Guide to the POSIX Threads Library

Order Number: AA–QSBPD–TE

April 2001

This guide reviews the principles of multithreaded programming, as reflected in the IEEE POSIX 1003.1-1996 standard, and provides implementation guidelines and reference information for the Compaq Multithreading Run-Time Library.

ZK 6493

The Compaq OpenVMS documentation set is available on CD–ROM.

This document was prepared using DECdocument, Version 3.3-1b.

Contents

Part I Compaq POSIX Threads Library Overview and Programming Guidelines

1 Introducing Multithreaded Programming

2 Objects and Operations

3   Programming with Threads

6 Examples

Part II POSIX.1 (pthread) Routines Reference

Part III Compaq Proprietary Interfaces: tis Routines Reference

Part IV Appendixes

A Considerations for Tru64 UNIX Systems

B Considerations for OpenVMS Systems

C Debugging Multithreaded Applications

D Migrating from the cma Interface

E Migrating from the d4 Interface

Glossary

Index

Examples

Figures

Tables

Preface

This guide describes the POSIX Threads Library, Compaq's Multithreading Run-Time Library. In addition to introducing components for building multithreaded applications and libraries to be called from either single-threaded or multithreaded programs, this guide reviews the key principles of multithreaded programming.

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

- For building portable, multithreaded applications, Compaq provides the **pthread** interface. This is an implementation of the POSIX standard 1003.1c-1995 (part of 1003.1-1996). This interface adds extensions specified in The Open Group's *Single Unix Specification, Version 2* (SUSV2), also known as XSH5, part of the UNIX98 brand.

- For building libraries whose routines can be called in either a single-threaded or multithreaded context, Compaq provides a proprietary thread-independent services (or **tis**) interface.

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

As a complement to this guide, and for a user's guide to multithreaded programming using the pthreads standard, we recommend the following:

> *Programming with POSIX Threads* by David R. Butenhof, published as part of the Addison-Wesley Professional Computing Series (ISBN 0-201-63392-2). *The Single UNIX Specification, Version 2*, The Open Group (ISBN 85912-181-0).
> Available online at http://www.opengroup.org/onlinepubs/7908799/toc.htm.

Intended Audience

This guide is for system and application programmers who use the POSIX Threads Library either to create multithreaded applications or to create thread-safe code libraries that can be called from either 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 the POSIX Threads Library.

- 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 POSIX Threads Library exception package.

- Chapter 6 contains example programs demonstrating how to call library 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 POSIX Threads Library issues and restrictions specific to Compaq Tru64 UNIX systems.

- Appendix B discusses POSIX Threads Library issues and restrictions specific to OpenVMS systems.

- Appendix C discusses debugging issues for a multithreaded program that uses the POSIX Threads Library.

- Appendix D summarizes the differences between the obsolete Compaq-proprietary CMA (or **cma**) interface and the Compaq **pthread** interface. Use this appendix to help you migrate your programs and applications to the **pthread** interface.

- Appendix E summarizes the differences between the retired POSIX 1003.4a/Draft 4 (**d4** or **DCE threads**) interface and the Compaq **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. This manual covers the version of the POSIX Threads Library available on the following platforms:

- Tru64 UNIX 5.0A or higher

- OpenVMS Alpha Version 7.3

- OpenVMS VAX Version 7.3

For a complete list and description of the books in the OpenVMS documentation set, see the *OpenVMS Version 7.3 New Features and Documentation Overview.*

Some books in the OpenVMS or Tru64 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 New Features Manual provides information on all of the books in the OpenVMS or Tru64 UNIX documentation set.

For additional information on the Open Systems Software Group (OSSG) products and services, access the Compaq World Wide Web site at the following location:

```
http://www.openvms.compaq.com/
```

Reader's Comments

Compaq welcomes your comments on this guide. Please send comments to any of the following addresses:

Fax	603 881-0120, Attention: Core Technology Group, ZKO2-3/Q18
Mail	Compaq Computer Corporation Core Technology Group, ZKO2-3/Q18 110 Spit Brook Rd. Nashua, NH 03062-2698

Conventions

VMScluster systems are now referred to as OpenVMS Cluster systems. Unless otherwise specified, references to OpenVMS Clusters or clusters in this document are synonymous with VMSclusters.

In this manual, every use of DECwindows and DECwindows Motif refers to DECwindows Motif for OpenVMS software.

The following conventions are used in this manual:

Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 x	A sequence such as PF1 x indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
	In the HTML version of this document, this convention appears as brackets, rather than a box.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities:
	• Additional optional arguments in a statement have been omitted.
	• The preceding item or items can be repeated one or more times.
	• Additional parameters, values, or other information can be entered.
. . .	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 command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{}	In command format descriptions, braces indicate required choices; you must choose one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
	In the HTML version of this document, this convention appears as italic text.

italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace text`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Part I

Compaq POSIX Threads Library Overview and Programming Guidelines

Part I contains chapters that provide an overview and concepts of the Threads Library as well as define programming disciplines and guidelines for writing a multithreaded program.

1

Introducing 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 POSIX Threads Library 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 POSIX Threads Library are capable of utilizing multiprocessors, if the target platform supports parallelism within a process. Compaq's Tru64 UNIX 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
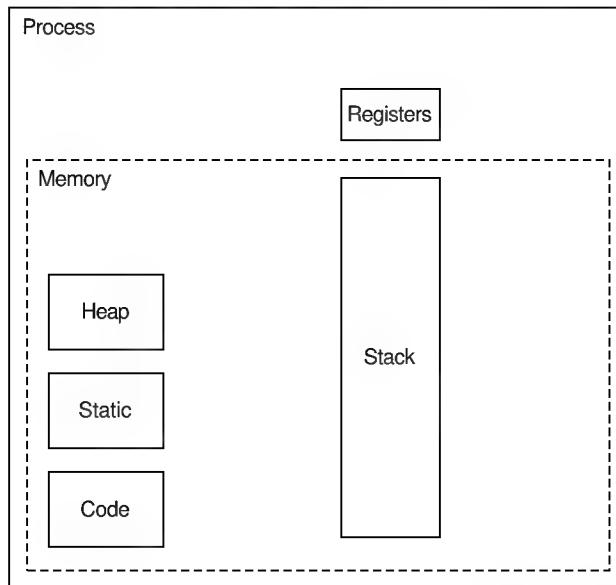
ZK–3913A–GE

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 Compaq's multithreading run-time library, a programmer can create several threads within a process. The process' 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' 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. The Threads Library provides routines that allow you to use these and other synchronization objects. Section 2.4 describes the synchronization objects that the Threads Library offers as well as the operations your program can perform on them.

Figure 1–2 Multithreaded Process
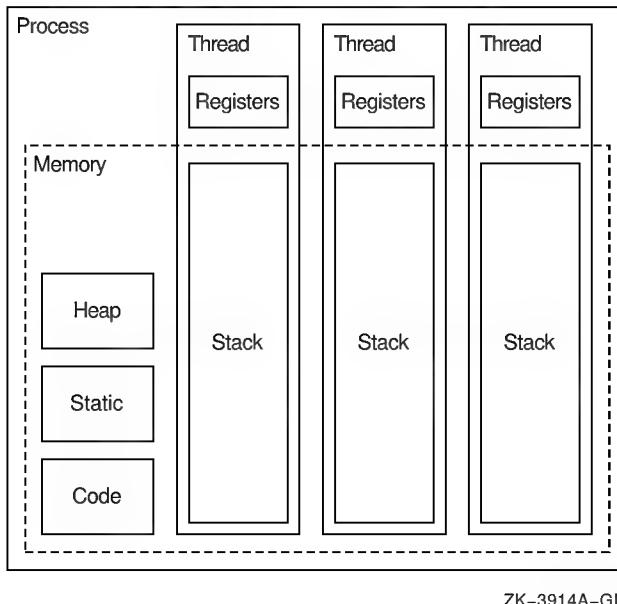
ZK–3914A–GE

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 thread scheduler, part of the run-time environment that the Threads Library establishes before your program begins running. Nevertheless, your program can influence how threads are scheduled 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 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:

- Blocked—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
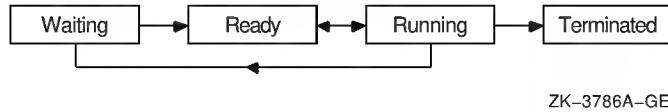
ZK–3786A–GE

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. When there are multiple bosses, this is often called producer/consumer.

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

TASK

Setup	Thread A	Cleanup
	Thread B	
	Thread C	

(Time)

ZK–3787A–GE
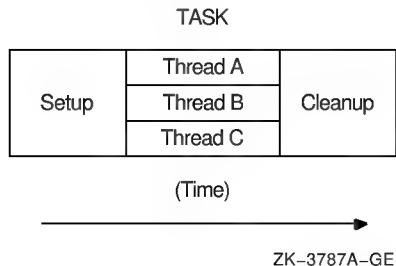
1.4.3 Pipelining Model

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

TASK

Thread A	Thread B	Thread C

(Time)

ZK–3788A–GE
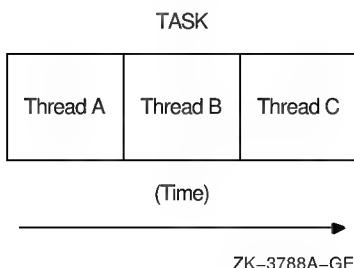
1.4.4 Combination of Functional Models

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 Programming Issues for Multithreaded Programs

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, Compaq's C, C++, Ada, Pascal, COBOL, FORTRAN 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 3.9.1 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 3.9.2.) 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.

When you design and code a multithreaded program, you must also 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 prevent conflicts with other threads that use the same nonreentrant routine or library. Section 3.9 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, which depends on, and varies with, the precise timing of the threads' execution. 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 POSIX Threads Libraries and Interfaces

As a package, the POSIX Threads Library 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 Threads Libraries depend.

From the programmer's view, the Threads 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 the Threads Library supports two interfaces that are documented in this guide:

- The **pthread** interface provides multithreading capability in your applications. This interface is based on the Single UNIX Specification (SUSV2), which incorporates the POSIX thread standard (1003.1c-1995). 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 Compaq 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 the **pthread** interface.

  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 the Threads Library includes interface definitions for the C programming language only. However, all Threads Library routines are callable from languages other than C. Your application must provide its own declarations for Threads Library routines in a manner appropriate for 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 the Threads Library also supports other interfaces that are not documented in this guide. See Section 1.6.3.

Special note when using the Threads Library from non-C languages:

Several Threads Library features and most Threads Library identifiers are provided as C language macros. As such, their definitions may not be available in other languages. Developers working in other languages will have to provide their own declarations of functions and constants. Features such as `TRY/CATCH` exception handling and POSIX push/pop cleanup handlers may be completely unavailable, although it may be possible to provide similar functionality using native exception handling facilities. Note that in this context, C++ is a non-C language; while the C language macros may compile successfully under C++, `TRY/CATCH` and push/pop cleanup handlers are not supported for C++ code. C++ code should use object destructors and C++ exception handlers.

1.6.1 The pthread Multithreading Interface

The **pthread** interface routines implement the IEEE Standard 1003.1-1996, Portable Operating System Interface (or POSIX) Application Program Interface, also known as **POSIX.1**. It also supports extensions specified in SUSV2 (UNIX98).

Table 1–1 lists and summarizes functionally the **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

- Read-write lock routines

- Read-write lock attributes object routines

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

- Thread execution routines

- Global mutex routines

- Mutex attributes routines

- Condition variable routines

- Object naming routines

- Exception object routines

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

This guide also documents several routines that are not declared entries in the **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 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

Table 1–1 (Cont.) pthread Routines Summary

Routine	Description
Thread Attributes Object Routines	
pthread_attr_destroy()	Destroys a thread attributes object
pthread_attr_getdetachstate()	Obtains the detachstate attribute of 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 of the specified thread attributes object
pthread_attr_getschedparam()	Obtains the scheduling parameters for the scheduling policy 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
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 of 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 of the specified thread attributes object
pthread_attr_setstacksize()	Changes the stacksize attribute of the specified thread attributes object

Table 1–1 (Cont.) 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 while the handler is on the "cleanup stack"
pthread_setcancelstate()	Sets the calling thread's cancelability state to enable or disable the delivery of cancelation requests
pthread_setcanceltype()	Sets the calling thread's cancelability type to enable or disable the delivery of cancelation requests
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
Thread-Specific Data Routines	
pthread_getspecific()	Obtains the thread-specific data value associated with the specified key
pthread_key_create()	Generates a unique thread-specific data key for the calling thread
pthread_key_delete()	Deletes a thread-specific data key
pthread_setspecific()	Changes the thread-specific data value associated with the specified key for the calling thread

Table 1–1 (Cont.) pthread Routines Summary

Routine	Description
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 before locking it
`pthread_mutex_trylock()`	Attempts to lock a mutex; returns immediately if mutex is already locked
`pthread_mutex_unlock()`	Unlocks a mutex locked by the calling thread
Mutex Attributes Object Routines	
`pthread_mutexattr_destroy()`	Destroys a mutex attributes object
`pthread_mutexattr_getpshared()`	Obtains the process-shared attribute from the specified mutex attributes object
`pthread_mutexattr_gettype()`	Obtains the mutex type attribute from the specified mutex attributes object
`pthread_mutexattr_init()`	Initializes a mutex attributes object
`pthread_mutexattr_setpshared()`	Changes the process-shared attribute in the specified mutex attributes object
`pthread_mutexattr_settype()`	Changes the mutex type attribute in the specified mutex attributes object
Condition Variable Routines	
`pthread_cond_broadcast()`	Wakes all threads currently 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 broadcast
`pthread_cond_wait()`	Causes a thread to wait for a condition variable to be signaled or broadcast

Table 1–1 (Cont.) pthread Routines Summary

Routine	Description
Condition Variable Attributes Object Routines	
pthread_condattr_destroy()	Destroys a condition variable attributes object
pthread_condattr_getpshared()	Obtains the process-shared attribute from the specified condition variable attributes object
pthread_condattr_init()	Initializes a condition variable attributes object
pthread_condattr_setpshared()	Changes the process-shared attribute in the specified condition variable attributes object
Read-Write Lock Routines	
pthread_rwlock_destroy()	Destroys a read-write lock object
pthread_rwlock_init()	Initializes a read-write lock object
pthread_rwlock_rdlock()	Acquires a read-write lock for read access; if locked, the caller waits for the lock to become available before locking it
pthread_rwlock_tryrdlock()	Acquires a read-write lock for read access without waiting
pthread_rwlock_trywrlock()	Acquires a a read-write lock for write access without waiting
pthread_rwlock_unlock()	Releases a read-write lock previously acquired by the calling thread
pthread_rwlock_wrlock()	Acquires a read-write lock for write access; if locked, the caller waits for the lock to become available before locking it
Read-Write Lock Attributes Object Routines	
pthread_rwlockattr_destroy()	Destroys a read-write lock attributes object
pthread_rwlockattr_getpshared()	Obtains the process-shared attribute from the specified read-write lock attributes object
pthread_rwlockattr_init()	Initializes a read-write lock attributes object
pthread_rwlockattr_setpshared()	Changes the process-shared attribute in the specified read-write lock attributes object

Table 1–1 (Cont.)   pthread Routines Summary

Routine	Description
Nonportable Extensions	
`pthread_delay_np()`	Pauses the calling thread's execution for the specified time interval
`pthread_get_expiration_np()`	Calculates a timeout for a timed condition variable wait
`pthread_getsequence_np()`	Gets a small integer specific to the calling thread
`pthread_attr_getstackaddr_np()`	Obtains the address and size of the specified thread attributes object
`pthread_attr_setstackaddr_np()`	Sets the address and size of the specified thread attributes object
`pthread_lock_global_np()`	Locks the global mutex
`pthread_unlock_global_np()`	Unlocks the global mutex
`pthread_cond_signal_int_np()`	Requests condition variable signal from software interrupt routine
`pthread_cond_sig_preempt_int_np()`	Wakes one thread that is waiting on the specified condition variable; called from software interrupt routine

`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_rwlock_getname_np()`
`pthread_rwlock_setname_np()`
`pthread_setname_np()`
	Gets/sets name associated with specific objects for debugging

`pthread_exc_get_status_np()`
`pthread_exc_matches_np()`
`pthread_exc_report_np()`
`pthread_exc_set_status_np()`
	Exception object routines (some are macros)

`pthread_yield_np()`	Notifies the scheduler that the current thread is willing to release its processor to other threads of the same or higher priority (alias for `sched_yield()`)

Table 1–1 (Cont.) 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 priority
sigwait()	Suspends a calling thread until a signal arrives

1.6.1.1 Optionally Implemented POSIX.1 Routines

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

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

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

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

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

- (Not supported for OpenVMS systems) Reported by the POSIX.1 `_POSIX_THREAD_PROCESS_SHARED` macro:

  ```
  pthread_condattr_getpshared()
  pthread_condattr_setpshared()
  pthread_mutexattr_getpshared()
  pthread_mutexattr_setpshared()
  pthread_rwlockattr_getpshared()
  pthread_rwlockattr_setpshared()
  ```

The POSIX.1 standard directs the Threads Library 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.

The Threads Library does provide the following macros specified in the POSIX.1 standard:

`_POSIX_THREADS`: threads are supported
`_POSIX_THREAD_SAFE_FUNCTIONS`: thread-safe libraries are supported
`_POSIX_THREAD_ATTR_STACKSIZE`: can specify stack size
`_POSIX_THREAD_ATTR_STACKADDR`: can specify stack address
`_POSIX_THREAD_PRIORITY_SCHEDULING`: real-time scheduling control is supported
`_POSIX_THREAD_PROCESS_SHARED`: cross-process synchronization is supported (Tru64 UNIX only)

1.6.2 Thread-Independent Services Interface

The Compaq 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 pthreads. That is, use **tis** routines to build thread-safe code libraries whose routines can be called from either a single-threaded or a multithreaded environment.

In the absence of threads, **tis** routines impose minimal overhead on the calling program. For instance, **tis** routines avoid the use of interlocked instructions and memory barriers.

When threads are present, **tis** routines provide full support for synchronization. Note that there are no **tis** routines for creating threads or thread objects, because these routines 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 ---

Unlike the other **tis** interfaces, the read-write lock functions work on a data type different from that used by the **pthread** read-write lock functions.

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

Table 1–2   tis Routines Summary

Routine	Description
General Routines	
`tis_once()`	Calls an initialization routine to be executed only once
`tis_self()`	Obtains the identifier of the calling thread
`tis_yield()`	Notifies the scheduler that the calling thread is willing to release its processor to other threads of the same or higher priority
Thread Cancelation Routines	
`tis_setcancelstate()`	Sets the calling thread's cancelability state to enable or disable the delivery of cancelation requests
`tis_testcancel()`	Requests delivery of any pending cancelation request to the calling thread
Thread-Specific Data Key Routines	
`tis_getspecific()`	Obtains the thread-specific data associated with the specified key for the calling thread
`tis_key_create()`	Generates a unique thread-specific data key
`tis_key_delete()`	Deletes a thread-specific data key
`tis_setspecific()`	Changes the thread-specific data value associated with the specified key for the calling thread

Table 1–2 (Cont.) tis Routines Summary

Routine	Description
Mutex Routines	
tis_lock_global()	Locks the global mutex
tis_mutex_destroy()	Destroys the specified mutex object
tis_mutex_init()	Initializes a mutex object
tis_mutex_lock()	Locks the specified mutex, if unlocked
tis_mutex_trylock()	Tries to lock the specified mutex
tis_mutex_unlock()	Unlocks the specified mutex when locked by the calling thread
tis_unlock_global()	Unlocks the global mutex
Condition Variable Routines	
tis_cond_broadcast()	Wakes all threads currently 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_timedwait()	Causes a thread to wait a specified period of time for a condition variable to be signaled or broadcast
tis_cond_wait()	Causes the calling thread to wait for the specified condition variable to be signaled or broadcast
tis_get_expiration()	Calculates a timeout for a timed condition variable wait
OpenVMS I/O Completion Routines	
tis_io_complete()	Completion AST service routine
tis_sync()	Thread-synchronous replacement for $SYNC system service

Table 1–2 (Cont.)   tis Routines Summary

Routine	Description
Read-Write Lock Routines	
tis_read_lock()	Acquires the specified read-write lock for read access
tis_read_trylock()	Acquires 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 by the calling thread
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()	Acquires 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 by the calling thread

1.6.3 Undocumented and Obsolete Interfaces

Previous versions of the Threads Library offered interfaces that under this version are no longer documented.

1.6.3.1 The cma Interface

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

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

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

1.6.3.2 The d4 (DCEthread) Interfaces

_____ **Note** _____

These obsolete interfaces will be removed in a future Compaq POSIX Threads release. As of that release, both source and binary code using the d4 (DCEthread) interfaces will no longer compile or execute.

_____

For backward compatibility only, this version of the Threads Library retains full binary support for the **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 **d4** interfaces will not be provided in a future release of the Threads Library. Compaq recommends that you migrate any **d4** code in your existing applications to the latest **pthread** interface, to take advantage of its standard features, portability, and future enhancements.

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

2

Objects and Operations

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

2.1 Threads and Synchronization Objects

A multithreaded program typically manipulates these objects:

- A **thread object** describes a **thread**, which refers to a distinct flow of control within a process. After a thread object is created, the Threads Library uses it to maintain information about the thread's state and its associated attributes.

- A **mutex** serves as a lock for data that is shared among the program's threads. To access data that is guarded by a mutex, a thread must acquire the mutex, access the data, and then release the mutex. Each instance of acquiring a mutex is called a **lock acquisition**. While a mutex is locked, if other threads attempt to acquire that mutex, those threads must wait for the mutex to be released.

- For data that is shared among a program's threads but is more frequently read than written, use a **read-write lock** to guard access to the data. Unlike a mutex, more than one thread can acquire the same read-write lock for read access at the same time.

- When associated with a shared data object and its mutex, a **condition variable** provides a mechanism that allows a thread to wait until a piece of shared data protected by a mutex is placed into a particular state.

2.2 Attributes Objects

When your program creates a thread, mutex, read-write lock or condition variable, it can accept the default attributes for that object or specify an existing attributes object (previously created by your program) that contains particular attribute values. You can also change some of the attributes of a thread after it has begun execution—for example, you can change the thread's priority. However, other attributes, such as stack size, are fixed at execution.

To initialize 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_mutexattr_init()` for mutex attributes
- `pthread_rwlockattr_init()` for read-write lock attributes
- `pthread_condattr_init()` for condition variable attributes

These routines initialize an attributes object with 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.

Initializing an attributes object (or changing the values in an attributes object) does not affect the attributes of existing threads, mutexes, read-write locks 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

- `pthread_rwlockattr_destroy()` for read-write lock attributes objects

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

2.3 Thread Operations

The following sections describe these operations on threads:

- Creating a thread

- Setting the attributes for a new thread

- Terminating a thread

- Detaching and destroying a thread

- Joining with another thread

- Controlling how a thread is scheduled

- Canceling a thread

2.3.1 Creating a Thread

Your program creates a thread using the `pthread_create()` routine. This routine creates a thread based on the settings of the thread attributes object if specified, which your program must have previously initialized. If called without a specified thread attributes object, `pthread_create` creates a new thread that has the default attributes.

The Threads Library 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 attributes, the new thread might preempt its creator (that is, it might start before the call to `pthread_create()` returns). The caller of `pthread_create()` can synchronize with the new thread using any mutually agreed upon mechanism or await its termination using `pthread_join()`.

The Threads Library assigns each new thread a thread identifier, which is written into the address specified as the `pthread_create()` routine's thread argument. The new thread's identifier is written before the new thread executes.

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_setdetachstate()` routine, to `PTHREAD_CREATE_DETACHED`. This is useful for creating a thread that your program knows will not be joined by any

other thread. That is, when such a thread terminates, the thread and its thread object are automatically destroyed.

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. Allocate a thread attributes object and then initialize it by calling the `pthread_attr_init()` routine. (Normally, you will initialize an extern or local variable of the appropriate type.)

2. Set values for the individual attributes of the thread attributes object. (The POSIX 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, it 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_init()` and `pthread_attr_destroy()` 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

By default, a new thread is created with the scheduling attributes (policy, parameters and contention scope) of its creator. If an attributes object is specified, the scheduling attribute values are ignored. When you want to create a thread with different scheduling attributes, you must set the attribute values, and also set the value of the inheritsched attribute to `PTHREAD_EXPLICIT_SCHED`. You do this by calling the `pthread_attr_setinheritsched()` routine. The default value is `PTHREAD_INHERIT_SCHED`.

2.3.2.2 Setting the Scheduling Policy Attribute

The scheduling policy attribute describes how new threads are scheduled for execution relative to the other threads in the process.

A thread has one of the following scheduling policies:

- `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 the current thread is preempted and the higher priority thread immediately begins running.

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

  On a multiprocessor, threads of varying policy and priority may run simultaneously. A high priority thread is not guaranteed exclusive use of a multiprocessor system. You must use synchronization, not scheduling attributes, to ensure exclusive access.

- `SCHED_OTHER` (Foreground or "throughput"; also known as `SCHED_FG_NP`)— This is the default scheduling policy. All threads are time sliced, and no thread with this policy will completely starve any other thread with this policy, regardless of any thread's priority. (**Time slicing** is a mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.) However, higher-priority threads tend to receive more execution time than lower-priority threads, if the threads behave similarly.

  Threads with this scheduling policy can be denied execution time by first-in/first-out (FIFO) or round-robin (RR) threads. Threads in this policy do not preempt other threads.

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

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 as shown in Section 2.3.2.1. Inheriting scheduling policy 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.

When you change the scheduling policy attribute, you must be sure the scheduling parameter attribute is compatible with the scheduling policy attribute before using the attributes object to create a thread.

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 does not execute often, the Threads Library tends to give it high access to the processor because it is not greatly affecting other threads. On the other hand, the Threads Library 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 the Threads Library to schedule such threads strictly by the specified priority. Because threads that use real-time scheduling policies require additional overhead, the 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-conformant implementations. The other scheduling policies are extensions to the POSIX standard.

Note

The SCHED_OTHER identifier is portable, but the POSIX standard does not specify the behavior that it signifies. For example, on non-Compaq platforms, the `SCHED_OTHER` scheduling policy could be identical to either the `SCHED_FIFO` or the `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. The Threads Library 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 parameters of an existing thread by calling the `pthread_setschedparam()` routine and requesting the current policy with the new parameters. 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.

Note

On Tru64 UNIX Systems:
There are system security issues for threads running with system contention scope. High priority threads may prevent other users from accessing the system. A system contention scope thread cannot have a priority higher than 19 (the default user priority). A system contention scope thread with `SCHED_FIFO` policy, because it will prevent execution by other threads of equal priority, cannot have a priority higher than 18.

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 the specified 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 for use as a thread's stack. The value of the stack address attribute represents the origin of the thread's stack (that is, the initial value to be placed in the thread's stack pointer register). 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.

Code using this attribute is nonportable because the meaning of "stack address" is undefined and untestable. Generally, implementations likely assume, as does the Threads Library, that you have specified the initial stack pointer; however, this is not required by the standards. Even so, some machines' stacks grow up while others grow down, and many may modify the stack pointer either before or after writing (or reading) data. In other words, one system may require that you pass the base, another `base - sizeof(int)`, another `base + size`, another `base + size + sizeof(long)`. Furthermore, the system cannot know the size of the stack, which may restrict the ability of debuggers and other tools to help you. As long as you are using an inherently nonportable interface, consider using `pthread_attr_setstackaddr_np()`.

You cannot create two concurrent threads that use the same stack address. The amount of storage you provide must be at least `PTHREAD_STACK_MIN` bytes.

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 and the stack. A guard area is a region of no-access memory that is allocated at the overflow end of the thread's writable stack. When the thread attempts to access a memory location within the guard area, 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-specific appendixes.

The Threads Library 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 is otherwise a waste of 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. In this case, no guard area or overflow warning area are allocated.

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.

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. You must also set the inheritsched attribute to `PTHREAD_EXPLICIT_SCHED` to prevent a new thread from inheriting its contention scope from the creator.

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.

The Threads Library selects at most one thread to execute on each processor at any point in time. The Threads Library 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. Attempting to set the contention scope attribute to a value not supported on your system will result in a return value of [ENOTSUP]. The following table summarizes support for thread contention scope by operating system:

Table 2–1 Support for Thread Contention Scope

Operating System	Available Thread Contention Scopes	Default Thread Contention Scope
Tru64 UNIX	Process System	Process
OpenVMS	Process	Process

_____ **Note** _____

On Tru64 UNIX systems:

When a thread creates a system contention scope thread, the creation can fail with an [EPERM] error condition. This is because system contention scope threads can only be created with priority above "default" priority if the process is running with root privileges.

2.3.3 Terminating a Thread

Terminating a thread means causing 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 value returned by the routine indicates the thread's exit status to a thread that joins with this thread.

- The thread calls the `pthread_exit()` routine. This routine accepts a status value in its value_ptr argument. The value returned by the routine 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, the Threads Library performs these actions:

1. It writes a return value into the terminated thread's thread object:

   - If the thread has been canceled, the value `PTHREAD_CANCELED` is written into the thread's thread object.

   - If the thread terminated by returning from its start routine, the return value is copied from the start routine into the thread's thread object. Alternatively, if the thread explicitly called `pthread_exit()`, the value received in the value_ptr argument (from `pthread_exit()`) is stored in 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 either a cancelation or a call to `pthread_exit()`, the Threads Library calls, in turn, each cleanup handler that this thread declared (using `pthread_cleanup_push()`) that had not yet been removed (using `pthread_cleanup_pop()`). (It also transfers control to any appropriate `CATCH`, `CATCH_ALL`, or `FINALLY` blocks, as described in Chapter 5. You can also use Compaq C's structured handling (SEH) extensions.)

   The Threads Library 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. You can also catch the exit or cancel exception using the `catch(...)`.

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

Your program can use the exception package to operate on the generated exception. (Note that the practice of using `CATCH` handlers in place of `pthread_cleanup_push()` is not portable.) Chapter 5 describes the exception package. The name of the native system extension, or that seen by C++, varies by platform.

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

   - The thread's value for the corresponding key is set to NULL.

   - The thread-specific data destructor function is called.

   This step is repeated until all thread-specific data values in the thread are NULL, or for up to a number of iterations equal to `PTHREAD_DESTRUCTOR_ITERATIONS` (4). This destroys all thread-specific data associated with the terminated thread. See Section 2.6 for more information about thread-specific data. Note that if after 4 iterations through the thread's thread-specific data values, there are still non-NULL values, they will be ignored. This may result in an application memory leak, and should be avoided.

4. The thread (if there is one) that is currently waiting to join with the terminated thread is awakened. That is, the thread that is waiting in a call to `pthread_join()` is awakened

5. If the thread is already detached or if there was a thread waiting in a call to `pthread_join()`, its storage is destroyed 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, it continues to exist as long as it is not detached. This means that storage, including stack, may remain allocated. 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 Tru64 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 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

- The raising or reraising of an exception

- A thread-directed Tru64 UNIX signal (for example, SIG_SEGV) while default signal action is in effect. (This raises an 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; however, pairs of calls can be nested.

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

If a thread has terminated, then detaching that thread causes the Threads Library to destroy it immediately. If a thread is detached before it terminates, then the Threads Library frees the thread's resources 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, the Threads Library 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. The thread should not be detached until no further references to it will be made.

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, the target thread is detached after it terminates.

Join is one form of thread synchronization. It is often useful when one thread needs to wait for another and possibly retrieve a single return value. (The value may be a pointer, for example to heap storage.) There is nothing special about join, though—similar results, or infinite variations, can be achieved by use of a mutex and condition variable.

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.

By default, the target thread of a join operation is created with the detachstate attribute of its thread attributes object set to `PTHREAD_CREATE_JOINABLE`. It should not be created with the detachstate attribute set to `PTHREAD_CREATE_DETACHED`.

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 the target thread is detached 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.

2.3.6 Scheduling a Thread

Scheduling means to evaluate and change the states of the process' threads. As your multithreaded program runs, the Threads Library detects whether each thread is ready to execute, is waiting for a synchronization object, or has terminated, and so on.

Also, for each thread, the Threads Library regularly checks whether that thread's scheduling priority and scheduling policy, when compared with those of the process' 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 the Threads Library 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 a substitute for 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 default scheduling policy and the lowest scheduling priority setting, the Threads Library 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. Refer to Section 3.1.3 for more information about using thread scheduling as thread synchronization.

The following figures demonstrate how the Threads Library 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

D ⟶ B ⟶ C ⟶ A ⟶
ZK–3789A–GE
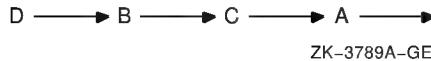
Thread D executes until it waits or terminates. Next, although thread B and thread C have the same priority, thread B starts because it has been waiting longer than thread C. Thread B executes until it waits or terminates, then thread C executes until it waits or terminates. Finally, thread A executes.

Figure 2–2 shows a flow with RR scheduling.

Figure 2–2   Flow with RR Scheduling

D ⟶ B ⟶ C ⟶ B ⟶ C ⟶ A ⟶
ZK–3790A–GE
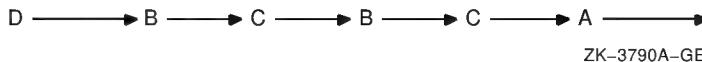
Thread D executes until it waits or terminates. Next, thread B and thread C are time sliced, because they both have the same priority. Finally, thread A executes.

Figure 2–3 shows a flow with Default scheduling.

Figure 2–3   Flow with Default Scheduling

D ⟶ B ⟶ C ⟶ A ⟶ B ⟶ C ⟶ . . . .
ZK–3791A–GE
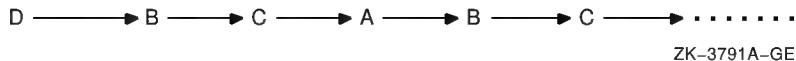
Threads D, B, C, and A are time sliced, even though thread A has a lower priority than the others. Thread A receives less execution time than thread D, B, or C if any of those are ready to execute as often as Thread A. However, the default scheduling policy protects thread A against indefinitely being 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 requesting 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 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. The target thread might have pushed cleanup handler routines (using the `pthread_cleanup_push()` routine) on its handler stack. When the target thread receives the cancelation request, the Threads Library unwinds the thread's call stack. For each frame, active exception handlers are invoked. These include cleanup handler routines, C++ object destructors, Compaq C SEH `except` clauses, and C++ `catch(...)` clauses.

2.3.7.1 Thread Cancelation Implemented Using Exceptions

The Threads Library implements thread cancelation using exceptions. Using the exception package, it is possible for a thread (to which a cancelation request has been delivered) explicitly to catch the thread cancelation exception (`pthread_cancel_e`) defined by the Threads Library 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 exception package.

2.3.7.2 Thread Return Value After Cancelation

When a thread is terminated due to cancelation, the Threads Library 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, instead, they remain pending.

If the thread's cancelability state is *enabled*, a thread may use the `pthread_testcancel()` routine to request the immediate delivery of any pending cancelation request. This routine enables the program to permit cancelation to occur at places where it is convenient, when it might not otherwise occur, such as 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 if your software depends on cancelation.

A thread can use the `pthread_setcancelstate()` routine to change its cancelability state.

A thread can use the `pthread_setcanceltype()` routine to change its **cancelability type**, which determines whether it responds to a cancelation request only 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, during a call to the `pthread_cond_wait()` routine. 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 Deferred Cancelation Points

A **cancelation point** is a routine that delivers a posted cancelation request to that request's target thread.

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

```
pthread_cond_timedwait()
pthread_cond_wait()
pthread_delay_np()
pthread_join()
pthread_testcancel()
```

The following routines in the **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 Tru64 UNIX

- Section B.9 for OpenVMS

2.3.7.5 Cleanup from Deferred 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.

The Threads Library 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 was not canceled (for example, releasing the mutex after waking up from a condition variable wait). (This usually corresponds to a `FINALLY` exception handler.)

Do not use pthread cleanup handlers in C++ code. Instead, rely on C++ object destructors.

* As described in Chapter 5, use the exceptions package `TRY/CATCH/CATCH_ALL` or `TRY/FINALLY` macros to clean up during a cancelation request. A cancelation request is sent to the thread by raising a special exception. Thus, code that contains a cancelation point can be placed inside a `TRY` block, and a `CATCH`, `CATCH_ALL` 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. You can also use C++ object destructors or `catch(...)`. Do not use the `TRY` macros from C++.

2.3.7.6 Cleanup from Asynchronous Cancelation

When an application sets the cancelability type to asynchronous, cancelation may occur at any instant, even within the execution of a single instruction. 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 looping code that is compute-bound and that makes no calls and allocates no resources.

While a thread's cancelability type is asynchronous, it should not call any routine unless that routine 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

In general, you should expect that no run-time library routine is safe for asynchronous cancelation, unless explicitly documented to the contrary. Only three routines are safe for asynchronous cancelation: `pthread_setcanceltype()`, `pthread_setcancelstate()` and `pthread_cancel()`.

For additional information about accomplishing asynchronous cancelation for your platform, see Section A.4 and Section B.9.

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.   */
       {
         .
         .
         .
         s = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &inner_c_s);
         if(s == 0)
            .
            .
            .
            /* Now cancelation is enabled.   */
            .
            .
            .
            /* Enable asynchronous cancelation this time.   */
               {
                 .
                 .
                 .
```

(continued on next page)

Example 2–1 (Cont.) pthread Cancel

```
                /* 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.
                       */
                }
             .
              .
               .
             }
  .
  .
  .
/* 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 three kinds of synchronization objects: mutexes, condition variables, and read-write locks.

2.4.1 Mutexes

A **mutex** (or mutual exclusion) 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
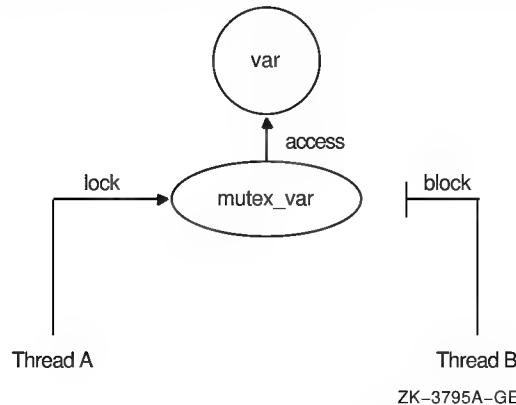
ZK–3795A–GE

Each mutex must be initialized before use. The Threads Library supports static initialization of static or extern mutexes at compile time, using the PTHREAD_MUTEX_INITIALIZER macro 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 the most efficient type of mutex, but also the least forgiving.

A normal mutex usually does not record or 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; this means that some potentially severe application errors may not be detected. Normal mutexes also provide less debugging information, because the owner cannot be identified.

2.4.1.2 Default Mutex

This is the name reserved by the Single UNIX Specification, Version 2, for a vendor's default mutex type. For the **pthread** interface, the "normal" mutex type is the "default" mutex type. Be aware that other implementations could implement "default" errorcheck, recursive, or even a nonportable mutex type.

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. The Threads Library will always detect and report an attempt by any thread other than the owner to unlock the mutex.

A recursive mutex is useful when a routine requires exclusive access to a piece of data, and cannot tell whether its caller already owns the mutex. This is common when converting old code to be thread-safe. However, the code must ensure that the shared data is in a consistent state before calling another routine which requires access to it under the lock.

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 may be slower than normal mutexes, because they do more internal tracking. The debugger can always display the current owner (if any) of an errorcheck mutex. Any correct program that works with normal mutexes will also work with errorcheck mutexes.

2.4.1.5 Mutex Operations

To lock a mutex, use one of the following routines, depending on what you want to happen if the mutex is already locked by another thread:

- `pthread_mutex_lock()`

  If the mutex is locked, the calling 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 a running unlocking thread attempts to relock the mutex before the first waiter gets a chance to run, the running 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 or has threads waiting on it.

_____ **Warning** _____

The Threads Library does not currently detect deadlock conditions involving more than one mutex, but may in the future. *Never* write code that depends upon the Threads Library **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.

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.

The process-shared attribute specifies whether a mutex can be operated upon by threads in only one process or by threads in more than one process, as follows:

- If the process-shared attribute's value is `PTHREAD_PROCESS_PRIVATE` (the default), the mutex can be operated upon only by threads in the same process as the thread that initialized that mutex.

- If the process-shared attribute's value is `PTHREAD_PROCESS_SHARED`, the mutex can be operated upon by any thread that has access to the memory where the mutex is allocated, even if these threads are in different processes.

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: for example, 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. You may have multiple condition variables associated with the same mutex—representing different states of the same data—but you cannot use the same condition variable with multiple mutexes.

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 or broadcasting the condition variable.

3. One thread (for signal), or all threads (for broadcast), comes out of the condition wait state. The threads always resume execution one at a time, because each thread must resume with the mutex locked (the condition wait relocks the mutex before returning from the thread). Other threads waiting on the condition variable remain blocked. Other threads awakened by a broadcast will block on the mutex until it is unblocked.

When a thread waits on a condition variable, it cannot assume that the predicate for which it is waiting will be satisfied when the condition variable wait returns. There are a number of reasons for this behavior. For instance, condition variable waits may return spuriously, meaning that the return may not be directly due to some other thread signaling or broadcasting the condition variable.

There are two reasons for these rules:

1. It can be extremely expensive, especially on a symmetric multiprocessor (SMP) system, for the implementation to ensure that a condition signal wakes one and only one thread. It is faster and easier to avoid the extra complication. Also, a thread awakened by the delivery of a UNIX signal will return from the condition wait when the signal is dismissed; the wait predicate may not have changed.

2. Spurious wakeups promote good programming practices. It may often be difficult to guarantee that a predicate will be true; most often, it is easy to determine that it might be true. It is often the case that, after signaling one thread that a predicate is true, another thread may manipulate the data in such a way that the predicate will not be true by the time the signaled thread runs. The solution is to recheck the predicate after the wait returns.

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.

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
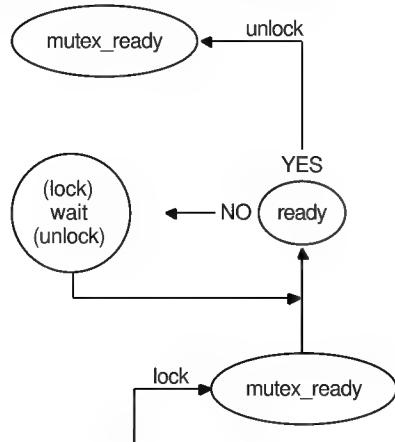
ZK–3793A–GE

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 and compares it against some expected value. 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. However, if the predicate indicates that thread B has not yet finished task X, 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

ZK–3792A–GE
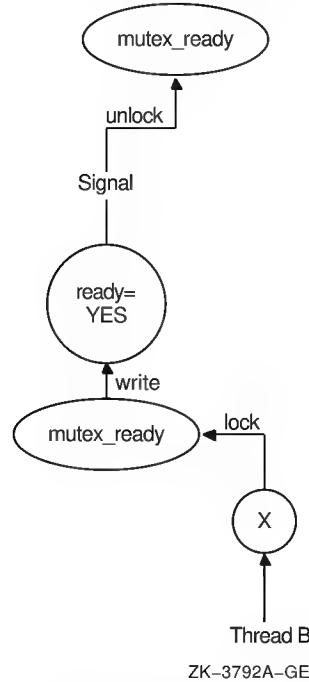
Thread B releases its lock on the shared variable's mutex. As a result of the
signal, thread A wakes up, implicitly regaining its lock on the condition variable's
mutex. It then verifies that the predicate is in the correct state, and proceeds to
execute task Y (see Figure 2–7).

Figure 2–7 Thread A Wakes and Proceeds
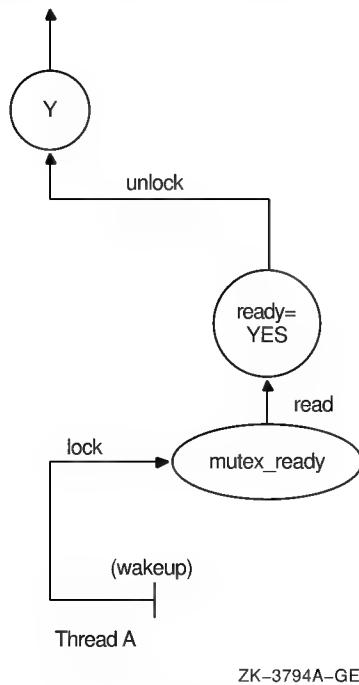
ZK–3794A–GE

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 3.8. You can also statically initialize extern or static 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 broadcast. 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()` or `pthread_cond_sig_preempt_int_np()`, to wake a thread from a signal handler (for Tru64 UNIX) or AST routine (for OpenVMS). There are special restrictions on these functions (see Part II).

If a thread signals or broadcasts 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 on a condition variable, 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 broadcast.

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

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

The process-shared attribute specifies whether a condition variable can be operated upon by threads in only one process or by threads in more than one process, as follows:

- If the process-shared attribute's value is PTHREAD_PROCESS_PRIVATE (the default), the condition variable can be operated upon only by threads in the same process as the thread that initialized that condition variable.

- If the process-shared attribute's value is PTHREAD_PROCESS_SHARED, the condition variable can be operated upon by any thread that has access to the memory where the condition variable is allocated, even if it those threads are in different processes.

2.4.4 Read-Write Locks

A **read-write lock** is a synchronization object for protecting shared data that can be accessed concurrently by more than one of the program's threads. Unlike a mutex, a read-write lock distinguishes between shared read and exclusive write operations on the shared data object.

Use a read-write lock to protect shared data that is read frequently but less frequently modified. For example, when you build a cache of recently accessed information, many threads might simultaneously examine the cache without conflict, but when a thread must update the cache it must have exclusive access.

When a thread locks a read-write lock, it must specify either shared read access or exclusive write access. Many threads may simultaneously acquire a read-write lock for read access, as long as there are no threads waiting for write access. A thread that wants read access cannot acquire the read-write lock if any thread has already acquired the read-write lock for write access; such a thread will block (wait) on the read-write lock. A thread trying to acquire the read-write lock for write access cannot continue if another thread has already acquired the read-write lock for either write access or read access; such a thread will block (wait) on the read-write lock.

2.4.4.1 Thread Priority and Writer Precedence for Read-Write Locks

If more than one thread is waiting for read access to a read-write lock, when the lock becomes available all of the threads will acquire the lock for read access.

If more than one thread is waiting for write access to a read-write lock, when the lock becomes available the thread in that group with the highest priority will acquire the lock for write access.

If both reader threads and writer threads are waiting for access to a read-write lock at the time the lock becomes available, one of the writer threads will acquire the lock, and the threads waiting for read access will continue to block.

The Threads Library implements "writer precedence" for read-write locks. A thread cannot acquire a read-write lock for read access if at least one thread is waiting for write access, even if other threads currently have read access. When a read-write lock is released, a waiting writer will be released if there are any, rather than releasing any waiting readers. Because readers usually outnumber writers, and read access occurs more frequently, writer precedence is needed to avoid "starvation". Without writer precedence, it would be possible that the read-write lock was always locked for read access, and writers would never run.

2.4.4.2 Initializing and Destroying a Read-Write Lock

Use the `pthread_rwlock_init()` routine to create and initialize a new read-write lock object.

Use the `pthread_rwlock_destroy()` routine to destroy a previously initialized read-write lock object.

You can initialize an extern or static read-write lock object using the `PTHREAD_RWLOCK_INITIALIZER` macro provided in the `pthread.h` header file.

2.4.4.3 Read-Write Lock Attributes

By default, a new read-write lock object's attributes have default values. To create a new read-write lock object with nondefault attributes, call the `pthread_rwlock_init()` routine and specify a read-write lock attributes object. Use the `pthread_rwlockattr_init()` routine to create a new read-write lock attributes object, and use the `pthread_rwlockattr_destroy()` routine to destroy a read-write lock attributes object.

There is one settable attribute for a read-write lock object, the process-shared attribute. To set and access the value of the process-shared attribute of a read-write lock attributes object, use the `pthread_rwlockattr_getpshared()` and `pthread_rwlockattr_setpshared()` routines, respectively.

2.5 Process-Shared Synchronization Objects

You can create synchronization objects (that is, mutexes, condition variables, and read-write locks) that protect data that is shared among threads running in different processes. These are called **process-shared** synchronization objects.

_____ Note _____

This version of the Threads Library supports process-shared synchronization objects for Tru64 UNIX systems only.

The following routines are not supported on OpenVMS Alpha and OpenVMS VAX systems:

```
pthread_condattr_getpshared()
pthread_condattr_setpshared()
```

```
pthread_mutexattr_getpshared()
pthread_mutexattr_setpshared()
pthread_rwlockattr_getpshared()
pthread_rwlockattr_setpshared()
```

2.5.1 Programming Considerations

On Tru64 UNIX systems, a process-shared synchronization object is a kernel object. Performing any operation on such an object requires a call into the kernel and thus is of higher cost than the same operation on a process-specific synchronization object.

When debugging a process-shared synchronization object, the debugger cannot currently display the mutex, nor its owner or waiting threads.

As is the case for process-specific synchronization objects, a process-shared synchronization object must be initialized only once; you cannot initialize it in each process that uses it. For independent processes that share a common synchronization protocol using process-shared synchronization objects, there must be some mechanism to determine which single process will initialize those objects.

For example, if multiple processes connect to a named memory section, all but one will fail, and the one successful process should have the responsibility of initializing any global process-shared synchronization objects in that memory section. (The other processes must also use some mechanism for waiting until the process-shared object is initialized before attempting to use the shared memory section.)

2.5.2 Process-Shared Mutexes

You can create a mutex that protects data that is shared among threads running in different processes. This is called a process-shared mutex.

Create a process-shared mutex by using the `pthread_mutexattr_setpshared()` routine to set the process-shared attribute in an initialized mutex attributes object and then use that attributes object in a call to `pthread_mutex_init()`.

2.5.3 Process-Shared Condition Variables

You can create a condition variable used to communicate changes to data that is shared among threads running in different processes. This is called a process-shared condition variable.

Create a process-shared condition variable by using the `pthread_condattr_setpshared()` routine to set the process-shared attribute in an initialized condition variable attributes object and then use that attributes object in a call to `pthread_cond_init()`.

2.5.4 Process-Shared Read-Write Locks

You can create a read-write lock that protects data that is shared among threads running in different processes. This is called a process-shared read-write lock.

Create a process-shared read-write lock by using the `pthread_rwlockattr_setpshared()` routine to set the process-shared attribute in an initialized read-write lock attributes object. Then use that attributes object in a call to `pthread_rwlock_init()`.

2.6 Thread-Specific Data

Each thread can use an area of memory private to the Threads Library where it stores thread-specific data. Use this memory to associate arbitrary data with a thread's context. This allows you to add user-specified fields to the current thread's context or define 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 3.8 for information about the one-time initialization in a threaded environment, or use the operating system's initialization mechanisms (the Tru64 UNIX `__init_*` routines or the OpenVMS `LIB$INITIALIZE` routine).

- Use `pthread_setspecific()` to associate thread-specific data values with a key value. Each thread can associate its own private data with the shared 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 Threads Library 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 in 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 or read-write locks

- A properly tested application predicate loop on a condition variable

- A call to join with a thread you expect to terminate

- An operating system synchronization mechanism, such as a file system read, an OpenVMS event flag wait, or a Tru64 UNIX semaphore wait

- An equivalent hardware construct, such as VAX interlocked instructions or Alpha load locked/store conditional sequences and memory barriers

On a uniprocessor, the Threads Library, in most cases, will context-switch threads in user mode, within a single operating system process. (This is true except for system contention scope threads on Tru64 UNIX.) 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 behavior might be termed "slightly asynchronous," because such a library tolerates many classes of errors in your application.

On a multiprocessor system, the Threads Library may run more than one application thread simultaneously. Many incautious programming techniques that will not usually cause trouble on a uniprocessor will cause trouble–often in ways that are difficult to isolate and fix–on a multiprocessor.

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 Objects Before Thread Creation

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

On "slightly asynchronous" uniprocessor systems this may seem safe, because the thread will probably not run until the creator blocks. Thus, the error can go undetected initially. On a multiprocessor, or even on a new release of the Threads Library with different timeslicing behavior, 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 Do Not Use Scheduling As Synchronization

Avoid using the 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 since a higher-priority thread cannot be preempted by a lower-priority running thread, 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 with the system's memory subsystem. While any written data will, eventually, be seen by other threads, it is essential for communication that some writes appear in a particular sequence. For example, you want a thread that follows a queue link to see the data written to the next queue entry. This requires explicit memory synchronization.

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

`fork()`	`pthread_cond_signal()`
`pthread_create()`	`pthread_cond_broadcast()`
`pthread_join()`	`sem_post()`
`pthread_mutex_lock()`	`sem_trywait()`
`pthread_mutex_trylock()`	`sem_wait()`
`pthread_mutex_unlock()`	`semop()`
`pthread_cond_wait()`	`wait()`
`pthread_cond_timedwait()`	`waitpid()`
`pthread_rwlock_*()`	
`$HIBER`	
`$WAKE`	
`$WAIT*`	

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 Sharing Memory Between Threads

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 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 use of the memory is invalid) 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` is shared by all scopes where the name is defined anywhere, and a `static` variable is private to the source file 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, page fault handling, or other interrupts. 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 or block 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 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, the Threads Library 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 Compaq VAX based systems do not probe the stack. This makes stack overflow failure modes unpredictable and difficult to analyze. Be especially careful to use as little stack memory as practical.

Most compilers on Compaq Alpha based systems generate code in the procedure prologue that probes the stack, which detects if there is not enough space for the procedure to run.

3.4.1 Sizing the Stack

To determine the required size of a thread's stack, add the sizes of the frames, including local variables, for the deepest call tree. Add to that number an extra amount of memory to accommodate interrupts and context switching. 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.

Compaq's Visual Threads includes a number of tools and procedures to measure and monitor stack use. See the Visual Threads product's online help for more information.

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 or monitoring tools typically increases the amount of stack memory that your program uses.

3.4.2 Using Stack Overflow Warning and Stack Guard Areas

By default, at the overflow end of each thread's stack, the Threads Library allocates an **overflow warning area** followed by a **guard area**. These two areas can help a multithreaded program detect overflow of a thread's stack.

Tru64 UNIX 5.0 and OpenVMS Alpha 7.3 include overflow warning support to allow the reporting of stack overflows while a thread can still be assured of executing code. The warning area is a page (or more) that is initially protected to trap writes, but then becomes writable so that it can be used to allow reporting or recovering from the overflow. (On Tru64 UNIX, the warning area is again protected once an overflow has been handled; on OpenVMS it remains unprotected.)

A guard area is a region of no access memory. 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 pages of memory that form a stack guard region are also known as **guard pages** or "red zone"; the overflow warning area is also known as a "yellow zone".

3.4.3 Diagnosing Stack Overflow Errors

A process can produce a memory access violation (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 thread'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' 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.

If a thread receives a memory access exception either during a routine call or when accessing a local variable, increase the size of the thread's stack. However, 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 could hang.

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.)

3.5 Scheduling Issues

The scheduling attributes of threads have unique programming issues.

3.5.1 Real-Time Scheduling

Use care when writing code that uses real-time scheduling (such as FIFO and RR policies) to control the priority of threads:

- Review Section 3.1. Scheduling of threads is not the same as synchronization 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.

- Avoiding multiple priorities and/or policies increases the complexity of the program: this complexity may cost more in performance than the addition of priorities provides, resulting in a performance loss over an application which does not use priorities.

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 Tru64 UNIX systems, to use high (real-time) thread scheduling priorities, a thread with system contention scope must run in a process with root privileges. On the other hand, a thread with process contention scope has access to all levels of priority without requiring special privileges.

Thus, if a process that is not privileged attempts to create another high priority thread with system contention scope, the creation will fail.

3.6 Using Synchronization Objects

The following sections discuss how to determine when to use a mutex with or without a condition variable, and how 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 short duration waits and fine-grained synchronization (memory access). 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 canceled.

Use a condition variable to wait for data to assume a desired state. Condition variables should be used for tasks with longer duration waits and coarse-grained synchronization (routine and system calls) Always use a condition variable with a mutex that protects the shared data being waited for. 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, the points at which each thread executes and waits and the point 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 the 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. You can also use hardware features such as Alpha land-locked/store-conditional instruction sequences.

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 three techniques you can use to avoid deadlocks:

- Use sequence numbers with 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 "try and back off" algorithm when acquiring multiple mutexes. Use `pthread_mutex_trylock()` to lock each additional mutex. If any call to `pthread_mutex_trylock()` returns `EBUSY`, unlock all of the mutexes

(including the first one locked with `pthread_mutex_trylock()`), and start over.

- Avoid locking more than one mutex at the same time.

3.6.4 Signaling a Condition Variable

Signaling the condition variable while holding the lock allows the Threads Library to perform certain optimizations which can result in more efficient behaviors in the working thread. In addition, doing so resolves a race condition which results if that signal might cause the condition variable to be deleted.

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);
                                        ❶
pthread_cond_signal (cv);               ❷
```

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);
```

❶ 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.

❷ 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);
```

3.6.5 Static Initialization Inappropriate for Stack-Based Synchronization Objects

Although it is acceptable to the compiler, you cannot use the following standard macros (or any other equivalent mechanism) to initialize synchronization objects that are allocated on the stack:

```
PTHREAD_MUTEX_INITIALIZER
PTHREAD_COND_INITIALIZER
PTHREAD_RWLOCK_INITIALIZER
```

The Threads Library detects some cases of misuse of static initialization of automatically allocated (stack-based) thread synchronization objects. For instance, if the thread on whose stack a statically initialized mutex is allocated attempts to access that mutex, the operation fails and returns [EINVAL]. If the application does not check status returns from Threads Library routines, this failure can remain unidentified. Further, if the operation was a call to `pthread_mutex_lock()`, the program can encounter a thread synchronization failure, which in turn can result in unexpected program behavior including memory corruption. (For performance reasons, the Threads Library does not currently detect this error when a statically initialized mutex is accessed by a thread other than the one on whose stack the object was automatically allocated.)

If your application must allocate a thread synchronization object on the stack, the application must initialize the object before it is used by calling one of the routines `pthread_mutex_init()`, `pthread_cond_init()`, or `pthread_rwlock_init()`, as appropriate for the object. Your application must also destroy the thread synchronization object before it goes out of scope (for instance, due to the routine's returning control or raising an exception) by calling one of the routines `pthread_mutex_destroy()`, `pthread_cond_destroy()`, or `pthread_rwlock_destroy()`, as appropriate for the object.

3.7 Granularity Considerations

Granularity refers to the smallest unit of storage (that is, bytes, words, longwords, or quadwords) that a host computer can load or store in one machine instruction. Granularity considerations can affect the correctness of a program in which concurrent or asynchronous access can occur to separate pieces of data stored in the same memory granule. This can occur in a multithreaded program, where different threads access the data, or in any program that has any of the following characteristics:

- Accesses data in memory that is shared with other processes

- Accesses data that can be accessed by asynchronous device drivers, signal handlers (on Tru64 UNIX), or ASTs (on OpenVMS)

- Accesses data objects that can be accessed by a continuable exception handler

The subsections that follow explain the granularity concept, the way it can affect the correctness of a multithreaded program, and techniques the programmer can use to prevent the granularity-related race condition known as **word tearing**.

3.7.1 Determinants of a Program's Granularity

A computer's processor typically makes available some set of granularities to programs, based on the processor's architecture, cache architecture, and instruction set. However, the computer's **natural granularity** also depends on the organization of the computer's memory and its bus architecture. For example, even if the processor "naturally" reads and writes 8-bit memory granules, a program's memory transfers may, in fact, occur in 32- or 64-bit memory granules.

On a computer that supports a set of granularities, the compiler determines a given program's **actual granularity** by the instructions it produces for the program to execute. For example, a given compiler on Alpha systems might generate code that causes every memory access to load or store a 64-bit word, regardless of the size of the data object specified in the application's source code. In this case, the application has a 64-bit word actual granularity. For this application, 8-bit, 16-bit, and 32-bit writes are not atomic with respect to other memory operations that overlap the same 64-bit memory granule.

To provide a run-time environment for applications that is consistent and coherent, an operating system's services and libraries should be built so that they provide the same actual granularity. When this is the case, an operating system can be said to provide a **system granularity** to the applications that it hosts. (A system's system granularity is typically reflected in the default actual granularity that the system's compilers encode when producing an object file.)

When preparing to port a multithreaded application from one system to another, you should determine whether there is a difference in the system granularities between the source and target systems. If the target system has a larger system granularity than the source system, you should become informed about the programming techniques presented in the sections that follow.

3.7.1.1 Alpha Processor Granularity

Systems based on the Alpha processor family have a quadword (64-bit) natural granularity.

Versions EV4 and EV5 of the Alpha processor family provide instructions for only longword- and quadword-length atomic memory accesses. Newer Alpha processors (EV5.6 and later) support byte- and word-length atomic memory accesses as well as longword- and quadword-length atomic memory accesses. (However, there is no way to ensure that a compiler uses the byte or word memory references when generating code for your application.)

_____ **Note** _____

On systems using Tru64 UNIX Version 4.0 and later:

If you use Compaq C or Compaq C++ to compile your application's modules on a system that uses the EV4 or EV5 version of the Alpha processor, you can use the –arch56 compiler switch to request the compiler to produce instructions available in the Alpha processor version EV5.6 or later, including instructions for byte- and word-length atomic memory access, as needed.

When an application compiled with the –arch56 switch runs under Tru64 UNIX Version 4.0 or later, with a newer Alpha processor (that is, EV5.6 or later), it utilizes that processor's full instruction set. When that same application runs under Tru64 UNIX Version 4.0 or later, with an older Alpha processor (that is, EV4 or EV5), the operating system performs a software emulation of each instruction that is not available to the

older processor; however, this is considerably slower than if the same application was run on a newer Alpha processor.

See the Compaq C and Compaq C++ compiler documentation for more information about the `-arch56` switch.

On Tru64 UNIX systems, use the `/usr/sbin/psrinfo -v` command to determine the version(s) of your system's Alpha processor(s).

3.7.1.2 VAX Processor Granularity

Systems based on the VAX processor family have longword (32-bit) natural granularity, but all instructions can access unaligned data safely (though perhaps with a substantial performance penalty).

For more information about the granularity considerations of porting an application from an OpenVMS VAX system to an OpenVMS Alpha systems, consult the document Migrating to an OpenVMS System[1].

3.7.2 Compiler Support for Determining the Program's Actual Granularity

Table 3–1 summarizes the actual granularities that are provided by the respective compilers on the respective Compaq platforms.

Table 3–1   Default and Optional Granularities

Platform	Compiler	Default Granularity Setting	Optional Granularity Settings
Tru64 UNIX Versions 4.0D and later (Alpha only)	C/C++	quadword	longword, byte/word on EV5.6
OpenVMS Alpha Version 7.3	C/C++	quadword	byte, word
OpenVMS VAX Version 7.3	C/C++	longword	None

Of course, for compilers that support an optional granularity setting, it is possible to compile different modules in your application with different granularity settings. You might do so either to avoid the possibility of word-tearing race condition, as described in Section 3.7.3, or to improve the application's performance.

3.7.3 Word Tearing

In a multithreaded application, concurrent access by different threads to data that occupy the same memory granule can lead to a race condition known as **word tearing**. This situation occurs when two or more threads independently read the same granule of memory, update different portions of that granule, then independently (that is, asynchronously) store their respective copies of that granule. Because the order of the store operations is indeterminate, it is possible that only the last thread to write the granule continues with a correct "view" of the granule's contents, and earlier writes could be "undone".

---

[1] This manual has been archived but is available on the OpenVMS Documentation CD-ROM.

In a multithreaded program the potential for a word-tearing race condition exists only when both of the following conditions are met:

- Two or more threads can concurrently write distinct pieces of data that occupy the same memory granule G, where G is a byte, word, longword, or quadword.

- The application's actual granularity is `sizeof(G)` or larger.

For instance, given a multithreaded program that has been compiled to have longword actual granularity, if any two of the program's threads can concurrently update different bytes or words in the same longword, then that program is, in theory, at risk for encountering a word-tearing race condition. However, in practice, language-defined restrictions on the alignments of data may limit the actual number of candidates for a word-tearing scenario, as described in the next section.

3.7.4 Alignments of Members of Composite Data Objects

The only data objects that are candidates for participating in a word-tearing race condition are members of composite data objects—that is, C language structures, unions, and arrays. In other words, the application's threads might access different data objects that are members of structures or unions, where those members occupy the same byte, word, longword, or quadword. Similarly, the application might access arrays whose elements occupy the same word, longword, or quadword.

On the other hand, the C language specification allows the compiler to allocate scalar data objects so that each is aligned on a boundary for the memory granule that the compiler prefers, as follows:

- For Compaq C and Compaq C++ on Tru64 UNIX Version 4.0D and higher (Alpha only), and OpenVMS Alpha Version 7.3 systems, alignment of scalars is always on quadword boundaries.

- For Compaq C and Compaq C++ on OpenVMS VAX Version 7.3 systems, alignment of scalars is always on longword boundaries.

For the details of the compiler's rules for aligning scalar and composite data objects, see the Compaq C and C++ compiler documentation for your application's host platforms.

3.7.5 Avoiding Granularity-Related Errors

Compaq recommends that you inspect your multithreaded application's code to determine whether a word-tearing race condition is possible for any two or more of the application's threads. That is, determine whether any two or more threads can concurrently write contiguously defined members of the same composite data object where those members occupy the same memory granule whose size is greater than or equal to the application's actual granularity.

If you find that you must change your application to avoid a word-tearing scenario, there are several approaches available. The simplest techniques require only that you change the definition of the target composite data object before recompiling the application. The following sections offers some suggestions.

3.7.5.1 Changing the Composite Data Object's Layout

If you can change the organization or layout of the composite data object's definition, you should do both of the following:

- Widen the structures or union members to the granule. If that is unacceptable, define padding storage after each structure or union member (except the last) or add padding storage to the array's element definition. This forces all members/elements to be placed in separate granules by the compiler.

- If your system's compiler offers a choice, compile the application's modules to produce the preferred actual granularity for the application's target system.

3.7.5.2 Maintaining the Composite Data Object's Layout

If you cannot change the organization or layout of the composite data object's definition, you should do one of the following:

- (On OpenVMS Alpha or OpenVMS VAX) Compile all application modules for byte actual granularity. Doing so automatically prevents word-tearing race conditions for structure or union members and array elements of size byte or larger that are accessed concurrently by different threads. No other program modification is required. This may have a performance penalty on Alpha EV4 and EV5 processors.

- (On Tru64 UNIX systems) For arrays, add the C language `volatile` storage qualifier to the definition of the entire array; for structures, add `volatile` to the declaration of only those members that share the pertinent memory granule. You must also compile the application's modules using the Compaq C or Compaq C++ compiler's `-strong-volatile` switch. Doing so causes the compiler to produce code that forces all accesses to those members to occur as atomic operations. See the description of the `-strong-volatile` switch in the Compaq C or Compaq C++ documentation and on the `cc` reference page. This may also have a severe performance penalty.

If you must maintain the composite data object's layout and cannot change the storage qualifiers for the application's composite objects, you can instead use the technique described in the next section.

3.7.5.3 Using One Mutex Per Composite Data Object

Your source code inspection may identify an array or a set of contiguously defined structure or union members that is subject to a word-tearing race condition. In this case, your program can use a mutex that is dedicated to protect all write accesses by all threads to those data objects, rather than change the definition of the composite data objects.

To use this technique, create a separate mutex for each composite data object where any members share a memory granule that is greater than or equal to the program's actual granularity. For example, given an application with quadword actual granularity, if structure members `M1` and `M2` occupy the same longword in structure `S` and those members can be written concurrently by more than one thread, then the application must create and reserve a mutex for use only to protect all write accesses by all threads to those two members.

In general, this is a less desirable technique due to performance considerations. However, if the absolute number of thread accesses to the target data objects over the application's run-time will be small, this technique provides explicit, portable correctness for all thread accesses to the target members.

3.7.6 Identifying Possible Word-Tearing Situations Using Visual Threads

For Tru64 UNIX systems, the Visual Threads tool can warn the developer at application run-time that a possible word-tearing situation has been detected. Enable the UnguardedData rule before running the application. This rule causes Visual Threads to track whether any memory location (that is, granule) in the application has been accessed from two threads without proper synchronization. This includes detection of word tearing as well as more straightforward synchronization errors. See the Visual Threads product's online help for more information.

Visual Threads is available as part of the Developer's Toolkit for Tru64 UNIX.

3.8 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.

You can 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 attempt to call the program initialization routine via the `pthread_once()` routine, and the Threads Library ensures that the specified initialization routine is called only once.

On the other hand, rather than use the `pthread_once()` routine, your program could 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, Tru64 UNIX dynamic loader `__init_` code.

3.9 Managing Dependencies Upon Other Libraries

Because multithreaded programming has become common only recently, many existing code libraries are incompatible with multithreaded uses. For example, many traditional 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 will obtain all entries in the password file. (This is not a problem on Tru64 UNIX, because the state is maintained using thread-specific data.)

Different library routines are compatible with multithreaded programming to different extents. The important distinctions are thread reentrancy and thread safety.

3.9.1 Thread Reentrancy

A routine is **reentrant** if it can be used simultaneously when called by different threads. For example, the standard C run-time library routine strtok() can be made 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.

A reentrant routine should have no dependency on static data. Because access to static data must be synchronized, there is always a performance penalty due to the cost of synchronizing. There is also a 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 or output parameters (where the caller passes the address and length of a buffer). You could also return information to the caller by allocating dynamic memory and requiring the caller to free that memory when finished. Avoid using errno or other global variables for returning error or diagnostic information; use routine return values instead.

3.9.2 Thread Safety

A routine is **thread-safe** if it can be called simultaneously from multiple threads without risk of corruption. If the routine is not actually reentrant, generally this means that it does some level of locking to prevent simultaneously active calls in different threads.

Thread-safe routines tend to be less efficient than reentrant routines. For example, a package that is thread-safe might still block all threads in the process while one thread executes the code.

Routines such as localtime() or strtok(), which traditionally rely on static storage, can be made thread-safe by using thread-specific data instead of static variables as is done on Tru64 UNIX. 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 often preferable.

3.9.3 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 reentrant or thread-safe, you can assume that it is not safe to use. Never assume that a routine is fully thread-safe 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-safe but that calls some other routine that is not thread-safe without proper protection, is itself not thread safe.

3.9.3.1 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 a single call, to routines that are not thread safe.

3.9.3.2 Using the Global Lock

To ensure serialization and exclusivity of the unsafe code, the Threads Library provides one **global lock** that can be used by all threads in a program when calling either routines or code that are not thread-safe while already holding the lock. 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.

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

The global lock allows a thread to acquire the lock recursively, so you do not need to be concerned if you call a routine that also may acquire the global lock.

Use the global lock whenever calling unsafe routines. All Threads Library routines are thread-safe.

3.9.3.3 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 either use or copy any static return values before releasing the mutex that is being held.

3.9.4 Use of Multiple Threads Libraries Not Supported

The Threads Library performs user-mode execution context-switching within a process 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 the Threads Library nor that other code can ever know the proper identity of the context which 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 Threads Library scheduling. Therefore, VAX Ada tasking cannot be used within a process that also uses the Threads Library. (This restriction does not exist for Compaq Ada for Tru64 UNIX, or Compaq Ada for OpenVMS Alpha, because they use the Threads Library.)

3.10 Detecting Error Conditions

The Threads Library can detect some of the following types of errors:

- Application programming interface (API) errors can occur when the program either specifies an invalid parameter or attempts an inappropriate operation on some Threads Library object.

- Internal errors can occur when the Threads Library determines that internal information has become corrupted to the point where it cannot continue operation.

The **pthread** interface reports API errors by returning an integer value indicating the type of error.

The Threads Library internal errors result in a **bugcheck**. The Threads Library writes a message that summarizes the problem to the process' current error device, and (on OpenVMS) writes a file that contains more detailed information. (On Tru64 UNIX systems, the core file is sufficient for analysis of the process using the Ladebug debugger.)

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

If the Threads Library 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.

3.10.1 Bugcheck Information

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

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

The next line states the reason for the failure. On Tru64 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 the Threads Library, 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.

3.10.2 Interpreting a Bugcheck

The fact that the Threads Library terminated the process with a bugcheck can mean that some subtle problem in the Threads Library has been uncovered. However, the Threads Library does not report all possible API errors, and there are a number of ways in which incorrect code in your program can lead to a 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 Tru64 UNIX signal handler or OpenVMS AST routine). This type of programming error most commonly results in a bugcheck that reports a "krnSpinLockPrm: deadlock detected" message or a "Can't find null thread" message. To prevent this type of error, do not use Threads Library routines other than those with the `_int` suffix in their names, such as `pthread_cond_signal_int_np()` from an interrupt routine.

In addition, the Threads Library 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 the Threads Library state by writing through invalid pointers, which can result in a bugcheck or other undesirable behavior.

If you encounter a bugcheck, first check your application for memory corruptions, calls from AST routines, and so on, and then contact your Compaq support representative and include this information file (or the Tru64 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.

4

Writing Thread-Safe Libraries

A **thread-safe library** consists of routines that are coded so that they are safe to be called from applications that use threads. The Threads Library provides the thread-independent services (or **tis**) interface to help you write 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 easy to maintain. 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 Threads Library synchronization.

The guidelines for using the **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 **tis** interface are:

- Synchronization without linking the thread library with some unique routines and some routines that correspond to those in the **pthread** interface

- Common synchronization data types (such as mutexes and condition variables) with the **pthread** interface

- Unique **tis** synchronization objects (such as the read-write lock which is different from the **pthread** read-write lock

- Support for thread-specific data objects

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

It is not difficult to create thread-safe code using the **tis** interface, and it should be straightforward to modify existing source code that is not thread-safe to make it 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 the Threads Library. See Section 3.9.4.

Although the C, C++, Pascal, BLISS, FORTRAN and COBOL programming languages do not support multithreaded programming directly, compilers for those languages generally create reentrant code.

4.1.2 Performance of tis Interface Routines

Routines in the **tis** interface are designed to impose low overhead when called from a single-threaded environment. For example, locking a mutex is essentially just setting a bit, and unlocking the mutex clears 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. During program initialization, the Threads Library automatically revectors the program's run-time linkages to most **tis** routines. This allows subsequent calls to those routines to use the normal 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 Threads Library operation is transparent to library code that uses the **tis** interface. For instance, if the Threads Library 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 the Threads Library 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 the Threads Library can be dynamically activated into the process at any time.

4.1.4 Cancelation Points

The following routines in the **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

Like the mutexes available through the other **pthread** interface, tis mutexes provide synchronization between multiple threads that share resources. In fact, you can statically initialize tis mutexes using the `PTHREAD_MUTEX_INITIALIZER` macro (see the Threads Library `pthread.h` header file).

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 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 global lock are also supported by **tis** interface routines. The global lock is a recursive mutex that is provided by the Threads Library for use by any thread. Your program can use the global lock without calling the **pthread** interface 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 PTHREAD_COND_INITIALIZER 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.

A condition variable wait is useful only when there are other threads. Your program can have more than one thread only if the Threads Library multithreading run-time environment is present. In a non-threaded environment, a wait aborts and signaling or broadcasting a tis mutex does nothing.

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. Please see the tis_io_complete() and tis_sync() reference pages.

4.4 Using Thread-Specific Data

The **tis** interface routines support the use of thread-specific data. If code in the process creates keys or sets thread-specific data values before the multithreading run-time environment is initialized, those keys and values continue to be available to your program in the initial thread.

4.5 Using Read-Write Locks

A **read-write lock** is an object that allows the application to control access to information that can be read concurrently by more than one thread and that needs to be read frequently and written only occasionally. Routines that manipulate the **tis** interface's read-write lock objects can control access to any shared resource.

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.

Tis read-write locks are completely different from the newer **pthread** read-write locks. Currently, the latter have no tis equivalent.

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 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, **tis** read-write locks give 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. (Note that this differs from **pthread** read-write locks, which have writer precedence.) 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.

Figure 4–1 Read-Write Lock Behavior

ZK–7929A–GE
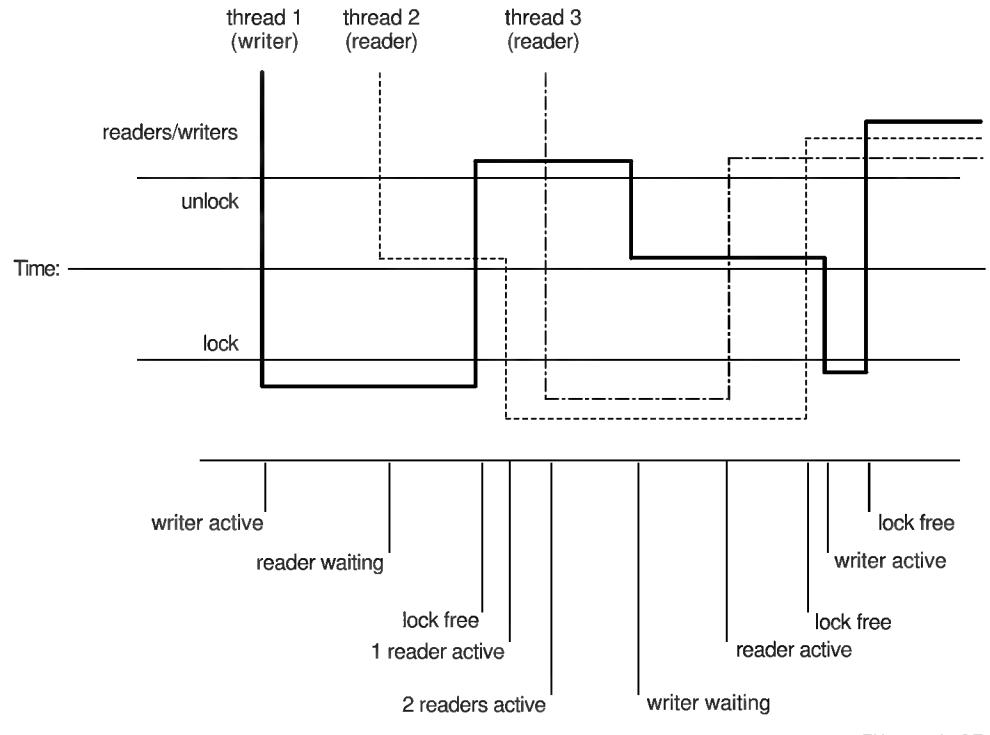
The `tis_rwlock_init()` routine initializes a read-write lock by initializing the supplied `tis_rwlock_t` structure.

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()` and `tis_write_trylock()` immediately return [EBUSY], rather than waiting.

If a non-threaded program manes a **tis** call that would block (such as a call to `tis_cond_wait()`, `tis_read_lock()` or `tis_write_lock()`), it is a fatal error that will abort the program.

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.

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

5

Using the Exceptions Package

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

This chapter does the following:

- Describes the exceptions package

- Shows how to declare, initialize, and handle an exception object in your program

- Describes the exceptions package's macros that support exception handling

- Describes the exceptions package's API-level routines that operate on exception objects

- Lists the names of exception objects that the exceptions package defines for its own use

5.1 About the Exceptions Package

The exceptions package is a part of the POSIX Threads Library. A C language header file (`pthread_exception.h`) provides an interface for defining and handling exceptions. It is designed for use with the pthreads interface routines. If you use the exceptions package, your application must be linked with the Threads Library.

5.1.1 Supported Programming Languages

You can use the exceptions package only when you are programming in the C language. While the exceptions will compile under C++, they will not behave properly. In addition, `gcc` lacks the Compaq C extensions that are needed to interact with the native exception handling system, and will not interoperate correctly with other language exception facilities.

You can use the C language exception handling mechanism (SEH) to catch exceptions. You can catch exceptions in C++ using `catch(...)`, and propagation of exceptions will run C++ object destructors. Currently, C++ code cannot catch specific exceptions. Also, `CATCH`, `CATCH_ALL` and `FINALLY` clauses will not run when C++ code raises an exception. (These restrictions will be reduced or removed in a future release.)

5.1.2 Relation of Exceptions to Return Codes and Signals

The Threads Library uses exceptions in the following cases:

- The `pthread_exit()` routine raises the exception `pthread_exit_e` defined by the Threads Library.

- Canceling a thread causes the Threads Library to raise the exception `pthread_cancel_e` defined by the Threads Library.

- On Tru64 UNIX, synchronous signals (such as SIGSEGV) are converted to exceptions unless a signal action is declared.

5.2 Why Use Exceptions

An **exception** is a mechanism for reporting an error condition. An exception is represented by an **exception subject**. Operations on exception objects allow your program to report and handle errors. If your program can handle an exception properly, the program can recover from errors. For example, while reading a tape, a program raises an exception from a parity error. The recovery action might be to retry reading the tape 100 times before giving up. However, if the program does not handle the exception, then the program terminates. Reporting errors via exceptions ensures that the error will not inadvertently go unnoticed and cause problems later.

You use exception programming to identify a portion of a routine, called an **exception scope**, where a calling thread wishes to 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 from within the exception scope, not from the point where the exception was raised.

- Finalize the exception scope. This means that the current scope'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 exception package supports finalization of a scope even when no exception was raised, so that resources are always released without duplication of code.

As a result, you can use the exceptions package to handle thread cancelation and thread exit in a unified and modular manner. Because the Threads Library implements both thread cancelation and thread exit by raising exceptions, your code can respond to these events in the same modular manner as it does for error conditions.

5.3 Exception Programming

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

To use exceptions, do the following:

1. Declare one exception object for each distinct error condition of interest to your program.

2. Code your program to invoke the `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 handled.

4. Optionally include the `CATCH` macro, which is associated with each exception scope, to define a block of exception handler code for each exception that your program wishes to handle at this point in its work. In this block your program can perform activities to respond to the particular error condition.

5. Optionally include the `CATCH_ALL` macro, which is associated with each exception scope, to define an exception handler to catch any other exception that might be raised, if your code needs to respond to such errors. Unless your code can fully recover from 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. Use the `FINALLY` macro, which is associated with each exception scope, to define **finalization code**, also known as **epilogue code**. This code is always executed when control leaves the `TRY` block, regardless of whether the code in the associated exception scope raised an exception. If this code is reached because of an exception being raised, the Threads Library automatically reraises the caught exception and passes it to the next outer exception scope.

When a thread in your program raises an exception, the Threads Library determines whether an exception scope has been defined in the current stack frame. If so, the Threads Library checks whether there is either a specific handler (`CATCH` code block) for the raised exception or an unspecified handler (`CATCH_ALL` or `FINALLY` code block). If not, the Threads Library 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 an exception object in your code, your program must declare and initialize the object. You must define an exception object (whether explicitly or implicitly) to be of `static` storage class.

The next sample code fragment demonstrates how a program declares and initializes an exception object.

```
static EXCEPTION parity_error;   /* Declare the exception */
EXCEPTION_INIT (parity_error);   /* Initialize the exception */
```

5.3.2 Raising an Exception

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

Example 5–1 demonstrates how to raise an exception.

Example 5–1   Raising an Exception

```
static EXCEPTION parity_error;

int  read_tape(void)
{
   int ret;

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

5.3.3 Catching an Exception

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

- A TRY code block, a lexical scope within which an exception will be handled if it is raised (if there is a matching CATCH block or a CATCH_ALL or FINALLY block).

- (Optionally) A CATCH code block, where your program handles a particular exception that was raised within the scope of this TRY block (a single TRY block may have more than one CATCH block for different exceptions).

- (Optionally) A CATCH_ALL code block, where your program handles any exception raised within the scope of this TRY block that is not named as an argument in a preceding CATCH block in this TRY block. The CATCH_ALL block must be the last block, following any CATCH blocks. (Only one CATCH_ALL block may be associated with a 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 (an exception scope with a FINALLY block cannot also have either a CATCH or CATCH_ALL block).

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 5–2   Catching an Exception Using CATCH

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

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, read aborted\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 default handler for unhandled exceptions receives the exception, produces a default error message that identifies the unhandled exception, then terminates the process.

Reraising is particularly appropriate for an exception caught in a `CATCH_ALL` block. Because this code block may catch exceptions that are unexpected by your program's code, it is unlikely that your code is able to fully recover from the error condition that the exception represents. Therefore, your code should allow the exceptions to continue to propagate, either so that it will reach a handler that can deal with it properly or the process can be terminated safely.

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 if no exception is raised. This means that those actions need not be duplicated in your code.

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

5.4 Exception Objects

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

An **exception object** 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. In the exceptions package, it is a statically allocated variable of type `EXCEPTION`.

5.4.1 Declaring and Initializing Exception Objects

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

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 exceptions 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.

The following code fragment shows how a program declares and initializes 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 using an exception object that has been properly initialized, the exception is identified by the address of the exception object. This form of exception object is called an **address exception**. Your program code that handles address exceptions is fully portable among supported platforms because address exceptions contain nothing that is platform dependent.

Use address exceptions if the error conditions that report in your program do not correspond to a system status code. 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 can use the exceptions package's `pthread_exc_set_status_np()` routine to assign a C errno code on Tru64 UNIX or a condition code on OpenVMS to the exception object. This form of exception object is called a **status exception**.

Given two different exception objects that have been set with the same status value, the exceptions package considers the two objects as representing the same exception. For example, if one of the two objects is used to raise an exception, the exception can be caught by specifying the other exception object that has been set to the same status value. In contrast, the Threads Library never considers two distinct address exception objects to match the same 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.

5.4.3 How Exceptions Terminate

Threads Library 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`, `CATCH_ALL` or `FINALLY` 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 control enters 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 thread handles the exception without reraising it, the Threads Library terminates the process, regardless of the state of the process' 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 either 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 Threads Library reserved macro, as described in Section 5.7.

Use the TRY/ENDTRY pair of macros to define an exception scope. (Throughout the discussion, this pair of macros is referred to 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 how a program defines an exception scope that encloses one operation, a call to the read_tape() routine.

Example 5–5   Defining an Exception Scope

```
EXCEPTION parity_error;

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) {

      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 will be handled if it is raised. Any exception raised within the block, or within any routines called directly or indirectly within the block, will 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. However, they may be nested–in fact, you can use TRY blocks not only inside other TRY blocks, but inside CATCH and FINALLY blocks as well.

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 exceptions package's `RAISE` macro to raise an exception.

When your program raises an exception, it 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. The program searches in the current scope, then in the next outer scope and so on, for an exception handler that explicitly or implicitly responds to the error (such as a `CATCH`, `CATCH_ALL` or `FINALLY` block).

2. The program invokes the handler code that is found.

3. If the exception is reraised, then the process resumes with the first step and the next outer scope.

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

Example 5–6 illustrates how a program raises an exception.

Example 5–6   Raising an 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 */
}
```

Threads Library exceptions are classified as terminating exceptions because after an exception is raised even if it is handled, 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 either explicitly or implicitly matches 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 exceptions 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 exception handler takes appropriate actions in response 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 it can mean fully responding to the error in such a manner that allows the routine with the handler 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 exceptions package, exception handlers are attached, which means that the handler code appears within the same routine where the specified exceptions are raised (directly or indirectly). This allows the code to access local commands when an exception occurs with that exception scope, and allows the error handling code to be positioned "close" to the code with which it is associated for readability and maintainability.

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 a matching exception propagates within the execution scope of the associated exception scope.

Use the exceptions package's CATCH macro to define an exception handler code block that is invoked when an exception matching the macro's specified exception object is propagated within the associated exception scope. Use the exceptions package's CATCH_ALL macro to define an exception handler code block that is invoked when any other exception is propagated within the associated exception scope.

An exception handler's code can reraise an exception. That is, the code can propagate an exception to the next outer exception scope for further processing. Use the exceptions package's RERAISE macro to do so. If appropriate, a handler may instead use the RAISE macro to raise a different exception.

Another form of exception handler code is finalization code, or epilogue code. You can define a block of epilogue code and associate it with an exception scope. When an exception is raised, epilogue code performs your cleanup actions within the current exception scope (such as releasing resources), then automatically propagates 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 exceptions package's FINALLY macro to define an epilogue code block. Note that, for a given exception scope, FINALLY blocks and CATCH and CATCH_ALL blocks are mutually exclusive.

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 particular exception by specifying a corresponding exception object 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 a matching exception object. If there is more than one CATCH code block that specifies a matching object within a single TRY/ENDTRY scope, only the first one gains control. (Thus, there is no point in having two CATCH blocks with matching or equivalent exceptions.)

To catch an address exception, the CATCH macro must specify the name of the exception object used in the invoked RAISE macro. 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, read aborted\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 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 execution within the associated exception scope.

If it is appropriate for the caught exception to be propagated to the next higher exception scope, the CATCH code block can use the RERAISE macro as its last action 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 that are not being uniquely handled on a per-exception basis in the program's code.

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.

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
```

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. This allows the next outer exception scope to handle the exception as it finds appropriate. Invoking the RERAISE macro is valid only within 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 or sufficient reason to attempt to recover fully. For example, a CATCH_ALL code block should always reraise the caught exception as its last action, because the exception handler cannot recover fully from the error since it does not know what the error specifically was.

Refer to Example 5–8 for an example of how a program invokes 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 may catch exceptions only for the purpose of performing cleanup actions, such as releasing resources. In many cases, these actions are performed when the TRY code block exits normally or after an exception has been caught. This requires duplicating code in the CATCH_ALL code block and following the exception scope (for the case when an exception does not occur).

The exceptions 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, when no exception is raised), so that they need not be coded more than once.

Example 5–9 demonstrates the FINALLY macro.

Example 5–9 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
```

In this example, if the thread was canceled while it was waiting, the
`pthread_cancel_e` exception would propagate out of the `pthread_cond_wait()`
call. The operations in the `FINALLY` code block release the mutex, after ensuring
that the shared data associated with the lock is correct for the next thread that
acquires the mutex.

_____ **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 exceptions
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 exception objects 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. You cannot use `THIS_CATCH` in a
`FINALLY` code block because there might not be an exception.

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 exceptions package propagates exception
objects, the address contained in `THIS_CATCH` might not be the actual

address of an address exception. To match `THIS_CATCH` against known exceptions, use the `pthread_exc_matches_np()` routine, as described in Section 5.8.5. Furthermore, the value of `THIS_CATCH` may become invalid when control leaves the `CATCH` or `CATCH_ALL` block.

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 address exception object. This converts an address exception object into a status exception object.

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

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

Example 5–10 demonstrates setting an error status in an address exception object.

Example 5–10 Setting an Error Status in an Exception Object

```
static EXCEPTION an_error;

unsigned long status_code = ENOMEM;
EXCEPTION_INIT (an_error);

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

pthread_exc_set_status_np (&an_error, status_code);
```

_____ Note _____

On OpenVMS systems:

Threads Library exception status values are OpenVMS condition codes with a SEVERE severity level. If necessary, the `pthread_exc_set_status_np()` routine will modify the severity level of the status code to SEVERE.

_____

5.8.3 Obtaining a System-Defined Error Status

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

Example 5–11 Obtaining the Error Status Value from a Status Exception Object

```
#include <pthread_exception.h>
.
.
.
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 %ld caught from system.\n", SOME_ERROR);
   else
     pthread_exc_report_np (THIS_CATCH);
   }
ENDTRY
```

Example 5–11 demonstrates using the `pthread_exc_get_status_np()` routine to obtain the status value associated with a caught status exception object.

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` or `CATCH_ALL` code block to report on a caught exception.

An exception in your program that has not been handled by a `CATCH` or `CATCH_ALL` causes the 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 Tru64 UNIX systems) or SYS$ERROR (on OpenVMS systems) that describes the exception.

Each defined exception has an associated message that describes the given error condition. Typically, external status values can also be reported. When an address exception is reported, the Threads Library can only report the fact that an exception has occurred and the address of the exception object.

See Example 5–11 for an example using the `pthread_exc_report_np()` routine to report an error.

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 exception objects, use this routine.

Example 5–12 demonstrates how to use the `pthread_exc_matches_np()` routine to test for the equivalence of two exception objects.

Example 5–12 Comparing Two Exception Objects

```
#include <pthread_exception.h>
.
.
.
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
```

5.9 Using Exceptions

This section presents guidelines for using exceptions in a modular way, so that independent software components can be written without requiring knowledge of each other, and includes tips on writing code using exceptions.

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 conflict. 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 5–13 demonstrates this misuse.

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.

Example 5–13 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
```

Example 5–14 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
```

Thus, the code in Example 5–13 should be rewritten as shown in Example 5–14.

Notice that the initialization code belongs prior to the invoking of the TRY macro, and the matching cleanup code belongs in the FINALLY code block. In this example, the open_file() call is moved to before the TRY macro, and the close() call is kept in the FINALLY 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.

5.9.4 Exiting an Exception Scope

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, such as by using a continue or break in an exception scope contained inside a loop or switch statement. After a given TRY macro is executed, the exceptions package requires that the corresponding ENDTRY macro is also executed unless an exception is raised or reraised.

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 prevents the compiler from producing certain optimizations about such variables that would be unsafe if an exception were raised. This ensures that such a variable's value is reliable in an exception handler after an exception is raised.

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 when 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 5–15 demonstrates the significance of using the `volatile` type qualifier for variables that are referenced within an exception scope.

Example 5–15 Use of the Volatile Type Qualifier Within an Exception Scope

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

    /*    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() */
```

❶ Values of `updated_volatile` and `updated_before_try` are reliable. Values of `updated` and `updated_static` are unreliable.

❷ Regardless of the path to this code, the values of `updated_volatile` and `updated_before_try` are reliable. 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 reliable. If this code is reached after the exception `fully_handled_exception` has been caught, the values of `updated` and `updated_static` are unreliable.

The code in Example 5–15 demonstrates:

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

- 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.

Test your program after compiling it with the "optimize" compiler option, to ensure that your program contains 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 always 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 stop the propagation of a thread cancelation exception or thread-exit request exception. The Threads Library 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, but they must always be reraised so that the thread will terminate properly.

5.9.7 Avoid Dynamically Allocated Exception Objects

Avoid dynamically allocated exception objects. Local exception objects should be declared (explicitly or implicitly) as `static`, and `extern` exception objects are acceptable.

5.10 Exceptions Defined by the POSIX Threads Library

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

Exception object names that begin with the prefix `pthread_` are raised within the 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 exceptions package or represent exceptions raised by the underlying system.

Table 5–1   Names of Exception Objects Defined by the Threads Library

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 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

Table 5–1 (Cont.) Names of Exception Objects Defined by the Threads Library

Exception	Definition
pthread_exc_fltdiv_e	Unhandled floating-point/decimal divide by zero exception
pthread_exc_fltovf_e	Unhandled floating-point overflow exception
pthread_exc_fltund_e	Unhandled floating-point underflow 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 exception
pthread_exc_intovf_e	Unhandled integer overflow exception
pthread_exc_noexcmem_e	Out of memory while processing an 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 ABORT
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 exiting using pthread_exit()
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 the Threads Library.

Currently on Tru64 UNIX systems, your program cannot use CATCH to catch a C++ or Ada exception.

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

5.12 Host Operating System Dependencies

This section mentions dependencies of the exceptions package on the operating system environment.

5.12.1 Tru64 UNIX Dependencies

Tru64 UNIX has an architecturally specified exception model that is used by the Threads Library as well as C++, Compaq Ada, and other languages that support exceptions. The Compaq C compiler has extensions that allow "native" exception handling.

5.12.2 OpenVMS Conditions and Exceptions

On OpenVMS, the Threads Library 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_cleanup_push()` routine, the exceptions package's `TRY` macro establishes 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 exception handler code.

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

When your program raises an exception, an OpenVMS condition has been signaled. Until the exception is actually caught (that is, before passing through any `TRY` blocks or 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` or `CATCH_ALL` block or implicitly at the end of a `FINALLY` block or a cleanup handler, the Threads Library 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. The Threads Library propagates the exception by calling LIB$STOP with the new argument array.

When a status exception is reraised, the Threads Library 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 the values of any local variables unavailable.

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 status exception; however, subsequent `CATCHES` will function properly.

- 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 indicates that some thread incurred an access violation that was propagated as an exception without being fully handled.

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

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 the location from which the exception (in this example, the ACCVIO condition) is originating.

6

Examples

This chapter presents two example programs that use routines in the **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 on their default status-returning mechanism to indicate routine completion status. Example 6–1 uses the POSIX cleanup handler mechanism to clean up from thread cancelation. In contrast, Example 6–2 uses the exception package to capture and clean up from thread cancelation and other synchronous fatal error conditions.

6.1 Prime Number Search Example

Example 6–1 shows the use of **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 obtain 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 the Threads Library 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 broadcast. 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 broadcast. 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, to control more easily 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 **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)

```
/*
 *
 * 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 */


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

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

(continued on next page)

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

```
/*
 * 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);

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

    while (not_done) {

        /* Test for cancelation 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;
```

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

```
        /* 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");

            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 cancelation */
            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.
     */
```

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 in order 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 canceled\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.
     */
}


/*
 * 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;
        }
    }
}
```

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

```
/*
 * 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 **pthread** interface routines but also uses the exception package to capture and clean up 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 it.

Example 6–2 C Program Example (Asynchronous User Interface)

```
/*
 *
 * 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) {            \
    fprintf (stderr, "%s status %d: %s\n", string, status, 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

/*
 * 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 */
```

Example 6–2 (Cont.)   C Program Example (Asynchronous User Interface)

```
/*
 * 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);

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");
```

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * 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 */

        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");
```

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");

        /*
         * 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;
```

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    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);
    }

}




/*
 * 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");
}
```

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;

    /*
     * Perform exit actions (thread was not canceled).
     */
    do_cleanup ((int)arg, ST_FINISHED);

    /*
     * All thread routines return a value.  This program does not check the
     * value, however.
     */
    return arg;
}
```

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```c
/*
 * 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 of which to query the progress.
 */
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 have 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 are 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;

        /*
         * 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;
```

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
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;
        }

        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");
    }
}
```

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);

    /*
     * All thread routines return a value.  This program does not check the
     * value, however.
     */
    return arg;
}

/*
 * Wait for the completion of the specified command.
 */
static void
do_wait (int index)
{
    int status;
    void *value;
```

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    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 canceled.\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;

    do {
        /*
         * We are 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 does not 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");
```

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
            /*
             * 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;

                /*
                 * 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 the Threads Library to 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;
}

/*
 * 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;
    }

}
```

Example 6–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * 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.1 (pthread) Routines Reference

Part II provides detailed descriptions of routines that constitute the **pthread** interface. These routines (with the prefix pthread_) implement the IEEE POSIX 1003.1-1996 (or POSIX.1) standard, subject to the capabilities of the host operating system.

_____ **Note** _____

The **pthread** routines described here are based on the final POSIX.1 standard approved by the IEEE.

Threads Library users should be aware that applications that use the obsolete **d4** interfaces will require significant modifications to upgrade to the **pthread** interface. (The obsolete **d4** interface corresponds to the IEEE POSIX 1003.4a/Draft 4 document.)

_____

The global errno variable is not used by the **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.1 standard. That is, the routine might not be available in implementations of the POSIX.1 standard other than the Threads Library.

The Threads Library adds the extensions 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 in the IEEE POSIX 1003.1-1996 standard; these routines include pthread_attr_getguardsize(), pthread_attr_setguardsize(), pthread_mutexattr_gettype(), and pthread_mutexattr_settype(). The Threads Library 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 portable to other implementations of the POSIX.1 standard.

pthread_atfork

Declares fork handler routines to be called when the calling thread's process forks a child process.

This routine is for Tru64 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>
 include <signal.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 by the parent process before creating the child process.

parent
Address of a routine that performs the fork parent handling. This routine is called by 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 by 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 Tru64 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, the new child process might have locked mutexes that are copies of mutexes that were locked in the parent process by threads that are not 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 does not apply...like heap owned */
      /* by other threads          */
      }

   {
      .
      .
      .

    pthread_atfork(my_prepare, my_parent, my_child);
      .
      .
    fork();
   }
```

If you do not want to use fork handlers, you can set any of this routine's arguments to NULL.

---------------------------------- **Note** ----------------------------------

It is illegal 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 to record the fork handler routines' addresses.

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 not a valid thread attributes object.

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.

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.

pthread_attr_getdetachstate

Return	Description
[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 the overflow of a thread's stack. A guard area is a region of no-access memory that the Threads Library 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.

pthread_attr_getguardsize

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_setguardsize()
pthread_attr_setstacksize()
pthread_create()
```

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 not a valid thread attributes object.

pthread_attr_getinheritsched

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
Reserved for future use. The value must be zero (0).

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.

pthread_attr_getname_np

The object name is a C language string and provides an identifier that is meaningful to a person debugging a 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 not a valid thread attributes object.

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 not a valid thread attributes object.

pthread_attr_getschedparam

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 not a valid thread attributes object.

pthread_attr_getschedpolicy

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**.

The Threads Library selects at most one thread to execute on each processor at any point in time. The Threads Library 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.
[EINVAL]	The value specified by attr is not a valid thread attributes object.
[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 when 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:

pthread_attr_getstackaddr

Return	Description
0	Successful completion.
[EINVAL]	The value specified by attr is not a valid thread attributes object.

Associated Routines

```
pthread_attr_getguardsize()
pthread_attr_getstacksize()
pthread_attr_init()
pthread_attr_setguardsize()
pthread_attr_setstackaddr()
pthread_attr_setstacksize()
pthread_create()
```

pthread_attr_getstackaddr_np

Obtains the stack address attribute of the specified thread attributes object.

Syntax

pthread_attr_getstackaddr_np(

        attr,

        stackaddr,

        size);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
stackaddr	void	write
size	size_t	write

C Binding

#include <pthread.h>

int
pthread_attr_getstackaddr (
        const pthread_attr_t   *attr,
        void   **stackaddr,
        size_t   *size);

Arguments

attr
Address of the thread attributes object whose stack address attribute is obtained.

stackaddr
Receives the address of the stack region of the thread attributes object specified by attr.

size
The size of the stack region in bytes.

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 when this routine is called.

The stack address attribute of a thread attributes object points to the origin of the stack for a new thread.

Unlike pthread_attr_getstackaddr(), this routine is a much more reliable portable interface. With the POSIX standard pthread_attr_getstackaddr(), a stack is specified using a single, undefined, address. An implementation of the standard can only assume that the specified value represents the value to which the thread's stack pointer should be set when beginning execution. However, this requires the application to know how the machine uses the stack. For example,

pthread_attr_getstackaddr_np

a stack may "grow" either up (to higher addresses) or down (to lower addresses), and may be decreased (or increased) either before or after storing a new value.

The Threads Library provides an alternative interface with `pthread_attr_getstackaddr_np()`. Instead of returning a stack address, it returns the base (lowest) address and the size.

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 not a valid thread attributes object.

Associated Routines

`pthread_attr_setstackaddr_np()`

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 bytes 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 not a valid stack attributes object.

pthread_attr_getstacksize

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 previously listed.

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.

pthread_attr_init

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 not a valid thread attributes object.
[ENOMEM]	Insufficient memory 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 PTHREAD_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.

pthread_attr_setdetachstate

When a thread that has not been detached completes execution, the Threads Library retains the state of that thread to allow another thread to join with it. If the thread is detached before it completes execution, the Threads Library 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, such as to join with it, 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 attr argument is not a valid threads attribute object or 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, with its associated overflow warning area, can help a multithreaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that the Threads Library allocates at the overflow end of the thread's stack, following the thread's overflow warning area. If the thread attempts to write in the overflow warning area, a stack overflow exception occurs. Your program can catch this exception and continue processing as long as the thread does not attempt to write in the guard area. When any thread attempts to access a memory location within the guard area, a memory addressing violation occurs without the possibility of recovery.

pthread_attr_setguardsize

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 the Threads Library 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.

The Threads Library 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 overflow warning or 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 not a valid thread attributes object, or the argument guardsize contains an invalid value.

Associated Routines

```
pthread_attr_init()
pthread_attr_getguardsize()
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.

pthread_attr_setinheritsched

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.

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]	The value specified by the attr argument is not a valid thread attributes object, or the inheritsched argument contains an invalid value.
[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_attr_setscope()
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
Reserved for future use. The value must be zero (0).

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 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.

pthread_attr_setname_np

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 not a valid thread attributes object, 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** _____

The Threads Library provides only the `sched_priority` scheduling parameter. 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 you are accessing shared data. With a sufficient number of processors
present, all ready threads, regardless of priority, execute simultaneously. Even
on a uniprocessor, a lower priority thread could either execute before or be
interleaved with a higher priority thread, for example due to page fault behavior.
See Chapter 1 and Chapter 2 for more information.

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, the Threads Library 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 platform. On Tru64 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 attr is not a valid thread attributes object, or 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.

Preemption is caused by both scheduling and policy. 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 attr is not a valid thread attributes object, or 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 contention 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.

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 Tru64 UNIX, the Threads Library supports both process contention scope and system contention scope threads. On OpenVMS, the Threads Library supports only process contention scope threads.

_____

The Threads Library selects at most one thread to execute on each processor at any point in time. The Threads Library 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 either 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.
[EINVAL]	The value specified by attr is not a valid thread attributes value, or the value specified by scope 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_attr_setinheritsched()
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 the Threads Library 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 Tru64 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, the Threads Library ignores the attributes object's guardsize attribute and provides no thread stack guard area or overflow warning 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.
[EINVAL]	The value specified by attr is not a valid thread attributes object.

Associated Routines

```
pthread_attr_getguardsize()
pthread_attr_getstackaddr()
pthread_attr_getstacksize()
pthread_attr_init()
pthread_attr_setguardsize()
pthread_attr_setstacksize()
pthread_create()
```

---

pthread_attr_setstackaddr_np

Changes the stack address and size of the specified thread attributes object.

Syntax

pthread_attr_setstackaddr_np(
>> *attr,*
>> *stackaddr,*
>> *size*);

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
stackaddr	void	read
size	size_t	read

C Binding

#include <pthread.h>

int
pthread_attr_setstackaddr_np (
        pthread_attr_t   *attr,
        void   *stackaddr,
        size_t   size);

Arguments

attr
Address of the thread attributes object whose stack address attribute is to be modified.

stackaddr
New value for the address of the stack region of the thread attributes object specified by attr.

size
The size of the stack region in bytes.

Description

This routine uses the values specified in the stackaddr and size arguments to set the base stack address and size 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 default value for the stack address attribute of an initialized thread attributes object is NULL.

Unlike pthread_attr_setstackaddr(), this routine is a much more reliable portable interface. With the POSIX standard pthread_attr_setstackaddr(), a stack is specified using a single, undefined, address. An implementation of the standard can only assume that the specified value represents the value to which the thread's stack pointer should be set when beginning execution. However, this requires the application to know how the machine uses the stack. For example,

a stack may "grow" either up (to higher addresses) or down (to lower addresses), and may be decreased (or increased) either before or after storing a new value.

The Threads Library provides an alternative interface with `pthread_attr_setstackaddr_np()`. Instead of specifying a stack address, you specify the base (lowest) address and the size.

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 not a valid thread attributes object.

Associated Routines

`pthread_attr_getstackaddr_np()`

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 the 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. On OpenVMS systems, only the initial thread can dynamically extend its stack. On Tru64 UNIX systems, very large stacks can be created, but only a few pages are committed.

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 not a valid thread attributes object, or the value specified by stacksize either is less than `PTHREAD_STACK_MIN` or exceeds a Threads Library-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 will receive 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 called with the arg argument (see the description for `pthread_cleanup_push()`) 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 the Threads Library 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 (}). This routine and

`pthread_cleanup_push()` are implemented as exceptions, and may not work in a C++ environment. (See Chapter 5 for more information.)

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 <phtread.h>

void
pthread_cleanup_push(
        void    (*routine)(void *),
        void    *arg);
```

Arguments

routine
Routine executed as the cleanup handler.

arg
Argument passed 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 called 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 the Threads Library 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 (}). This routine and `pthread_cleanup_pop()` are implemented as exceptions, and may not work in a C++ environment. (See Chapter 5 for more information.)

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. Call this routine when a condition variable attributes object will no longer be referenced.

Condition variables that were created using this attributes object are not affected by the destruction of the condition variable attributes object.

The results of calling this routine are unpredictable if the value specified by the attr argument refers to a condition variable 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 attributes object specified by attr is invalid.

Associated Routines

pthread_condattr_init()

pthread_condattr_getpshared

Obtains the process-shared attribute of the specified condition variable attributes object.

This routine is for Tru64 UNIX systems only.

Syntax

pthread_condattr_getpshared(

                attr,

                pshared);

Argument	Data Type	Access
attr	opaque pthread_condattr_t	read
pshared	int	write

C Binding

#include <pthread.h>

int
pthread_condattr_getpshared (
        const pthread_condattr_t   *attr,
        int   *pshared);

Arguments

attr
Address of the condition variable attributes object whose process-shared attribute is obtained.

pshared
Receives the value of the process-shared attribute of the condition variable attributes object specified by attr.

Description

This routine obtains the value of the process-shared attribute of the condition variable attributes object specified by the attr argument and stores it in the location specified by the pshared argument. The specified attributes object must already be initialized at the time this routine is called.

Creating a condition variable whose process-shared attribute is set to PTHREAD_PROCESS_PRIVATE permits it to be operated upon by threads created within the same process as the thread that initialized that condition variable. If threads in other processes attempt to operate on such a condition variable, the behavior is undefined.

The default value of the process-shared attribute of an initialized condition variable attributes object is PTHREAD_PROCESS_PRIVATE.

Creating a condition variable whose process-shared attribute is set to PTHREAD_PROCESS_SHARED permits it to be operated upon by any thread that has access to the memory where that condition variable is allocated, even if it is allocated in memory that is shared by multiple processes.

pthread_condattr_getpshared

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 not a valid attributes object.

Associated Routines

```
pthread_condattr_destroy()
pthread_condattr_init()
pthread_condattr_setpshared()
pthread_cond_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, on OpenVMS systems, no attributes affecting condition variables are defined; you cannot change any attributes in the condition variable attributes object. On Tru64 UNIX systems, the PSHARED attribute is defined.

The pthread_condattr_init() and pthread_condattr_destroy() routines are provided for future expandability of the **pthread** interface and to conform with the POSIX.1 standard. These routines serve no useful function, because there are no pthread_condattr_set*() type routines available at this time.

pthread_condattr_init

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 not a valid condition variable attributes object.
[ENOMEM]	Insufficient memory exists to initialize the condition variable attributes object.

Associated Routines

```
pthread_condattr_destroy()
pthread_cond_init()
```

pthread_condattr_setpshared

Changes the process-shared attribute of the specified condition variable attributes object.

This routine is for Tru64 UNIX systems only.

Syntax

pthread_condattr_setpshared(
                attr,
                pshared);

Argument	Data Type	Access
attr	opaque pthread_condattr_t	write
pshared	int	read

C Binding

#include <pthread.h>

int
pthread_condattr_setpshared (
        pthread_condattr_t   *attr,
        int   pshared);

Arguments

attr
Address of the condition variable attributes object whose process-shared attribute is to be modified.

pshared
New value for the process-shared attribute of the condition variable attributes object specified by attr.

Description

This routine uses the value specified in the pshared argument to set the process-shared attribute of the condition variable attributes object specified in the attr argument.

Creating a condition variable whose process-shared attribute is set to PTHREAD_PROCESS_PRIVATE, permits it to be operated upon by threads created within the same process as the thread that initialized that condition variable. If threads of differing processes attempt to operate on such a condition variable, the behavior is undefined.

The default value of the process-shared attribute of an initialized condition variable attributes object is PTHREAD_PROCESS_PRIVATE.

Creating a condition variable whose process-shared attribute is set to PTHREAD_PROCESS_SHARED permits it to be operated upon by any thread that has access to the memory where that condition variable is allocated, even if it is allocated in memory that is shared by multiple processes.

pthread_condattr_setpshared

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 not a valid attributes object, or the value specified by *pshared* is outside the range of legal values for that attribute.

Associated Routines

```
pthread_condattr_destroy()
pthread_condattr_init()
pthread_condattr_getpshared()
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 one of those threads 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 not a valid condition variable.

pthread_cond_broadcast

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 the Threads Library 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 either 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 not a valid condition variable.
[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.

pthread_cond_destroy

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 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.

pthread_cond_getname_np

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 not a valid condition variable.

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 specified in the attr argument. 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 macro PTHREAD_COND_INITIALIZER to initialize statically allocated condition variables to the default condition variable attributes. To invoke 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().

pthread_cond_init

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. 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 not a valid attributes object.
[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 for 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
Reserved for future use. The value must be zero (0).

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 multithreaded application. The maximum number of characters in the object name is 31.

pthread_cond_setname_np

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 not a valid condition variable object, 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 of the threads 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.

If the calling thread holds the lock to the target condition variable's associated mutex while setting the variable's wait predicate, that thread can call pthread_cond_signal() to signal the variable even after releasing the lock on that mutex. However, for more predictable scheduling behavior, call pthread_cond_signal() before releasing the target condition variable's associated mutex.

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 not a valid condition variable.

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 Tru64 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. If no threads are waiting on the condition variable at the time of the call to pthread_cond_signal_int_np(), the next future waiting thread will be automatically released (that is, it will not actually wait). This routine establishes a "pending" wake if necessary.

You can call this routine regardless of whether the associated mutex is either locked or unlocked. (Never lock a mutex from an interrupt handler routine.)

_____ **Note** _____

This routine allows you to signal a condition variable from a software interrupt handler. Do not call this routine from noninterrupt code. To signal a condition variable from the normal noninterrupt level, use pthread_cond_signal().

_____

pthread_cond_signal_int_np

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 not a valid condition variable.

Associated Routines

```
pthread_cond_broadcast()
pthread_cond_signal()
pthread_cond_sig_preempt_int_np()
pthread_cond_timedwait()
pthread_cond_wait()
```

pthread_cond_sig_preempt_int_np

Wakes one thread that is waiting on the specified condition variable (called from interrupt level only).

Syntax

pthread_cond_sig_preempt_int_np (cond)

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

C Binding

```
 void
pthread_cond_sig_preempt_int_np (
          pthread_cond_t    *cond);
```

Arguments

cond
Condition variable signaled.

Description

This routine wakes one thread waiting on a condition variable. It can only be called from a software interrupt handler routine. Calling this routine implies that it might be possible for a single waiting thread to proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true.

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.

You can call this routine when the associated mutex is either locked or unlocked. (Never try to lock a mutex from an interrupt handler.)

This routine allows you to signal a thread from a software interrupt handler. Do not call this routine from noninterrupt code. If you want to signal a thread from the normal noninterrupt level, use pthread_cond_signal.

_____ **Note** _____

If a waiting thread has a preemptive scheduling policy and a higher priority than the thread which was running when the interrupt occurred, then the waiting thread will preempt the interrupt routine and begin to run immediately. This is unlike pthread_cond_signal_int_np() which causes the condition variable to be signaled at a safe point after the interrupt has completed. pthread_cond_sig_preempt_int_np() avoids the possible latency which pthread_cond_signal_int_np() may introduce; however, a side effect of this is that during the call to pthread_cond_sig_preempt_int_np() other threads may run if a preemption occurs. Thus, once an interrupt routine calls pthread_cond_sig_preempt_int_np() it can no longer rely on

pthread_cond_sig_preempt_int_np

any assumptions of exclusivity or atomicity which are typically provided by interrupt routines. Furthermore, once the call to `pthread_cond_sig_preempt_int_np()` is made, in addition to other threads running, subsequent interrupts may be delivered at any time as well (that is, they will not be blocked until the current interrupt completes). For this reason, it is recommended that `pthread_cond_sig_preempt_int_np()` be called as the last statement in the interrupt 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 cond is not a valid condition variable.

Associated Routines

```
pthread_cond_broadcast()
pthread_cond_signal()
pthread_cond_signal_int_np()
pthread_cond_timedwait()
pthread_cond_wait()
```

pthread_cond_timedwait

Causes a thread to wait for the specified condition variable to be signaled or broadcast, 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 broadcast. 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.

Description

This routine causes a thread to wait until one of the following occurs:

* The specified condition variable is signaled or broadcast.

* The current system clock time is greater than or equal to the time specified by the abstime argument.

pthread_cond_timedwait

This routine is identical to `pthread_cond_wait()`, except that this routine can return before a condition variable is signaled or broadcast, 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. When the thread regains control after calling `pthread_cond_timedwait()`, the mutex is locked and the thread is the owner. This is true regardless of why the wait ended. If general cancelability is enabled, the thread reacquires the mutex (blocking for it if necessary) before the cleanup handlers are run (or before the exception is raised).

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 have locked the mutex specified in mutex. The results of this routine are unpredictable if this routine is called without first locking the mutex. The only routines that are supported for use with asynchronous cancelability enabled are those that disable asynchronous cancelability.

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]	The Threads Library 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 broadcast.

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 broadcast. 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. When the thread regains control after calling `pthread_cond_wait()`, the mutex is locked and the thread is the owner. This is true regardless of why the wait ended. If general cancelability is enabled, the thread reacquires the mutex (blocking for it if necessary) before the cleanup handlers are run (or before the exception is raised).

pthread_cond_wait

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 broadcast. 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.

The only routines that are supported for use with asynchronous cancelability enabled are those that disable asynchronous cancelability.

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.
[ENOMEM]	The Threads Library 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:

- The Threads Library 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 Tru64 UNIX, or the `pthread.h` header file on other 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

The Threads Library 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, condition variables or read-write locks.

For the duration of the new thread's existence, the Threads Library 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).

The Threads Library assigns each new thread a thread identifier, which is written into the address specified as the `pthread_create()` routine's *thread* argument. The new thread's thread identifier is written 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 Tru64 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, the following actions are performed:

1. A return value (if one is available) is written into the terminated thread's thread object, as follows:

   - If the thread has been canceled, the value PTHREAD_CANCELED is written into the thread's thread object.

   - If the thread terminated by returning from its start routine, the return value is copied from the start routine (if one is available) into the thread's thread object. Alternatively, if the thread explicitly called pthread_exit(), the value received in the value_ptr argument (from pthread_exit()) is stored in 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(), the Threads Library calls, in turn, each cleanup handler that this thread declared (using pthread_cleanup_push()) and that is not yet removed (using pthread_cleanup_pop()). (The Threads Library also transfers control to any appropriate CATCH, CATCH_ALL, or FINALLY blocks , as described in Chapter 5 .)

   The Threads Library 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(), the Threads Library raises the pthread_exit_e exception. To exit the terminated thread due to cancelation, the Threads Library raises the pthread_cancel_e exception.

   Your program can use the 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 exception package.

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

   - The thread's value for the corresponding key is set to NULL.

   - Call each thread-specific data destructor function in this multithreaded process' list of destructors.

   Repeat 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 2.6 for more information about thread-specific data.

5. Awaken the thread (if there is one) that is currently waiting to join with the terminated thread. That is, awaken the thread that is waiting in a call to `pthread_join()`.

6. If the thread is already detached, destroy 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 not a valid attributes block.
[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 either 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.

You can create a thread 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 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
Threads Library status exception object whose status code is obtained.

code
Receives the system-specific status code associated with the specified status exception object.

Description

This routine obtains and returns the system-specific status value from the 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 Threads Library status exceptions, use this routine within a `CATCH` or `CATCH_ALL` 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 the Threads Library.

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 status exception, it sets the code argument and returns zero (0). Possible return values are as follows:

pthread_exc_get_status_np

Return	Description
0	Successful completion.
[EINVAL]	The exception argument is not a valid status exception object.

Associated Routines

pthread_exc_set_status_np()

pthread_exc_matches_np

(Macro) Determines whether two Threads Library 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
Threads Library exception object.

exception2
Threads Library exception object.

Description

This routine compares two 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 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 Threads Library 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
Threads Library exception object that has been set with a status value.

Description

This routine produces a text message on the `stderr` device (Tru64 UNIX 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 status exceptions, use this routine within a `CATCH` or `CATCH_ALL` code block to produce the message associated with a caught exception. Note that any exception objects set to the same status value are considered equivalent by the Threads Library.

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 Threads Library 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
Threads Library 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 address exception object. This transforms the address exception object into a status exception object.

The exception argument must already have been initialized with the exception package's EXCEPTION_INIT macro.

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 the Threads Library.

Return Values

None

pthread_exc_set_status_np

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. The Threads Library 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 are 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.

The Threads Library issues an implicit call to pthread_exit() when a thread returns from the start routine that was used to create it. The Threads Library 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, do not use references to local variables of the existing thread for the value_ptr argument of the pthread_exit() routine.

pthread_exit

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 the Threads Library
automatically manages the concurrency of all threads in a multithreaded process,
it ignores this concurrency level value.

The concurrency level value has no effect on the behavior of a multithreaded
program that uses the Threads Library. 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 the Threads Library
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' concurrency level global
variable. If this process has never called the pthread_setconcurrency() routine,
this routine returns zero (0).

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 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 policies and their meanings.

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 either 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 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 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.

Description

This routine obtains the thread-specific data associated with the specified key for the current thread. Obtain this key by calling the pthread_key_create() routine. 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).

Description

This routine adds a specified interval to the current absolute system time and returns a new absolute time. This new absolute time may then be 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-bit pointer environments for OpenVMS Alpha. For information regarding 32- and 64-bit pointers, see Appendix B. Ensure that your compiler provides 64-bit support before you use `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 from its start routine).

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.

_____ **Note** _____

For OpenVMS Alpha systems:
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()`. 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()`. 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), the Threads Library 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
Location where the new thread-specific data key will be stored.

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. The initial value of the key in all threads is NULL.

The Threads Library 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, the thread is terminated. 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.

Attempting to use the 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 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 not a valid key.

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 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 not a valid key.

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 key object.

mbz
Reserved for future use. The value must be zero (0).

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 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 not a valid key, 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 Tru64 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>
 include <signal.h>

int
pthread_kill (
     pthread_t   *thread,*
     int   *sig*);

Arguments

thread
Thread to receive a signal request.

sig
A signal request.

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 Threads Library global mutex.

Syntax

pthread_lock_global_np( );

C Binding

#include <pthread.h>

int
pthread_lock_global_np (void);

Arguments

None

Description

This routine locks the Threads Library global mutex. If the global mutex is currently held by another thread when a thread calls this routine, the calling thread waits for the global mutex to become available and then locks it.

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 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 should use 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, the Threads Library 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 not a valid attributes object.

Associated Routines

```
pthread_mutexattr_init()
```

pthread_mutexattr_getpshared

Obtains the value of the process-shared attribute of the specified mutex attributes object.

This routine is for Tru64 UNIX systems only.

Syntax

pthread_mutexattr_getpshared(
                              attr,
                              pshared);

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read
pshared	int	write

C Binding

#include <pthread.h>

int
pthread_mutexattr_getpshared (
        const pthread_mutexattr_t   *attr,
        int   *pshared);

Arguments

attr
Address of the mutex attributes object whose process-shared attribute is obtained.

pshared
Value received from process-shared attribute of the mutex attributes object specified in attr.

Description

This routine obtains the value of the process-shared attribute of the mutex attributes object specified by the attr argument and stores it in the location specified by the pshared argument. This attributes object must already be initialized at the time this routine is called.

Setting the process-shared attribute to PTHREAD_PROCESS_PRIVATE permits a mutex to be operated upon by threads created within the same process as the thread that initialized the mutex. If threads of differing processes attempt to operate on such a mutex, the behavior is undefined.

The default value of the process-shared attribute of a mutex attributes object is PTHREAD_PROCESS_PRIVATE.

Setting the process-shared attribute to PTHREAD_PROCESS_SHARED permits a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

pthread_mutexattr_getpshared

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 not a valid attributes object.

Associated Routines

```
pthread_mutex_init()
pthread_mutexattr_destroy()
pthread_mutexattr_init()
pthread_mutexattr_setpshared()
```

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:

pthread_mutexattr_gettype

Return	Description
0	Successful completion.
[EINVAL]	The value specified by attr is not a valid mutex attributes object.

Associated Routines

```
pthread_mutexattr_init()
pthread_mutexattr_settype()
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:

pthread_mutexattr_init

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_setpshared

Changes the value of the process-shared attribute of the specified mutex attributes object.

This routine is for Tru64 UNIX systems only.

Syntax

pthread_mutexattr_setpshared(

                     attr,

                     pshared);

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write
pshared	int	read

C Binding

#include <pthread.h>

int
pthread_mutexattr_setpshared (
        pthread_mutexattr_t   *attr,
        int   pshared);

Arguments

attr
Address of the mutex attributes object whose process-shared attribute is to be modified.

pshared
Value to set in the process-shared attribute of the mutex attributes object specified by attr.

Description

This routine uses the value specified in the pshared argument to set the value of the process-shared attribute of an initialized mutex attributes object specified in the attr argument.

Setting the process-shared attribute to PTHREAD_PROCESS_PRIVATE permits a mutex to be operated upon by threads created within the same process as the thread that initialized the mutex. If threads of differing processes attempt to operate on such a mutex, the behavior is undefined.

The default value of the process-shared attribute of a mutex attributes object is PTHREAD_PROCESS_PRIVATE.

Setting the process-shared attribute to PTHREAD_PROCESS_SHARED permits a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

pthread_mutexattr_setpshared

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 not a valid mutex attributes object, or the new value specified for the attribute is outside the range of legal values for that attribute.

Associated Routines

```
pthread_mutex_init()
pthread_mutexattr_destroy()
pthread_mutexattr_init()
pthread_mutexattr_getpshared()
```

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:

pthread_mutexattr_settype

Return	Description
0	Successful completion.
[EINVAL]	The value specified by attr or type is not a valid mutex attributes type.
[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_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,
the Threads Library 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.
[EINVAL]	The value specified by mutex is not a valid mutex.

pthread_mutex_destroy

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 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.

pthread_mutex_getname_np

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 not a valid mutex.

Associated Routines

pthread_mutex_setname_np()

pthread_mutex_init

Initializes a mutex.

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 to be initialized.

attr
Mutex attributes object that defines the characteristics of the mutex to be initialized.

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.

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.

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. If a mutex is allocated on a stack, static initializers cannot be used on the mutex.

pthread_mutex_init

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.
[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 not a valid mutex.
[ENOMEM]	Insufficient memory exists to initialize the mutex.
[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.

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 its 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.

pthread_mutex_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.
[EDEADLK]	A deadlock condition is detected.
[EINVAL]	The value specified by mutex is not a valid mutex.

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
Reserved for future use. The value must be zero (0).

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 multithreaded application. The maximum number of characters in the object name is 31.

pthread_mutex_setname_np

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 not a valid mutex, or the length in characters of name exceeds 31.
[ENOMEM]	Insufficient memory 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 by the calling thread.

- 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.

pthread_mutex_trylock

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 not a valid mutex.

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 Tru64 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.

pthread_mutex_unlock

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 not a valid mutex.
[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 a routine that is executed by a single thread, once.

Syntax

pthread_once(
         once_control,
         routine);

Argument	Data Type	Access
once_control	opaque pthread_once_t	modify
routine	procedure	read

C Binding

#include <pthread.h>

int
pthread_once (
         pthread_once_t   *once_control,
         void   (*routine) (void));

Arguments

once_control
Address of a record that controls the one-time execution code. Each one-time execution routine must have its own unique pthread_once_t record.

routine
Address of a procedure to be executed once. This routine is called only once, regardless of the number of times it and its associated once_control block 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 routine with no arguments. Subsequent calls to pthread_once() with the same once_control will not call the routine. On return from pthread_once(), it is guaranteed that the 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 a routine that directly or indirectly results in a recursive call to pthread_once() and that specifies the same routine argument, the recursive call can result in a deadlock.

_____

pthread_once

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)
    {
     /* initialization code goes here */
     flag = TRUE;
    }
 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_rwlockattr_destroy

Destroys a previously initialized read-write lock attributes object.

Syntax

pthread_rwlockattr_destroy(
                    attr);

Argument	Data Type	Access
attr	opaque pthread_rwlockattr_t	write

C Binding

#include <pthread.h>

int
pthread_rwlockattr_destroy (
            pthread_rwlockattr_t   *attr);

Arguments

attr
Address of the read-write lock attributes object to be destroyed.

Description

This routine destroys the read-write lock attributes object referenced by attr; that is, the object becomes uninitialized.

After successful completion of this routine, the results of using attr in a call to any routine (other than `pthread_rwlockattr_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 value specified by attr is not a valid attributes block.

Associated Routines

```
pthread_rwlockattr_init()
pthread_rwlock_init()
```

pthread_rwlockattr_getpshared

Obtains the value of the process-shared attribute of a read-write lock attributes object.

This routine is for Tru64 UNIX systems only.

Syntax

pthread_rwlockattr_getpshared(*attr,*

pshared);

Argument	Data Type	Access
attr	opaque pthread_rwlockattr_t	read
pshared	int	write

C Binding

#include <pthread.h>

int
pthread_rwlockattr_getpshared (
        const pthread_rwlockattr_t   *attr,
        int   *pshared);

Arguments

attr
Address of the read-write lock attributes object whose process-shared attribute is to be obtained.

pshared
Receives the value of the process-shared attribute of the read-write lock attributes object specified by attr.

Description

This routine obtains the value of the process-shared attribute from the read-write lock attributes object specified by the attr argument and stores it in the location specified by the pshared argument. This attributes object must already be initialized at the time this routine is called.

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 not a valid attributes object.

Associated Routines

```
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_unlock()
pthread_rwlock_wrlock()
pthread_rwlockattr_init()
pthread_rwlockattr_setpshared()
```

pthread_rwlockattr_init

Initializes a read-write lock attributes object.

Syntax

pthread_rwlockattr_init(
                    attr);

Argument	Data Type	Access
attr	opaque pthread_rwlockattr_t	write

C Binding

#include <pthread.h>

int
pthread_rwlockattr_init (
        pthread_rwlockattr_t   *attr);

Arguments

attr
Address of the read-write lock attributes object to be initialized.

Description

This routine initializes the read-write lock attributes object referenced by attr and sets its attributes with default values.

The results of calling this routine are undefined if attr references an already initialized read-write lock attributes object.

After an initialized read-write lock attributes object has been used to initialize one or more read-write lock objects, any operation on that attributes object (including destruction) has no effect on those read-write lock objects.

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 initialize the read-write lock attributes object

Associated Routines

```
pthread_rwlockattr_destroy()
pthread_rwlock_init()
```

pthread_rwlockattr_setpshared

Sets the value of the process-shared attribute of a read-write lock attributes object.

This routine is for Tru64 UNIX systems only.

Syntax

pthread_rwlockattr_setpshared(*attr*,

pshared);

Argument	Data Type	Access
attr	opaque pthread_rwlockattr_t	write
pshared	int	read

C Binding

#include <pthread.h>

int
pthread_rwlockattr_setpshared (
        pthread_rwlockattr_t   *attr,
        int   pshared);

Arguments

attr
Address of the read-write lock attributes object whose process-shared attribute is to be modified.

pshared
New value for the process-shared attribute of the read-write lock attributes object specified by attr.

Description

This routine uses the value specified in the pshared argument to set the process-shared attribute of the read-write lock attributes object specified by the attr argument. This attributes object must already be initialized at the time this routine is called.

If the process-shared attribute is set to PTHREAD_PROCESS_PRIVATE, the read-write lock object can only be operated upon by threads created within the same process as the thread that initialized the read-write lock object. If threads of differing processes attempt to operate on such a read-write lock object, the behavior is undefined.

The default value of the process-shared attribute of a read-write lock attributes object is PTHREAD_PROCESS_PRIVATE.

If the process-shared attribute of a read-write lock attributes object is set to PTHREAD_PROCESS_SHARED, the read-write lock object can be operated upon by any thread that has access to the memory where that object is allocated, even if that object is allocated in memory that is shared by multiple processes.

pthread_rwlockattr_setpshared

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 not a valid attributes object, or the value *pshared* is outside the range of legal values for that attribute.

Associated Routines

```
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_unlock()
pthread_rwlock_wrlock()
pthread_rwlockattr_init()
pthread_rwlockattr_getpshared()
```

pthread_rwlock_destroy

Destroys a read-write lock object.

Syntax

pthread_rwlock_destroy(
                    rwlock);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	write

C Binding

#include <pthread.h>

int
pthread_rwlock_destroy (
            pthread_rwlock_t   *rwlock);

Arguments

rwlock
Address of the read-write lock object to be destroyed.

Description

This routine destroys the specified read-write lock object by uninitializing it, and should be called when the object is no longer referenced in your program. After this routine is called, the Threads Library may reclaim internal storage used by the specified read-write lock object. The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to pthread_rwlock_init().

It is illegal to destroy a locked read-write lock.

The results of this routine are unpredictable if the specified read-write lock object does not currently exist or is not initialized. This routine destroys the read-write lock object specified by the rwlock argument and releases any resources that the object used.

A destroyed read-write lock object can be reinitialized using the pthread_rwlock_init() routine. The results of otherwise referencing a destroyed read-write lock object are undefined.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

pthread_rwlock_destroy

Return	Description
0	Successful completion.
[EBUSY]	An attempt was made to destroy the object referenced by rwlock while it is locked or referenced.

Associated Routines

```
pthread_rwlock_init()
```

pthread_rwlock_getname_np

Obtains the object name from a read-write lock object.

Syntax

pthread_rwlock_getname_np(
                                rwlock,
                                name,
                                len);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	read
name	char	write
len	size_t	read

C Binding

#include <pthread.h>

int
pthread_rwlock_getname_np (
            pthread_rwlock_t   *rwlock,
            char   *name,
            size_t   len);

Arguments

rwlock
Address of the read-write lock 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 read-write lock object specified by rwlock to the buffer at the location name. 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 multithreaded application. The maximum number of characters in the object name is 31.

If the specified read-write lock object has not been previously set with an object name, this routine copies a C language null string into the buffer at location name.

pthread_rwlock_getname_np

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 rwlock is not a valid read-write lock.

Associated Routines

```
pthread_rwlock_setname_np()
```

pthread_rwlock_init

Initializes a read-write lock object.

Syntax

pthread_rwlock_init(*rwlock,*

attr);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	write
attr	opaque pthread_rwlockattr_t	read

C Binding

#include <pthread.h>

int
pthread_rwlock_init (
       pthread_rwlock_t   *rwlock,
       const pthread_rwlockattr_t   *attr);

Arguments

rwlock
Read-write lock object to be initialized.

attr
Read-write lock attributes object that defines the characteristics of the read-write lock to be initialized.

Description

This routine initializes a read-write lock object with the attributes specified by the read-write lock attributes object specified in attr. A read-write lock is a synchronization object that serializes access to shared information that needs to be read frequently and written only occasionally. A thread can acquire a read-write lock for shared read access or for exclusive write access.

Upon successful completion of this routine, the read-write lock is initialized and set to the unlocked state. If attr is set to NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used any number of times without being reinitialized.

Results of calling this routine are undefined if attr specifies an already initialized read-write lock or if rwlock is used without first being initialized.

If this routine returns unsuccessfully, rwlock is not initialized and the contents of rwlock are undefined.

A read-write lock is a resource of the process, not part of any particular thread. A read-write lock is neither destroyed not unlocked automatically when any thread exits. Because read-write locks are shared, they may be allocated in heap or static memory, but not on a stack.

pthread_rwlock_init

In cases where default read-write lock attributes are appropriate, you may use the `PTHREAD_RWLOCK_INITIALIZER` macro to statically initialize the lock object without calling this routine. The effect is equivalent to dynamic initialization by a call to `pthread_rwlock_init()` with attr specified as NULL, except that no error checks are performed. Statically initialized read-write locks need not be destroyed using `pthread_rwlock_destroy()`.

Use the `PTHREAD_RWLOCK_INITIALIZER` macro as follows:

`pthread_rwlock_t` rwlock= `PTHREAD_RWLOCK_INITIALIZER;`

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.
[EAGAIN]	The system lacks the necessary resources to initialize the read-write lock.
[EBUSY]	The Threads Library has detected an attempt to reinitialize the read-write lock (a previously initialized, but not yet destroyed, read-write lock object).
[EINVAL]	The value specified by attr is not a valid attributes block.
[ENOMEM]	Insufficient memory exists to initialize the read-write lock.
[EPERM]	The caller does not have privileges to perform this operation.

Associated Routines

`pthread_rwlock_destroy()`

pthread_rwlock_rdlock

Acquires a read-write lock object for read access.

Syntax

pthread_rwlock_rdlock(
                    rwlock);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	write

C Binding

#include <pthread.h>

int
pthread_rwlock_rdlock (
            pthread_rwlock_t   *rwlock);

Arguments

rwlock
Address of the read-write lock object to acquire for read access.

Description

This routine acquires a read-write lock for read access. If no thread already holds the lock for write access and there are no writers waiting to acquire the lock, the lock for read access is granted to the calling thread and this routine returns. If a thread already holds the lock for read access, the lock is granted and this routine returns.

A thread can hold multiple, concurrent locks for read access on the same read-write lock. In a given thread, for each call to this routine that successfully acquires the same read-write lock for read access, a corresponding call to pthread_rwlock_unlock must be issued.

If some thread already holds the lock for write access, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks until it can acquire the lock for read access. Results are undefined if the calling thread has already acquired a lock for write access on rwlock when this routine is called.

If the read-write lock object referenced by rwlock is not initialized, the results of calling this routine are undefined.

If a thread is interrupted (via a Tru64 UNIX signal or an OpenVMS AST) while waiting for a read-write lock for read access, upon return from the interrupt routine the thread resumes waiting for the lock as if it had not been interrupted.

pthread_rwlock_rdlock

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 read-write lock object was acquired for read access.
[EINVAL]	The value specified by rwlock does not refer to an initialized read-write lock object.
[EDEADLCK]	The calling thread already owns the specified read-write lock object for write access.
[EAGAIN]	The lock for read access could not be acquired because the maximum number of read lock acquisitions for rwlock has been exceeded.

Associated Routines

```
pthread_rwlock_init()
pthread_rwlockattr_init()
pthread_rwlock_tryrdlock()
pthread_rwlock_wrlock()
pthread_rwlock_unlock()
```

pthread_rwlock_setname_np

Changes the object name in a read-write lock object.

Syntax

pthread_rwlock_setname_np(

        rwlock,

        name,

        mbz);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	write
name	char	read
mbz	void	read

C Binding

#include <pthread.h>

int
pthread_rwlock_setname_np (
        pthread_rwlock_t   *rwlock,
        const char   *name,
        void   *mbz);

Arguments

rwlock
Address of the read-write lock object whose object name is to be changed.

name
Object name value to copy into the read-write lock object.

mbz
Reserved for future use. The value must be zero (0).

Description

This routine changes the object name in the read-write lock object specified by rwlock to the value specified by name. To set a new read-write lock object's object name, call this routine immediately after initializing the read-write lock object.

The object name is a C language string and provides an identifier that is meaningful to a person debugging a multithreaded application. The maximum number of characters in the object name is 31.

pthread_rwlock_setname_np

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 read-write lock object was acquired for read access.
[EINVAL]	The value specified by rwlock is invalid, or the length in characters of name exceeds 31.
[ENOMEM]	Insufficient memory to create a copy of the object name string.

Associated Routines

```
pthread_rwlock_getname_np()
pthread_rwlock_init()
```

pthread_rwlock_tryrdlock

Attempts to acquire a read-write lock object for read access without waiting.

Syntax

pthread_rwlock_tryrdlock(

                rwlock);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	write

C Binding

#include <pthread.h>

int
pthread_rwlock_tryrdlock (
        pthread_rwlock_t   *rwlock);

Arguments

rwlock
Address of the read-write lock object to acquire for read access.

Description

This routine attempts to acquire a read-write lock for read access, but does not wait for the lock if it not immediately available.

If no thread already holds the lock for write access and there are no writers waiting to acquire the lock, the lock for read access is granted to the calling thread and this routine returns. If a thread already holds the lock for read access, the lock is granted and this routine returns.

If some thread already holds the lock for write access, the calling thread will not acquire the read lock. Results are undefined if the calling thread has already acquired a lock for write access on rwlock when this routine is called.

A thread can hold multiple, concurrent locks for read access on the same read-write lock. In a given thread, for each call to this routine that successfully acquires the same read-write lock for read access, a corresponding call to pthread_rwlock_unlock() must be issued.

If the read-write lock object referenced by rwlock is not initialized, the results of calling this routine are undefined.

Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

pthread_rwlock_tryrdlock

Return	Description
0	Successful completion; the read-write lock object was acquired for read access.
[EAGAIN]	The lock for read access could not be acquired because the maximum number of read lock acquisitions for rwlock has been exceeded.
[EBUSY]	The read-write lock could not be acquired for read access because another thread already acquired it for write access or is blocked and waiting for it for write access.
[EDEADLCK]	The current thread already owns the read-write lock for writing.
[EINVAL]	The value specified by rwlock does not refer to an initialized read-write lock object.

Associated Routines

```
pthread_rwlockattr_init()
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_unlock()
pthread_rwlock_wrlock()
```

pthread_rwlock_trywrlock

Attempts to acquire a read-write lock object for write access without waiting.

Syntax

pthread_rwlock_trywrlock(
                    rwlock);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	write

C Binding

#include <pthread.h>

int
pthread_rwlock_trywrlock (
            pthread_rwlock_t   *rwlock);

Arguments

rwlock
Address of the read-write lock object to acquire for write access.

Description

This routine attempts to acquire the read-write lock referenced by rwlock for write access. If any thread already holds that lock for write access or read access, this routine fails and returns [EBUSY] and the calling thread does not wait for the lock to become available.

Results are undefined if the calling thread holds the read-write lock (whether for read or write access) at the time this routine is called.

If the read-write lock object referenced by rwlock is not initialized, the results of calling this routine are undefined.

Realtime applications can encounter **priority inversion** when using read-write locks. The problem occurs when a high-priority thread acquires a read-write lock that is about to be unlocked (that is, posted) by a low-priority thread, but the low-priority thread is preempted by a medium-priority thread. This scenario leads to priority inversion in that a high-priority thread is blocked by lower-priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of priority inversion and can deal with it in a number of ways, such as by having critical sections that are guarded by read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

pthread_rwlock_trywrlock

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 read-write lock object was acquired for write access.
[EBUSY]	The read-write lock could not be acquired for write access because it was already locked for write access or for read access.
[EDEADLCK]	The current thread already owns the read-write lock for write or read access.
[EINVAL]	The value specified by rwlock does not refer to an initialized read-write lock object.

Associated Routines

```
pthread_rwlockattr_init()
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_unlock()
pthread_rwlock_wrlock()
```

pthread_rwlock_unlock

Unlocks a read-write lock object.

Syntax

pthread_rwlock_unlock(
                    rwlock);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	write

C Binding

#include <pthread.h>

int
pthread_rwlock_unlock (
            pthread_rwlock_t   *rwlock);

Arguments

rwlock
Address of the read-write lock object to be unlocked.

Description

This routine releases a lock acquisition held on the read-write lock object referenced by rwlock. Results are undefined if rwlock is not held by the calling thread.

If this routine is called to release a lock for read access on rwlock and the calling thread also currently holds other locks for read access on rwlock, the read-write lock object remains in the read locked state. If this routine releases the calling thread's last lock for read access on rwlock, the calling thread is no longer one of the owners of the lock object.

If this routine is called to release a lock for write access on rwlock, the lock object is put in the unlocked state with no owners.

If a call to this routine results in the read-write lock object becoming unlocked and there are multiple threads waiting to acquire that lock for write access, the Threads Library uses the scheduling policy of those waiting threads to determine which thread next acquires the lock object for write access. If there are multiple threads waiting to acquire the read-write lock object for read access, the Threads Library uses the scheduling policy of those waiting threads to determine the order in which those threads acquire the lock for read access. If there are multiple threads waiting to acquire the read-write lock object for both read and write access, it is unspecified whether a thread waiting for read access or for write access next acquires the lock object.

If the read-write lock object referenced by rwlock is not initialized, the results of calling this routine are undefined.

pthread_rwlock_unlock

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 values specified by rwlock does not refer to an initialized read-write lock object.
[EPERM]	The calling thread does not hold the read-write lock object.

Associated Routines

```
pthread_rwlockattr_init()
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_wrlock()
```

pthread_rwlock_wrlock

Acquires a read-write lock for write access.

Syntax

pthread_rwlock_wrlock(
                        rwlock);

Argument	Data Type	Access
rwlock	opaque pthread_rwlock_t	write

C Binding

#include <pthread.h>

int
pthread_rwlock_wrlock (
            pthread_rwlock_t   *rwlock);

Arguments

rwlock
Address of the read-write lock object to acquire for write access.

Description

This routine attempts to acquire a read-write lock for write access. If any thread already has acquired the lock for write access or read access, the lock is not granted and the calling thread blocks until it can acquire the lock. A thread can hold only one lock for write access on a read-write lock.

Results are undefined if the calling thread holds the read-write lock (whether for read or write access) at the time this routine is called.

If the read-write lock object referenced by rwlock is not initialized, the results of calling this routine are undefined.

If a thread is interrupted (via a Tru64 UNIX signal or an OpenVMS AST) while waiting for a read-write lock for write access, upon return from the interrupt routine the thread resumes waiting for the lock as if it had not been interrupted.

Realtime applications can encounter **priority inversion** when using read-write locks. The problem occurs when a high-priority thread acquires a read-write lock that is about to be unlocked (that is, posted) by a low-priority thread, but the low-priority thread is preempted by a medium-priority thread. This scenario leads to priority inversion in that a high-priority thread is blocked by lower-priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of priority inversion and can deal with it in a number of ways, such as by having critical sections that are guarded by read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

pthread_rwlock_wrlock

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 read-write lock object was acquired for write access.
[EDEADLCK]	The calling thread already owns the read-write lock for write or read access.
[EINVAL]	The value specified by rwlock does not refer to an initialized read-write lock object.

Associated Routines

```
pthread_rwlockattr_init()
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_trywrlock()
pthread_rwlock_unlock()
```

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 type is `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 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.

—————————————————— **Caution** ——————————————————

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 POSIX
Threads 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 the Threads Library automatically manages the concurrency of all threads in a multithreaded process, it ignores this concurrency level value.

"Concurrency level" is a parameter used to coerce "simple" 2-level schedulers into allowing application concurrency. The Threads Library supplies the maximum concurrency at all times, automatically. It has no need for coercion, and calls `pthread_setconcurrency()` merely to determine the value returned by the next call to `pthread_getconcurrency()`.

The concurrency level value has no effect on the behavior of a multithreaded program that uses the Threads Library. 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 the Threads Library 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.
[EAGAIN]	The value specified by *new_level* would cause a system resource to be exceeded.
[EINVAL]	The value specified by *new_level* is negative.

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
Reserved for future use. The value must be zero (0).

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 either 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 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.
[EINVAL]	The length in characters of name exceeds 31.
[ENOMEM]	Insufficient memory to create a copy of the object name string.
[ESRCH]	The thread specified by thread does not exist.

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, the Threads Librray 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 platform. On Tru64 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 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 either 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.

pthread_setschedparam

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." The Threads Library simply stores value for later retrieval.

pthread_setspecific

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 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 Tru64 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>
 include <signal.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).

pthread_sigmask

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 will be delivered 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 Threads Library 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 Threads Library 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 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 to try to acquire the global mutex. The scheduling policy is used by this routine 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()
```

pthread_yield_np

Notifies the scheduler that the current thread is willing to release its processor to other threads of the same or higher priority.

Syntax

pthread_yield_np( );

C Binding

int
pthread_yield_np (void);

Arguments

None

Description

This routine notifies the thread scheduler that the current thread is willing to release its processor to other threads of equivalent or greater scheduling precedence. (A thread generally will release its processor to a thread of a greater scheduling precedence without calling this routine.) If no other threads of equivalent or greater scheduling precedence are ready to execute, the thread continues.

This routine can allow knowledge of the details of an application to be used to improve its performance. If a thread does not call pthread_yield_np(), other threads may be given the opportunity to run at arbitrary points (possibly even when the interrupted thread holds a required resource). By making strategic calls to pthread_yield_np(), other threads can be given the opportunity to run when the resources are free. This improves performance by reducing contention for the resource.

As a general guideline, consider calling this routine after a thread has released a resource (such as a mutex) which is heavily contended for by other threads. This can be especially important either if the program is running on a uniprocessor machine, or if the thread acquires and releases the resource inside a tight loop.

Use this routine carefully and sparingly, because misuse can cause unnecessary context switching which will increase overhead and actually degrade performance. For example, it is counter-productive for a thread to yield while it holds a resource that the threads to which it is yielding will need. Likewise, it is pointless to yield unless there is likely to be another thread that is ready to run.

––––––––––––––––––––– **Note** –––––––––––––––––––––

pthread_yield_np() is equivalent to sched_yield(). Use sched_yield() since it is part of the standard portable POSIX Threads Library.

–––

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]	The routine `pthread_yield_np()` is not supported by this implementation.

Associated Routines

```
pthread_attr_setschedparam()
pthread_getschedparam()
pthread_setschedparam()
```

sched_get_priority_max

Returns the maximum priority for the specified scheduling policy.

Syntax

sched_get_priority_max(
                    policy);

Argument	Data Type	Access
policy	integer	read

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.

sched_get_priority_min

Returns the minimum priority for the specified scheduling policy.

Syntax

sched_get_priority_min(
              policy);

Argument	Data Type	Access
policy	integer	read

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.

---

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.1-1996 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.
[ENOSYS]	The routine sched_yield() is not supported by this implementation.

Associated Routines

pthread_attr_setschedparam()
pthread_getschedparam()
pthread_setschedparam()

sigwait

Suspends a calling thread until a signal arrives.

This routine is for Tru64 UNIX systems only.

Syntax

sigwait(
        set,
        signal);

Argument	Data Type	Access
set	sigset_t	read
signal	integer	write

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 blocks 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.

Part III

Compaq Proprietary Interfaces: tis Routines Reference

Part III provides detailed descriptions of the Compaq proprietary 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 similar to the **pthread** functions for synchronization. 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** _____

Never use `tis_cond_wait()` in a nonthreaded environment. It cannot wait, as there would be no thread able to awaken the waiter. The **tis** "stub" will abort your 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 has no effect.

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 not a valid condition variable.

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, the Threads Library 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.
[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.)
[EINVAL]	The value specified by cond is not a valid condition variable.

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 Threads Library default condition variable attributes.

A condition variable is a synchronization object used 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 default condition variable attributes. Static initialization can be used only for a condition variable with storage class "extern" or "static" — "automatic" (stack local) objects must be initialized by calling tis_cond_init(). 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().

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 condition 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 cond is not a valid condition variable.
[ENOMEM]	Insufficient memory 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. When threads are not present, this routine has no effect.

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 not a valid condition variable.

Associated Routines

```
tis_cond_broadcast()
tis_cond_destroy()
tis_cond_init()
tis_cond_wait()
```

tis_cond_timedwait

Causes a thread to wait for the specified condition variable to be signaled or broadcast, such that it will awake after a specified period of time.

Syntax

tis_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 <tis.h>

int
tis_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 broadcast. See the `tis_get_expiration()` 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.

Description

If threads are not present, this function is equivalent to `sleep()`.

This routine causes a thread to wait until one of the following occurs:

- The specified condition variable is signaled or broadcast.

- The current system clock time is greater than or equal to the time specified by the abstime argument.

tis_cond_timedwait

This routine is identical to tis_cond_wait(), except that this routine can return before a condition variable is signaled or broadcast, specifically, when the specified time expires. For more information, see the tis_cond_wait() description.

This routine atomically releases the mutex and causes the calling thread to wait on the condition. When the thread regains control after calling tis_cond_timedwait(), the mutex is locked and the thread is the owner. This is true regardless of why the wait ended. If general cancelability is enabled, the thread reacquires the mutex (blocking for it if necessary) before the cleanup handlers are run (or before the exception is raised).

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 have locked the mutex specified in mutex. The results of this routine are unpredictable if this routine is called without first locking the mutex. The only routines that are supported for use with asynchronous cancelability enabled are those that disable asynchronous cancelability.

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 tis_cond_timedwait() operations or 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.
[ETIMEDOUT]	The time specified by abstime expired.
[ENOMEM]	The Threads Library cannot acquire memory needed to block using a statically initialized condition variable.

Associated Routines

```
tis_cond_broadcast()
tis_cond_destroy()
tis_cond_init()
tis_cond_signal()
tis_cond_wait()
tis_get_expiration()
```

tis_cond_wait

Causes a thread to wait for the specified condition variable to be signaled or broadcast.

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 broadcast.

Calling this routine in a single-threaded environment is a coding error. Because no other thread exists 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 is not a valid condition variable or the value specified by mutex is not a valid mutex, 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_get_expiration

Obtains a value representing a desired expiration time.

Syntax

tis_get_expiration(
         delta,
         abstime);

Argument	Data Type	Access
delta	struct timespec	read
abstime	struct timespec	write

C Binding

#include <tis.h>

int
tis_get_expiration (
         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).

Description

If threads are not present, this routine has no effect.

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 tis_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

```
tis_cond_timedwait()
```

tis_io_complete

AST completion routine to VMS I/O system services.
This routine is for OpenVMS systems only.

Syntax

tis_io_complete( );

C Binding

#include <tis.h>

int
tis_io_complete (void);

Description

When you are performing thread-synchronous "wait-form" system service calls
on OpenVMS such as $QIOW, $ENQW, $GETJPIW, and so on, you should use this
routine and tis_sync() with the asynchronous form of the service (in other
words, without the "W"), and specify the address of tis_io_complete() as the
completion AST routine (the AST argument if any is ignored). That must also
specify an IOSB (or equivalent, such as an LKSB) and if possible a unique event
flag (see lib$get_ef). Once the library code is ready to wait for the I/O, it simply
calls tis_sync() (just as if it were calling $SYNC).

Return Values

None.

Associated Routines

tis_sync()

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 obtain 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.

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.

tis_key_create

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.

The Threads Library 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.
[EINVAL]	The value specified by key is invalid.
[ENOMEM]	Insufficient memory to create the key.

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. If the cleanup is done after this routine is called, the application must have a private mechanism to access any and all thread-specific values, contexts, and so on.

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.

tis_key_delete

Associated Routines

```
tis_getspecific()
tis_key_create()
tis_setspecific()
```

tis_lock_global

Locks the Threads Library global mutex.

Syntax

tis_lock_global( );

C Binding

#include <tis.h>

int
tis_lock_global (void);

Arguments

None

Description

This routine locks the 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, the Threads Library 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 either 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 not a valid mutex.
[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 Threads Library 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.

Your program can use the PTHREAD_MUTEX_INITIALIZER macro to statically initialize a mutex object without calling this routine. Static initialization can be used only for a condition variable with storage class "extern" or "static" — "automatic" (stack local) objects must be initialized by calling tis_mutex_init(). 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. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize a mutex.
[EBUSY]	The implementation has detected an attempt to reinitialize mutex (a previously initialized, but not yet destroyed, mutex).

Return	Description
[EINVAL]	The value specified by mutex is not a valid mutex.
[ENOMEM]	Insufficient memory to initialize the mutex.
[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. (The Threads Library may 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.
[EDEADLK]	A deadlock condition is detected.
[EINVAL]	The value specified by mutex is not a valid mutex.

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 not a valid mutex.

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 not a valid mutex.
[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. Any one-time initialization routine in static storage specified by once_control 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.

_____ **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.

_____

tis_once

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 simply 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.

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 not a valid read-write lock.

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.
[EINVAL]	The value specified by lock is not a valid read-write lock.

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 not a valid read-write lock.

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.
[EINVAL]	The value specified by lock is not a valid read-write lock.

tis_rwlock_destroy

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.

Note

The **tis** read-write lock has no relationship to the Single UNIX Specification, Version 2 (SUSV2, or UNIX98) read-write lock routines (such as `pthread_rwlock_init()`). The `tis_rwlock_t` type, in particular, cannot be used with the **pthread** read-write lock functions, nor can a `pthread_rwlock_t` type be used with the **tis** read-write lock functions.

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 not a valid read-write lock.
[ENOMEM]	Insufficient memory to initialize lock.

tis_rwlock_init

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 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 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. Doing so could lead to a memory leak or an infinite loop.

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 key is not a valid key.
[ENOMEM]	Insufficient memory to associate the value with the key.

Associated Routines

```
tis_getspecific()
tis_key_create()
tis_key_delete()
```

tis_sync

Used as the synchronization point for asynchronous I/O system services. This routine is for OpenVMS systems only.

Syntax

tis_sync(
    efn,
    iosb);

Argument	Data Type	Access
efn	unsigned long	read
iosb	void *	read

C Binding

#include <tis.h>

int
tis_sync (
    unsigned long   *efn,*
    void   *iosb);

Arguments

efn
The event flag specified with the OpenVMS system service routine.

iosb
The IOSB specified with the OpenVMS system service routine.

Description

When you are performing thread-synchronous "wait-form" system service calls on OpenVMS such as `$QIOW`, `$ENQW`, `$GETJPIW`, and so on, you should use this routine and `tis_io_complete()` with the asynchronous form of the service (that is, without the "W") and specify the address of `tis_io_complete()` as the completion AST routine (the AST argument, if any, is ignored). The call must also specify an IOSB (or equivalent, such as an LKSB) and if possible a unique event flag (see `lib$get_ef`). Once the library code is ready to wait for the I/O, it simply calls `tis_sync()` (just as if it were calling `$SYNC`).

Return Values

This routine has the same return values as the OpenVMS `$SYNC()` routine.

Associated Routines

    `tis_io_complete()`

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 Threads Library global mutex.

Syntax

tis_unlock_global( );

C Binding

#include <tis.h>

int
tis_unlock_global (void);

Arguments

None

Description

This routine unlocks the 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 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_write_lock (
        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 not a valid read-write lock.

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.
[EINVAL]	The value specified by lock is not a valid read-write lock.

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:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by lock is not a valid read-write lock.

Associated Routines

```
tis_read_lock()
tis_read_trylock()
tis_read_unlock()
tis_rwlock_init()
tis_rwlock_destroy()
tis_write_lock()
tis_write_trylock()
```

tis_yield

Notifies the scheduler that the current thread is willing to release its processor to other threads of the same or higher priority.

Syntax

tis_yield( );

C Binding

int
tis_yield (void);

Arguments

None

Description

When threads are not present, this routine has no effect.

This routine notifies the thread scheduler that the current thread is willing to release its processor to other threads of equivalent or greater scheduling precedence. (A thread generally will release its processor to a thread of a greater scheduling precedence without calling this routine.) If no other threads of equivalent or greater scheduling precedence are ready to execute, the thread continues.

This routine can allow knowledge of the details of an application to be used to improve its performance. If a thread does not call tis_yield(), other threads may be given the opportunity to run at arbitrary points (possibly even when the interrupted thread holds a required resource). By making strategic calls to tis_yield(), other threads can be given the opportunity to run when the resources are free. This improves performance by reducing contention for the resource.

As a general guideline, consider calling this routine after a thread has released a resource (such as a mutex) which is heavily contended for by other threads. This can be especially important if the program is running on a uniprocessor machine, or if the thread acquires and releases the resource inside a tight loop.

Use this routine carefully and sparingly, because misuse can cause unnecessary context switching that will increase overhead and actually degrade performance. For example, it is counter-productive for a thread to yield while it holds a resource that the threads to which it is yielding will need. Likewise, it is pointless to yield unless there is likely to be another thread that is ready to run.

tis_yield

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]	The routine `tis_yield()` is not supported by this implementation.

Part IV

Appendixes

Part IV contains appendixes that provide supporting information about the POSIX Threads Library, such as operating system-specific information, debugging information, and additional reference information.

A

Considerations for Tru64 UNIX Systems

This appendix discusses Threads Library issues specific to Tru64 UNIX systems.

A.1 Overview

The Tru64 UNIX operating system supports multiple concurrent "execution contexts" within a process. The Threads Library 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 Threaded Applications

The following sections discuss points to consider when building using the Threads Library.

A.2.1 Including Threads Header Files

Include one of the Threads Library header files shown in Table A–1 in your program to use the appropriate Threads library.

Table A–1 Header Files

Header File	Interface
pthread.h	POSIX 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 Threads Libraries

Multithreaded applications are built using shared libraries. For a description of shared libraries, see the Tru64 UNIX Programmer's Guide.

Table A–2 contains the libraries supported for multithreaded programming.

Table A–2 Tru64 UNIX Shared Libraries for Multithreaded Programs

libpthreads.so	Shared version of Threads Library "legacy" package, implementing the Compaq-proprietary CMA (or **cma**) and POSIX 1003.4a/Draft 4 (**d4** or **DCE threads**) interfaces.
libpthread.so	Shared version of the POSIX threads package. Requires libexc.so and libc.so

Table A–2 (Cont.)   Tru64 UNIX Shared Libraries for Multithreaded Programs

`libexc.so`	Shared version of Tru64 UNIX exception support package.
`libc.so`	Shared version of the C language run-time library (`libc.so`).

Build a multithreaded application using shared versions of `libexc`, `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 provide the `–D_REENTRANT` compilation switch (or equivalent) at compilation, and link as shown in Section A.2.3.

A.2.3 Linking Multithreaded Shared Libraries

The `ld` command does not support the `–pthread` or `–threads` switch. Normally, programs can be compiled and linked from the `cc` command. If you must link using the `ld` command, you must list the shared libraries in the proper order. The `libc` library should be the last library referenced, `libexc` should immediately precede `libc`, and the thread libraries should precede `libexc`.

For libraries that use only the **pthread** interface, use the following:

```
ld <...> –lpthread –lexc –lc
```

If using the **cma** or **d4** interfaces, use the following:

```
ld <...> –lpthreads –lpthread –lexc –lc
```

Also, `cc –pthread` (or `cc –threads`) causes the compiler to replace any libraries that have special thread-safe alternatives. These libraries have the same name ending in `–r`. For example, `cc –pthread –o foo –lbar`, if there is a `libbar.so` and `libbar_r.so`, would use the latter. When linking with the `ld` command, you must perform that search and replacement yourself.

Note

If you build software (whether applications or libraries) that links against the static version of a Threads library, you must not require developers who use your software to link against any library that dynamically loads any Threads shared library, such as `libpthread.so`.

A.2.4 Compiling Applications With the tis Interface

Applications that use the Compaq-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 Tru64 UNIX Systems

Under Tru64 UNIX Version 4.0 and later, the Threads Library implements a **two-level scheduling** model. The thread library schedules "user threads" onto kernel execution contexts (often known as "kernel threads" or "virtual processors"), just as Tru64 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, the Threads Library schedules a new user thread to run. While the Threads Library is scheduling user threads onto kernel threads, the Tru64 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 the Threads Library and the Tru64 UNIX kernel. When a thread blocks in the kernel, the Threads Library 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 Use of Kernel Threads

Tru64 UNIX kernel threads are created as they are needed by the application. The number of kernel threads that the Threads Library creates is limited by normal Tru64 UNIX configuration limits regarding user and system thread creation. Normally, however, the Threads Library creates one kernel thread for each actual processor on the system, plus a "manager thread" for bookkeeping operations.

The Threads Library 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. (These idled kernel threads are deleted by the kernel if they remain idle for a long time.) When the process terminates, all kernel threads in the process are reclaimed by the kernel.

The Threads Library 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 (in "native" $threadlevel) can change at any time.

A.3.2 Support for Realtime Scheduling

The Threads Library supports Tru64 UNIX realtime 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.1 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 maximum priority 63 will not preempt a thread in another process running with `SCHED_FIFO` and lower priority.

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 *Tru64 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 Tru64 UNIX Version 4.0, all threads used only system contention scope. In Tru64 UNIX Version 4.0, all threads created using the **pthread** interface, by default, have process contention scope.

A.4 Thread Cancelability of System Services

Tru64 UNIX supports the required system cancelation points specified by the POSIX.1 standard and by the Single UNIX Specification, Version 2 (UNIX98).

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 **pthread** interface.

_____ **Note** _____

It is not legal, or supported, to call any Tru64 UNIX system function with asynchronous cancelability type. You cannot "work around" the lack of system call cancelation using asynchronous cancelability.

_____

For more information, see Section 2.3.7.

A.4.1 Cancelation Points

The following functions are cancelation points (as defined by the Single UNIX Specification, Version 2 (SUSV2)):

```
accept()                        send()
aio_suspend()                   sendmsg()
close()                         sendto()
connect()                       shutdown()
creat()                         sigpause()
fcntl() (for cmd F_SETLKW)      sigsuspend()
fsync()                         sigtimedwait()
getmsg()                        sigwait()
getpmsg()                       sigwaitinfo()
lockf()                         sleep()
mq_receive()                    system()
mq_send()                       tcdrain()
msgrcv()                        t_close()
msgsnd()                        t_connect()
msync()                         t_listen()
nanosleep()                     t_rcv()
open()                          t_rcvconnect()
pause()                         t_rcvrel()
poll()                          t_rcvreldata()
pread()                         t_rcvudata()
pthread_cond_timedwait()        t_rcvv()
pthread_cond_wait()             t_rcvvudata()
pthread_delay_np()              t_snd()
pthread_join()                  t_sndrel()
pthread_testcancel()            t_sndreldata()
putmsg()                        t_sndudata()
putpmsg()                       t_sndv
pwrite()                        t_sndvudata()
read()                          usleep()
readv()                         wait()
recv()                          wait3()
recvfrom()                      waitid()
recvmsg()                       waitpid()
select()                        write()
sem_wait()                      writev()
```

A.4.2 Conditional or Future Cancelation Points

These functions may not cause delivery of a pending cancel, and cancelation may not interrupt a blocking state. Some will recognize cancelation only under some conditions (for example, if `printf()` flushes a standard I/O buffer to the file stream). Others may currently not be coded to recognize cancelation, but may be changed in the future. All code should be prepared to handle cancelation at these calls, but must not depend on cancelation at these calls.

```
closedir()                   getchar()                perror()
closelog()                   getchar_unlocked()       popen()
ctermid()                    getcwd()                 printf()
dbm_close()                  getdate()                putc()
dbm_delete()                 getgrent()               putc_unlocked()
dbm_fetch()                  getgrgid()               putchar()
dbm_nextkey()                getgrgid_r()             putchar_unlocked()
dbm_open()                   getgrnam()
dbm_store()                  getgrnam_r()             puts()
dlclose()                    gethostbyaddr()          pututxline()
dlopen()                     gethostbyname()          putw()
endgrent()                   gethostent()             putwc()
endhostent()                 gethostname()            putwchar()
endnetent()                  getlogin()               readdir()
endprotoent()                getlogin_r()             readdir_r()
endpwent()                   getnetbyaddr()           remove()
endservent()                 getnetbyname()           rename()
endutxent()                  getnetent()              rewind()
fclose()                     getprotobynumber()       rewinddir()
fcntl()  (for any cmd)       getprotobyname()         scanf()
fflush()                     getpwent()               seekdir()
fgetc()                      getpwnam()               semop()
fgetpos()                    getpwnam_r()             setgrent()
fgets()                      getpwuid()               sethostent()
fgetwc()                     getpwuid_r()             setnetent()
fgetws()                     gets()                   setprotoent()
fopen()                      getservbyname()          setpwent()
fprintf()                    getservbyport()          setservent()
fputc()                      getservent()             setutxent()
fputs()                      getutxent()              strerror()
fputwc()                     getutxid()               syslog()
fputws()                     getutxline()             tmpfile()
fread()                      getw()                   tmpname()
freopen()                    getwc()                  ttyname()
fscanf()                     getwchar()               ttyname_r()
fseek()                      getwd()                  ungetc()
fseeko()                     glob()                   ungetwc()
fsetpos()                    iconv_close()            unlink()
ftell()                      iconv_open()             vfprintf()
ftello()                     ioctl()                  vfwprintf()
ftw()                        lseek()                  vprintf()
fwprintf()                   mkstemp()                vwprintf()
fwrite()                     nftw()                   wprintf()
fwscanf()                    opendir()                wscanf()
getc()                       openlog()
getc_unlocked()              pclose()
```

Note that appropriate non-standard functions that do not appear in the preceding list might become cancelation points in the future. Tru64 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.1 standard.

Tru64 UNIX Version 4.0 introduced the full POSIX.1 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 Tru64 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 Tru64 UNIX Version 4.0 and later, all signal actions are processwide. That is, when any thread uses sigaction or equivalent to either 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 Tru64 UNIX Version 4.0, asynchronous signals were processed only in the main thread. In Tru64 UNIX Version 4.0, any thread that does not have the signal masked can process the signal.

_____ **Note** _____

To support binary compatibility, for a thread created by a **cma** or **d4** interface routine, the thread starts with all asynchronous signals blocked.

_____

A.5.1 POSIX sigwait Service

The POSIX 1003.1 `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 with threads. For example, techniques that rely on timer and I/O signals are usually more complicated and errorprone than techniques that rely on simply waiting synchronously within a thread. Furthermore, most of the thread 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.1 standard requires that the thread block the signals for which it will wait before calling `sigwait()`. For reliable operation, the signals should be blocked in all threads. Otherwise, the signal might be delivered to another thread before the `sigwait` thread calls `sigwait()`, or after it has returned with another signal.

A.5.2 Handling Synchronous Signals as Exceptions

For the signals traditionally representing synchronous errors in the program, the Threads Library 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 Tru64 UNIX signals that are reported as 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`
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, the Threads Library potentially rounds up the value specified in the object's guardsize attribute. The Threads Library 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 the number of bytes equal to the setting of `PAGESIZE`.

A.7 Thread Stack and Backing Store Allocation

Starting in Version 5.0, for threads that accept the default stack address attribute, the Threads Library allocates a thread's writable stack area from uncommitted virtual memory, then commits predefined increments of the writable stack area to the thread only as it is needed. The stack's corresponding backing store is also reserved incrementally as the stack is committed. In this way, no more backing store is reserved than the stack actually requires.

Because Tru64 UNIX 5.0 does not commit backing store (or physical pages) for stacks until the pages are used by the program, the default stack size has been increased. The previous default of about 24Kb (3 pages) has been increased to 5Mb.

A.8 Dynamic Activation

Dynamic activation of the Threads Library run-time environment, or of code that depends on the Threads Library, is currently not supported.

A.9 Pagefaults and Realtime Scheduling

Like normal file I/O operations, pagefaults are "thread synchronous". A thread that incurs a "hard" pagefault (reading the page from backing store) will be blocked while other threads continue to run on the "virtual processor" (or on others). This has implications for realtime scheduling, especially of SCHED_FIFO policy threads, that do not expect to block except for explicit I/O synchronization. To write a SCHED_FIFO thread that cannot block unexpectedly, you must use mlockall to lock the application into memory, preventing pagefaults.

B

Considerations for OpenVMS Systems

This appendix discusses POSIX Threads Library issues and restrictions specific to the OpenVMS operating system.

B.1 Overview

The OpenVMS Alpha operating system supports multiple concurrent "execution contexts" within a process. The Threads Library 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. Even without kernel threads, OpenVMS Alpha "upcalls" support smooth integration between the Threads Library and kernel scheduler. See Section B.12 for more information, including how to enable kernel threads and upcalls in your application. OpenVMS VAX supports neither kernel threads nor upcalls.

B.2 Compiling Under OpenVMS

The C language header files shown in Table B–1 provide interface definitions for the **pthread** and **tis** interfaces.

Table B–1 Header Files

Header File	Interface
pthread.h	POSIX.1 style routines
tis.h	Compaq 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

The Threads Library is supplied only as shareable images. It is not supplied as object libraries.

When you link an image that calls Threads Library routines, you must link against the appropriate images listed in Table B–2.

Table B–2 Threads Library Images

Image	Routine Library
PTHREAD$RTL.EXE	POSIX.1 style interface
CMA$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 uses the CMA$OPEN_LIB_SHR.EXE and CMA$OPEN_RTL.EXE images, you must specify them in a Linker options file.

Note

While this version of the POSIX Threads Library for OpenVMS supports upward compatibility of source and binaries for the **d4** interface, it 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 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 the Threads Library 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 the Threads Library:

- 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 Threads Library functions that may block the thread while it has disabled ASTs.

- Do not call Threads Library routines, except those that have the _int (interrupt) suffix in their names, from within an AST routine. Calling any other Threads Library routines from code running in an AST can be unreliable or cause unexpected behavior.

- For OpenVMS Alpha, ASTs are handed off to the Threads Library 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.

- For synchronous I/O completion, use the asynchronous variant of the system service routine and the `tis_io_complete()` and `tis_sync()` routines.

B.5 Dynamic Activation

Certain run-time libraries use conditional synchronization mechanisms. These mechanisms typically are enabled during image initialization when the run-time library is loaded, and only if the process is multithreaded (that is, if the core run-time library PTHREAD$RTL has been linked in). If the process is not multithreaded, the synchronization is disabled.

If your application were to dynamically activate PTHREAD$RTL, any run-time library that uses conditional synchronization may not behave reliably. Thus, dynamic activation of the core run-time library PTHREAD$RTL is not supported.

If your application must dynamically activate an image that depends upon PTHREAD$RTL (that is, the image must run, or can be run, in a multithreaded environment), you must build the application by explicitly linking the image calling LIB$FIND_IMAGE_SYMBOL against PTHREAD$RTL.

Use the OpenVMS command ANALYZE/IMAGE to determine whether an image depends upon PTHREAD$RTL. For more information see your OpenVMS documentation.

Libraries that wish to use thread-safe synchronization only when threads are present should use the **tis** functions instead of dynamically activating `PTHREAD$RTL.EXE`.

B.6 Default and Minimum Thread Stack Size

As of OpenVMS Version 7.2, the Threads Library has increased the default thread stack size for both OpenVMS Alpha and OpenVMS VAX. Applications that create threads using the default stack size (or a size calculated from the default) will be unaffected by this change.

As of OpenVMS Version 7.2, the Threads Library has increased the minimum thread stack size (based on the `PTHREAD_STACK_MIN` constant) for OpenVMS VAX only. Existing applications that were built using a version prior to OpenVMS Version 7.2 and that base their thread stack sizes on this minimum must be recompiled.

B.7 Requesting a Specific, Absolute Thread Stack Size

Prior to OpenVMS Version 7.2, when an application requested to allocate a thread stack of a specific, absolute size, the Threads Library would increase the size by a certain quantity, then round up that sum to an integral number of pages. This process resulted in the actual stack size being considerably larger than the caller's request, possibly by more than one page.

Starting with OpenVMS Version 7.2, when an application requests the Threads Library to allocate a thread stack of a specific, absolute size, no additional space is added, but the allocation is still rounded up to an integral number of pages.

Any application that uses default-sized stacks is unlikely to experience problems due to this change. Similarly, any application that sets its thread stack allocations in terms of either the default or the allowable minimum stack size is unlikely to experience problems due to this change; however, depending on the allocation calculation used, the application might receive more memory for thread stacks.

Starting with OpenVMS Version 7.2, any thread that is created with a stack allocation of a specific, absolute size might fail during execution because of insufficient stack space. This failure indicates an existing bug in the application that was made manifest by the change in the Threads Library.

When the application requests to allocate a thread stack of a specific size, it must allow for not only the space that the application itself requires, but also sufficient stack space for context switches and other activity. The Threads Library only occasionally uses this additional stack space, such as during timeslice interruptions. A thread with inadequate stack space might not encounter problems during development and testing because of timing vagaries—for instance, a thread might not experience problems until a timeslice occurs while the thread is at its maximum stack utilization—and this situation might never arise during in-house testing. In a different system environment, such as in a production environment, the timing might be different, possibly resulting in occasional failures when certain conditions are met.

B.8 Declaring an OpenVMS Condition Handler

This section discusses a restriction on declaring an OpenVMS condition handler while using exceptions, and 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 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, exceptions will not be handled correctly until the next TRY statement is executed. The TRY statement restores the 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, the Threads Library saves the previous handler address. When the Threads Library receives a condition it does not handle

(including SS$_UNWIND, SS$_DEBUG, or a condition code that does not have a SEVERE severity), it invokes the saved condition handler. The condition handler will be reestablished when the `TRY` block exits.

B.9 Thread Cancelability of System Services

On OpenVMS Alpha, system calls are not cancelation points for threads created using the POSIX 1003.1 style interface. System calls are not cancelation points for threads in legacy multithreaded applications that were created using the Compaq proprietary CMA (or **cma**) or POSIX 1003.4a/Draft 4 (**d4** or DCE *threads*) interfaces. None of the system calls should be called with asynchronous cancelation enabled. For more information, see Section 2.3.7.

B.10 Using OpenVMS Alpha 64-Bit Addressing

On OpenVMS Alpha, the Threads Library 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 Threads Library functions have special 64-bit versions because the OpenVMS Alpha calling standard always supports 64-bit arguments and return values.

B.11 Condition Values

Table B–3 lists the condition values for OpenVMS systems and provides an explanation and user action.

Table B–3 Condition Values

Condition Value	Explanation and User Action
CMA$_EXCCOP	Exception raised, OpenVMS condition code follows.
	Explanation: One of the exception commands (RAISE or RERAISE) raised or reraised an exception condition originating outside the Threads Library. The secondary condition code in the signal vector will be the original code.
	User Action: See the documentation for the software that your program is calling to determine the reason for this exception.
CMA$_EXCCOPLOS	Exception raised, some information lost.

Table B–3 (Cont.) Condition Values

Condition Value	Explanation and User Action
	Explanation: CMA$_EXCCOPLOS is nearly the same as CMA$_EXCCOP except that the Threads Library 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, the Threads Library clears the condition codes' arguments in the resignaled vector. In most cases, the Threads Library 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 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 retrieving message text, and it cannot be interpreted by other facilities.
	User Action: None.

B.12 Two-Level Scheduling on OpenVMS Alpha Systems

This section applies to OpenVMS Alpha systems only.

Under OpenVMS Alpha Version 7.0 and later, the Threads Library implements a two-level scheduling model. 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.

The Threads Library schedules threads onto virtual processors similar to the way that OpenVMS schedules processes onto the processors of a multiprocessing machine. Thus, to the runtime environment, a scheduled thread is executed on a virtual processor until it blocks or until it exhausts its timeslice quantum; then the Threads Library schedules a new thread to run.

While the Threads Library 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 the Threads Library 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 Threads Library 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 Threads Library scheduler to notify it of the change in the thread's status.

This upcall gives the Threads Library 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 the Threads Library that a previously blocked thread is now eligible to run again. The Threads Library schedules that thread to run based on its scheduling policy and priority.

B.12.1 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 whether the process can create multiple kernel threads and whether the OpenVMS Alpha kernel's support for 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. This qualifier must be applied to a "main" image. If used on a shared library image, it will be ignored.

The use of kernel threads and upcalls is also limited by the kernel sysgen parameter MULTITHREAD. If set to 0, no process may use upcalls or create kernel threads. A value of 1 allows upcalls, but not kernel threads. A higher value represents the maximum number of kernel threads each process may use. (You cannot have multiple kernel threads without upcalls.)

Table B–4 Results of Keyword Arguments to /THREADS_ENABLE Qualifier

Keywords Specified	Result
/NOTHREADS_ENABLE	No kernel threads support
/THREADS_ENABLE or: /THREADS_ENABLE=(MULTIPLE_ KERNEL_THREADS,UPCALLS)	Full kernel threads support, including the ability to run multiple user threads simultaneously on different CPUs on a multiprocessor machine

(continued on next page)

Table B–4 (Cont.)   Results of Keyword Arguments to /THREADS_ENABLE Qualifier

Keywords Specified	Result
/THREADS_ENABLE=MULTIPLE_KERNEL_THREADS	Same behavior as if /NOTHREADS_ENABLE is specified (without support for upcalls, the Threads Library cannot reliably use multiple kernel threads)
/THREADS_ENABLE=UPCALLS	Upcall support (such as making system calls thread-synchronous), but restricts the process' threads to one CPU on a multiprocessor machine

_____ **Note** _____

Under no circumstances should a process explicitly create kernel threads. The Threads Library creates them as needed when allowed to do so. Explicit creation of kernel threads by an application disrupts the operation of the runtime environment and causes incorrect and/or unreliable application behavior.

_____

B.12.2 Setting Kernel Threads Support in Existing Images

Under OpenVMS Alpha only, use the THREADCP tool to set or show the kernel threads features described earlier for an existing main 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, as shown in Section B.12.2.1.

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.12.2.1 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.12.3 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 upcalls and the creation of multiple kernel threads are disabled.
1	Upcalls are enabled; the creation of multiple kernel threads is disabled.
2 through 16	Both 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.12.4 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 the Threads Library 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, the Threads Library creates no more virtual processors than there are user threads (excluding internal threads).

The Threads Library 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 Threads Library 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.12.5 Delivery of ASTs

When a user mode AST becomes deliverable to a Threads Library process, the OpenVMS scheduler makes an upcall to the Threads Library, passing the information that is required to deliver the AST (service routine address, argument, and target user thread ID). The Threads Library 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 either 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 all OpenVMS releases to date, 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 the Threads Library 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 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 a thread can 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.12.6 Blocking System Services

In OpenVMS Alpha Version 7.0 and later, with few exceptions a blocking system service call is thread synchronous—that is, only the calling thread is blocked. The exceptions are services that do not block in user mode and services that set common event flags. (See also Section B.12.8.)

When a thread calls a system service that must block, the OpenVMS scheduler makes an upcall to allow the Threads Library 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 the Threads Library 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.12.7 $HIBER and $WAKE

$HIBER and $WAKE result in upcalls to the Threads Library. 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. (The `sleep()` routine does

this deliberately to mimic the ANSI C required behavior of returning when interrupted by a signal. Though OpenVMS does not have UNIX signals, an asynchronous $WAKE is similar in intent.)

B.12.8 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.

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.

A Threads Library process is rarely in LEF state. In general, instead of blocking for an event flag wait, the Threads Library schedules another thread to be run. However, if no threads are available, the Threads Library schedules a "null" thread, which places the virtual processor in HIB state until it is needed to execute a thread.

_____ **Note** _____

If a thread calls a system service that uses a common event flag, the calling thread's virtual processor blocks until the wait is satisfied. (That is, no upcall is made to the OpenVMS kernel to schedule another thread.) On a uniprocessor, such a system service call will most likely cause all threads in the process to block.

_____

B.12.9 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 the Threads Library takes place, and the Threads Library places the thread in a blocked state. The Threads Library schedules another thread to run in its place.

  When the pagefault resolution is complete, another upcall occurs, and the Threads Library 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.

- Threads Library timeslicing changed slightly for OpenVMS Alpha Version 7.0 and later. Prior to Version 7.0, the Threads Library timeslicer was implemented using an OpenVMS timer. This caused a Threads Library 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 the Threads Library 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.12.10 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 the Threads Library 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 the Threads Library 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.12.11 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 the Threads Library reverts to its behavior prior to OpenVMS Version 7.0—that is, no upcalls can occur, and it does not use all processors in multiprocessor systems.

B.12.12 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.12.12.1 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, such as when at a breakpoint in the debugger, to see all but one of your kernel threads in SUSP state. This effect is a part of the debugging support and is not the result of calling $SUSPND.

B.12.12.2 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.12.12.3 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.12.4 for more information.

B.13 Interoperability with POSIX for OpenVMS

Previous releases of the POSIX for OpenVMS layered product had very limited interoperability with the Threads Library. Under OpenVMS Version 7.0 and later, using the Threads Library with the POSIX for OpenVMS layered product is not supported.

C

Debugging Multithreaded Applications

The debugging information in this appendix is specific to applications that use the POSIX Threads Library.

C.1 Using PTHREAD_CONFIG

During initialization of the Threads Library run-time environment, the PTHREAD_CONFIG environment variable (on Tru64 UNIX 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 Threads Library application.

C.1.1 Major and Minor Keywords

As summarized in Table C–1, PTHREAD_CONFIG takes "major keywords" as arguments. Use a "minor keyword" to specify a value for each major keyword.

Table C–1 PTHREAD_CONFIG Settings

Major keyword	Minor keyword	Meaning
dump=	file-path	Path of bugcheck file (OpenVMS only)
meter=	condition	Meter condition variable operations
	mutex	Meter mutex operations
	stack	Record thread greatest stack extent
	all	Meter all available operations
	none	No metering
width=	bugcheck_output_width	Width of output from bugcheck output

C.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 DCL under OpenVMS, you can set PTHREAD_CONFIG as follows:

```
$ define PTHREAD_CONFIG "meter=(stack,mutex);dump=/tmp/dump-d.dmp;width=132"
```

C.2 Running in Metered Mode

Metering tells the Threads Library 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 the Threads Library 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 the Threads Library to gather and record statistics and history information for all synchronization operations.

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.

C.3 Visual Threads

We recommend Visual Threads to debug Threads Library applications on Tru64 UNIX and OpenVMS systems. Visual Threads can be used to automatically diagnose common problems associated with multithreading, including deadlock, protection of shared data (on Tru64 UNIX systems only), and thread usage errors. It can also be used to monitor the thread-related performance of an application, helping you to identify bottlenecks or locking granularity problems. It is a unique debugging tool because it can be used to identify problem areas even if an application does not show any specific symptoms.

See the online Visual Threads documentation at `www.compaq.com/visualthreads/` for more information.

C.4 Using Ladebug on Tru64 UNIX Systems

The Compaq Ladebug debugger provides commands to display the state of threads, mutexes, and condition variables.

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 ...] — Displays mutex state.
- `show condition` [n ...] — Displays condition variable state.

Refer to the Ladebug documentation for further details.

C.5 Debugging Threads on OpenVMS Systems

This section presents particular topics that relate to debugging a multithreaded application under OpenVMS.

C.5.1 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 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

Migrating from the cma Interface

This appendix presents information that helps you migrate existing programs and applications that use the Compaq proprietary CMA (or **cma**) interface to use the **pthread** interface, based on the IEEE POSIX 1003.1c-1995 standard.

Note

In future 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 **pthread** interface is the most portable, efficient, and robust multithreading run-time library offered by Compaq.

D.1 Overview

The **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 the POSIX Threads Library.

You should include the C language `pthread.h` header file for prototypes of the **pthread** routines.

D.2 cma Handles

A **cma handle** is storage, similar to a pointer, that refers to a specific Threads Library 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 the Threads Library.

In the **cma** interface, because objects are accessed only by handles, you can think of the handle as if it were the object itself. Threads Library objects are accessed by handles (rather than pointers), because handles allow for greater robustness and portability. Handles allow the Threads Library 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 **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.)

D.3 Interface Routine Mapping

As summarized in Table D–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 D–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()	
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()	

Table D–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()	
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_np()	

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).

D.4 New pthread Routines

The following are **pthread** interface routines that have no functional similarities in the **cma** interface:

```
pthread_atfork() (Tru64 UNIX only)
pthread_attr_getdetachstate()
pthread_attr_getscope()
pthread_attr_setdetachstate()
pthread_attr_setscope()
pthread_condattr_getpshared()
pthread_condattr_setpshared()
pthread_key_delete()
pthread_kill() (Tru64 UNIX only)
pthread_mutexattr_getpshared()
pthread_mutexattr_setpshared()
All pthread_rwlockattr_ and pthread_rwlock_ routines
pthread_sigmask() (Tru64 UNIX only)
sigwait()
```

E

Migrating from the d4 Interface

This appendix provides migration information for the routines in the POSIX 1003.4a/Draft 4 (or **d4**) interface.

Note

Applications that use the **d4** routines require significant modification to be migrated to the **pthread** interface described in Part II.

E.1 Overview

Routines in the **pthread** interface differ significantly from the original POSIX 1003.4a/Draft 4 implementation. This section describes the major changes between the interfaces.

E.2 Error Status and Function Returns

The **pthread** interface does not use the global variable *errno*. (Note that the Threads Library provides a thread-specific *errno* for use by libraries and application code, but the **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 **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).

E.3 Replaced or Renamed Routines

Many routines in the **d4** interface have been replaced or renamed in the **pthread** interface, as shown in Table E–1.

Table E–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/getguardsize_np()`	`pthread_attr_set/getguardsize()`

(continued on next page)

Table E–1 (Cont.) pthread Routines That Replace d4 Routines

d4 Routine	Replacement pthread Routine
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()
pthread_setasynccancel()	pthread_setcanceltype()
pthread_setcancel()	pthread_setcancelstate()
pthread_set/getprio()	pthread_set/getschedparam()
pthread_set/getscheduler()	pthread_set/getschedparam()
pthread_yield()	sched_yield()

E.4 Routines with No Changes to Syntax

Except for the return value, the following routines in the **d4** interface have no changes to syntax in the **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).

E.5 Routines with Prototype or Syntax Changes

Table E–2 shows the routines in the **d4** interface that have changes to their argument syntax in the **pthread** interface.

Table E–2   d4 Routines With Syntax Changes as pthread Routines

Old Syntax	New Syntax

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*)

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 E–3 shows routines in the **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 the Threads Library.

If your original code used the standard Threads Library datatypes, then this migration requirement might not impact your code.

Table E–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_)

E.6 New Routines

The following are routines in the **pthread** interface that did not exist at the time of the implementation of the **d4** interface:

```
pthread_atfork() (Tru64 UNIX only)
pthread_attr_getscope()
pthread_attr_setscope()
pthread_key_delete()
pthread_kill() (Tru64 UNIX only)
```
All pthread_rwlockattr_ and pthread_rwlock_ routines
```
pthread_sigmask() (Tru64 UNIX only)
```
All _getpshared and _setpshared routines

Glossary

actual granularity

Granularity for a program; limited by the granularities made available by the processor, but determined by the code produced by the compiler. See also granularity, natural granularity, and system granularity.

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 Threads Library 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 Threads Library 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 cancelation) 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 Threads Library 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 the Threads Library 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.

granularity

Smallest unit of storage (that is, bytes, words, longwords, or quadwords) that a computer can load or store in one machine instruction. See also actual granularity, natural granularity, and system granularity.

guard area

Area at the overflow end of a thread's writable stack and the stack overflow warning area that is inaccessible to the thread. If the thread attempts to access a memory location within the guard area, a memory addressing violation occurs. See also overflow warning area.

guard pages

Low-level memory regions that form a stack guard region.

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 Threads Library object.

inherit scheduling attribute

Attribute of a thread that specifies whether a newly created thread inherits the scheduling attributes (scheduling priority, policy and contention scope) 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.

lock acquisition

Each instance of acquiring a mutex or read-write lock.

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.

natural granularity

Granularity of a processor; determined by the processor's architecture, cache architecture, and instruction set. See also actual granularity, granularity, and system granularity.

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 the Threads Library, this kind of mutex offers the best performance. See also mutex, errorcheck mutex, and recursive mutex.

overflow warning area

Area between the overflow end of the thread's writable stack and the stack guard area. If the thread attempts to access a memory location within the overflow warning area, a stack overflow exception occurs. The program can catch this exception and continue processing. See also guard area.

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.

reentrant

Refers to a routine that functions normally despite being called simultaneously or sequentially in different threads.

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 Threads Library scheduling algorithm that, in combination, determine which thread will be allowed to run when a scheduling decision is made. Scheduling decisions are made either 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 broadcast), 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 Tru64 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.

system granularity

Granularity provided by an operating system's run-time libraries, to provide a consistent and coherent environment for applications. See also actual granularity, granularity, and natural granularity.

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-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 Threads Library **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.

time slicing

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 the Threads Library **tis** interface routines.

tis mutex

Mutex object that can be created using the Threads Library **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 the Threads Library that a kernel execution context is available. When a kernel execution context becomes available, the Threads Library scheduler schedules the thread with highest scheduling precedence that is ready to run onto the available kernel execution context.

word tearing

Form of race condition in a multithreaded program where two or more threads independently read the same granule of memory, update different portions of that granule, then independently (that is, asynchronously) store their respective copies of that granule. Can occur due to programmer's inattention to granularity considerations.

Index

A

Address exceptions, 5–7
Addressing, 64-bit, B–5
API
  See Application Programming Interface (API)
Application programming interface (API)
  POSIX Threads error conditions from, 3–17
ASTs (asynchronous system traps)
  POSIX Threads' delivery of, B–10
  restrictions on use with POSIX Threads, B–2
Asynchronous programming techniques, in a
    multithreaded program, A–7
Asynchronous thread cancelation, 2–15
  cleanup from, 2–17
Asynchronous user interface example program,
    6–8
Attributes
  of condition variables, 2–27
  of mutexes, 2–22
    mutex type, 2–22
  of threads, 2–3
    contention scope, 2–7
    guardsize, 2–7
    inherit scheduling, 2–3
    scheduling policy, 2–3
    scheduling priority, 2–5
    stack address, 2–6
    stack size, 2–6
Attributes objects, 2–1
  creating, 2–1
  destroying, 2–2

B

Background scheduling policy, 2–3
Boss/worker functional model, 1–4
  work queue variation, 1–4
Bugchecks, 3–18
  configuring output, 3–18
  contents of dump file, 3–18
  interpreting output, 3–18

C

Cancelability state, 2–15
Cancelability state attribute
  of thread attributes object, 2–15
    setting, pthread–174, tis–40
Cancelability type, 2–15
Cancelability type attribute
  of thread attributes object, 2–15
    setting, pthread–176
Cancelation points
  in multithreaded code, 2–16
  POSIX Threads routines that provide, 2–16,
    4–2
  system service routines that provide
    under OpenVMS Alpha, B–5
    under Tru64 UNIX, A–5
Cancelation requests
  delivering, pthread–50, pthread–189, tis–45
  sending, pthread–50
CATCH macro, 5–10
CATCH_ALL macro, 5–11
Cleanup handlers, 2–11
  executing, pthread–52
  registering, pthread–54
cma interface
  See Compaq-proprietary CMA (cma) interface
CMA interface, D–1
Compaq proprietary CMA (cma) interface, 1–19,
  D–1
Compiling applications
  under OpenVMS, B–1
  under Tru64 UNIX, A–1
Concurrency level
  of threads
    obtaining, pthread–99
    setting, pthread–178
Condition handlers (OpenVMS), declaring, B–4
Condition values (OpenVMS), used by POSIX
  Threads, B–5
Condition variable attributes objects, 2–1, 2–27
  creating, pthread–59
  destroying, pthread–56
  initializing, pthread–59
  obtaining process-shared attribute value,
    pthread–57

Word tearing, 3–12
   identifying scenarios, 3–13, 3–15
Work crew functional model, 1–4
Work queues, variation of boss/worker functional
    model, 1–4

Y

Yielding to another thread, pthread–192, tis–51