVMS Version 7.0 and later, using DECthreads with the POSIX for OpenVMS layered product is not supported.

C

Considerations for Windows NT Systems

This appendix discusses DECthreads issues and restrictions specific to its implementation under the Windows NT operating system.

C.1 DECthreads Interfaces on Windows NT Systems

The Win32 subsystem provides support for multithreading through the Win32 Application Programming Interface (API). The Win32 API allows for thread creation, termination, synchronization, and other thread functions.

The DECthreads and Win32 threads libraries are interoperable. Threads created using the Win32 API can use DECthreads synchronization primitives, for example, and DECthreads-created threads can use the Win32 synchronization primitives. See Section C.4 for more information.

C.1.1 Pthread Interface

To add value to the Win32 API, DECthreads provides its POSIX 1003.1c-1995 (**pthread**) interface, which is available across all DIGITAL platforms.

Note these differences in support for the DECthreads **pthread** interface routines on Windows NT systems:

- Due to the design of Win32 threads, the DECthreads **pthread** interface supports creating threads with system contention scope only.
- On Windows NT systems, the DECthreads scheduler supports creating threads with the `SCHED_RR` scheduling policy only. You can use **pthread** interface routines to set and refer to other POSIX.1c scheduling policies, but DECthreads implicitly changes the scheduling policy setting to `SCHED_RR` for any thread it creates or modifies.

In addition, the DECthreads exception package is also available to support handling of DECthreads exceptions in conjunction with use of the **pthread** interface. See Section C.4 for more information.

The following DECthreads **pthread** interface routines are not implemented for Windows NT systems, and also do not return errors or generate exceptions:

```
pthread_attr_getinheritsched( )
pthread_attr_setinheritsched( )
```

Do not use these routines. In a future DECthreads release they will be changed to return errors and generate exceptions:

The `pthread_attr_setstacksize( )` routine does not change the stack size of newly created threads. On Windows NT a DECthreads-created thread is created with a stack of the same size as for its process's primary thread. The stack size grows as needed. (See the Win32 documentation for the `CreateThread( )` routine.) No errors or exceptions are generated as a result of using `pthread_attr_setstacksize( )`.

The corresponding `pthread_attr_getstacksize( )` routine returns the value that was set using `pthread_attr_setstacksize( )`, but this is not useful because a DECthreads-created thread is not created using the thread attributes object's stack size attribute.

An attempt to set the scheduling policy or priority attributes of an existing thread or in a thread attributes object using of the following routines produces a return value of [ENOTSUP]:

```
pthread_attr_setschedparam( )
pthread_attr_setschedpolicy( )
pthread_setschedpolicy( )
```

C.1.2 Other Interfaces

The DECthreads thread-independent services (or **tis**) interface is not yet available for Windows NT systems.

The DECthreads proprietary CMA (or **cma**) and POSIX 1003.4a/Draft 4 (or **d4**) interfaces are no longer offered for Windows NT systems. See Appendix E and Appendix F for information about migrating your applications from the DECthreads **cma** and **d4** interfaces, respectively, to the DECthreads **pthread** interface.

C.2 Compiling DECthreads Applications

For a dynamically linked program, compile with the `/MT` switch. For a statically linked program, compile with the `/MD` switch. These switches ensure that reentrant definitions, such as *errno*, are used. For example:

```
% CL /c myprog.c /MT
```

C.3 Linking DECthreads Applications

Applications that use the C run-time library should be linked against one of two C run-time libraries that support multithreading, as follows:

- Link against the `libcmt.lib` library when compiling with the `/MT` switch.
- Link against the `msvcrt.lib` library when compiling with the `/MD` switch.

Multithreaded applications should not link against `libc.lib` because it does not support multithreading.

C.4 Interoperability of Win32 API and DECthreads pthread Routines

Win32 threads can create, use, and operate on DECthreads primitives, such as synchronization objects, as any DECthreads-created thread can. Win32 threads can operate on both DECthreads-created threads and other Win32 threads using the DECthreads **pthread** interface routines—that is, Win32 threads can join with them, cancel them, and so on.

A Win32 thread can operate directly on a DECthreads-created thread using the Win32 API, provided that it can obtain a Win32 handle to that thread. A Win32 thread can synchronize with a DECthreads-created thread and with other Win32 threads using DECthreads synchronization objects, just as DECthreads-created threads synchronize with each other. A Win32 thread can synchronize with a DECthreads-created thread using Win32 synchronization objects the same way that Win32 threads synchronize with each other.

A DECthreads-created thread can use the Win32 API to obtain a Win32 handle to itself, which allows it to be operated on via Win32 API routines. It can operate on Win32 threads and DECthreads-created threads (including itself) using Win32 API routines the same way that Win32 threads operate on each other. It can synchronize with Win32 threads and other DECthreads-created threads using Win32 synchronization objects the same way that Win32 threads synchronize with each other. It can synchronize with Win32 threads using DECthreads synchronization objects. It can operate on Win32 threads, use `pthread_cancel( )` to cancel them, use `pthread_join( )` to join with them, and so on—but only if the Win32 thread has previously made a call into the DECthreads library, which registers the Win32 thread with the DECthreads run-time environment.

It is strongly recommended that a DECthreads-created thread *not* call `TerminateThread( )`. This routine terminates a thread immediately and does not provide any means for the thread to clean up any context it might have acquired while executing. Context that is left in an inconsistent state when

calling `TerminateThread( )`, such as a locked mutex, can cause unpredictable results in your program.

Using `pthread_cancel( )` to cancel the initial (or “primary”) thread of a Win32 process causes the process to exit in the same manner as if it were terminated as the result of a call to the Win32 routine `ExitThread( )`.

DECthreads structured exception handling is interoperable with Win32 exception handling, but has a slightly different syntax. DECthreads supports a different set of tokens for defining exception blocks. The DECthreads tokens include `TRY`, `CATCH`, `CATCH_ALL`, `FINALLY`, and `ENDTRY`. The Microsoft tokens include `__try`, `__except`, and `__finally`.

The DECthreads exception semantics support multiple catch clauses and provide support for creating unique exceptions (that is, DECthreads address exceptions). For more information, see Chapter 5.

DECthreads exception blocks can be nested inside of Win32 exception blocks, and vice versa. The DECthreads `CATCH_ALL` macro will catch all exceptions—those raised via DECthreads exception handling macros, those raised via Win32, and system-raised exceptions.

C.5 Thread Cancelability of System Services

Win32 system calls are not cancelation points. None of the system calls should be called with asynchronous cancelation enabled. For more information, see Section 2.3.7.

D

Debugging Multithreaded Applications

The debugging information in this appendix is specific to applications that use DECthreads.

D.1 Using PTHREAD_CONFIG

During initialization of the DECthreads run-time environment, the `PTHREAD_CONFIG` environment variable (on DIGITAL UNIX and Windows NT systems) or logical symbol (on OpenVMS systems), if defined, is used to set static options for the multithreaded program. You can set `PTHREAD_CONFIG` to assist you in debugging a DECthreads application.

D.1.1 Major and Minor Keywords

As summarized in Table D–1, `PTHREAD_CONFIG` takes "major keywords" as arguments. Use a "minor keyword" to specify a value for each major keyword.

Table D–1 PTHREAD_CONFIG Settings

Major keyword	Minor keyword	Meaning
`dump=`	*file-path*	Path of DECthreads bugcheck file
`meter=`	`condition`	Meter condition variable operations
	`mutex`	Meter mutex operations
	`stack`	Record thread greatest stack extent
	`thread`	(Unused)
	`all`	Meter all available operations
	`none`	No metering
`width=`	*bugcheck_output_width*	Width of output from DECthreads bugcheck output

D.1.2 Specifying Multiple Values

When setting PTHREAD_CONFIG, use a semicolon to separate major keyword expressions and use a comma to separate minor keyword values. For example, using the C shell under DIGITAL UNIX, you can set PTHREAD_CONFIG as follows:

```
% setenv PTHREAD_CONFIG meter=thread,mutex;dump=/tmp/dump-%d.dmp;width=132
```

On DIGITAL UNIX systems only, DECthreads forms the bugcheck output file's name using the process ID number (PID) as the %d format item.

On DIGITAL UNIX systems only, PTHREAD_CONFIG can be the object of the export command.

D.2 Running DECthreads in Metered Mode

Metering tells DECthreads to collect statistical and historical information about the use of synchronization objects within your program. This affects all synchronization within the program, including that within DECthreads itself and any other libraries that use threads. Therefore, metering provides a very powerful tool for debugging multithreaded code.

To enable metering, define PTHREAD_CONFIG prior to running any threaded application. The variable should have a value of meter=all to enable metering. This causes DECthreads to gather and record statistics and history information for all synchronization operations. Additionally, when running in metered mode, DECthreads marks all thread stacks (except the main thread stack) with a specific pattern.

Programs running in metered mode are somewhat slower than unmetered programs. Also, normal mutexes that are metered can behave like errorcheck mutexes in many ways. This does not affect the behavior of correct programs, but you should be aware of some differences between normal and errorcheck mutexes. The most important difference is that normal mutexes do not report a number of usage errors, while errorcheck mutexes do.

Because it can be expensive to detect these conditions, a normal mutex may not always report these errors. Regardless of whether the program seems to work correctly under these circumstances, the operations are illegal. A metered normal mutex will report these errors under more circumstances than will an unmetered normal mutex.

D.3 Using Ladebug on DIGITAL UNIX

The DIGITAL Ladebug debugger provides commands to display the state of threads, mutexes, and condition variables, without using the built-in DECthreads debug facility.

Using the Ladebug commands, you can examine core files and remote debug sessions, as well as run processes.

The basic commands are:

- `thread` *n* — Sets the current thread context to *n*.
- `show thread [`*n* `...]` — Displays thread state (more information displayed if `$verbose=1`)
- `show mutex [`*n* `...]` — Display mutex state.
- `show condition [`*n* `...]` — Display condition variable state.

Refer to the Ladebug documentation for further details.

D.4 Debugging Threads on OpenVMS Systems

This section presents particular topics that relate to debugging a multithreaded application under OpenVMS.

D.4.1 Display of Stack Trace from Unhandled Exception

When a program incurs an unhandled exception, a stack trace is produced that shows the call frames from the point where the exception was raised or, if DECthreads `TRY/CATCH`, `TRY/FINALLY`, or POSIX cleanup handlers are used, from the point where it was last reraised to the bottom of the stack.

D.5 Debugging Threads on Windows NT Systems

The WinDbg Debugger, provided with the Windows NT Software Development Kit (SDK), provides support for debugging a threaded program. When working with DECthreads in WinDbg, be aware that terminating a thread with `pthread_exit()` raises an exception as a normal part of the exiting process. Unless you explicitly tell WinDbg to ignore this exception, it catches the exception and stops execution. To allow the thread to exit properly and your application to continue, use the WinDbg `gn` (go not handled) command. This command tells WinDbg to ignore the exception.

WinDbg also stops at each DECthreads exception block for this exception and has to be continued using the WinDbg `gn` command until the thread's stack is completely unwound. The exception used by DECthreads for thread rundown is `0x177db052`.

DECthreads also generates an exception in a thread as a normal part of thread cancelation. When canceling a thread with `pthread_cancel( )`, an exception value of `0x177db048` is raised. This exception should be allowed to propagate all the way to the base of the thread's stack. If WinDbg halts execution of the program and indicates that this exception is being raised, use the WinDbg `gn` command to allow the exception to continue.

You can use the Microsoft Visual C++ debugger's graphical user interface to debug exceptions and to indicate how to handle various exceptions. In this case, choose "Stop if not handled" for the exception codes mentioned above.

However, be aware that you cannot fully debug an application that uses `pthread_exit( )` to exit a Win32 thread or uses `pthread_cancel( )` to cancel a Win32 thread. For a thread created using the Win32 API, using `pthread_exit( )` to exit, or using `pthread_cancel( )` to cancel, that thread causes a library-defined unhandled exception filter routine to be invoked. Unhandled exception filter routines cannot be executed in a debugger environment. Any unhandled exception causes the process to exit.

E

Migrating from the cma Interface

This appendix presents information that helps you migrate existing programs and applications that use the DIGITAL-proprietary DECthreads CMA (or **cma**) interface to use the DECthreads **pthread** interface, based on the IEEE POSIX 1003.1c-1995 standard.

Note

In future DECthreads releases, the **cma** interface will continue to exist and be supported, but it will no longer be documented or enhanced. Therefore, it is recommended that you port your **cma**-based programs and applications to the **pthread** interface as soon as possible. The DECthreads **pthread** interface is the most portable, efficient, and robust multithreading run-time library offered by DIGITAL.

E.1 Overview

The DECthreads **pthread** interface differs significantly from the **cma** interface, though there are many similarities between the functions that individual routines perform. This section gives hints about the relationship between the two sets of routines, to assist you in migrating applications.

Note that routines whose names have the `_np` suffix are *not portable*—that is, the routine might not be available except in DECthreads.

You need not prototype the **pthread** routines if you include the C language `pthread.h` header file.

E.2 cma Handles

A **cma handle** is storage, similar to a pointer, that refers to a specific DECthreads object (thread, mutex, condition variable, queue, or attributes object).

Handles are allocated by the user application. They can be freely copied by the program and stored in any class of storage; objects are managed by DECthreads.

In the **cma** interface, because objects are accessed only by handles, you can think of the handle as if it were the object itself. DECthreads objects are accessed by handles (rather than pointers), because handles allow for greater robustness and portability. Handles allow DECthreads to detect the following types of run-time errors:

- Using an uninitialized handle
- Using a corrupted handle
- Using a handle whose object no longer exists (a dangling handle)

Handles are not supported in the DECthreads **pthread** interface. Although this provides less robustness due to more limited error checking, it allows better performance by decreasing memory use and memory access. (That is, handles result in pointers to pointers.)

E.3 Interface Routine Mapping

As summarized in Table E–1, many **cma** routines perform functions nearly identical to corresponding routines in the **pthread** interface. The syntax and semantics differ, but the similarities are also notable.

Table E–1 Corresponding cma and pthread Routines

cma Routine	pthread Routine	Notes
`cma_alert_disable_asynch( )`	`pthread_setcancelstate( )` `/pthread_setcanceltype( )`	
`cma_alert_disable_general( )`	`pthread_setcancelstate( )` `/pthread_setcanceltype( )`	
`cma_alert_enable_asynch( )`	`pthread_setcancelstate( )` `/pthread_setcanceltype( )`	

(continued on next page)

Table E–1 (Cont.) Corresponding cma and pthread Routines

cma Routine	pthread Routine	Notes
cma_alert_enable_general( )	pthread_setcancelstate( ) /pthread_setcanceltype( )	
cma_alert_restore( )	pthread_setcancelstate( ) /pthread_setcanceltype( )	
cma_alert_test( )	pthread_testcancel( )	
cma_attr_create( )	pthread_attr_init( )	
cma_attr_delete( )	pthread_attr_destroy( )	
cma_attr_get_guardsize( )	pthread_attr_getguardsize_np( )	
cma_attr_get_inherit_sched( )	pthread_attr_getinheritsched( )	
cma_attr_get_mutex_kind( )	pthread_mutexattr_gettype_np( )	
cma_attr_get_priority( )	pthread_attr_setsched_param( )	
cma_attr_get_sched( )	pthread_attr_getschedpolicy( )	
cma_attr_get_stacksize( )	pthread_attr_getstacksize( )	
cma_attr_set_guardsize( )	pthread_attr_setguardsize_np( )	
cma_attr_set_inherit_sched( )	pthread_attr_setinheritsched( )	
cma_attr_set_mutex_kind( )	pthread_mutexattr_settype_np( )	
cma_attr_set_priority( )	pthread_attr_setsched_param( )	
cma_attr_set_sched( )	pthread_attr_setschedpolicy( )	
cma_attr_set_stacksize( )	pthread_attr_setstacksize( )	
cma_cond_broadcast( )	pthread_cond_broadcast( )	
cma_cond_create( )	pthread_cond_init( )	
cma_cond_delete( )	pthread_cond_destroy( )	
cma_cond_signal( )	pthread_cond_signal( )	
cma_cond_signal_int( )	pthread_cond_signal_int_np( )	
cma_cond_timed_wait( )	pthread_cond_timedwait( )	
cma_cond_wait( )	pthread_cond_wait( )	
cma_delay( )	pthread_delay_np( )	

(continued on next page)

Table E–1 (Cont.) Corresponding cma and pthread Routines

cma Routine	pthread Routine	Notes
cma_handle_assign( )	none	Use Language assignment operator.
cma_handle_equal( )	pthread_equal( )	
cma_init( )	none	Not necessary.
cma_key_create( )	pthread_key_create( ) (Note: pthread_key_delete( ) is available as well.)	
cma_key_get_context( )	pthread_getspecific( )	
cma_key_set_context( )	pthread_setspecific( )	
cma_lock_global( )	pthread_lock_global_np( )	
cma_mutex_create( )	pthread_mutex_init( )	
cma_mutex_delete( )	pthread_mutex_delete( )	
cma_mutex_lock( )	pthread_mutex_lock( )	
cma_mutex_try_lock( )	pthread_mutex_trylock( )	
cma_mutex_unlock( )	pthread_mutex_unlock( )	
cma_once( )	pthread_once( )	
cma_stack_check_limit_np( )		
cma_thread_alert( )	pthread_cancel( )	
cma_thread_bind_to_cpu( )	none	
cma_thread_create( )	pthread_create( )	
cma_thread_detach( )	pthread_detach( )	
cma_thread_exit_error( )	pthread_exit( )	With Status.
cma_thread_exit_normal( )	pthread_exit( )	With Status.
cma_thread_get_priority( )	pthread_getschedparam( )	
cma_thread_get_sched( )	pthread_setschedparam( )	
cma_thread_get_self( )	pthread_self( )	

(continued on next page)

Table E–1 (Cont.) Corresponding cma and pthread Routines

cma Routine	pthread Routine	Notes
cma_thread_join( )	pthread_join( )	
cma_thread_set_priority( )	pthread_setschedparam( )	
cma_thread_set_sched( )	pthread_setschedparam( )	
cma_time_get_expiration( )	pthread_get_expiration_np( )	
cma_unlock_global( )	pthread_unlock_global_np( )	
cma_yield( )	pthread_yield( )	

Notice that the **cma** routine cma_cond_timed_wait( ) requires the time argument *expiration* to be specified in local time; whereas the **pthread** routine pthread_cond_timedwait( ) requires the time argument *abstime* to be specified in Universal Coordinated Time (UTC).

E.4 New pthread Routines

The following are **pthread** interface routines that have no functional similarities in the **cma** interface:

pthread_atfork( ) (DIGITAL UNIX only)
pthread_attr_getdetachstate( )
pthread_attr_setdetachstate( )
pthread_key_delete( )
pthread_kill( ) (DIGITAL UNIX only)
pthread_sigmask( ) (DIGITAL UNIX only)

F

Migrating from the d4 Interface

This appendix provides migration information for the routines in the DECthreads POSIX 1003.4a/Draft 4 (or **d4**) interface.

Note

Applications that use the DECthreads **d4** routines require significant modification to be migrated to the **pthread** interface described in Part II.

F.1 Overview

Routines in the DECthreads **pthread** interface differ significantly from the original DECthreads POSIX 1003.4a/Draft 4 implementation. This section describes the major changes between the interfaces.

F.2 Error Status and Function Returns

The DECthreads **pthread** interface does not use the global variable *errno*. (Note that DECthreads provides a thread-specific *errno* for use by libraries and application code, but the DECthreads **pthread** interface does not write to it.)

If an error condition occurs, a **pthread** routine returns an integer value that indicates the type of error. For example, a call to the DECthreads **d4** interface's implementation of `pthread_cond_destroy( )` that returned a –1 and set *errno* to [EBUSY], returns [EBUSY] as the routine's return value in the **pthread** interface implementation.

On successful completion, most **pthread** interface routines return zero (0).

F.3 Replaced or Renamed Routines

Many routines in the DECthreads **d4** interface have been replaced or renamed in the **pthread** interface, as shown in Table F–1.

Table F–1 pthread Routines That Replace d4 Routines

d4 Routine	Replacement pthread Routine
pthread_attr_create( )	pthread_attr_init( )
pthread_attr_delete( )	pthread_attr_destroy( )
pthread_attr_set/getdetach_np( )	pthread_attr_set/getdetachstate( )
pthread_attr_set/getprio( )	pthread_attr_set/getschedparam( )
pthread_attr_set/getsched( )	pthread_attr_set/getschedpolicy( )
pthread_condattr_create( )	pthread_condattr_init( )
pthread_condattr_delete( )	pthread_condattr_destroy( )
pthread_keycreate( )	pthread_key_create( )
pthread_mutexattr_create( )	pthread_mutexattr_init( )
pthread_mutexattr_delete( )	pthread_mutexattr_destroy( )
pthread_mutexattr_get /setkind_np( )	pthread_mutexattr_get/settype_np( )
pthread_setasynccancel( )	pthread_setcanceltype( )
pthread_setcancel( )	pthread_setcancelstate( )
pthread_set/getprio( )	pthread_set/getschedparam( )
pthread_set/getscheduler( )	pthread_set/getschedparam( )
pthread_yield( )	sched_yield( )

F.4 Routines with No Changes to Syntax

Except for the return value, the following routines in the DECthreads **d4** interface have no changes to syntax in the DECthreads **pthread** interface:

```
pthread_attr_setinheritsched( )
pthread_cancel( )
pthread_cond_broadcast( )
pthread_cond_destroy( )
pthread_cond_signal( )
pthread_cond_signal_int_np( )
pthread_cond_timedwait( )
pthread_cond_wait( )
```

```
pthread_delay_np( )
pthread_equal( )
pthread_exit( )
pthread_get_expiration_np( )
```
`pthread_join( )` **(now detaches the thread)**
```
pthread_mutex_destroy( )
pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
pthread_once( )
```

The following routines have no changes in syntax or return value:

```
pthread_self( )
pthread_testcancel( )
```

Notice that the **d4** routine `pthread_cond_timedwait( )` requires the time argument *abstime* to be specified in local time; whereas the **pthread** routine `pthread_cond_timedwait( )` requires the time argument *abstime* to be specified in Universal Coordinated Time (UTC).

F.5 Routines with Prototype or Syntax Changes

Table F–2 shows the routines in the DECthreads **d4** interface that have changes to their argument syntax in the DECthreads **pthread** interface.

Table F–2 d4 Routines With Syntax Changes as pthread Routines

Old Syntax	New Syntax
unsigned long **pthread_attr_getguardsize_np**(pthread_attr_t *attr*)	int **pthread_attr_getguardsize_np**(const pthread_attr_t **attr*, size_t **guardsize*)
int **pthread_attr_getinheritsched**(pthread_attr_t *attr*)	int **pthread_attr_getinheritsched**(const pthread_attr_t **attr*, int **inheritsched*)
unsigned long **pthread_attr_getstacksize**(pthread_attr_t *attr*)	int **pthread_attr_getstacksize**(const pthread_attr_t **attr*, size_t **stacksize*)

(continued on next page)

Table F–2 (Cont.) d4 Routines With Syntax Changes as pthread Routines

Old Syntax	New Syntax
unsigned long **pthread_attr_setguardsize_np**(pthread_attr_t **attr*, long *guardsize*)	int **pthread_attr_setguardsize_np**(pthread_attr_t **attr*, size_t *guardsize*)
unsigned long **pthread_attr_setstacksize**(pthread_attr_t **attr*, long *stacksize*)	int **pthread_attr_setstacksize**(const pthread_attr_t **attr*, size_t *stacksize*)
int **pthread_cleanup_pop**(int *execute*)	void **pthread_cleanup_pop**(int *execute*)
int **pthread_cond_init**(pthread_cond_t **cond*, pthread_condattr_t *attr*)	int **pthread_cond_init**(pthread_cond_t **cond*, pthread_condattr_t **attr*)
int **pthread_create**(pthread_t **thread*, pthread_attr_t *attr*, pthread_startroutine_t *start_routine*, pthread_addr_t *arg*)	int **pthread_create**(pthread_t **thread*, const pthread_attr_t **attr*, void* (**start_routine*)(void *), void **arg*)
int **pthread_detach**(pthread_t **thread*)	int **pthread_detach**(pthread_t *thread*)
int **pthread_getspecific**(pthread_key_t *key*, void ***value*)	void ***pthread_getspecific**(pthread_key_t *key*)
void **pthread_lock_global_np**( )	int **pthread_lock_global_np**(void)
void **pthread_unlock_global_np**( )	int **pthread_unlock_global_np**(void)
int **pthread_mutex_init**(pthread_mutex_t **mutex*, pthread_mutexattr_t *attr*)	int **pthread_mutex_init**(pthread_mutex_t **mutex*, const pthread_mutexattr_t **attr*)

Table F–3 shows routines in the DECthreads **d4** interface that have a corresponding **pthread** routine that does not support the obsolete **d4**-style datatypes. These datatypes were documented for previous releases of DECthreads.

If your original code used the standard DECthreads datatypes, then this migration requirement might not impact your code.

Table F–3 d4 Routines Whose pthread Counterpart Uses Standard Datatypes

New Standard Datatype Syntax	Nonstandard Datatype Syntax
void **pthread_cleanup_push**(void (**routine*)(void *), void **arg*)	int **pthread_cleanup_push**(pthread_cleanup_t **routine*, pthread_addr_t *arg*)
int **pthread_create**(pthread_t **thread*, const pthread_attr_t **attr*, void *(**start_routine*)(void *), void **arg*)	int **pthread_create**(pthread_t **thread*, pthread_attr_t *attr*, pthread_startroutine_t *start_routine*, pthread_addr_t *arg*)
int **pthread_exit**(void **value_ptr*)	int **pthread_exit**(pthread_addr_t *status*)
void ***pthread_getspecific**(pthread_key_t *key*)	int **pthread_getspecific**(pthread_key_t *key*, pthread_addr_t **value*)
int **pthread_join**(pthread_t *thread*, void ***value_ptr*)	int **pthread_join**(pthread_t *thread*, pthread_addr_t **status*)
int **pthread_once**(pthread_once_t **once_control*, void (**init_routine*)(void))	int **pthread_once**(pthread_once_t **once_block*, pthread_initroutine_t *init_routine*)
int **pthread_setspecific**(pthread_key_t *key*, const void **value*)	int **pthread_setspecific**(pthread_key_t *key*, pthread_addr_t *value*)

F.6 New Routines

The following are routines in the DECthreads **pthread** interface that did not exist at the time of the implementation of the **d4** interface:

pthread_atfork( ) (DIGITAL UNIX only)
pthread_attr_getdetachstate( )
pthread_attr_setdetachstate( )
pthread_key_delete( )
pthread_kill( ) (DIGITAL UNIX only)
pthread_sigmask( ) (DIGITAL UNIX only)

Glossary

address exception

An exception whose identity is based on where in the program it was raised. *See also* exception and status exception.

alert

See cancelation request.

alertable routine

See cancelable routine.

AST

Mechanism that signals an asynchronous event to a process.

asynchronous cancelability

If enabled, allows a thread to receive a cancelation request at any time (not only at cancelation points). *See also* general cancelability.

asynchronous signal

Signal that is the result of an event that is external to the process and is delivered at any point in a thread's execution when such an event occurs. *See also* synchronous signal.

attributes

Individual components of the attributes object. Attributes specify detailed properties about the objects to be created. *See also* attributes object.

attributes object

Object used to describe DECthreads objects (thread, mutex, condition variable, or queue). This description consists of the individual attribute values that are used to create an object. *See also* attributes.

bugcheck

An error condition internal to the DECthreads run-time environment that causes it to produce a specially formatted error message. Output of this message can be controlled using the `PTHREAD_CONFIG` environment variable or logical symbol.

cancelability state

Attribute of a thread that determines whether it currently receives cancelation requests.

cancelability type

Attribute of a thread that determines whether it responds to a cancelation request at cancelation points (synchronous concelation) or at any point in its execution (asynchronous cancelation).

cancelation point

A routine that, when called, determines whether a cancelation request is pending for this thread.

cancelation request

Mechanism by which one thread requests termination of another thread (or itself).

condition variable

Object that allows a thread to block its own execution until some shared data reaches a particular state.

condition variable attributes object

Object that allows you to specify values for condition variable attributes when you create a condition variable.

contention scope

Attribute of a thread that specifies the set of threads with which it competes for processing resources. *See also* process contention scope and system contention scope.

deadlock

Condition involving one or more threads and a set of one or more resources in which each of the threads is blocked waiting for one of the resources and all of the resources are held by the threads such that none of the threads can continue. For example, a thread will enter a self-deadlock when it attempts to lock a normal mutex a second time. Likewise, two threads will enter a deadlock when each attempts to lock a second mutex that is already held by the other. The introduction of additional threads and synchronization objects allows for more complex deadlock configurations.

dynamic memory

Memory that is allocated by the program as a result of a call to some memory management function, and that is referenced through pointer variables. *See also* static memory and stack memory.

epilogue code

Block of code, associated with a DECthreads exception scope, that finalizes the context of an exception scope. Epilogue code is always executed, regardless of whether the code in the associated exception scope raised an exception.

errorcheck mutex

Mutex that can be locked exactly once by a thread, like a normal mutex. If a thread tries to lock the mutex again without first unlocking it, the thread receives an error instead of deadlocking. *See also* deadlock, mutex, normal mutex, and recursive mutex.

exception

Object that describes an error condition.

exception scope

Block of code where exceptions are handled.

finalization code

See epilogue code.

general cancelability

If enabled, allows a thread to receive a cancelation request at specific cancelation points. If disabled, the thread cannot be canceled. *See also* asynchronous cancelability.

global lock

Single recursive mutex provided by DECthreads for use by all threads in a process when calling routines or code that is not thread safe to ensure serialized, exclusive access to the unsafe code.

guard area

Area at the overflow end of the thread stack that is inaccessible to the thread. This helps prevent or detect overflow of the thread's stack.

guardsize attribute

Attribute of a thread that specifies the minimum size (in bytes) of the guard area for a thread's stack.

handle

Storage, similar to a pointer, that refers to a specific DECthreads object.

inherit scheduling attribute

Attribute of a thread that specifies whether a newly created thread inherits the scheduling attributes (scheduling priority and policy) of the creating thread or uses the scheduling attributes stored in the attributes object. *See also* thread attributes object.

kernel execution context

Entity managed by the operating system kernel that uses processing resources. Also known as a kernel thread or virtual processor.

lifetime

Length of time memory is allocated for a particular purpose.

multithreaded programming

Division of a program into multiple threads that execute concurrently.

mutex

Mutual exclusion, an object that multiple threads use to ensure the integrity of a shared resource that they access (most commonly shared data) by allowing only one thread to access it at a time. *See also* normal mutex, errorcheck mutex, and recursive mutex.

mutex attributes object

Object that allows you to specify values for mutex attributes when you create a mutex.

mutex kind attribute

Mutex attribute that specifies whether its kind is normal, recursive, or errorcheck.

nonterminating signal

Signal that does not result in the termination of the process by default. *See also* terminating signal.

normal mutex

A kind of mutex that can be locked exactly once by a thread. It does not perform error checks. If a thread tries to lock the mutex again without first unlocking it, the thread waits for itself to release the lock and deadlocks. In DECthreads, this kind of mutex offers the best performance. *See also* mutex, errorcheck mutex, and recursive mutex.

per-thread context

See thread-specific data.

predicate

Boolean expression that defines a particular state of shared data; threads wait on a condition variable for shared data to enter the defined state. *See also* condition variable.

priority inversion

Occurs when interaction among three or more threads blocks the highest-priority thread from executing until after the lowest-priority thread can execute.

process contention scope

Setting for the contention scope attribute of a thread. Specifies that a thread competes for processing resources only with other threads in the same process. *See also* contention scope and system contention scope.

race condition

Occurs when two or more threads perform an operation, and the result of the operation depends on unpredictable timing factors.

read-write lock

An object that serializes access, in a thread-safe manner, to a data object that is shared among threads and that is frequently read but less frequently written.

recursive mutex

Mutex that can be locked more than once by a given thread without causing a deadlock. The thread must call the `pthread_mutex_unlock( )` routine the same number of times that it called the `pthread_mutex_lock( )` routine before another thread can lock the mutex. *See also* deadlock, mutex, normal mutex, and errorcheck mutex.

scheduling policy attribute

Attribute of a thread that describes how the thread is scheduled for execution relative to the other threads in the program. *See also* thread attributes object.

scheduling precedence

The set of characteristics of threads and the DECthreads scheduling algorithm that, in combination, determine which thread will be allowed to run when a scheduling decision is made. Scheduling decisions are made when a thread becomes ready to run (for example, when a mutex on which it was waiting is unlocked, or a condition variable on which it was waiting is signaled or broadcasted), or when a thread is blocked (for example, when it attempts to lock a locked mutex or when it waits on a condition variable).

scheduling priority attribute

Attribute of a thread that specifies the execution priority of a thread, expressed relative to other threads in the same policy. *See also* thread attributes object.

scope

Areas of a program where code can access memory.

software interrupt handler

A routine that is executed in response to an interrupt generated by the operating system or equivalent support software. For example, an AST service routine handles interrupts on OpenVMS systems; a signal handler routine handles interrupts on Digital UNIX systems.

stack memory

Memory that is allocated from a thread's stack area at run time by code generated by the language compiler, generally when a routine is initially called. *See also* dynamic memory and static memory.

stacksize attribute

Attribute of a thread that specifies the minimum size (in bytes) of the memory required for its stack.

start routine

Routine in your program where a newly created thread begins executing.

static memory

Any variable that is permanently allocated at a particular address for the life of the program. *See also* dynamic memory and stack memory.

status exception

An exception whose identity is based on the status value it contains. *See also* exception and address exception.

synchronous signal

Signal that is the result of an event that occurs inside a process and is delivered synchronously with respect to that event. *See also* asynchronous signal.

system contention scope

Setting for the contention scope attribute of a thread. Specifies that a thread competes for processing resources with all other threads in the system. *See also* contention scope and process contention scope.

terminating signal

Signal that results in the termination of the process by default. *See also* nonterminating signal.

thread

Single, sequential flow of control within a program. Within a single thread, there is a single point of execution.

thread attributes object

Object that allows you to specify values for thread attributes when you create a thread.

thread object

Data structure that describes a thread.

thread reentrant

Refers to a routine that functions normally despite being called simultaneously or sequentially in different threads.

thread safe

Refers to a routine that can be called simultaneously from multiple threads without risk of corruption. Refers to a library that typically consists of routines that do not themselves create or use threads but which can be called safely from applications that use threads.

thread-independent services

Routines in the DECthreads **tis** interface that support building thread-safe libraries.

thread-specific data

User-specified fields of arbitrary data that can be added to a thread's context.

timeslicing

Mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.

tis condition variable

Condition variable object that can be created using DECthreads **tis** interface routines.

tis mutex

Mutex object that can be created using DECthreads **tis** interface routines.

two-level scheduling

Thread scheduling model that schedules user threads onto kernel execution contexts, just as the operating system schedules processes onto the processors of a multiprocessor machine.

upcall

Technique for the operating system kernel to inform DECthreads that a kernel execution context is available. When a kernel execution context becomes available, the DECthreads scheduler schedules the thread with highest scheduling precedence that is ready to run onto the available kernel execution context.

Index

A

Address exceptions, 5–8
Addressing
  64-bit, B–5
API
  See Application Programming Interface (API)
Application programming interface (API)
  DECthreads error conditions from, 3–15
ASTs
  See Asynchronous system traps
Asynchronous programming techniques
  in a multithreaded program, A–9
Asynchronous system traps (AST)
  DECthreads's delivery of, B–12
  restrictions on use with DECthreads, B–3
Asynchronous thread cancelation, 2–20
  cleanup from, 2–22
Asynchronous user interface example program, 6–10
Attributes
  of condition variables, 2–34
  of mutexes, 2–29
    mutex type, 2–29
  of threads, 2–4
    contention scope, 2–9
    guardsize, 2–9
    inherit scheduling, 2–5
    scheduling policy, 2–5
    scheduling priority, 2–7
    stack address, 2–8
    stack size, 2–8
Attributes objects, 2–1
  creating, 2–2
  destroying, 2–2

B

Background scheduling policy, 2–5
Boss/worker functional model, 1–6
  work queue variation, 1–6
Bugchecks, 3–16
  configuring output, 3–16
  contents of dump file, 3–16
  forming dump file name, D–2
  interpreting output, 3–17

C

Cancelability state, 2–20
Cancelability state attribute
  of thread attributes object, 2–20
    setting, pthread–169, tis–46
Cancelability type, 2–20
Cancelability type attribute
  of thread attributes object, 2–20
    setting, pthread–171
Cancelation points
  DECthreads routines that provide, 2–21, 4–3
  in multithreaded code, 2–21
  system service routines that provide
    under DIGITAL UNIX, A–6
    under OpenVMS Alpha, B–5
Cancelation requests
  delivering, pthread–58, pthread–185, tis–50

Cancelation requests (cont'd)
sending, pthread–58
CATCH macro, 5–13
CATCH_ALL macro, 5–14
Cleanup handlers, 2–14
executing, pthread–60
registering, pthread–62
cma interface
See DIGITAL-proprietary CMA (cma) interface
Compiling applications
under DIGITAL UNIX, A–1
under OpenVMS, B–2
under Windows NT, C–2
Concurrency level
of threads
getting, pthread–110
setting, pthread–174
Condition handlers (OpenVMS)
declaring, B–4
Condition values (OpenVMS)
used by DECthreads, B–5
Condition variable attributes objects, 2–1, 2–34
creating, pthread–66
destroying, pthread–64
initializing, pthread–66
Condition variables, 2–29
creating, pthread–74, tis–7
destroying, pthread–70, tis–5
distinguishing from mutexes, 3–9
initializing, pthread–74, tis–7
naming, pthread–72, pthread–77
signaling, 2–30, 3–10
under the DECthreads thread-independent services (tis) interface, 4–4
using in thread-safe library code, 4–4
waiting a specified time interval for, 2–34, pthread–83
waiting indefinitely for, 2–30, pthread–86, tis–11
wakeups for waiting threads, pthread–68, pthread–79, pthread–81, tis–3, tis–9
Contention scope, 2–9
interaction with thread scheduling attributes, 3–8
Contention scope attribute
of thread attributes object, 2–9
getting, pthread–23
setting, pthread–50

D

d4 interface
See POSIX 1003.4a/Draft 4 (d4) interface
Data
See Thread-specific data
Deadlocks, 1–9, 3–10
avoiding, 3–10
Debugging tools
for DECthreads applications, D–1
metered mode, D–2
under DIGITAL UNIX, D–3
under OpenVMS, D–3
under Windows NT, D–3
DECthreads
64-bit addressing, B–5
blocking OpenVMS system services, B–13
bugcheck feature
See Bugchecks
cancelability of system services, A–5, B–5, C–4
compiling applications
under DIGITAL UNIX, A–1
under OpenVMS, B–2
under Windows NT, C–2
condition values used, B–5
debugging applications, D–1
declaring OpenVMS condition handlers, B–4
delivery of OpenVMS ASTs, B–12
dynamic activation
under DIGITAL UNIX, A–10
under OpenVMS, B–4
effects of OpenVMS DCL commands, B–17
error conditions

DECthreads
error conditions (cont'd)
application programming interface level, 3–15
internal, 3–15
exiting from OpenVMS images, B–16
global lock
avoiding software that lacks thread safety, 3–14
using from the tis interface, 4–3
header files
under DIGITAL UNIX, A–1
under OpenVMS, B–2
interactions with OpenVMS, B–15
interfaces, 1–9
DIGITAL-proprietary CMA (cma), 1–23
in C language, 1–9
in languages other than C, 1–9
obsolete, 1–23
POSIX.1c (pthread), 1–10
POSIX 1003.4a/Draft 4 (d4), 1–24
thread-independent services (tis), 1–20
undocumented but supported, 1–23
interoperability
with errno variable, pthread–1, tis–1
with POSIX for OpenVMS layered product, B–17
with signals (DIGITAL UNIX), A–8
with Win32 application programming interface, C–3
libraries, 1–9
linking applications
under DIGITAL UNIX, A–2
under Windows NT, C–3
linking with shared libraries (DIGITAL UNIX), A–2
platform dependencies
for DIGITAL UNIX, A–1
for OpenVMS, B–1
for Windows NT, C–1
POSIX.1003.4a/Draft 4 (d4) interface, pthread–1
POSIX.1c (pthread) interface, pthread–1
DECthreads (cont'd)
real-time scheduling, A–4
thread-independent services (tis) interface, 1–20, 4–1, tis–1
two-level scheduling
under DIGITAL UNIX, A–3
under OpenVMS Alpha, B–6
use of kernel threads
under DIGITAL UNIX, A–4
under OpenVMS Alpha, B–6
virtual processors (OpenVMS), B–11
DECthreads exceptions package, 1–11
DECthreads global mutex
locking, pthread–136, tis–20
unlocking, pthread–186, tis–51
Default mutexes, 2–27
Detachstate attribute
of thread attributes object
getting, pthread–9
setting, pthread–32
DIGITAL-proprietary CMA (cma) interface, 1–23, E–1
DIGITAL UNIX operating system
using DECthreads with, A–1
Dynamic activation
of DECthreads
under DIGITAL UNIX, A–10
under OpenVMS, B–4
Dynamic memory, 3–5
using from threads, 3–5

E

errno variable, pthread–1, tis–1
Errorcheck mutexes, 2–28
Error conditions
detecting, 3–15
from DECthreads application programming interface, 3–15
internal to DECthreads, 3–15
Event flags (OpenVMS), B–14
Example programs
asynchronous user interface, 6–10
prime number search, 6–1

Exceptions
address, 5–8
cancelation of threads, 2–19
catching
all, 5–14
specific, 5–13
CATCH macro, 5–13
CATCH_ALL macro, 5–14
conventions for modular use, 5–21
debugging when unhandled (OpenVMS), D–3
DECthreads-defined objects, 5–27
DECthreads exceptions package, 5–1
DECthreads package, 1–11
epilogue actions for, 5–16
failing
due to condition handlers, B–4
FINALLY macro, 5–16, 5–22
importing error status into, pthread–106
interoperability of, 5–28
language-specific, 5–28
matching two, 5–20, pthread–102
naming conventions for, 5–21
obtaining error status from, 5–19, pthread–100
operations on, 5–17
programming for, 5–3
programming languages supported for, 5–1
pthread_exc_get_status_np() routine, 5–19, pthread–100
pthread_exc_matches_np() routine, 5–20, pthread–102
pthread_exc_report_np() routine, 5–20, pthread–104
pthread_exc_set_status_np() routine, 5–18, pthread–106
purpose of, 5–2
RAISE macro, 5–11
raising, 5–11
referencing when caught, 5–17
relation to return codes and signals, 5–2
reporting, pthread–104
reporting when caught, 5–20
RERAISE macro, 5–15, 5–26
Exceptions (cont'd)
reraising, 5–15
scope of, 5–10
setting error status in, 5–18
status, 5–8
synchronous signals reported as, A–9
termination of, 5–9
THIS_CATCH exception object, 5–17
TRY macro, 5–10
unhandled, D–3
Exception scopes, 5–10
Expiration time
obtaining, pthread–120

F

FINALLY macro, 5–16, 5–22
First-in/first-out (FIFO) scheduling policy, 2–5
Foreground scheduling policy, 2–5
Fork handlers (DIGITAL UNIX), pthread–3
Functional models
for multithreaded programming, 1–6
boss/worker, 1–6
combinations, 1–8
pipelining, 1–7
work crew, 1–6

G

$GETJPI system service (OpenVMS)
MULTITHREAD item code, B–11
Global lock
See DECthreads, global lock
Guardsize attribute
of thread attributes object, 2–9, 3–6
getting, pthread–11, pthread–13
setting, pthread–34, pthread–37

H

Handlers
cleanup, 2–14
condition (OpenVMS), B–4
fork (DIGITAL UNIX), pthread–3
interrupt, pthread–81

$HIBER system service (OpenVMS), B–14

I

Images (OpenVMS)
  compiling for DECthreads, B–2
  linking DECthreads-based, B–2
Inherit scheduling attribute
  of thread attributes object, 2–5
    getting, pthread–15
    setting, pthread–40
Interfaces
  to DECthreads, 1–9
    DIGITAL-proprietary CMA (cma), 1–23, E–1
    in C language, 1–9
    in languages other than C, 1–9
    obsolete, 1–23
    POSIX.1c (pthread), 1–10
    POSIX 1003.4a/Draft 4 (d4), 1–24, F–1
    thread-independent services (tis), 1–20, 4–1, tis–1
    undocumented but supported, 1–23
Interrupt handlers
  for threads, pthread–81

K

Kernel threads
  enabling in existing OpenVMS images, B–9
  OpenVMS linker options, B–8
  querying use of (OpenVMS), B–11
  relation to user threads
    under DIGITAL UNIX, A–4
    under OpenVMS, B–6
  virtual processors for (OpenVMS), B–11

L

Ladebug debugger (DIGITAL UNIX), D–3
Libraries
  for DECthreads, 1–9
  lacking thread safety, 1–8, 3–14
  shared (DIGITAL UNIX)
Libraries
  shared (DIGITAL UNIX) (cont'd)
    linking with DECthreads, A–2
    using with DECthreads, A–2
  thread-safe, 3–13, 4–1
Linking applications
  under DIGITAL UNIX, A–2
  under OpenVMS, B–2
Locks
  global
    See DECthreads, global lock
  read-write, 4–4

M

Macros
  CATCH, 5–13
  CATCH_ALL, 5–14
  FINALLY, 5–16, 5–22
  PTHREAD_COND_INITIALIZER, 4–4, pthread–75
  PTHREAD_COND_INITWITHNAME_NP, 4–4
  PTHREAD_MUTEX_INITIALIZER, 4–3, pthread–155
  PTHREAD_MUTEX_INITWITHNAME, 4–3
  PTHREAD_ONCE_INIT, pthread–165
  RAISE macro, 5–11
  RERAISE, 5–15, 5–26
  TRY, 5–10
    restrictions, B–4
Memory
  dynamic, 3–5
  shared, 3–4
  stack, 3–5
    identifying overflow, 2–9
  static, 3–4
  synchronizing threads' access to, 3–3
Multiprocessor systems, 1–2
Multithreaded programming
  asynchronous programming techniques, A–9
  asynchronous thread execution, 3–1
  cancelation point routines, 4–3

Multithreaded programming (cont'd)
cancelation points, 2–21
dependencies upon other libraries, 3–12
multiple thread libraries unsupported, 3–15
not thread-safe, 3–14
thread-reentrant, 3–13
thread-safe, 3–13
detecting DECthreads error conditions, 3–15
example programs
asynchronous user interface, 6–10
prime number search, 6–1
thread cancelation, 2–23
functional models, 1–6
boss/worker, 1–6
combinations, 1–8
pipelining, 1–7
work crew, 1–6
managing a thread's stack, 3–6
one-time initialization, 3–12, pthread–164, tis–32
potential issues, 1–8
deadlocks, 1–9
dependence upon nonreentrant software, 1–8, 3–13
priority inversion, 1–9
program complexity, 1–8
race conditions, 1–9
programming errors
initializing DECthreads objects after thread creation, 3–2
passing stack local data, 3–2
thread scheduling as thread synchronization, 3–3
scheduling threads, 3–7
interaction with thread contention scope, 3–8
priority inversion, 3–8
real-time, 3–7
signals (DIGITAL UNIX)
avoiding use of, A–8
synchronizing memory access, 3–3, 3–9
avoiding deadlocks, 3–10
avoiding race conditions, 3–9

Multithreaded programming
synchronizing memory access (cont'd)
distinguishing mutexes and condition variables, 3–9
signaling a condition variable, 3–10
using memory
dynamic, 3–5
shared, 3–4
stack, 3–5
static, 3–4
writing thread-safe libraries, 4–1
yielding thread execution, pthread–192
Mutex attributes objects, 2–1, 2–29
creating, pthread–144
destroying, pthread–138
initializing, pthread–144
mutex type attribute, 2–29
getting, pthread–140, pthread–142
setting, pthread–146, pthread–148
Mutexes, 2–26
creating, pthread–154, tis–24
DECthreads global
locking, pthread–136, tis–20
unlocking, pthread–186, tis–51
destroying, pthread–150, tis–22
distinguishing from condition variables, 3–9
initializing, pthread–154, tis–24
in thread-safe library code, 4–3
locking, pthread–156, pthread–160, tis–26, tis–28
locking, before signaling a condition variable, 3–10
naming, pthread–152, pthread–158
operations on, 2–28
protecting call to code lacking thread safety, 3–14
types of
default, 2–27
errorcheck, 2–28
normal, 2–27
recursive, 2–27
under the DECthreads thread-independent services (tis) interface, 4–3

Mutexes (cont'd)
unlocking, pthread–162, tis–30
using static data before release of, 3–14
Mutex type attribute
of mutex attributes object, 2–29

N

Naming conventions
for DECthreads exception objects, 5–21
Normal mutexes, 2–27

O

Object names
obtaining, pthread–17, pthread–72, pthread–112, pthread–130, pthread–152
setting, pthread–43, pthread–77, pthread–132, pthread–158, pthread–176
One-time initialization of threads, 3–12
OpenVMS operating system
64-bit addressing, B–5
condition values used by DECthreads, B–5
DCL command operation with DECthreads, B–17
debugging DECthreads applications, D–3
interactions with DECthreads, B–15
linker options for kernel threads, B–8
linking DECthreads-based images, B–2
system services
blocking, B–13
using DECthreads with, B–1

P

PAGESIZE environment variable (DIGITAL UNIX)
relation to size of thread stack guard region, A–10
Pipelining functional model, 1–7
POSIX.1003.4a/Draft 4 (d4) interface, pthread–1
POSIX.1003.4a/Draft 4 document, pthread–1
POSIX.1c (pthread) interface, 1–10, pthread–1
optionally implemented routines, 1–20
summary of routines, 1–10
POSIX.1c standard, 1–9, pthread–1
optionally implemented routines, 1–20
POSIX 1003.1c-1995 standard
See POSIX.1c standard
POSIX 1003.4a/Draft 4 (d4) interface, 1–24, F–1
POSIX for OpenVMS layered product
interoperability with DECthreads, B–17
Prime number search example program, 6–1
Priority inversion, 1–9, 3–8
avoiding, 3–8
Process contention scope, 2–10, A–5
Processes
child
creating, pthread–3
pthread.h header file, 1–9, A–1, B–2
pthread interface
See POSIX.1c (pthread) interface
pthread_atfork() routine, pthread–3
pthread_attr_destroy() routine, pthread–7
using, 2–2
pthread_attr_getdetachstate() routine, pthread–9
pthread_attr_getguardsize() routine, pthread–11
using, 2–9
pthread_attr_getguardsize_np() routine, pthread–13
using, 2–9
pthread_attr_getinheritsched() routine, pthread–15
pthread_attr_getname_np() routine, pthread–17

pthread_attr_getschedparam() routine,
pthread–19
pthread_attr_getschedpolicy() routine,
pthread–21
pthread_attr_getscope() routine, pthread–23
using, 2–10
pthread_attr_getstackaddr() routine,
pthread–25
using, 2–8
pthread_attr_getstacksize() routine,
pthread–27
pthread_attr_init() routine, pthread–29
using, 2–2
pthread_attr_setdetach() routine
using, 2–4
pthread_attr_setdetachstate() routine,
pthread–32
pthread_attr_setguardsize() routine,
pthread–34
using, 2–9
pthread_attr_setguardsize_np() routine,
pthread–37
using, 2–9
pthread_attr_setinheritsched() routine,
pthread–40
using, 2–5
pthread_attr_setname_np() routine,
pthread–43
pthread_attr_setschedparam() routine,
pthread–45
using, 2–8
pthread_attr_setschedpolicy() routine,
pthread–48
using, 2–6
pthread_attr_setscope() routine, pthread–50
using, 2–10
pthread_attr_setstackaddr() routine,
pthread–53
using, 2–8
pthread_attr_setstacksize() routine,
pthread–56
using, 2–8
pthread_attr_stacksize() routine
using, 3–7
pthread_cancel() routine, pthread–58
using, 2–11, 2–19
PTHREAD_CANCELED return value, 2–19
pthread_cleanup_pop() routine, pthread–60
using, 2–12, 2–14, 2–19, 2–21
pthread_cleanup_push() routine,
pthread–62
using, 2–12, 2–14, 2–19, 2–21
pthread_condattr_destroy() routine,
pthread–64
using, 2–2, 2–34
pthread_condattr_init() routine, pthread–66
using, 2–2, 2–34
pthread_cond_broadcast() routine,
pthread–68
using, 2–34, 3–3
pthread_cond_destroy() routine, pthread–70
using, 2–34
pthread_cond_getname_np() routine,
pthread–72
pthread_cond_init() routine, pthread–74
using, 2–33
PTHREAD_COND_INITIALIZER macro,
4–4, pthread–75
PTHREAD_COND_INITWITHNAME_NP
macro, 4–4
pthread_cond_setname_np() routine,
pthread–77
pthread_cond_signal() routine, pthread–79
using, 2–31, 2–34, 3–3
pthread_cond_signal_int_np() routine,
pthread–81
using, 2–34, 3–17
pthread_cond_timedwait() routine,
pthread–83
using, 2–28, 2–34, 3–3
pthread_cond_wait() routine, pthread–86
using, 2–28, 2–31, 2–33, 3–3
PTHREAD_CONFIG, D–1
configuring DECthreads bugcheck output,
3–16
major and minor keyword settings, D–1
specifying multiple values, D–2

pthread_create() routine, pthread–89
  using, 2–3, 3–3
pthread_delay_np() routine, pthread–94
pthread_detach() routine, pthread–96
  using, 2–15
pthread_equal() routine, pthread–98
pthread_exceptions.h header file, 1–11
pthread_exc_get_status_np() routine, pthread–100
  using, 5–19
pthread_exc_matches_np() routine, pthread–102
  using, 5–20
pthread_exc_report_np() routine, pthread–104
  using, 5–20
pthread_exc_set_status_np() routine, pthread–106
  using, 5–18
pthread_exit() routine, pthread–108
  using, 2–11, 2–12
pthread_getconcurrency() routine, pthread–110
pthread_getname_np() routine, pthread–112
pthread_getschedparam() routine, pthread–114
pthread_getsequence_np() routine, pthread–116
pthread_getspecific() routine, pthread–118
  using, 2–35
pthread_get_expiration_np() routine, pthread–120
pthread_join() routine, pthread–122
  using, 2–15, 3–3
pthread_key_create() routine, pthread–125
  using, 2–35
pthread_key_delete() routine, pthread–128
pthread_key_getname_np() routine, pthread–130
pthread_key_setname_np() routine, pthread–132
pthread_kill() routine, pthread–134
pthread_lock_global_np() routine, pthread–136
  using, 3–14
pthread_mutexattr_destroy() routine, pthread–138
  using, 2–2
pthread_mutexattr_gettype() routine, pthread–140
  using, 2–29
pthread_mutexattr_gettype_np() routine, pthread–142
  using, 2–29
pthread_mutexattr_init() routine, pthread–144
  using, 2–2
pthread_mutexattr_settype() routine, pthread–146
  using, 2–29
pthread_mutexattr_settype_np() routine, pthread–148
  using, 2–29
pthread_mutex_destroy() routine, pthread–150
  using, 2–29
pthread_mutex_getname_np() routine, pthread–152
pthread_mutex_init() routine, pthread–154
  using, 2–26
PTHREAD_MUTEX_INITIALIZER macro, 4–3, pthread–155
PTHREAD_MUTEX_INITWITHNAME macro, 4–3
pthread_mutex_lock() routine, pthread–156
  using, 2–27, 2–28, 3–3
pthread_mutex_setname_np() routine, pthread–158
pthread_mutex_trylock() routine, pthread–160
  using, 2–28, 3–3
pthread_mutex_unlock() routine, pthread–162
  using, 2–27, 2–28, 3–3
pthread_once() routine, pthread–164
  using, 3–2, 3–12

PTHREAD_ONCE_INIT macro, pthread–165
pthread_once_t data structure, pthread–164, tis–32
pthread_self() routine, pthread–167
pthread_setcancelstate() routine, pthread–169
  using, 2–20
pthread_setcanceltype() routine, pthread–171
  using, 2–20
pthread_setconcurrency() routine, pthread–174
pthread_setname_np() routine, pthread–176
pthread_setschedparam() routine, pthread–178
  using, 2–6, 2–8
pthread_setspecific() routine, pthread–181
  using, 2–35
pthread_sigmask() routine, pthread–183
pthread_testcancel() routine, pthread–185
  using, 2–20
pthread_unlock_global_np() routine, pthread–186
  using, 3–14

R

Race conditions, 1–9
  avoiding, 3–9
RAISE macro, 5–11
Read-write locks, 4–4
  creating, tis–43
  DECthreads tis interface routines for, 4–7
  destroying, tis–41
  initializing, tis–43
  locking
    for read access, tis–35, tis–37
    for write access, tis–53, tis–55
  under the DECthreads thread-independent services (tis) interface, 4–4
  unlocking
    for read access, tis–39
Read-write locks
  unlocking (cont'd)
    for write access, tis–57
  using in thread-safe library code, 4–4
Recursive mutexes, 2–27
Reentrant code
  See also Thread-reentrant code
  required for multithreaded programming, 1–5
  required for thread-safe code, 4–2
RERAISE macro, 5–15, 5–26
Round-robin (RR) scheduling policy, 2–5

S

Scheduling parameters
  of threads
    getting, pthread–114
    setting, pthread–178
Scheduling parameters attribute
  of thread attributes object
    getting, pthread–19
    setting, pthread–45
Scheduling policy attribute
  of thread attributes object, 2–5
    getting, pthread–21
    setting, pthread–48
Scheduling priority attribute
  of thread attributes object, 2–7
sched_get_priority_max() routine, pthread–188
sched_get_priority_min() routine, pthread–190
sched_yield() routine, pthread–192
Sequence numbers
  See Thread sequence numbers
Shared memory
  using from threads, 3–4
Signal masks (DIGITAL UNIX)
  See Thread signal masks
Signals (DIGITAL UNIX)
  per-thread usage, A–8
  synchronous
    reported as exceptions, A–9

sigwait() routine, pthread–194
sigwait() service (DIGITAL UNIX), A–8
Stack address attribute
 of thread attributes object, 2–8
  getting, pthread–25
  setting, pthread–53
Stack memory, 3–5
 using from threads, 3–5
Stacks
 of threads
  See Thread stacks
Stacksize attribute
 of thread attributes object, 2–8
  getting, pthread–27
  setting, pthread–56
Static memory, 3–4
 using before release of mutex, 3–14
 using from threads, 3–4
Status exceptions, 5–8
Synchronization objects
 condition variables, 2–29
 mutexes, 2–26
Synchronous thread cancelation, 2–20
 cleanup from, 2–21
SYSGEN (OpenVMS)
 MULTITHREAD parameter, B–16
System contention scope, 2–10, A–4
System services
 cancelability from DECthreads, A–5, B–5

T

THIS_CATCH exception object, 5–17
Thread attributes objects, 2–1
 cancelability state attribute, 2–20
  setting, pthread–169, tis–46
 cancelability type attribute, 2–20
  setting, pthread–171
 contention scope attribute, 2–9, pthread–23, pthread–50
 creating, pthread–29
 destroying, pthread–7
 detachstate attribute, pthread–9, pthread–32
Thread attributes objects (cont'd)
 guardsize attribute, 2–9, 3–6, pthread–11, pthread–13, pthread–34, pthread–37
 inherit scheduling attribute, 2–5, pthread–15, pthread–40
 initializing, pthread–29
 naming, pthread–17, pthread–43
 scheduling parameters, pthread–19, pthread–45
 scheduling policy attribute, 2–5, pthread–21, pthread–48
 scheduling priority attribute, 2–7
 setting attributes in, 2–4
 stack address attribute, 2–8, pthread–25, pthread–53
 stacksize attribute, 2–8, pthread–27, pthread–56
THREADCP tool (OpenVMS), B–9
Thread-independent services (tis) interface, 1–20, tis–1
 condition variables, 4–4
 features of, 4–1
 mutexes, 4–3
 performance of routines, 4–2
 read-write locks, 4–4
 run-time linkages to routines, 4–2
 summary of routines, 1–21
 thread-specific data, 4–4
Thread objects
 naming, pthread–112, pthread–176
Thread-reentrant code, 3–13
Threads
 See also Multithreaded programming
 advantages of, 1–1
 attributes of, 2–4
 avoiding nonreentrant routines, 1–8
 cancelability state, 2–20
 cancelability type, 2–20
 canceling, 2–19, pthread–58
  asynchronously, 2–20
  code example, 2–23
  control of, 2–20
  delivery of cancelation request, pthread–185

Threads
canceling (cont'd)
exception-based implementation, 2–19
PTHREAD_CANCELED return value, 2–19
synchronously, 2–20
whether enabled, 2–20
changes of state, 1–5
cleanup
from asynchronous cancelation, 2–22
from synchronous cancelation, 2–21
cleanup handlers, 2–14, pthread–60, pthread–62
concurrency level, pthread–110, pthread–174
contention scope, 2–9
context-switching
in kernel, 3–2
in user mode, 3–1
creating, 2–3, pthread–89
deadlocks among, 1–9
delaying execution of, pthread–94
delivering cancelation requests, tis–50
destroying, 2–14, pthread–96
detaching, 2–14, pthread–96
executing, 1–4
identifiers
comparing, pthread–98
getting, pthread–167, tis–45
joining with another thread, 2–15, pthread–122
locking mutexes, pthread–160, tis–28
one-time initialization of, 3–12, pthread–164, tis–32
on multiprocessor systems, 1–2
overview of, 1–2
priority inversion among, 1–9
process contention scope, A–5
race conditions among, 1–9
reentrant code for, 1–5
scheduling, 2–16
alternative policies, 2–5
alternative priorities, 2–7
calculating priority, 2–16

Threads
scheduling (cont'd)
effects of scheduling policy, 2–17
inheriting attributes, 2–5
issues, 3–7
real-time (DIGITAL UNIX), A–4
scheduling parameters
getting, pthread–114
setting, pthread–178
sending signals to, pthread–134
sequence numbers
getting, pthread–116
signal masks for (DIGITAL UNIX)
getting, pthread–183
setting, pthread–183
starting, 2–3
synchronizing memory access, 3–3
system contention scope, A–4
terminating, 2–11
due to error, pthread–89
normally, pthread–89
series of actions, 2–11, pthread–91
via pthread_exit() routine, pthread–108
thread-specific data, 2–34
timeslicing of, 2–5
unlocking DECthreads global mutex, pthread–186
unlocking mutexes, pthread–162, tis–30
unlocking the DECthreads global mutex, tis–51
using a stack guard area, 2–9
using dynamic memory, 3–5
using shared memory, 3–4
using stack memory, 3–5
using static memory, 3–4
waiting for another thread to terminate, 2–15, pthread–122
waiting on mutexes, pthread–156
wakeups for
broadcasting, pthread–68, tis–3
signaling, pthread–79, pthread–81, tis–9
yielding to another thread, pthread–192

Thread-safe code, 3–13
  in libraries, 4–1
  requires reentrant compilation, 4–2
  using condition variables, 4–4
  using mutexes, 4–3
  using read-write locks, 4–4
  using thread-specific data, 4–4
Thread sequence numbers
  getting, pthread–116
Thread signal masks (DIGITAL UNIX)
  getting, pthread–183
  setting, pthread–183
Thread-specific data, 2–34
  keys
    creating, pthread–125, tis–15
    destroying, pthread–128, tis–18
    getting, pthread–118, tis–13
    naming, pthread–130, pthread–132
    setting, pthread–181, tis–48
  under the DECthreads thread-independent services (tis) interface, 4–4
  using in thread-safe library code, 4–4
Thread stacks, 3–5
  handling overflow, 3–7
  identifying overflow, 3–6
  identifying overflow of, 2–9
  managing, 3–6
  setting the origin address, 2–8
  size of, 3–6
  tracing, D–3
  using a stack guard area, 2–9, 3–6
    under DIGITAL UNIX, A–10
Throughput scheduling policy, 2–5
Time
  expiration
    obtaining, pthread–120
Timeslicing
  of threads, 2–5
tis interface
  See Thread-independent services (tis) interface
tis_cond_broadcast() routine, tis–3
tis_cond_destroy() routine, tis–5
tis_cond_init() routine, tis–7
tis_cond_signal() routine, tis–9
tis_cond_wait() routine, tis–11
  using, 4–3
tis_getspecific() routine, tis–13
tis_key_create() routine, tis–15
tis_key_delete() routine, tis–18
tis_lock_global() routine, tis–20
  using, 4–3
tis_mutex_destroy() routine, tis–22
tis_mutex_init() routine, tis–24
  using, 4–3
tis_mutex_lock() routine, tis–26
tis_mutex_trylock() routine, tis–28
tis_mutex_unlock() routine, tis–30
tis_once() routine, tis–32
tis_read_lock() routine, tis–35
  using, 4–5
tis_read_trylock() routine, tis–37
  using, 4–5
tis_read_unlock() routine, tis–39
tis_rwlock_destroy() routine, tis–41
  using, 4–5
tis_rwlock_init() routine, tis–43
  using, 4–5
tis_self() routine, tis–45
tis_setcancelstate() routine, tis–46
tis_setspecific() routine, tis–48
tis_testcancel() routine, tis–50
  using, 4–3
tis_unlock_global() routine, tis–51
  using, 4–3
tis_write_lock() routine, tis–53
  using, 4–5
tis_write_trylock() routine, tis–55
  using, 4–5
tis_write_unlock() routine, tis–57
TRY macro, 5–10
  restrictions, B–4
Two-level scheduling
  under DIGITAL UNIX, A–3
  under OpenVMS Alpha, B–6

U

Upcalls
- under DIGITAL UNIX, A–3
- under OpenVMS, B–7
  - due to $HIBER and $WAKE system services, B–14

User threads, A–3
- relation to kernel threads
  - under DIGITAL UNIX, A–4
  - under OpenVMS, B–6, B–11

V

Virtual processors (OpenVMS)
- for kernel threads, B–11

W

$WAKE system service (OpenVMS), B–14

Wakeups
- for threads
  - broadcasting, pthread–68, tis–3
  - signaling, pthread–79, pthread–81, tis–9

Win32 application programming interface
- interoperability with DECthreads, C–3

WinDbg debugger (Windows NT), D–3

Windows NT operating system
- DECthreads interfaces, C–1
- supported DECthreads interfaces, C–1
- using DECthreads with, C–1

Work crew functional model, 1–6

Work queues
- variation of boss/worker functional model, 1–6