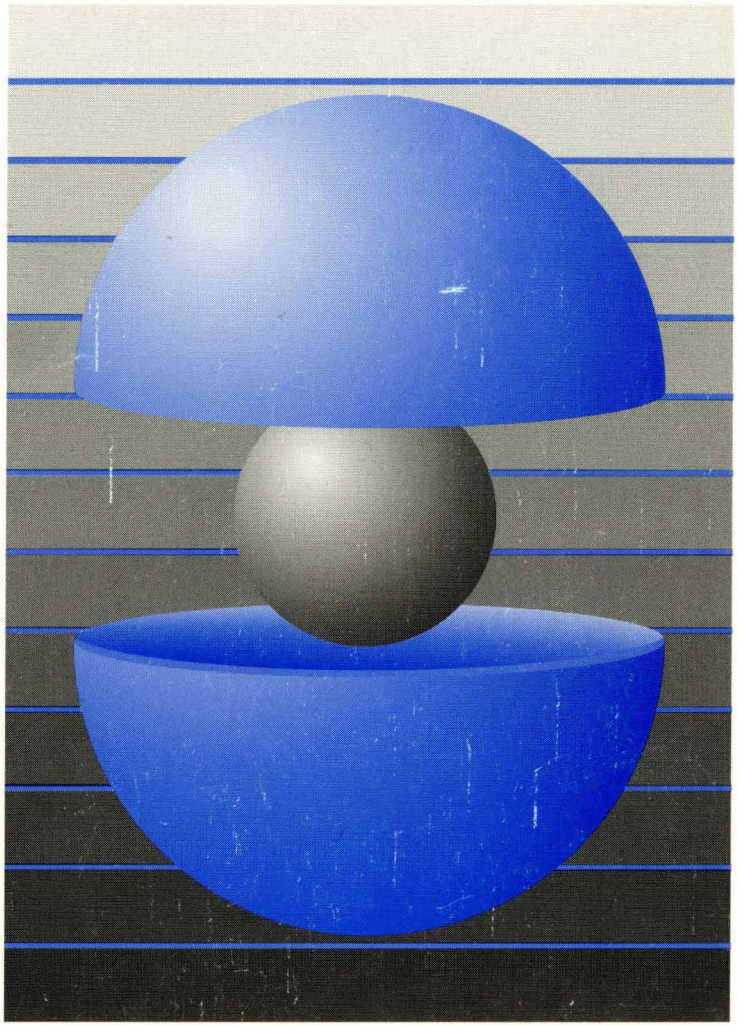
DEC OSF/1

digital

Guide to DECthreads



Part Number: AA-Q2DPA-TK

DEC OSF/1

Guide to DECthreads

Order Number: AA–Q2DPA–TK

February 1994

This manual provides guidelines and reference information for DECthreads, Digital's Multithreading Run-Time Library.

Digital Equipment Corporation
Maynard, Massachusetts

February 1994

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK6101

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

3 Programming with Threads

4 Using the DECthreads Exception Package

5 DECthreads Examples

Part II POSIX 1003.4a (pthread) Reference

Part III Digital Proprietary Interface Reference

Index

Examples

Figures

Tables

About This Manual

This manual provides usage and reference information on DECthreads routines.

DECthreads provides three complete and distinct interfaces that allow you to perform multithreaded operations: pthread, pthread exception-returning, and cma. The interface you select depends upon your goals and environment.

Audience

This manual is intended for system and application programmers who want to create a multithreaded program using DECthreads routines.

Organization

This manual consists of the following:

Part I

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

Part II

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

Part III

- This part provides detailed reference information on each DECthreads cma routine. Routine descriptions appear in alphabetical order by routine name.

Part IV

- This part provides detailed reference information on each DECthreads library routine. Routine descriptions appear in alphabetical order by routine name.

Appendixes

- Appendix A discusses DECthreads issues specific to systems based on UNIX software.
- Appendix B discusses DECthreads issues and restrictions specific to OpenVMS systems.
- Appendix C lists the DECthreads exceptions with an explanation and a recommended user action for each exception.
- Appendix D lists the DECthreads nonstandard data types that correspond to the types specified by the POSIX 1003.4a standard, with an explanation of each type.

Glossary

- See the Glossary for definitions of terms used in this manual. The terms are listed alphabetically.

Related Documents

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

- DEC OSF/1 AXP Version 1.0 or higher
- ULTRIX Version 4.2 or higher
- OpenVMS AXP Version 1.0 or higher
- VMS Version 5.5 or higher

The printed version of the DEC OSF/1 documentation set is color coded to help specific audiences quickly find the books that meet their needs. This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General Users	G	Teal
System Administrators	S	Red
Network Administrators	N	Yellow
Programmers	P	Blue
Reference Page Users	R	Black

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the DEC OSF/1 documentation set.

Reader's Comments

Digital welcomes your comments on this or any other DEC OSF/1 manual. You can send your comments in the following ways:

- Internet electronic mail: `readers_comment@ravine.zk3.dec.com`
- Fax: 603-881-0120 Attn: USG Documentation, ZK03-3/Y32
- A completed Reader's Comments form (postage paid, if mailed in the United States). Two Reader's Comments forms are located at the back of each printed manual.

If you have suggestions for improving particular sections or find any errors, please indicate the title, order number, and section numbers. Digital also welcomes general comments.

Conventions

Table 1 shows the conventions used in this manual.

Table 1 Conventions

Convention	Description
% $	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne and Korn shells.
#	A number sign represents the superuser prompt.
`cat(1)`	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat(1)` indicates that you can find information on the `cat` command in Section 1 of the reference pages.
Ctrl/*x*	The key combination Ctrl/*x* indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/Y or Ctrl/Z.
`monospaced text`	This typeface indicates the name of a command, routine, service, exception, or file. This typeface is also used in interactive examples and other screen displays.
`monospaced text`	This bolded typeface represents user input in interactive examples in the hardcopy and online versions of this guide.
. . .	A horizontal ellipsis in a figure or example indicates that not all of the statements are shown.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
()	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[]	In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term.

(continued on next page)

Table 1 (Cont.) Conventions

Convention	Description
italic text	Italic text represents book titles, parameters, arguments, and information that can vary in system messages (for example, Internal error *number*).
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.
mouse	The term *mouse* refers to any pointing device, such as a mouse, a puck, or a stylus.

Part I

DECthreads Overview and Programming Guidelines

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

1

Introduction to Multithreaded Programming

This chapter introduces **multithreaded programming**, which is the division of a program into multiple threads that execute concurrently. It also describes four software models that can be used as a basis for constructing multithreaded programming programs and applications.

This chapter also introduces concepts and techniques that are defined in more detail in Chapter 2.

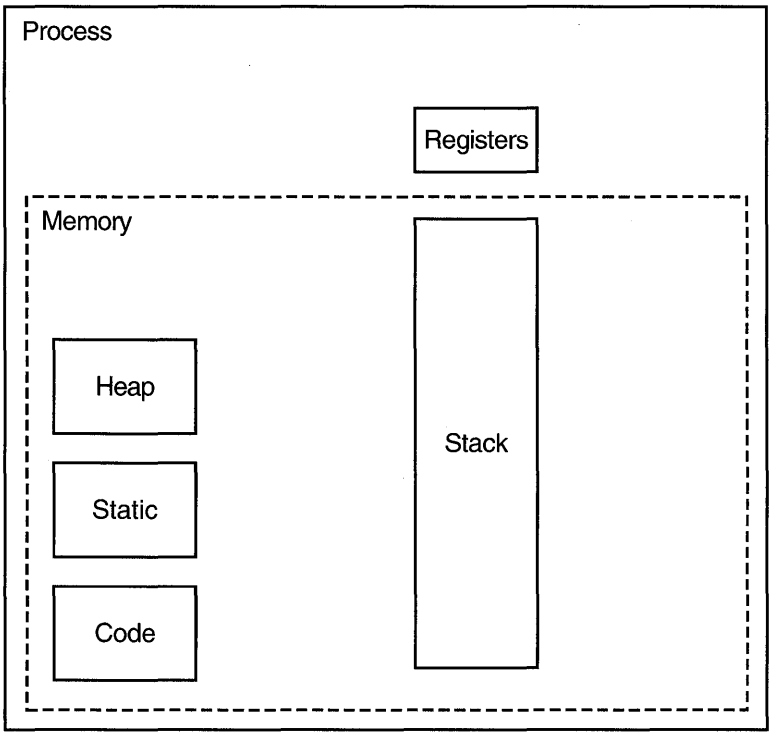
1.1 Overview of Threads

A **thread** is a single, sequential flow of control within a program. Within a single thread, there is a single point of execution. Most traditional programs consist of a single thread.

Using DECthreads, Digital's multithreading run-time library, a programmer can create several threads within a program. Threads execute concurrently, and, within a multithreaded program, there are at any time multiple points of execution. Threads execute within (and share) a single address space. Therefore, threads read and write the same memory locations. Synchronization elements such as mutexes and condition variables ensure that the shared memory is accessed correctly. DECthreads provides routines that allow you to create and use these synchronization elements. Mutexes and condition variables are discussed in Section 2.4.1 and Section 2.4.2.

Figure 1–1 and Figure 1–2 show the differences between a single threaded process and a multithreaded process.

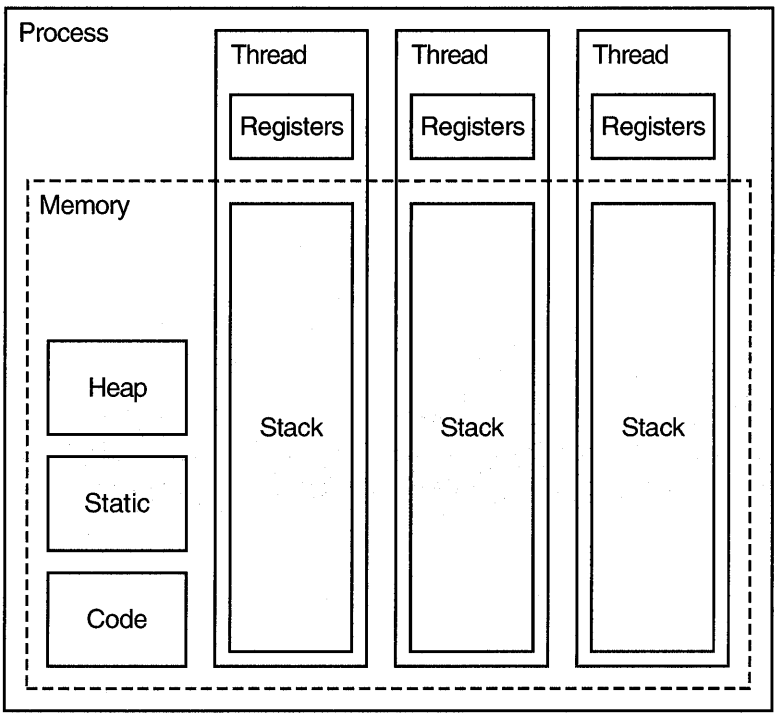
Figure 1–1 Single Threaded Process



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.

Figure 1–2 Multithreaded Process



ZK–3914A–GE

DECthreads provides the following three separate interfaces with which you can perform multithreaded operations:

- pthread interface

  The pthread interface is based on Draft 4 of the proposed IEEE standard for multithreaded programming, POSIX 1003.4a. This interface reports errors by setting a C language construct, the global *errno* value, and returning –1.

  Applications consistent with P1003.4a/D4 might require significant modifications for those future versions of DECthreads that conform to subsequent drafts or the final 1003.4a standard. However, other vendors have implemented the P1003.4a interface; therefore, programs using the pthread interface are more portable to non-Digital platforms than programs using the cma interface.

- pthread exception-returning interface

The pthread exception-returning interface has the same syntax as the pthread interface (and might also require modifications as the P1003.4a standard evolves). However, like the cma interface, the pthread exception-returning interface reports errors by raising exceptions. This simplifies the development of robust programs and reduces the chances of unpredictable behavior caused by ignored errors.

- cma interface

  The cma interface is a proprietary interface designed by Digital. Programs written to use the cma interface will run without need for modification on future versions of DECthreads. The cma interface reports errors by raising exceptions. This interface is usually not available on non-Digital platforms.

1.2 Advantages of Using DECthreads

Threads are used to improve the performance (throughput, computational speed, responsiveness—or some combination) of a program. Multiple threads are useful in a multiprocessor system where threads run concurrently on separate processors. Threads created using the DECthreads library are capable of utilizing multiprocessors if the operating system on that platform supports kernel threads. Multiple threads also improve program performance on single processor 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).

It is also advantageous to use threads when constructing a user interface. Consider the typical arrangement of a window system. Each time the user invokes an action (by clicking on a mouse button for example), a separate thread can be used to implement the action. If the user invokes multiple actions, multiple threads perform the actions in parallel. (Note that the implementation of the window system can also use a thread to handle the mouse actions, because the mouse is an example of a slow device.)

Threads are 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 one server process per client, at great expense).

1.3 Thread Execution

You can view multiple threads in a program as executing simultaneously. However, you cannot make any assumptions about the relative start or finish times of threads or the sequence in which they execute. Nevertheless, you can influence the scheduling of threads.

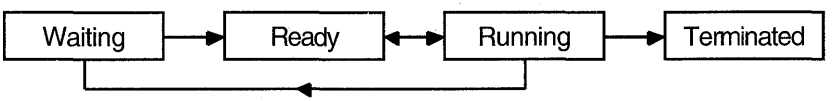
Each thread has its own thread identifier, which allows it to be uniquely identified. Also associated with a thread are its scheduling policy and priority, thread-specific data values, and the required system resources to support a flow of control.

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

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

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

Figure 1–3 Thread State Transition Diagram



ZK–3786A–GE

Note

A multithreaded program must be reentrant. Therefore, be sure that your compiler generates reentrant code before you do multithreading design or coding work. (C, Ada, Pascal, and BLISS compilers generate reentrant code by default.)

If your program is nonreentrant, it may be impossible to keep the program's threads from interfering with each other. See Section 3.2.2 for more information about thread reentrancy.

1.4 Software Models for Multithreaded Programming

The following sections describe four software models for which multithreaded programming is especially well suited:

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

1.4.1 Boss/Worker Model

In a boss/worker model of program design, one thread functions as the boss because it assigns tasks to worker threads for them to perform. Each worker performs a different 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 or not each worker is ready to receive another task.

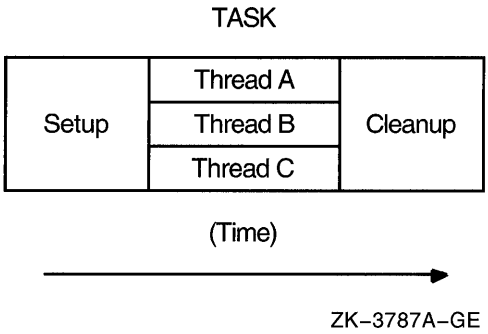
A variation of the boss/worker model is the work queue model. The boss places tasks in a queue, and workers check the queue and take tasks to perform. An example of the work queue model in an office environment is a secretarial typing pool. The office manager (boss) puts documents to be typed in a basket and typists (workers) 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 horizontally 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 house. Each person cleans certain rooms or performs certain types of work (washing floors, polishing furniture, and so forth), and each works independently.

Figure 1–4 shows a task performed by three threads in a work crew model.

Figure 1–4 Work Crew Model of Thread Operation



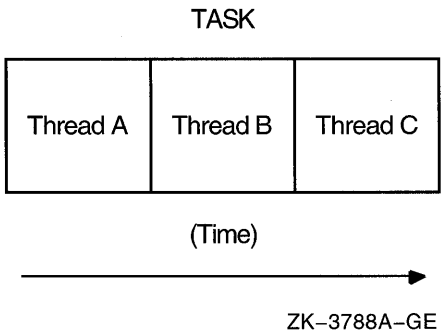
1.4.3 Pipelining Model

In the pipelining model, a task is divided vertically into steps. The steps must be performed in sequence to produce a single instance of the desired output, 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 program is designed to produce multiple instances of the desired output, and the steps are designed to operate in parallel so that while one step is performed on one instance of the output, the preceding step can be performed on the next instance of the output.

An example of the pipelining model in a factory environment 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 using the pipelining model, each thread executes a step in the task. Figure 1–5 shows a task performed by three threads in a pipelining model.

Figure 1–5 Pipelining Model of Thread Operation



1.4.4 Combinations of Models

You may find it appropriate to combine the software models in a single program if your task is complex. For example, a program could be designed using the pipelining model, but with one or more steps handled by a work crew. In addition, threads could be assigned to a work crew by taking a task from a work queue and deciding (based on the task characteristics) which threads are needed for the work crew.

1.5 Potential Problems with Multithreaded Programming

When you design and code a multithreaded program, consider the following problems and accommodate or eliminate each as appropriate:

- Program complexity

  Program complexity is the most significant problem 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 synchronization and interplay among threads is correct. This level of expertise is higher than for most single-threaded programs.

- Race conditions

  A type of programming error called a race condition causes unpredictable and erroneous program behavior. Section 3.7.2 discusses race conditions in more detail.

- Deadlocks

  A type of programming error called a deadlock causes two or more threads to be blocked from executing indefinitely. Section 3.7.3 discusses deadlocks in more detail.

- Priority inversion

  Priority inversion prevents high-priority threads from executing when interdependencies exist among three or more threads of different priorities. Section 3.6 discusses priority inversion in more detail.

- Nonreentrant software

  If a thread calls a routine or library that is not equipped to deal with threads, use the global locking mechanism to prevent conflicts with other threads using the same routine or library. Section 3.2.2 discusses nonreentrant software in more detail.

2

Thread Concepts and Operations

This chapter discusses concepts and techniques related to DECthreads.

For detailed information on the multithreading routines referred to in this chapter, see the description of that routine in the appropriate reference section.

2.1 Thread Operations

The following sections describe the operations you can perform with threads.

2.1.1 Starting a Thread

To start a thread, you can create it using the `cma_thread_create` or `pthread_create` routine. These routines create the thread object, based on the specified or default attributes, and start execution of the function you specified as the thread's start routine.

2.1.2 Terminating a Thread

A thread exists until it terminates and the `cma_thread_detach` or `pthread_detach` routine has been called for the thread. If the thread terminates before a detach routine has been called for it, then the thread continues to exist and other threads can join it until it is detached.

Note

When the initial thread returns from the main routine, the entire process (on UNIX systems) or image (on OpenVMS systems) terminates, just as it does when a thread calls `exit( )` (on systems based on UNIX software) or SYS$EXIT (on OpenVMS systems).

A thread terminates for any of the following reasons:

- The thread returns from its start routine. This is the usual case.
- The thread calls the `cma_thread_exit_normal`, `cma_thread_exit_error`, or `pthread_exit` routine.

The cma_thread_exit_normal routine terminates the calling thread when successful completion occurs prematurely; the cma_thread_exit_error routine terminates the calling thread when an error occurs. The pthread_exit routine terminates the calling thread and returns a status value to indicate the thread's exit status.

- The thread is terminated prematurely after having been specified in a call to the cma_thread_alert or pthread_cancel routine.

  The cma_thread_alert and pthread_cancel routines request termination of a specified thread if alerts or cancelation are permitted. See Section 2.7 for more information on canceling threads and controlling whether or not cancelation is permitted.

2.1.3 Normal Termination

Normal termination occurs when a thread returns from its *start_routine* function or calls cma_thread_exit_normal or pthread_exit. In the case of a normal termination, the following actions are performed:

1. The return value of the start routine function and the normal termination status (cma_c_term_normal) are copied into the thread object. This permits both to be obtained when another thread later calls the cma_thread_join routine. (The pthread_join routine returns only the start routine function value). If the start routine returns normally and the start routine is a procedure that does not return a value, then the return value obtained by cma_thread_join or pthread_join will be unpredictable.
2. Each thread-specific data destructor is removed from the list of destructors for this thread, and then is called. This step destroys all the thread-specific data associated with the current thread. See Section 2.6 for more information on thread-specific data.

   If an unhandled exception occurs when calling a destructor, the steps described in the following section on error termination are performed.
3. In the pthread interface to DECthreads, each cleanup handler that has been declared by pthread_cleanup_push and not yet removed by pthread_cleanup_pop is called. The most recently pushed handler is called first.
4. A broadcast is made so that all threads currently waiting in a call to cma_thread_join or pthread_join can return from the call.
5. The thread object is marked to indicate that it is no longer needed by the thread itself. A check is made to determine if the thread object is no longer needed by other threads, that is, if cma_thread_detach or pthread_detach has already been called. If that routine has been called, then the thread

object is deallocated. Otherwise, the thread object is retained for use by subsequent joiners until it is detached.

2.1.4 Error Termination

Error termination occurs when a thread calls the cma_thread_exit_error routine or calls the pthread_exit routine specifying an error status as an argument. In the case of an error termination, the following actions are performed:

1. The exit status of the thread object is set to the error value.
2. The unwind handlers of any stack frames that are still active are invoked.
3. Execution continues at step 2 in Section 2.1.3.

2.1.5 Waiting for a Thread to Terminate

A thread waits for the termination of another thread by calling the cma_thread_join or pthread_join routine. Execution in the current thread is suspended until the specified thread terminates. If multiple threads call these routines and specify the same thread, all threads resume execution when the specified thread terminates.

If you specify the current thread with the cma_thread_join or pthread_join routine, a deadlock results. See Section 3.7.3 for more information about deadlocks.

Do not confuse cma_thread_join and pthread_join with other routines that cause waits and that are related to the use of a particular DECthreads feature. For example, use the cma_cond_wait or pthread_cond_wait routines, or the cma_cond_timedwait or pthread_cond_timedwait routines to wait for a condition variable to be signaled or broadcast. (See Section 2.4.2 for more information on condition variables.)

2.1.6 Deleting a Thread

Once a thread has been marked for deletion, it is automatically deleted after it terminates; that is, no explicit deletion operation is required. If the thread has not yet terminated, the cma_thread_detach or pthread_detach routine marks the thread for deletion, and its storage is reclaimed immediately when the thread terminates. A thread cannot be joined or canceled after the cma_thread_detach or pthread_detach routine has been called for the thread (even if the thread has not yet terminated).

If a thread that has not been detached terminates, its storage is retained so that other threads can join with it. The storage is reclaimed when the thread is detached.

2.2 Handles

A **handle** is storage, similar to a pointer, that refers to a specific DECthreads object (thread, mutex, condition variable, queue, or attributes object).

Handles are allocated by the user application. They can be freely copied by the program and stored in any class of storage; objects are managed by DECthreads.

Because DECthreads objects are only accessed by handles, you can think of the handle as if it were the object itself.

DECthreads objects are accessed by handles, rather than pointers, because handles allow for greater robustness and portability. Handles allow DECthreads to detect the following types of run-time errors:

- Using an uninitialized handle
- Using a handle that has been corrupted
- Using a handle whose object no longer exists (a dangling handle)

2.3 Attributes Objects

An **attributes object** is used to describe DECthreads objects. This description consists of the individual attribute values that are used to create an object. An attributes object is analogous to a type definition in a programming language; it describes details of the objects to be created.

When you create an object, you can accept the default attributes for that object or specify an attributes object that contains specific attributes that you have set. For a thread, you can also change certain attributes after thread execution starts—for example, you can change the thread's priority.

The following sections describe how to create and delete attributes objects, and describe the individual attributes that you can specify for different objects.

2.3.1 Creating an Attributes Object

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

- `cma_attr_create` for any type of cma attributes object
- `pthread_attr_create` for pthread thread attributes
- `pthread_condattr_create` for pthread condition variable attributes
- `pthread_mutexattr_create` for pthread mutex attributes

These routines create an attributes object containing default values for the individual attributes. To modify any attribute values in an attributes object, use one of the set routines described in the following sections.

Creating an attributes object or changing the values in an attributes object does not affect the attributes of objects previously created.

In the cma interface to DECthreads, there is no distinction between thread, mutex, or condition variable attributes objects. All attributes objects are identical and can be used to create any type of object. However, in the pthread interface, each type of object has a distinct type of attributes object.

2.3.2 Deleting an Attributes Object

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

- `cma_attr_delete` for any type of cma attributes object
- `pthread_attr_delete` for pthread thread attributes objects
- `pthread_condattr_delete` for pthread condition variable attributes objects
- `pthread_mutexattr_delete` for pthread mutex attributes objects

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

2.3.3 Thread Attributes

A **thread attributes object** allows you to specify values for thread attributes other than the defaults when you create a thread with the `cma_thread_create` or `pthread_create` routine. To use a thread attributes object, perform the following steps:

1. Create a thread attributes object by calling the `cma_attr_create` or `pthread_attr_create` routine.
2. Call the routines discussed in the following sections to set the individual attributes of the thread attributes object.
3. Create a new thread by calling the `cma_thread_create` or `pthread_create` routine and specifying the handle of the thread attributes object.

You have control over the following attributes of a new thread:

- Scheduling inheritance
- Scheduling policy
- Scheduling priority

- Stack size
- Stack guard size

2.3.3.1 Inherit Scheduling Attribute

The **inherit scheduling attribute** specifies whether a newly created thread inherits the scheduling attributes (scheduling priority and policy) of the creating thread (the default) or uses the scheduling attributes stored in the attributes object. You can set this attribute by calling the `cma_attr_set_inherit_sched` or `pthread_attr_setinheritsched` routine.

2.3.3.2 Scheduling Policy Attribute

The **scheduling policy attribute** describes how the thread is scheduled for execution relative to the other threads in the program. (In the following list, the lowercase policy corresponds to the cma interface and the uppercase policy corresponds to the pthread interface.) A thread has one of the following scheduling policies:

- `cma_c_sched_fifo` or `SCHED_FIFO` (first-in/first-out (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 it preempts the current thread and begins running immediately.
- `cma_c_sched_rr` or `SCHED_RR` (round-robin (RR))—The highest priority thread runs until it blocks; however, threads of equal priority, if that priority is the highest among other threads, are timesliced. (**Timeslicing** is a mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.) If a thread with this policy becomes ready, and it has a higher priority than the currently running thread, then it preempts the current thread and begins running immediately.
- `cma_c_sched_throughput` (also known as `cma_c_sched_default`) or `SCHED_FG_NP` (also known as `SCHED_OTHER`) (Default)—All threads are timesliced. Under this policy, all threads receive some scheduling regardless of priority. Therefore, no thread is completely denied execution time. Nevertheless, higher priority threads receive more execution time than lower priority threads. Threads with the default scheduling policy can be denied execution time by FIFO or RR threads.

- cma_c_sched_background or SCHED_BG_NP (Background)—Like the default (throughput) scheduling policy, this policy ensures that all threads, regardless of priority, receive some scheduling. However, background threads can be denied execution time by FIFO or RR threads, and receive less execution time than default policy threads.

You can use either of the following methods to set the 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 this, call the cma_attr_set_sched or pthread_attr_setsched routine. This allows the creator of a thread to establish the created thread's initial scheduling policy and priority. Note that this value is used only if the attributes object is set so that the created thread does not inherit its priority from the creating thread. Inheriting priority is the default behavior.
- Change the scheduling policy of an existing thread (and, at the same time, the scheduling priority) by calling the cma_thread_set_sched or pthread_setscheduler routine. This allows a thread to change its own scheduling policy and priority but has no effect on an attributes object.

Section 2.8 describes and shows the effect of scheduling policy on thread scheduling.

2.3.3.3 Scheduling Priority Attribute

The **scheduling priority attribute** specifies the execution priority of a thread. This attribute is expressed relative to other threads in the same policy on a continuum of minimum to maximum for each scheduling policy. A thread's priority falls within one of the following ranges, depending on its scheduling policy.

For cma routines:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

For pthread routines:

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 2.8 describes how to specify priorities between the minimum and maximum values, and it also discusses how priority affects thread scheduling.

You can use either of the following methods to set the scheduling priority attribute:

- Set the scheduling priority attribute in the attributes object, which establishes the execution priority of a new thread when it is created. To do this, call the `cma_attr_set_priority` or `pthread_attr_setprio` 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 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 priority attribute of an existing thread by calling the `cma_thread_set_priority` or `pthread_setprio` routine. (Call the `cma_thread_set_sched` or `pthread_setscheduler` routine to change both the scheduling priority and scheduling policy of an existing thread.) This allows a thread to change its own execution priority.

2.3.3.4 Stacksize Attribute

The **stacksize attribute** is the minimum size (in bytes) of the memory required for a thread's stack. To increase or decrease the size of the stack for the thread about to be created, call the `cma_attr_set_stacksize` or `pthread_attr_setstacksize` routine, and use this attributes object when creating the thread and stack. You cannot change the size of a thread's stack after the thread has been created. See Section 3.5.2 for more information on sizing a stack.

2.3.3.5 Guardsize Attribute

The **guardsize attribute** is the minimum size (in bytes) of the guard area for the stack of a thread. A **guard area** is a reserved area designed to help prevent or detect, or both, overflow of the thread's stack. The guard area is a region of memory that cannot be accessed by a thread. It is located adjacent to the last page in the thread's stack. To increase or decrease the size of the guard area for the thread about to be created, call the `cma_attr_set_guardsize` or `pthread_attr_setguardsize_np` routine.

2.3.4 Mutex Attributes

A **mutex attributes object** allows you to specify values other than the defaults for mutex attributes when you create a mutex with the `cma_mutex_create` or `pthread_mutex_init` routine.

Section 2.4.1 describes the purpose and types of mutexes.

2.3.4.1 Mutex Type Attribute

The **mutex type attribute** specifies whether a mutex is fast, recursive, or nonrecursive. (See Section 2.4.1 for more information.) You can set the mutex type attribute by calling the `cma_attr_set_mutex_kind` or `pthread_mutexattr_setkind_np` routine. If you do not use a mutex attributes object to select a mutex type, calling the `cma_mutex_create` or `pthread_mutex_init` routine creates a fast mutex by default.

2.3.5 Condition Variable Attributes

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

Section 2.4.2 describes the purpose and uses of condition variables.

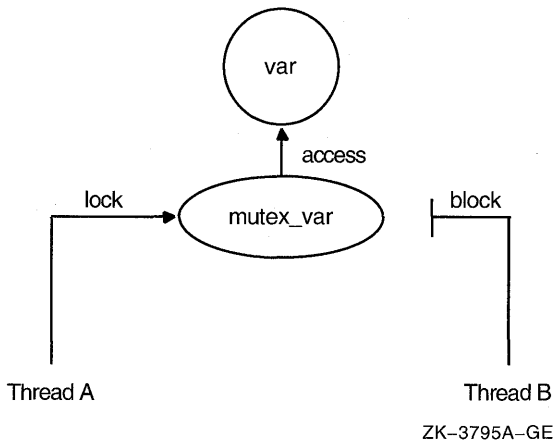
2.4 Synchronization Objects

In a multithreaded program, you must use synchronization objects whenever there is a possibility of corruption of shared data or conflicting scheduling of threads that have mutual scheduling dependencies. The following sections discuss DECthreads synchronization objects: mutexes and condition variables.

2.4.1 Mutexes

A **mutex** (*mut*ual *ex*clusion) is 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. A mutex has two states, locked and unlocked. 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 2–1.)

Figure 2–1 Only One Thread Can Lock a Mutex



Each mutex must be created. (To create mutexes as part of your program's one-time initialization code, see Section 2.5.) To create a mutex, use the `cma_mutex_create` or `pthread_mutex_init` routine. These routines allow 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 Fast Mutex

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

This type of mutex is called fast because it can be locked and unlocked more rapidly than a recursive or nonrecursive mutex. It is the most efficient form of mutex.

2.4.1.2 Recursive Mutex

A **recursive mutex** can be locked more than once by a given thread without causing a deadlock. The thread must call the `cma_mutex_unlock` or `pthread_mutex_unlock` routine the same number of times that it called the `cma_mutex_lock` or `pthread_mutex_lock` routine before another thread can lock the mutex. Recursive mutexes have the notion of a mutex owner. When a thread successfully locks a recursive mutex, it owns that mutex and the lock count is set to 1. Any other thread attempting to lock the mutex blocks until the mutex becomes unlocked. If the owner of the mutex attempts to lock the mutex again, the lock count is incremented, and the thread continues running. When an owner unlocks a recursive mutex, the lock count is decremented. The mutex remains locked and owned until the count reaches zero. It is an error for any thread other than the owner to attempt to unlock the mutex.

A recursive mutex is useful if a thread needs exclusive access to a piece of data, and it needs to call another routine (or itself) that needs exclusive access to the data. A recursive mutex allows nested attempts to lock the mutex to succeed rather than deadlock.

This type of mutex is called recursive because it allows you a capability not permitted by a fast (default) mutex. However, its use requires more careful programming. A recursive mutex should never be used with condition variables, because the unlock performed for a `cma_cond_wait` or `pthread_cond_wait`, or a `pthread_cond_timedwait` or `cma_cond_timedwait` might not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate, and the thread waits indefinitely. See Section 2.4.2 for information on the condition variable wait and timed wait routines.

2.4.1.3 Nonrecursive Mutex

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

2.4.1.4 Mutex Operations

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

- `cma_mutex_lock` or `pthread_mutex_lock`

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

- `cma_mutex_trylock` or `pthread_mutex_trylock`

  This routine returns immediately with a Boolean value 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 `cma_mutex_unlock` or `pthread_mutex_unlock` routine. If another thread is waiting on the mutex, it is placed in the ready state. If more than one thread is waiting on the mutex, the scheduling policy (see Section 2.3.3.2) and the scheduling priority (see Section 2.3.3.3) determine which thread is readied, and the next running thread that requests it locks the mutex. The mutex is not automatically granted to the first waiter. If the unlocking thread attempts to relock the mutex before the first waiter gets a chance to run, the unlocking thread will succeed in relocking the mutex, and the first waiter may be forced to reblock.

You can delete a mutex and reclaim its storage by calling the `cma_mutex_delete` or `pthread_mutex_destroy` routine. Use these routines only after the mutex is no longer needed by any thread. A mutex cannot be deleted while it is locked. Mutexes are automatically deleted when the program terminates.

Note

There is no deadlock detection for mutexes other than nonrecursive mutexes or for deadlocks composed of combinations of multiple mutexes or condition variables.

2.4.2 Condition Variables

A **condition variable** allows a thread to block its own execution until some shared data reaches a particular state. A condition variable is a synchronization object used in conjunction with a mutex. 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 Boolean expression called a **predicate**.

Cooperating threads check the predicate and wait on the condition variable. 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.

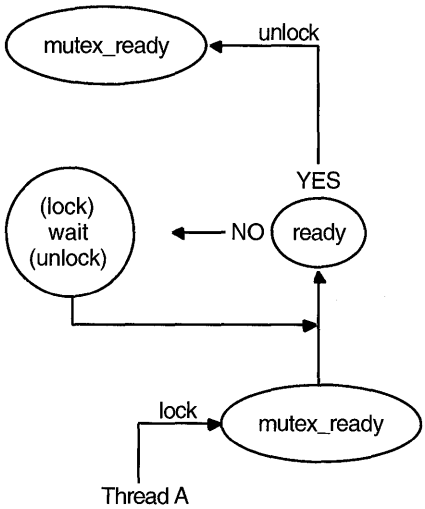
A condition variable is used for tasks with coarse granularity; a thread can wait on a condition variable for long periods. A mutex is used for synchronization with fine granularity and should be held only for short periods of time.

You must associate a mutex with a condition variable. A thread locks a mutex for some shared data and then checks whether or not the shared data is in the proper state. If it is not in the proper state, the thread waits on the appropriate condition variable. 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. When the thread that acquires the mutex puts the data in the appropriate state, it wakes a waiting thread by signaling the condition variable. One thread comes out of its wait state with the mutex locked (the thread relocks the mutex before returning from the wait); other threads waiting on the condition variable remain blocked.

It is important to evaluate the predicate in a while loop. This ensures that the program will check the predicate after it returns from the condition wait and guards against the predicate becoming false again as the waiting thread completes the 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 2–2).

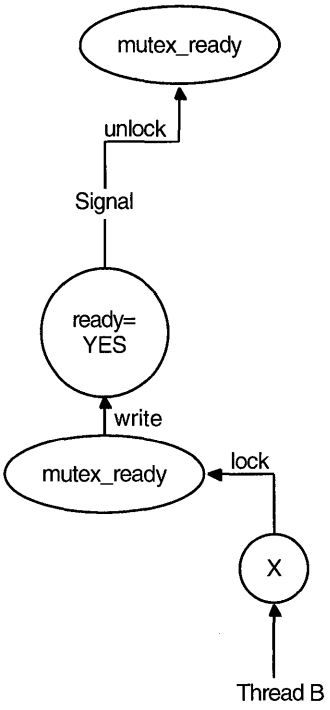
Figure 2–2 Thread A Waits on Condition Ready



ZK–3793A–GE

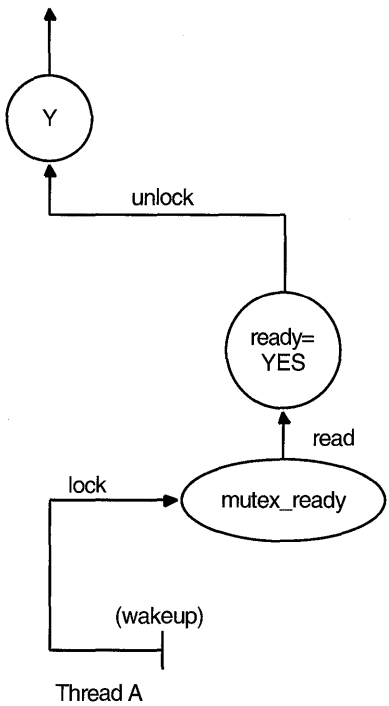
First, Thread A locks the mutex named *mutex_ready* that is associated with the shared variable named *ready*. Then it reads the value in *ready*. This test is called the predicate. If the predicate indicates that Thread B has finished Task X, then Thread A can unlock the mutex and proceed with Task Y. If the predicate indicates that Thread B has not yet finished Task X, however, then Thread A waits for the predicate to change by calling the `cma_cond_wait` or `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 `cma_cond_signal` or `pthread_cond_signal` routine (see Figure 2–3).

Figure 2–3 Thread B Signals Condition Ready



Thread B releases its lock on the shared variable's mutex. As a result of the signal, Thread A wakes up, implicitly regaining its lock on the condition variable's mutex. It then verifies that the predicate is in the correct state, and proceeds to execute Task Y (see Figure 2–4).

Figure 2–4 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.

You can use the `cma_cond_init` or `pthread_cond_init` routine to create a condition variable. To create condition variables as part of your program's one-time initialization code, see Section 2.5.

Use the `cma_cond_wait` or `pthread_cond_wait` routine to cause a thread to wait until the condition is signaled or broadcast. These routines specify a condition variable and a mutex that you have locked. (If you have not locked the mutex, the results of `cma_cond_wait` or `pthread_cond_wait` are unpredictable.) These routines automatically unlock the mutex and cause the calling thread to

wait on the condition variable until another thread calls one of the following routines:

- `cma_cond_signal` or `pthread_cond_signal` to wake one thread that is waiting on the condition variable
- `cma_cond_broadcast` or `pthread_cond_broadcast` to wake all threads that are waiting on a condition variable

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

If you want to limit the time that a thread waits for a condition to be signaled or broadcast, use the `cma_cond_timedwait` or `pthread_cond_timedwait` routine. These routines specify 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 delete a condition variable and reclaim its storage by calling the `cma_cond_delete` or `pthread_cond_destroy` routine. Use one of these routines only after the condition variable is no longer needed by any thread. A condition variable cannot be deleted while one or more threads are waiting on it. Condition variables are automatically deleted when the program terminates.

2.4.3 Other Synchronization Methods

Another synchronization method that you can use is calling `cma_thread_join` or `pthread_join`. These routines allow a thread to wait for a specific thread to complete its execution. When the specified thread terminates, the joining thread is unblocked and continues its execution. See Section 2.1.2 for information on terminating a thread.

2.5 One-Time Initialization Routines

You will probably 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, you may want to create mutexes and condition variables—each of which must be created only once—in an initialization routine. Multiple threads can call the `cma_once` or `pthread_once` routine, or one of those routines can be called multiple times in the same thread, resulting in only one call to the specified routine.

Use the cma_once or pthread_once routine to ensure that your initialization routine is executed only a single time, that is, by the first thread that tries to initialize the facility. Note that it is extremely difficult to perform portable one-time initialization reliably in a multithreaded environment without using these routines.

2.6 Per-Thread Context and Thread-Specific Data

Each thread has an area in which thread-specific data information is kept.

Note

The cma routines use the term **per-thread context**, while the pthread routines use the term **thread-specific data**. For simplicity, the term **thread-specific data** is used in this section to describe both per-thread context and thread-specific data.

You can associate arbitrary data with a thread's context. Think of this as the ability to add one or more user-specified fields to the current thread's context.

Thread-specific data is like a global variable in that it is accessible to the thread anywhere in the program. Each thread can store its own value in the thread-specific data.

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

- cma_key_create or pthread_keycreate to create a unique key value

  These routines generate a unique key value that is shared by all threads in the process. This keeps your data separate from other thread-specific data. One call to the cma_key_create or pthread_keycreate routine creates a cell in all threads. A routine can be specified in the call to these routines that destroy the context value associated with this key when the thread terminates, for example, to free storage pointed to by an address in the thread-specific data.

- cma_key_set_context or pthread_setspecific to associate data with a key

  These routines associate some data with a specific key. Multiple threads associate different data with the same key. For example, each thread stores the value of a pointer in a different block of dynamically allocated memory that it has reserved. The same key is used in each thread to store and retrieve that thread's thread-specific data value. You may want to call cma_once to create the key. See Section 2.5 for more information.

- `cma_key_get_context` or `pthread_getspecific` to obtain the data associated with a key

  These routines obtain the thread-specific context value associated with a specified key. Use these routines to locate the data associated with the current thread's context.

2.7 Thread Cancelation

Canceling (or alerting) is a mechanism by which one thread requests termination of another thread (or itself).

Note

The cma routines use the term *alert*, while the pthread routines use the term *cancel*. For simplicity, the term *cancel* is used in this section to describe both cancels and alerts.

When you request that a thread be canceled, you are requesting that it terminate as soon as possible. However, the target thread can control how quickly it terminates by controlling its general cancelability and its asynchronous cancelability.

The following is a list of routines that are **cancelation points:**

- `cma_alert_enable_asynch` or `pthread_setasynccancel`
- `cma_alert_test` or `pthread_testcancel`
- `cma_delay` or `pthread_delay_np`
- `cma_thread_join` or `pthread_join`
- `cma_cond_wait` or `pthread_cond_wait`
- `cma_cond_timed_wait` or `pthread_cond_timedwait`

General cancelability is enabled by default. **General cancelability** determines whether a thread can receive a cancelation request. If general cancelability is disabled, the thread does not receive any cancelation requests. Normally threads receive cancelation requests only at cancelation points—for example, when a call to the `cma_cond_wait` or `pthread_cond_wait` routine is made. However, if you enable **asynchronous cancelability**, cancelation requests can be delivered at any time.

If general cancelability is enabled, you can request the delivery of any pending cancel request by using the cma_alert_test or pthread_testcancel routine. These routines allow you to permit cancelation to occur at places where it might not otherwise be permitted under general cancelability, and it is especially useful within very long loops to ensure that cancel requests are noticed within a reasonable time.

If you disable general cancelability, the thread cannot be terminated by any cancel request. Disabling general cancelability means that a thread could wait indefinitely if it does not come to a normal conclusion; therefore, be careful about disabling general cancelability.

Because it is impossible to predict exactly when an asynchronous cancelation request will be delivered, it is extremely difficult to recover properly when an asynchronous cancelation request is delivered. For this reason, asynchronous cancelation should only be enabled across regions of code that do not need to clean up in any way (such as unlocking mutexes or freeing storage). For the same reason, you should not call any library subroutines while asynchronous cancelation is enabled, including all DECthreads routines except cma_alert_restore, pthread_setcancel, and pthread_setasynccancel. Asynchronous cancelability is disabled by default.

Note

If general cancelability is disabled, the thread cannot be canceled, regardless of whether asynchronous cancelability is enabled or disabled. The setting of asynchronous cancelability is relevant only when general cancelability is enabled.

Use the following routines to control the canceling of threads:

- cma_alert_enable_general and cma_alert_disable_general or pthread_setcancel to enable and disable general cancelability
- cma_alert_test or pthread_testcancel to request delivery of a pending cancel to the current thread
- cma_alert_enable_asynch and cma_alert_disable_asynch or pthread_setasynccancel to enable and disable asynchronous cancelability
- cma_alert_restore to restore a prevous cancelation state
- cma_thread_alert or pthread_cancel to request that a thread be canceled

2.8 Thread Scheduling

Threads are scheduled according to their scheduling priority and how the scheduling policy treats those priorities. To understand the discussion in this section, you must understand the concepts in the following sections:

- Section 2.3.3.2 on scheduling policies, including how each policy handles thread scheduling priority
- Section 2.3.3.3 on thread scheduling priorities
- Section 2.3.3.1 on inheriting of scheduling attributes by created threads

To specify the minimum or maximum priority, use the appropriate symbol—for example, `cma_c_prio_through_min` or `PRI_OTHER_MIN`, or `cma_c_prio_through_max` or `PRI_OTHER_MAX`. The cma interface to DECthreads specifies a mid-range value, but the pthread interface does not. To specify a value between the minimum and maximum priority, use an appropriate arithmetic expression. For example, to specify a priority midway between the minimum and maximum for the default scheduling policy in the pthread interface, specify the following concept using your programming language's syntax:

```
pri_other_mid = (PRI_OTHER_MIN + PRI_OTHER_MAX) / 2
```

You should avoid using specific numerical values because the range of priorities can change from implementation to implementation.

Values outside the range of minimum to maximum result in an error. Priority values are integers.

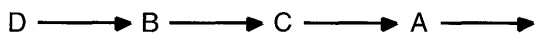
To show results of the different scheduling policies, consider the following example: A program has four threads, called A, B, C, and D. For each scheduling policy, three scheduling priorities have been defined: minimum, middle, and maximum. The threads have the following priorities:

A	minimum
B	middle
C	middle
D	maximum

The following figures show execution flows depending on whether the first-in /first-out (FIFO), round-robin (RR), or throughput (Default) scheduling policy is in effect. 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 (during the flow shown in each figure).

Figure 2–5 shows a flow with FIFO scheduling.

Figure 2–5 Flow with FIFO Scheduling

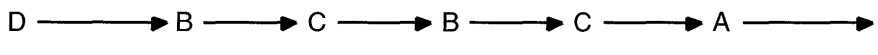


ZK–3789A–GE

Thread D executes until it waits or terminates; then Thread B starts because it has been waiting longer than Thread C, and it executes until it waits or terminates; then Thread C executes until it waits or terminates; then Thread A executes.

Figure 2–6 shows a flow with RR scheduling.

Figure 2–6 Flow with RR Scheduling

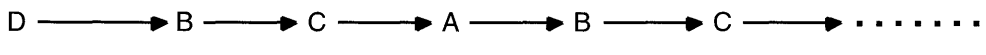


ZK–3790A–GE

Thread D executes until it waits or terminates; then Threads B and C are timesliced, because they both have middle priority; then Thread A executes.

Figure 2–7 shows a flow with Default scheduling.

Figure 2–7 Flow with Default Scheduling



ZK–3791A–GE

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

Because low-priority threads eventually run, the default scheduling policy protects against the problems of starvation and priority inversion, discussed in Section 3.6.

2.9 Atomic Queues

Atomic queues are DECthreads Library objects that you can use to communicate information among threads or among routines in a single thread.

Note

Atomic queues are documented in Part IV, the DECthreads Library reference section. The queue routines begin with the `cma_lib_` prefix; they are not included with the pthread interface. You can call these routines from programs running on OpenVMS and OSF/1 systems only.

Operations on queues are atomic because any operation on the queue is guaranteed to complete before any other operation on that same queue can begin. Queue operations are not interruptable.

An atomic queue contains preallocated queue items. Each element (an integer identifier or a pointer to a block of data) inserted into the queue consumes a queue item. The number of elements allowed on the queue is called the **queue size**. The queuesize attribute can be specified by calling the `cma_lib_attr_set_queuesize` routine. The default is 128 queue items.

You can insert an element at the end of a queue by calling `cma_lib_queue_enqueue` or at the front of a queue by calling `cma_lib_queue_requeue`. You can remove an element from a queue by calling `cma_lib_queue_dequeue`. In each of these routines, if the element cannot be inserted or removed because the queue is full or empty, the calling thread is blocked until the action can be performed.

If you want to insert or remove an element but return with a status value if the queue is full or empty rather than cause the thread to wait, you can call the `cma_lib_queue_try_enqueue`, `cma_lib_queue_try_requeue`, or `cma_lib_queue_try_dequeue` routines. These routines return a Boolean value indicating whether or not the routine succeeded.

The `cma_lib_queue_try_enqueue_int` routine works exactly like `cma_lib_queue_try_enqueue` except that it can be called from an interrupt handler. Only routines with an `_int` suffix can be called from interrupt routines.

3

Programming with Threads

This chapter discusses issues of which a programmer using threads must be aware. 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 Design for Asynchronous Execution

When programming with threads, always keep in mind that the execution of a thread is inherently asynchronous with respect to other threads running the system (or in the process). You *cannot* depend upon *any* synchronization between two threads unless you explicitly code that synchronization into your program using one of the following:

- Mutexes
- A properly tested application predicate loop on a condition variable
- A call to join with a thread you expect to terminate
- An equivalent platform dependent programming construct (such as VAX interlocked instructions)

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

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

Some examples of common programming errors that may work often under some implementations but not at all under others are as follows:

1. 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.

   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.

   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.

2. Initializing DECthreads objects (such as mutexes) or global data that is to be used by another thread *after* creating the thread.

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

Before you create a thread, you should set up all requirements that the thread will need to execute. If you need to set the thread scheduling parameters, for example, do so with attributes objects when you create it, rather than trying to use `cma_thread_set_priority`, `pthread_setprio`, or other routines afterwards. If you need to set global data for the thread or create synchronization objects, do these before you create the thread or set them in a `cma_once` or `pthread_once` initialization routine that is called from each thread.

3.2 Using Existing Libraries

Because multithreaded programming has only recently become common, many existing code libraries are incompatible with threads. Many of the traditional C run-time library routines, for example, 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 `getpwent(2)` routine returns the entries in

the password file in sequence. If multiple threads call `getpwent` repetitively, even if the calls are serialized, no thread will see all entries in the password file.

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

3.2.1 Thread Safety

A routine is called **thread-safe** if it can be called simultaneously from multiple threads without risk of corruption. Generally this means that it does some simple level of locking (possibly using the DECthreads global lock) to prevent simultaneously active calls in different threads. See Section 3.3.1 for information about the global lock.

Although these routines are thread-safe, they may be inefficient and are frequently inconvenient. For example, a UNIX `stdio` package that is thread-safe might still block all threads in the process while waiting to read or write data to a file. Routines such as `localtime(3)` return a pointer to static storage, so that callers do not have to deallocate the returned buffer. If one thread calls `localtime`, and then another thread calls `localtime` before the first thread has retrieved the returned value, both threads will retrieve the value returned by the second call. Some other routines maintain an implicit context so that a series of calls can be made to return sequential information, for example, the `strtok` routine returns a sequence of delimited tokens. Interleaved calls from multiple threads might disrupt that implicit context, even if the routine has protected itself against corruption from multiple simultaneous calls.

Programmers utilizing thread-safe code from multithreaded programs must be aware of these limitations and carefully take them into consideration. For example, when using a hypothetical thread-safe implementation of `localtime`, the caller must lock a mutex before calling the routine, and keep it locked until after the return value is processed or copied into local storage. In the case of `strtok`, the mutex must be held before retrieving the first token value of a string and until after the final token value has been safely copied or used.

3.2.2 Thread Reentrancy

A routine is **thread-reentrant** when it functions normally despite being called simultaneously or sequentially by different threads. Again, taking the hypothetical example of `strtok`, the traditional interface could most efficiently be made thread-reentrant by adding an argument that specifies a context for the sequence of tokens. Thus, multiple threads could simultaneously parse different strings without interfering with each other.

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

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

3.2.3 Working with Code that is not Thread-Safe

When you must call code that is not thread-safe, you need to ensure serialization and exclusivity of the unsafe routine across all threads in the program. Using the DECthreads global lock when calling any unsafe code accomplishes this. All threads and libraries using the routine can 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.

Furthermore, as when using thread-safe routines, you must be aware that in many cases you need to protect more than just the call itself. You need to use or copy any static return values before releasing the mutex, and you may need to protect a sequence of calls rather than just a single call.

If a routine is not specifically documented as thread-reentrant or thread-safe, you should assume that it is unsafe. You should never assume that a routine is fully thread-reentrant unless that is specifically documented; many times, routines can rely on static data in ways that are not completely obvious from the interface. A routine carefully written to be thread-reentrant but that calls some other routine that is unsafe without proper protection, is actually unsafe itself.

3.3 Converting Code to be Thread-Safe

Often, existing code can be converted from unsafe to thread-safe without a great deal of difficulty, if the source code is available. One important consideration is the language compiler used in translating the source code.

Note

Most Ada compilers generate inherently reentrant code because Ada supports multithreaded programming. Although the C, Pascal, and Bliss languages do not support multithreaded programming directly, compilers for those languages generally create reentrant code. However, the Fortran and COBOL languages are defined in such a way that they generally make implicit use of static storage, and some Fortran and COBOL compilers do not generate reentrant code. It is difficult to write reentrant code in a nonreentrant language.

As long as your compiler provides reentrant code sequences, DECthreads provides two tools that help convert existing code to be thread-safe—the global lock and thread-specific data.

3.3.1 DECthreads Global Lock

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

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

Use the global lock whenever calling unsafe routines—and when unsure, always assume a routine is unsafe. All DECthreads routines are thread-safe (and all, except some of the UNIX system jacket routines, are also thread-reentrant).

3.3.2 Thread-Specific Data

When an existing unsafe interface uses static storage for context information, and you need to make that interface thread-safe or thread-reentrant, one alternative for managing the static context is the use of thread-specific data. **Thread-specific data** consists of user-specified fields of arbitrary data that can be added to a thread's context.

Instead of using static storage, you can declare a thread-specific data key to maintain the address of a dynamically allocated record. Use `cma_once` or `pthread_once` routine to create the data key the first time the routine is called using `cma_once`. Use `cma_key_get_context` or `pthread_getspecific` to retrieve the current value of the key for that thread. If there is none (the value is NULL), allocate a new record and use `cma_key_set_context` or `pthread_setspecific` to store the address of the record. Thereafter, you can reuse that same record to maintain the static data and return the address of that record. Although this method is less efficient than passing explicit context, it allows the calls in different threads to be independent without changing interfaces. It also allows each thread to maintain an independent stream of information, as required by routines such as `strtok`.

3.4 Shared Variables

Most threads do not operate entirely 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 atomic hardware operations.

Most mechanisms for thread communication involve the use of shared memory, taking advantage of 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** (the areas of the program where code can access the memory) and **lifetime** (the length of time the memory exists) of each of the three types of memory.

3.4.1 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, `extern` variables can be accessed anywhere, and static variables can be referenced within the source module or routine, depending on where they are declared. Note that the static memory described in this section is not the same as the C language static storage class; static memory refers to any variable that is permanently allocated at a particular address for the life of the program.

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

3.4.2 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 may be accessible).

Generally, the storage is valid for the entire execution of the routine, and the actual address can be calculated and passed to other threads, but 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 may be altered by subsequent calls. The other threads will not be able to determine that this has happened, and erroneous behavior will result.

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

3.4.3 Dynamic Memory

Dynamic memory is allocated by the program as a result of a call to some memory management function (for example, the C language run-time function `malloc( )` or the OpenVMS common run-time function `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 (or, in a language supporting automatic garbage collection, it will exist until the run-time system detects that there are no references to it).

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

3.4.4 Use of Shared Memory

Although it can be difficult to safely share stack memory between multiple threads, both static and dynamic memory are almost essential to multithreaded programs.

Static memory is appropriate 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.

Dynamic memory is usually appropriate to manage persistent context. For example, in a thread-reentrant routine that is called multiple times to return a stream of information (for example, to list all active connections to a server or to return a list of users), using dynamic memory allows multiple contexts that are independent of threads. Multiple threads may be able to share a given context, or a single thread may have more than one context.

3.5 Stack Management

DECthreads protects the ends of stack memory and sets a default stack size that is acceptable to most applications. 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.

See the `cma_stack_check_limit_np` routine description for information on how to determine whether sufficient space exists on the current thread's stack to allocate the requested number of bytes of local storage. (There is no stack limit checking routine in the pthread interface to DECthreads.)

3.5.1 Stack Overflow

A program can receive a memory error (access violation, bus error, or segmentation fault) when it overflows its stack. It is often necessary to run the program under control of your system's debugger to determine where these errors occur. (However, if the debugger needs to allocate space on the stack, it may not function properly if the stack overflows.)

If a thread receives a memory access exception during a routine call or when accessing a local variable, increase the size of the stack. (To increase the thread's stack size attribute before creating it, call the `cma_attr_set_stacksize` or `pthread_attr_setstacksize` routine. See Section 2.3.3.4 for more information.) However, not all memory access exceptions indicate a stack overflow.

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

3.5.2 Sizing the Stack

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

3.6 Priority Inversion

Priority inversion occurs when interaction among three or more threads blocks the highest-priority thread from executing. For example, a high-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 high-priority thread is made to wait while a thread of lower priority (the middle-priority thread) executes.

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 raise its priority to that associated with the object.

The Default (throughput) scheduling policy prevents priority inversion from causing a complete blockage of the high-priority thread, because the low-priority thread is eventually permitted to execute and release the resource. The FIFO and RR policies, however, do not provide for resumption of the low-priority thread if the middle-priority thread executes indefinitely.

3.7 Using Synchronization Objects

The following sections discuss when to use a mutex and when to use a condition variable and the use of mutexes to prevent two potential problems: race conditions and deadlocks. This section also discusses why you should signal a condition variable with the associated mutex locked.

3.7.1 Mutex or Condition Variable

Use a mutex for tasks with fine granularity. Examples of fine-grained tasks are those that serialize access to shared memory or make simple modifications to shared memory (critical sections of a few program statements or less). Mutex waits are not interruptable—threads waiting to lock a mutex cannot be alerted or canceled.

A condition variable is not used to protect data. It is used to wait for data to assume a desired state. A condition variable is always used with a mutex that protects the shared data. Condition variable waits are interruptable.

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

3.7.2 Race Conditions

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

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

1. Thread A increments variable X.
2. Thread A is interrupted by Thread B.
3. Thread B starts and increments variable X.
4. Thread B is interrupted by Thread A.
5. Thread A checks the value of X and takes some action based on that value.

   The value of X is different from what it was when Thread A incremented it, and the program's behavior is incorrect.

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

See Section 3.7.4 for another example of a race condition.

3.7.3 Deadlocks

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

Following are two techniques you can use to avoid deadlocks:

- Use sequence numbers with fast mutexes.

  Associate a sequence number with each mutex and lock mutexes in sequence. Never attempt to lock a mutex with a sequence number lower than that of a mutex the thread already holds.

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

- Use a recursive mutex.

  This method is useful when a thread needs to lock the same mutex more than once before unlocking it. This technique can help prevent a thread from deadlocking on itself.

3.7.4 Signaling a Condition Variable

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

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

```
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_destory routine before Thread A resumes execution.

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

The previous code fragments also demonstrate a race condition. The program depends on a sequence of events among multiple threads, but it does not enforce the desired sequence. Signaling the condition variable while holding the associated mutex eliminates the race condition. That 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);
```

3.8 DECthreads Error Reporting

DECthreads can detect the following types of errors:

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

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

- The POSIX 1003.4a (pthread) interface returns a function value of −1 and sets the per-thread *errno* variable to an error code.

- The CMA interface and the "pthread exception" interface raise exceptions to indicate error conditions.

DECthreads internal errors result in a bugcheck. DECthreads writes a message to the current error device (UNIX stderr or OpenVMS SYS$ERROR) summarizing the problem, and creates a file containing more detailed information. By default, the file is named cma_dump.log and is created in the current default directory. You can redirect the information to a different file by defining an OpenVMS logical name or a UNIX environment variable CMA_DUMP before running any program that uses DECthreads.

The following example shows how to define an OpenVMS logical name:

```
$ define cma_dump sys$manager:server_001.thd_dmp
```

You can define a UNIX (Korn Shell) environment variable with a command such as the following:

```
$ CMA_DUMP=/server_001.thd_dmp;export CMA_DUMP
```

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

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

```
%Internal DECthreads problem (version V2.12-078), terminating execution.
```

The subsequent line is the reason for the failure, and the final line written to the error device (usually) is the location of the detailed state information, as in the following example:

```
% See 'sys$manager:server_001.thd_dmp' for state information.
```

The detailed information file contains information you can get from the cma_debug( ) interface. This information is usually necessary to track down the problem. If you submit a problem report involving a DECthreads bugcheck, please include this information file along with sample code and output.

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

One common example is the use of any mutex operation or certain condition variable operations from within a UNIX signal handler or OpenVMS AST routine. This type of programming error most commonly results in bugchecks reporting "enter_kernel: deadlock" or "Can't find null thread". To prevent this error, avoid using any condition variables operations other than the following:

- `cma_cond_signal_int`
- `cma_cond_signal_preempt_int`
- `pthread_cond_signal_int_np`
- `pthread_cond_sig_preempt_int_np`

In addition, DECthreads maintains a variety of state information in memory which is writable by user mode code. Therefore, it is possible for applications to accidentally modify DECthreads state by writing through invalid pointers, which can result in a bugcheck or other undesirable behavior.

4

Using the DECthreads Exception Package

DECthreads provides interfaces that report errors in the following two ways:

- The routine returns a status value to the thread.
- The routine raises an exception.

Note

The DECthreads cma routines raise exceptions only. However, DECthreads provides two versions of the pthread routines: one that raises exceptions and one that returns status values.

Before you write a multithreaded program using the pthread routines, you must choose only one of the previous two methods of receiving status. These two methods cannot be used together in the same source code module.

The DECthreads exception package is most useful when you are programming in the C language. The exception-returning interfaces are only usable from languages other than C when run on OpenVMS systems. (On OpenVMS systems you can use the OpenVMS condition-handling facility to catch exceptions.)

The POSIX P1003.4a standard specifies that status values be returned to the thread. The pthread reference section of this manual documents that status value-returning interface. However, an alternative to status values is provided by DECthreads in the pthread exception-returning interface. The cma interface also reports errors by raising exceptions.

This chapter introduces and provides conventions for the modular use of the exception-returning interface.

4.1 Invoking the pthread Exception-Returning Interface

To invoke the exception-returning interface for pthread calls, replace `#include <pthread.h>` in your program with the following include statement:

```
#include <pthread_exc.h>
```

Access to exceptions from the C language is defined by the macros in the `exc_handling.h` file, which is automatically included by `cma.h` and `pthread_exc.h`.

4.2 Overview of Exceptions

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

Using a few simple macros, C functions can declare a block of code (an **exception scope**) where exceptions are to be caught, and can define a block of code within an exception scope to process a specific exception (or all exceptions). DECthreads exception handlers are **attached**, which means that the handler code appears within the block where exceptions are caught. This allows you to see what actions will be taken when an exception occurs.

There are two ways to process an exception that occurs within the exception scope:

- The exception is **caught**. This means that the code handles all effects of the error and continues normal operation.
- The scope is **finalized**. This means that the current context is cleaned up and resources (such as mutexes) are released. The exception is then passed on 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.

4.2.1 Types of Exceptions

There are two types of exceptions: **address** exceptions and **status** exceptions. An exception is initialized as an address exception, but it can be modified (before it is used) by defining a status value for it. Following are the primary differences between address and status exceptions:

- Different exception objects can be declared with the same status value, and those exceptions are considered identical by the exception package. For example, if one exception is raised, it can be caught by specifying another

exception object having the same status. Two different address exceptions can never match each other.

- If the platform supports a universal definition of error status, then status exceptions can often be used to import and export system status values. When a facility called by DECthreads raises a system exception, DECthreads and its clients can catch the exception using a DECthreads status exception. Similarly, when a function raises a DECthreads exception, a caller might be able to handle it using facilities provided by the language or platform.

Status values used in exceptions can be interpreted, handled, and reported in a universal manner, regardless of which facility defined the status value. Use address exceptions if your code does not have a range of status codes assigned to it. Address exceptions are always unique so you do not risk colliding with another facility's status codes and inadvertently handling the wrong exception. Also, address exceptions are more portable because status codes are likely to be different on each platform.

4.2.2 Terminating Exception Semantics

DECthreads exceptions are terminating exceptions. This means that control never returns to the instruction following a `RAISE` statement. When an exception occurs because of a hardware condition such as an illegal address, execution cannot be resumed at the failing instruction. An exception causes execution of handlers that have been declared (starting with the most recently declared handler and proceeding backwards) until a `CATCH` or `CATCH_ALL` clause is reached that does not end with `RERAISE`. At this point, execution continues at the first statement beyond the `ENDTRY` that terminates that current handler.

4.3 Exception Operations

The DECthreads Exception Package allows you to perform the following operations on exceptions:

- Declare and initialize an exception object
- Raise an exception
- Define a region of code over which exceptions are caught
- Catch a particular exception or all exceptions
- Reraise the current exception
- Define epilogue actions for a block
- Import a system-defined error status into the program as an exception

- Extract a system-defined error status from an exception
- Report an exception
- Determine whether two exceptions match

These operations are discussed in the following sections.

4.3.1 Declaring and Initializing an Exception Object

An exception object is an opaque type which should only be manipulated by the exception package functions. The actual contents of the type may differ from one implementation to another.

Declaring and initializing an exception object documents that a program reports or handles a particular error. Having the error expressed as an exception object provides future extensibility as well as portability.

An exception is declared as a variable of type `EXCEPTION`. In general, you should declare the type as `static` or `extern`. For example:

```
static EXCEPTION an_error;
```

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

Following is an example of declaring and initializing an exception object:

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

4.3.2 Raising an Exception

Raising an exception reports an error not by returning a value, but by propagating the exception. Propagation involves searching all active scopes for code written to handle the error or code written to perform finalization actions in case of any error and causing that code to execute. If a scope does not define a handler or finalization block, then the scope is simply torn down as the exception propagates up the stack. This is sometimes referred to as **unwinding** the stack. Because DECthreads exceptions are terminating, there is no option to make execution resume at the point of the error. (Execution resumes at the point where the exception is caught.)

If an exception is unhandled, the process is terminated with a core dump (on systems based on UNIX software) or traceback (on OpenVMS systems).

Termination prevents the unhandled error from affecting other areas of the program. An example of raising an exception is as follows:

```
RAISE (parity_error);
```

4.3.3 Defining a Region of Code over which Exceptions are Caught

The TRY macro defines the beginning of an exception scope, and the ENDTRY macro defines the end of the scope. These macros allow the programmer to define a scope (a block) wherein exceptions can be caught. Any exceptions raised within the block, or within any functions that are called directly or indirectly by the block, pass through the control of this scope. These exceptions can be caught and reraised if it is desirable to continue propagation, or ignored (which implicitly reraises them).

Following is an example of defining an exception-handling region (without indicating any recovery actions):

```
TRY {
   read_tape ();
}
ENDTRY
```

4.3.4 Catching a Particular Exception

The exception scope can express interest in any number of specific exceptions by naming them in CATCH expressions. When an exception reaches the exception scope, control is transferred to the first CATCH clause in the block that matches the exception. If there is more than one CATCH for a given exception within the scope of a single TRY/ENDTRY scope, then only the first one matching the current exception gains control.

To catch an address exception, the CATCH macro must specify the same exception object as the 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. In general, you should RAISE and CATCH using the same exception object even when using status exceptions.

Following is an example of catching a particular exception and specifying the recovery action (in this case, a message). After catching the exception and

executing the recovery action, the exception is explicitly reraised (causing it to propagate to its callers):

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

4.3.5 Catching All Exceptions

The exception scope can express interest in all exceptions using the CATCH_ALL macro. No CATCH macros can follow the CATCH_ALL macro within an exception scope.

Any exception that is caught using a CATCH_ALL macro should be reraised. It is inappropriate to absorb exceptions that your code is not explicitly aware of. Because you cannot necessarily predict all possible exceptions that your code might encounter, you cannot assume that your code can recover in every possible situation. Therefore, your CATCH_ALL clause should reraise all exceptions to allow an outer scope to catch this specific exception and perform the appropriate recovery.

Following is an example of the CATCH_ALL macro.

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

4.3.6 Reraising the Current Exception

Within the code block of a CATCH or CATCH_ALL macro, you can use RERAISE to allow outer exception scopes the chance to handle the exception. Do this when the current scope needs to restore some permanent state (for example, releasing resources such as memory or a mutex), but does not have enough context about the error to attempt to recover.

The RERAISE function is only valid in the code of a CATCH or CATCH_ALL clause. For example:

```
int *local_mem;
local_mem = malloc (sizeof (int));

TRY {
   operation(local_mem);      /* May raise an exception */
   free (local_mem);
}
CATCH (an_error) {
   free (local_mem);
   RERAISE;
}
```

4.3.7 Defining Epilogue Actions for a Block

Frequently, the only reason a block of code needs to catch exceptions is to perform cleanup actions, such as releasing resources. In many cases, the same operations are performed whether the block exits normally or with an exception; under many exception models, this requires duplicating code (both within a CATCH_ALL type construct, and following the exception scope in case an exception does not occur).

The FINALLY macro catches an exception and then reraises the exception for outer scopes to handle. The actions defined by a FINALLY clause are also performed when the scope exits normally without an exception, so that they do not need to be duplicated.

Do not combine the FINALLY clause with CATCH or CATCH_ALL. Doing so results in unpredictable behavior.

Following is an example of the FINALLY macro:

```
int *local_mem;
local_mem = malloc (sizeof (int));
TRY {
   operation(local_mem);      /* May raise an exception */
}
FINALLY {
   free (local_mem);
}
ENDTRY
```

4.3.8 Determining the Current Exception

The current exception object can be referenced within a CATCH or CATCH_ALL block by using the name THIS_CATCH. The exception object THIS_CATCH has a type of EXCEPTION *. This value can be passed to exc_get_status( ), exc_report( ), or exc_matches( ) (defined in Section 4.3.10, Section 4.3.11, and Section 4.3.12).

Because of the way exceptions are propagated, the address contained in THIS_CATCH might not be the actual address of an address exception; if you need to match THIS_CATCH against known exceptions, use exc_matches( ).

4.3.9 Importing a System-Defined Error Status into the Program as an Exception

The exc_set_status( ) function can be used to create a status exception. The exception object must already have been initialized with EXCEPTION_INIT. Any system specific status value may be used. All exception objects set to the same status value are considered the same by the exception facility.

An example of importing an error status into an exception is as follows:

```
void exc_set_status (EXCEPTION *exception,
       unsigned int code);

static EXCEPTION an_error;

EXCEPTION_INIT (an_error);
exc_set_status (&an_error, ENOMEM);
```

4.3.10 Exporting a System-Defined Error Status

The exc_get_status( ) function can be used to retrieve the system status value from a status exception, for example, after an exception is caught. If the exception object specified is a status exception, exc_get_status( ) sets the status value argument and returns 0; otherwise, it returns –1 and does not set the status value argument. For example:

```
int exc_get_status (EXCEPTION *exception,
       unsigned int *status);
```

```
TRY {
   operation ();
    }
CATCH_ALL {
   int  status;
   if (exc_get_status (THIS_CATCH, &status) == 0
        && status < sys_nerr)
     fprintf (stderr, "%Exception %s\n",
        strerror (status));
   else
     exc_report (THIS_CATCH);
   }
ENDTRY
```

4.3.11 Reporting an Exception

DECthreads reports an exception only when it is raised without a CATCH or CATCH_ALL, immediately before the process is terminated. Sometimes client code might wish to report an exception as part of error recovery. The exc_report( ) function prints a message to stderr or SYS$ERROR describing the exception.

All predefined exceptions have an associated message describing the error. Normally, when the DECthreads exception package has been well-integrated with a platform status mechanism, external status values can also be reported. However, when an address exception is reported, DECthreads can only report the fact that an exception has occurred, and the address of the exception object.

Following is an example of reporting an error:

```
void exc_report (EXCEPTION *exception);
```

For example:

```
exc_report (&exc_e_illinstr);
```

4.3.12 Determining Whether Two Exceptions Match

The exc_matches( ) function compares two exception objects, taking into consideration whether they are address or status exceptions, and possibly other system-specific rules for matching status values. Whenever you need to compare two exceptions, you should use this function. For example:

```
int exc_matches (EXCEPTION *exception1,
      EXCEPTION *exception2);

EXCEPTION my_status;
```

```
EXCEPTION_INIT (&my_status);
exc_set_status (&my_status, status_code);
   .
   .
   .
if (exc_matches (THIS_CATCH, &my_status))
   fprintf (stderr, "This is my exception\n");
```

4.4 C Language Syntax

The following example shows the syntax for handling exceptions:

```
TRY
    try_block
[CATCH (exception_name)
    handler_block]...
[CATCH_ALL
    handler_block]
ENDTRY
```

A **try_block** or a **handler_block** is a sequence of statements, the first of which may be declarations, as in a normal block. If an exception is raised in the try_block, the catch clauses are evaluated in order to see if any one matches the current exception.

The CATCH or CATCH_ALL clauses absorb an exception—they can catch an exception propagating out of the try_block and direct execution into the associated handler_block. Propagation of the exception, by default, then ends. Within the lexical scope of a handler, it is possible to cause propagation of the same exception to resume (this is called reraising the exception), or it is possible to raise some new exception.

The RERAISE statement is allowed in any handler statements and causes the current exception to be reraised. Propagation of the caught exception resumes.

The RAISE (exception_name) statement is allowed anywhere and causes a particular exception to start propagating. For example:

```
TRY
    sort(); /* Call a function that may raise an exception.
             * An exception propogates by transferring control
             * out of some nested routine back to the TRY
             * clause.  Any output parameters or return values
             * of the called routine are therefore indeterminate.
             */

CATCH (pthread_cancel_e)
    printf("Canceled while sorting\n");
    RERAISE;
```

```
CATCH_ALL
    printf("Some other exception while sorting\n");
    RERAISE;

ENDTRY
```

In the previous example, if the pthread_cancel_e exception propagates out of the function call, the first printf is executed. If any other exception propagates out of sort, the second printf is executed. In either situation, propagation of the exception resumes because of the RERAISE statement. (If the code is unable to fully recover from the error, or does not understand the error, it needs to do as in the previous example and further propagate the error to its callers.)

The following example shows the syntax for an epilogue:

```
TRY
    try_block
FINALLY
    final_block
ENDTRY
```

The final_block is executed regardless of whether the try_block executes to completion without raising an exception or if an exception is raised in the try_block. If an exception is raised in the try_block, propagation of the exception is resumed after executing the final_block.

Note that a CATCH_ALL handler and RERAISE could be used to do this, but the epilogue code would then have to be duplicated in two places, as follows:

```
TRY
    try_block
CATCH_ALL
    final_block
    RERAISE;
ENDTRY
{ final_block }
```

A FINALLY statement has exactly this meaning but avoids code duplication.

Note

The behavior of FINALLY along with CATCH or CATCH_ALL clauses is unpredictable. Do not combine them for the same try_block.

Another example of the FINALLY statement is as follows:

```
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);
    /* 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 the previous example, the call to pthread_cond_wait could raise the pthread_cancel_e exception if the thread was canceled while it was waiting. The final_block ensures that the shared data associated with the lock is correct for the next thread that acquires the mutex.

4.5 Rules and Conventions for Modular Use of Exceptions

The following rules ensure that exceptions are used in a modular way (so that independent software components can be written without requiring knowledge of each other):

- Use unique names for exceptions.

  A naming convention ensures that the names for exceptions that are declared EXTERN from different modules do not clash. The following conventions are recommended.

  For the cma interface:

  <facility-prefix>_e_<error-name>

  For example, cma_e_alerted.

  For the pthread interface:

  <facility-prefix>_<error-name>_e

  For example, pthread_cancel_e.

- Avoid putting code in a TRY macro that belongs before it.

  The TRY macro should only guard statements for which the statements in the FINALLY or CATCH, or CATCH_ALL clauses are always valid to execute.

A common misuse of TRY is to put code in the try_block that should be placed before TRY. An example of this misuse is as follows:

```
TRY
    handle = open_file (file_name);
    /* Statements that may raise an exception here */
FINALLY
    close (handle);
ENDTRY
```

The previous FINALLY code assumes that no exception is raised by open_file. Otherwise, the code would access an invalid identifier in the FINALLY part if open_file is modified to raise an exception. The previous example should be rewritten as follows:

```
handle = open_file (file_name);
TRY
    /* Statements that may raise an exception here */
FINALLY
    close (handle);
ENDTRY
```

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

- Raise exceptions prior to performing side-effects.

  Write functions that propagate exceptions to their callers so that the function does not modify any persistent process state before raising the exception. A call to the matching close call is required only if the open 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 open_file raises an exception, the identifier will not have been written, so open must not require that close be called when open raises an exception. This property is also what allows the call to be moved to open_file prior to the TRY.

- Do not place a return or go-to between TRY and ENDTRY.

  It is invalid to return or go-to or leave by some other means a TRY, CATCH, CATCH_ALL, or FINALLY block. Special code is generated by the ENDTRY macro and it must be executed.

- Use the ANSI C volatile attribute.

  Variables that are read or written by exception handling code must be declared with the ANSI C volatile attribute. Run your tests with the optimize compiler option to ensure that the compiler thoroughly tests your exception handling code.

- Reraise exceptions that are not fully handled.

  Reraise any exception that you catch, unless your handler has performed the complete recovery action for the error. This rule permits an unhandled exception to propagate to some final default handler that knows how to recover fully.

  A corollary of this rule is that `CATCH_ALL` handlers must reraise, since they may catch any exception, and usually cannot do recovery actions that are proper for every exception.

  Following this convention is important so that you also do not absorb a cancel or thread-exit request. These are mapped into exceptions so that exception handling has the full power 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.)

- Declare only static exceptions.

4.6 Pthread Exceptions and Definitions

Table 4–1 lists the pthread exceptions and briefly explains the meaning of each exception.

Exception names beginning with the prefix pthread_ or cma_ are raised as the result of something happening internal to the DECthreads facility and are not meant to be raised by user code. Exceptions beginning with exc_ are generic and belong to the exception facility and/or the underlying system.

See Appendix C for a list and descriptions of the cma interface exceptions. Most pthread exceptions correspond directly to a cma interface exception.

Table 4–1 Pthread Exceptions

Exception	Definition
pthread_cancel_e	Thread cancelation in progress
pthread_existence_e	Object referenced does not exist
pthread_in_use_e	Object referenced is already in use
pthread_use_error_e	Requested operation is improperly invoked
pthread_badparam_e	Improper parameter

(continued on next page)

Table 4–1 (Cont.) Pthread Exceptions

Exception	Definition
pthread_stackovf_e	Attempted stack overflow was detected
pthread_nostackmem_e	No space is currently available to create a new stack
pthread_unimp_e	Unimplemented feature
pthread_notstack_e	The current stack was not created by DECthreads
exc_illaddr_e	Data or object could not be referenced
exc_exquota_e	Operation failed due to insufficient quota
exc_insfmem_e	Insufficient virtual memory for requested operation
exc_nopriv_e	Insufficent privilege for requested operation
exc_uninitexc_e	Uninitialized exception raised
exc_SIGILL_e	Unhandled illegal instruction signal
exc_SIGTRAP_e	Unhandled trace or breakpoint trap signal
exc_SIGIOT_e	Unhandled IOT trap signal
exc_SIGEMT_e	Unhandled EMT trap signal
exc_SIGFPE_e	Unhandled floating point exception signal
exc_SIGBUS_e	Unhandled bus error signal
exc_SIGSEGV_e	Unhandled segmentation violation signal
exc_SIGSYS_e	Unhandled bad system call signal
exc_SIGPIPE_e	Unhandled broken pipe signal
exc_SIGXCPU_e	Unhandled CPU-time limit exceeded signal
exc_SIGXFSZ_e	Unhandled file-size limit exceeded signal
exc_intovf_e	Unhandled integer overflow trap exception
exc_intdiv_e	Unhandled integer divide by zero trap exception
exc_fltovf_e	Unhandled floating point overflow trap exception
exc_fltdiv_e	Unhandled floating point/decimal divide by zero trap exception
exc_fltund_e	Unhandled floating point underflow trap exception
exc_decovf_e	Unhandled decimal overflow trap exception
exc_subrng_e	Unhandled subscript out of range trap exception
exc_resaddr_e	Unhandled reserved addressing fault exception

(continued on next page)

Table 4–1 (Cont.) Pthread Exceptions

Exception	Definition
exc_privinst_e	Unhandled privileged instruction fault exception
exc_resoper_e	Unhandled reserved operand fault exception

5

DECthreads Examples

This chapter contains two examples that show the use of the DECthreads pthread routines from programs written in the C language. Example 5–1 uses the status-returning interface (the default) to perform a prime number search. Example 5–2 uses the exception-returning interface to DECthreads to demonstrate an asynchronous user interface.

5.1 Prime Number Search Example

Example 5–1 shows the use of the DECthreads pthread 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 if it is a prime, records it if it is prime, and then takes another number—and so on.

This program shows the work crew model of programming (see Section 1.4.2.) The workers (threads) increment a number (`current_num`) in order to get their next work assignment. As a whole, the worker threads are responsible for finding a specified number of prime numbers, at which point their work is completed.

The number of workers to be used and the requested number of prime numbers to be found are defined constants. A macro is used to check for error status and to print a given string and the associated error value. Data to be accessed by all threads (mutexes, condition variables, and so forth) are declared as global items.

Worker threads execute the prime search routine, which begins by synchronizing with the parent thread utilizing a predicate and a condition variable. Enclose a condition wait in a predicate loop to prevent a 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. The same mutex must be used for all operations performed on a specific condition variable.

After the parent sets the predicate and broadcasts, the workers begin finding prime numbers until canceled by a fellow worker who has found the last requested prime number. Upon each iteration the workers increment the current number to be worked on and take the new value as their work item. A mutex is locked and unlocked around getting the next work item, in order 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.

Each worker thread then determines if its current work item (number) is prime by trying to divide numbers into it. If the number proves to be nondivisible, it is put on the list of primes. Cancels are turned off while working with list of primes in order to better control any cancels that do occur. The list of primes and its current count are protected by locks, which also protect the cancelation process of all other worker threads upon finding the last requested prime. While still under the prime list lock, the current worker checks to see if it has found the last requested prime, and, if so, unsets a predicate and cancels all other worker threads. Cancels are then reenabled. 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.

  Setting up of the environment means initializing mutexes and the one condition variable used in the example.

- Create worker threads.

  Creation of worker threads is straightforward and utilizes the default attributes (`pthread_attr_default`). Worker threads immediately wait on a condition variable.

- Broadcast to the worker threads that they may start.

- Join each thread as it finishes.

  As the parent joins each of the returning worker threads, it receives an exit value from them which indicates whether a thread exited normally or not. In this case the exit values on all but one of the worker threads should be –1, indicating that they were canceled.

- Sort and print the list of primes.

The following pthread routines are used in Example 5–1:

pthread_cancel
pthread_cond_broadcast
pthread_cond_init
pthread_cond_wait

pthread_create
pthread_detach
pthread_exit
pthread_join
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_unlock
pthread_setcancel
pthread_testcancel

Example 5–1 C Program Example (Prime Number Search)

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

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

/*
 * Constants used by the example.
 */

#define     workers     5           /* Threads to perform prime check  */
#define     request     110         /* Number of primes to find        */

/*
 * Macros
 */

#define check(status,string) \
    if (status == -1) perror (string)

/*
 * Global data
 */
```

(continued on next page)

Example 5–1 (Cont.) C Program Example (Prime Number Search)

```
pthread_mutex_t prime_list;     /* Mutex for use in accessing the prime     */
pthread_mutex_t current_mutex;  /* Mutex associated with current number     */
pthread_mutex_t cond_mutex;     /* Mutex used for ensuring CV integrity     */
pthread_cond_t  cond_var;       /* 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 (pthread_addr_t arg)
    {
    int     status;                 /* Hold status from pthread calls   */

    status = pthread_mutex_unlock (&cond_mutex);
    check(status,"3:Mutex_unlock bad status\n");
    }

/*
 * 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 (pthread_addr_t arg)
    {
    div_t   div_results;            /* DIV results: quot and rem        */
    int     numerator;              /* Used for determing primeness     */
    int     denominator;            /* Used for determing 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              */

    my_number = (int)arg;

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

(continued on next page)

Example 5–1 (Cont.) C Program Example (Prime Number Search)

```
status = pthread_mutex_lock (&cond_mutex);
check(status,"1:Mutex_lock bad status\n");

pthread_cleanup_push (unlock_cond, NULL);

    while (thread_hold) {
        status = pthread_cond_wait (&cond_var, &cond_mutex);
        check(status,"2:Cond_wait bad status\n");
        }

pthread_cleanup_pop (1);

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

while (not_done) {

    /* Cancelation point */
    pthread_testcancel ();

    /* Get next integer to be checked */
    status = pthread_mutex_lock (&current_mutex);
    check(status,"4:Mutex_lock bad status\n");
    current_num = current_num + 2;              /* Skip even numbers */
    numerator = current_num;
    status = pthread_mutex_unlock (&current_mutex);
    check(status,"5:Mutex_unlock bad status\n");

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

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

    if (prime != 0) {

        /* Explicitly turn off all cancels */
        pthread_setcancel(CANCEL_OFF);

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

(continued on next page)

Example 5–1 (Cont.) C Program Example (Prime Number Search)

```
            status = pthread_mutex_lock (&prime_list);
            check(status,"6:Mutex_lock bad status\n");

            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,"12:Cancel bad status\n");
                        }
                    }
                }

            status = pthread_mutex_unlock (&prime_list);
            check(status,"13:Mutex_unlock bad status\n");

            /* Reenable cancels */
            pthread_setcancel(CANCEL_ON);
            }

        pthread_testcancel ();
        }

    pthread_exit ((pthread_addr_t)my_number);
    }

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     not_done;       /* Indicates swap made in sort          */

    /*
     * Create mutexes
     */
```

(continued on next page)

Example 5–1 (Cont.) C Program Example (Prime Number Search)

```
    status = pthread_mutex_init (&prime_list, pthread_mutexattr_default);
    check(status,"7:Mutex_init bad status\n");
    status = pthread_mutex_init (&cond_mutex, pthread_mutexattr_default);
    check(status,"8:Mutex_init bad status\n");
    status = pthread_mutex_init (&current_mutex, pthread_mutexattr_default);
    check(status,"9:Mutex_init bad status\n");

    /*
     * Create conditon variable
     */

    status = pthread_cond_init (&cond_var, pthread_condattr_default);
    check(status,"10:Cond_init bad status\n");

    /*
     * Create the worker threads.
     */

    for (worker_num = 0; worker_num < workers; worker_num++) {
        status = pthread_create (
            &threads[worker_num],
            pthread_attr_default,
            prime_search,
            (pthread_addr_t)worker_num);
        check(status,"11:Pthread_create bad status\n");
    }

    /*
     * 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,"12:Mutex_lock bad status\n");

    thread_hold = 0;
    status = pthread_cond_broadcast (&cond_var);

    status = pthread_mutex_unlock (&cond_mutex);
    check(status,"13:Mutex_unlock bad status\n");

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

(continued on next page)

Example 5–1 (Cont.) C Program Example (Prime Number Search)

```
    for (worker_num = 0; worker_num < workers; worker_num++) {
        status = pthread_join (
            threads[worker_num],
            &exit_value );
        check(status,"14:Pthread_join bad status\n");

        if (exit_value == (pthread_addr_t)worker_num)
            printf("Thread terminated normally\n");

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

        status = pthread_detach ( &threads[worker_num] );
        check(status,"15:Pthread_detach bad status\n");
        }

   /*
    * 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.
    */

    not_done = 1;
    for (index1 = 1; ((index1 < request) && (not_done)); index1++) {
        for (index2 = 0; index2 < index1; index2++) {
            if (primes[index1] < primes[index2]) {
                temp = primes[index2];
                primes[index2] = primes[index1];
                primes[index1] = temp;
                not_done = 0;
                }
            }
        }

    /*
     * Print out the list of prime numbers that the worker threads
     * found.
     */

    printf ("The list of %d primes follows:\n", request);
    printf("%d",primes[0]);
    for (list = 1; list < request; list++) {
        printf (",\t%d", primes[list]);
        }
    printf ("\n");
```

(continued on next page)

Example 5–1 (Cont.) C Program Example (Prime Number Search)

```
}
```

5.2 Asynchronous User Interface Example

Example 5–2 shows the use of the DECthreads pthread routines in a C program that features a text-based asynchronous user interface. This interface reads and writes commands to the terminal.

This example uses the DECthreads exception package to allow the pthread routines to report errors by raising exceptions rather than by returning status values.

Example 5–2 allows you to start multiple commands that run concurrently and report their results when complete. You can monitor the status of running commands or cancel the commands.

Asynchronous Commands

The asynchronous commands are `date` and `time`. These commands cause the program to wait a certain number of seconds before executing. The number of seconds is the argument to the command; for example, `time 10` causes the program to wait 10 seconds before reporting the time.

Housekeeping Commands

The housekeeping commands are as follows:

- `status`

  Displays the state of a command.

- `wait`

  Waits for a command to finish.

- `cancel`

  Stops a command.

The argument to these commands is the number of the command (assigned and displayed when the asynchronous command starts). This program is limited to four outstanding commands.

The following pthread routines are used in Example 5–2:

pthread_cancel
pthread_cond_init

pthread_cond_signal
pthread_cond_wait
pthread_create
pthread_delay_np
pthread_detach
pthread_join
pthread_lock_global_np
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_unlock
pthread_once
pthread_unlock_global_np
pthread_yield

Example 5–2 C Program Example (Asynchronous User Interface)

```
/*
 *
 * DECthreads example program featuring an asynchronous user interface
 *
 */

/*
 * Include files
 */
#include <pthread_exc.h> ❶
#undef printf ❷
#undef gets
#include <stdio.h>
#include <time.h>

/*
 * Local definitions
 */
#define PROMPT     "Info> "            /* Prompt string */
#define MAXLINSIZ   81                 /* Command line size */
#define THDNUM       5                 /* Number of server threads */
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * 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       0
#endif

/*
 * Global variables
 */
struct THREAD_DATA {
    pthread_t       thread_handle;      /* Server thread handle */
    pthread_mutex_t mutex_handle;       /* 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_cond_t      free_thread_cv;     /* Condition variable for same */
int                 free_thread;        /* Flag indicating a free thread */

/*
 * Local Routines
 */
static void
dispatch_task (pthread_startroutine_t routine, char task, int time);

static void
do_cancel (int index);

static void
do_cleanup (int index, int final_state);

static pthread_addr_t
do_date (pthread_addr_t arg);

static void
do_delay (int index);
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
static void
do_status (int index);

static pthread_addr_t
do_time (pthread_addr_t arg);

static void
do_wait (int index);

static int
find_free_thread (int *index);

static char *
get_cmd (char *buffer);

static int
get_y_or_n (char *query, char defans);

static void
init_routine (void);

static void
print_error (char *message);

static void
print_help (void);

static void
this_shouldnt_happen (char *message);

/*
 * The main program:  Description TBS.
 */
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 */
    pthread_startroutine_t  routine;    /* Routine to execute in a thread */
    static pthread_once_t   once_block = pthread_once_init; ❸

    /*
     * Perform program initialization.
     */
    pthread_once (&once_block, init_routine); ❹
```

(continued on next page)

Example 5–2 (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.
    */
    pthread_yield (); ❺
    if (get_cmd(cmdline)) {
        cmd_cnt = sscanf (cmdline, "%s %d", cmd_wd, &cmd_arg);
        routine = (pthread_startroutine_t )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;
                    }
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * These commands require an argument
 */
case 'c':   /* "Cancel" */
case 'd':   /* "Date" */
case 't':   /* "Time" */
case 'w':   /* "Wait" */
    {
    if (cmd_cnt != 2) {
        print_error ("Missing command argument.");
        }
    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;
                }
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
                            case 'w':           /* "Wait" */
                                {
                                do_wait (cmd_arg);
                                break;
                                }
                                default:
                                {
                                this_shouldnt_happen("Inner switch.");
                                break;
                                }
                            }
                        }
                    break;
                    }
                default:
                    {
                    print_error ("Unrecognized command.");
                    break;
                    }
                }
            }
        else if (cmd_cnt != EOF)              /* Ignore blank command line */
            this_shouldnt_happen ("Unexpected parse error.");

        /*
         * 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 (pthread_startroutine_t routine, char task, int time)
    {
    int     i;                /* Index of free thread slot */
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    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.
         */
        pthread_mutex_lock (&thread_data[i].mutex_handle);
        thread_data[i].time = time;
        thread_data[i].task = task;
        thread_data[i].state = ST_RUNNING;
        pthread_mutex_unlock (&thread_data[i].mutex_handle);

        /*
         * Create the thread, using the default attributes.  The thread will
         * execute the specified routine and get its data from array slot 'i'.
         */
        pthread_create (
                &thread_data[i].thread_handle,
                pthread_attr_default,
                routine,
                (pthread_addr_t)i);

        pthread_lock_global_np (); ❻
        printf ("This is command #%d.\n\n", i);
        pthread_unlock_global_np ();
        }
    }

/*
 * Wait for the completion of the specified command.
 */
static void
do_cancel (int index)
    {
    int cancelable;

    if ((index < 0) || (index >= THDNUM))
    print_error ("Bad command number.");
    else {
        pthread_mutex_lock (&thread_data[index].mutex_handle);
        cancelable = (thread_data[index].state == ST_RUNNING);
        pthread_mutex_unlock (&thread_data[index].mutex_handle);
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
        if (cancelable)
            pthread_cancel (thread_data[index].thread_handle); ❼
        else
            print_error ("Command is not active.");
        }
    }
/*
 * Post-task clean-up routine.
 */
static void
do_cleanup (int index, int final_state)
    {
    /*
     * 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".
     */
    pthread_mutex_lock (&free_thread_mutex); ❽

    /*
     * Mark the thread as finished with its task.
     */
    pthread_mutex_lock (&thread_data[index].mutex_handle);
    thread_data[index].state = final_state;
    pthread_mutex_unlock (&thread_data[index].mutex_handle);

    /*
     * Set the flag indicating that there is a free thread, and signal the
     * main thread, in case it is waiting.
     */
    free_thread = TRUE; ❾
    pthread_cond_signal (&free_thread_cv);
    pthread_mutex_unlock (&free_thread_mutex);
    }
```

(continued on next page)

Example 5–2 (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 pthread_addr_t
do_date (pthread_addr_t 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);

        pthread_lock_global_np (); ❿
        TRY {
            clock_time = time ((time_t *)0); ⓫
            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);
            }
        FINALLY { ⓬
            pthread_unlock_global_np ();
            }
        ENDTRY
        }
    CATCH (pthread_cancel_e) { ⓭
        pthread_lock_global_np ();
        printf ("%d) Canceled.\n", arg);
        pthread_unlock_global_np ();
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
        /*
         * Perform exit actions
         */
         do_cleanup ((int)arg, ST_CANCELED);

        RERAISE;
        }
    CATCH_ALL {
        pthread_lock_global_np ();
        printf ("%d) ", arg);
        exc_report (THIS_CATCH); ⓮
        pthread_unlock_global_np ();

        /*
         * 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 doesn't check the
     * value, however.
     */
    return arg;
    }

/*
 * Delay routine
 *
 * Since the actual tasks that threads do in this program take so little time
 * to perform, execute a delay to make it seem like they are taking a long
 * time.  Also, this will give the user something to query the progress of.
 */
static void
do_delay (int index)
    {
    static struct timespec interval = {1, 0}; ⓯
    int done;                                   /* Loop exit condition */
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    while (TRUE) {
        /*
         * Decrement the global count, so the main thread can see how much
         * progress we've made.  Keep decrementing as long as the remaining
         * time is greater than zero.
         *
         * Lock the mutex to ensure no conflict with the main thread that
         * might be reading the time remaining while we're decrementing it.
         */
        pthread_mutex_lock (&thread_data[index].mutex_handle);
        done = ((thread_data[index].time--) <= 0);
        pthread_mutex_unlock (&thread_data[index].mutex_handle);

        /*
         * 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 */

    if ((index < -1) || (index >= THDNUM))
        print_error ("Bad command number.");
    else {
        if (index == -1)
            start = 0, end = THDNUM;
        else
            start = index, end = start + 1;
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
pthread_lock_global_np ();
for (i = start; i < end; i++) {
    pthread_mutex_lock (&thread_data[i].mutex_handle);
    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;
                }
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
                    case ST_RUNNING:
                        {
                        if (thread_data[i].time < 0)
                            printf ("waiting to print");
                        else
                            printf (
                                    "%d seconds remaining",
                                     thread_data[i].time);
                        break;
                        }
                    default:
                        {
                        this_shouldnt_happen ("Thread state default");
                        break;
                        }
                    }

                printf (".\n");
                output = TRUE;
                }

            pthread_mutex_unlock (&thread_data[i].mutex_handle);
            }

        if (!output) printf ("No such command.\n");

        printf ("\n");

        pthread_unlock_global_np ();
        }
    }

/*
 * Thread routine that prints out the date.
 */
static pthread_addr_t
do_time (pthread_addr_t arg)
    {
    time_t  clock_time;             /* Julian time */
    char    *date_str;              /* Pointer to string returned from ctime */
    char    time_str[8];            /* Piece of ctime string */
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    TRY {
        /*
         * Pretend that this task actually takes a long time to perform.
         */
         do_delay ((int)arg);

        pthread_lock_global_np ();
        TRY {
            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');
            }
        FINALLY {
            pthread_unlock_global_np ();
            }
        ENDTRY
        }
    CATCH (pthread_cancel_e) {
        pthread_lock_global_np ();
        printf ("%d) Canceled.\n", arg);
        pthread_unlock_global_np ();

        do_cleanup ((int)arg, ST_CANCELED);

        RERAISE;
        }
    CATCH_ALL {
        pthread_lock_global_np ();
        printf ("%d) ", arg);
        exc_report (THIS_CATCH);
        pthread_unlock_global_np ();

        do_cleanup ((int)arg, ST_ERROR);

        RERAISE;
        }
    ENDTRY

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

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    /*
     * All thread routines return a value.  This program doesn't check the
     * value, however.
     */
    return arg;
    }

/*
 * Wait for the completion of the specified command.
 */
static void
do_wait (int index)
    {
    if ((index < 0) || (index >= THDNUM))
        print_error ("Bad command number.");
    else
        pthread_join (thread_data[index].thread_handle, (pthread_addr_t)NULL);⓰
    }

/*
 * 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;              /* Flag:  free thread found */
    int      retry = FALSE;     /* Flag:  look again for finished threads */

    do {
        /*
         * We're about to look for a free thread, so prevent the data state
         * from changing while we are looking.
         *
         * Note that when attempting to lock more than one mutex at a time,
         * always lock the mutexes in the same order everywhere in the code.
         * The ordering here is the same as in "do_cleanup".
         */
        pthread_mutex_lock (&free_thread_mutex);
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Find a slot that doesn't have a running thread in it.
 *
 * Before checking, lock the mutex to prevent conflict with the thread
 * if it is running.
 */
for (i = 0, found = FALSE; i < THDNUM; i++) {
    pthread_mutex_lock (&thread_data[i].mutex_handle);
    found = (thread_data[i].state != ST_RUNNING);
    pthread_mutex_unlock (&thread_data[i].mutex_handle);

    /*
     * 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) (17)
              pthread_cond_wait (&free_thread_cv, &free_thread_mutex);
        }
    }
pthread_mutex_unlock (&free_thread_mutex);
} while (retry);
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
    if (found) {
        /*
         * Request DECthreads reclaim its internal storage for this old thread
         * before we use the handle to create a new one.
         */
        pthread_detach (&thread_data[i].thread_handle); ⓲

        *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)
    {
    char    *ret_val;                    /* Return from gets() */

    /*
     * The following calls to stdio routines are protected from concurrent
     * calls by other threads by locking the "global lock".
     */
    pthread_lock_global_np();           /* Serialize calls to stdio */
    printf (PROMPT);
    ret_val = gets (buffer);
    pthread_unlock_global_np();         /* Release for other threads */

    return ret_val;
    }
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Get a yes or no answer to a query.  A "blank" answer uses default answer.
 *
 * Synchronize I/O with other threads to prevent conflicts if the stdio
 * routines are not thread-safe.
 *
 * 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 */
        pthread_lock_global_np ();
        printf ("%s [%c] ", query, defans);
        gets (buffer);
        pthread_unlock_global_np ();
       if (buffer[0] == '\0') buffer[0] = defans;       /* Apply default */
        switch (buffer[0]) {
            case 'y':
            case 'Y':
                {
                answer = TRUE;
                retry = FALSE;
                break;
                }
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
            case 'n':
            case 'N':
                {
                answer = FALSE;
                retry = FALSE;
                break;
                }
            default:
                {
                pthread_lock_global_np ();
                printf ("Please enter \"Y\" or \"N\".\n");
                pthread_unlock_global_np ();
                retry = TRUE;
                break;
                }
            }
        } while (retry);

    return answer;
    }

/*
 * Initialization routine;
 *
 * Called as a one-time initialization action.
 */
static void
init_routine (void) ⓳
    {
    int i;

    for (i = 0; i < THDNUM; i++) {
        pthread_mutex_init (
                &thread_data[i].mutex_handle,
                pthread_mutexattr_default);
        thread_data[i].time = 0;
        thread_data[i].task = '\0';
        thread_data[i].state = ST_INIT;
        }

    pthread_mutex_init (
            &free_thread_mutex,
            pthread_mutexattr_default);

    pthread_cond_init (
            &free_thread_cv,
            pthread_condattr_default);
    }
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Print user-error message.
 *
 * Synchronize I/O with other threads to prevent conflicts if the stdio
 * routines are not thread-safe.
 */
static void
print_error (char *message)
    {
    pthread_lock_global_np();
    printf ("%s\n\n", message);
    pthread_unlock_global_np();
    }

/*
 * Print help text.
 *
 * Synchronize I/O with other threads to prevent conflicts if the stdio
 * routines are not thread-safe.
 */
static void
print_help (void)
    {
    pthread_lock_global_np();
    printf ("Commands are formed by a verb and an 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 the 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");
    pthread_unlock_global_np();
    }
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
/*
 * Print error message for unexpected execution event.
 *
 * Synchronize I/O with other threads to prevent conflicts if the stdio
 * routines are not thread-safe.
 */
static void
this_shouldnt_happen (char *message)
    {
    pthread_lock_global_np();
    printf ("\n\nThis shouldn't happen: %s\n\n", message);
    pthread_unlock_global_np();
    }
```

❶ This program uses the pthread exception interface.

❷ This program uses `printf` with `gets` to prompt for user input. To ensure that no other thread can print or read between the `printf` and `gets`, the program uses a local function to lock the global lock around `printf` and `gets`. Although this does not conflict with the `stdio` jacket functions provided by DECthreads, removing the macros may provide slightly better performance.

❸ Control block for one-time initialization.

❹ Initializes program global objects.

❺ Ensures that other threads (of equal priority) have a chance to run without mutex conflicts.

❻ Locks the global lock before calling `printf` because the DECthreads jacket function for `printf` was disabled.

❼ Requests that the command thread terminate.

❽ Also locks a second mutex. When two mutexes are locked at the same time, lock them in the same order to prevent a deadlock.

❾ Sets the predicate variable and signals the condition variable while the mutex is locked to avoid a race condition.

❿ Locks the global lock before entering the `TRY` block. Otherwise, the `FINALLY` clause cannot determine whether or not the lock completed.

⓫ Locks the global mutex around calls to these C run-time functions (which do not have DECthreads jacket routines). This avoids a race condition.

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

⓬ Ensures that the mutex is unlocked, even if an exception is raised.

⓭ Performs cleanup and RERAISEs the exception if the function is canceled.

⓮ Prints a message describing the caught exception.

⓯ Specifies the preset delay interval: `tv.sec = 1, tv.nsec = 0`. This results in a delay of 1 second.

⓰ Waits for the thread to complete. NULL is passed for the return status value because the program is not interested in what the thread returns.

⓱ The condition variable wait is placed inside a loop that tests the wait predicate. The condition wait is not called if the predicate is satisfied, and the predicate is rechecked each time the condition wait returns.

⓲ Detaches the thread because no other threads will join with it.

⓳ Function called by pthread_once to initialize the program.

One example of the output generated by this program is as follows:

```
RUN DECTHREADS_EXAMPLE

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
```

(continued on next page)

Example 5–2 (Cont.) C Program Example (Asynchronous User Interface)

```
<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
```

Part II

POSIX 1003.4a (pthread) Reference

Part II provides detailed descriptions of the POSIX 1003.4a (pthread) Interface to DECthreads.

Note

The pthread routines in DECthreads are based on P1003.4a Draft 4. A future version of DECthreads is expected to provide support for the final 1003.4a standard. Users should be aware that applications consistent with P1003.4a/D4 might require significant modifications for those future versions of DECthreads that conform to subsequent drafts of the final 1003.4a standard.

To indicate errors, the pthread routines return status values by default. (See Chapter 4 for a method of allowing the pthread routines to raise exceptions.)

You need not initialize the pthread routines.

Routine names ending with the `_np` suffix denote that the routine is not portable—the routine might not be available in implementations of POSIX 1003.4a other than DECthreads. Also, DECthreads defines some types (such as `pthread_addr_t`) that are not specified by POSIX 1003.4a. These types are provided to allow code to be less dependent on ANSI C features (for example, "void *"). These types are documented in Appendix D.

pthread_attr_create

Creates a thread attributes object.

Syntax

pthread_attr_create(
attr);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

C Binding

int
pthread_attr_create (
pthread_attr_t **attr*);

Arguments

attr
Thread attributes object created.

Description

This routine creates a thread attributes object that is used to specify the attributes of threads when they are created. The attributes object created by this routine is only used in calls to `pthread_create`.

The individual attributes (internal fields) of the attributes object are set to default values. (The default values of each attribute are discussed in the descriptions of the following routines.) Use the following routines to change the individual attributes:

```
pthread_attr_setinheritsched
pthread_attr_setprio
pthread_attr_setsched
pthread_attr_setstacksize
```

When an attributes object is used to create a thread, the values of the individual attributes determine the characteristics of the new object. Attributes objects perform similar to additional arguments to object creation. Changing individual attributes does not affect any objects that were previously created using the attributes object.

When you set the scheduling policy or priority, or both, in an attributes object, you must disable scheduling inheritance before the scheduling attributes are used.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[ENOMEM]	Insufficient memory exists to create the thread attributes object.
–1	[EINVAL]	The value specified by *attr* is invalid.

pthread_attr_delete

Deletes a thread attributes object.

Syntax

pthread_attr_delete(
attr);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

C Binding

int
pthread_attr_delete (
pthread_attr_t **attr*);

Arguments

attr
Thread attributes object deleted.

Description

This routine deletes a thread attributes object. This routine gives permission to reclaim storage for the thread attributes object. Threads that were created using this thread attributes object are not affected by the deletion 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 –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[ENOMEM]	Insufficient memory exists to create the thread attributes object.
–1	[EINVAL]	The value specified by *attr* is invalid.
–1	[ESRCH]	The value specified by *attr* does not refer to an existing thread attributes object.

pthread_attr_getguardsize_np

Obtains the guardsize attribute of the specified thread attributes object.

Syntax

pthread_attr_getguardsize_np(
attr);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

C Binding

unsigned long
pthread_attr_getguardsize_np (
pthread_attr_t *attr*);

Arguments

attr
Thread attributes object whose guardsize attribute is obtained.

Description

This routine obtains the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas are necessary when threads might allocate large structures on the stack.

Return Values

On successful completion, this routine returns the guardsize attribute value.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Guardsize attribute		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.

pthread_attr_getinheritsched

Obtains the inherit scheduling attribute of the specified thread attributes object.

Syntax

pthread_attr_getinheritsched(
attr);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

C Binding

int
pthread_attr_getinheritsched (
pthread_attr_t *attr*);

Arguments

attr
Thread attributes object whose inherit scheduling attribute is obtained.

Description

This routine obtains the value of the inherit scheduling attribute in 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`.

The default value of the inherit scheduling attribute is `PTHREAD_INHERIT_SCHED`.

Return Values

On successful completion, this routine returns the inherit scheduling attribute value.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Inherit scheduling attribute		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.
–1	[ESRCH]	The value specified by *attr* does not refer to an existing thread attributes object.

pthread_attr_getprio

Obtains the scheduling priority attribute of the specified thread attributes object.

Syntax

pthread_attr_getprio(
attr);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

C Binding

int
pthread_attr_getprio (
pthread_attr_t *attr*);

Arguments

attr
Thread attributes object whose priority attribute is obtained.

Description

This routine obtains the value of the scheduling priority of threads created using the thread attributes object specified by the *attr* argument.

Return Values

On successful completion, this routine returns the scheduling priority attribute value.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Scheduling priority attribute		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.
–1	[ESRCH]	The value specified by *attr* does not refer to an existing thread attributes object.

pthread_attr_getsched

Obtains the scheduling policy attribute of the specified thread attributes object.

Syntax

pthread_attr_getsched(
attr);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

C Binding

int
pthread_attr_getsched (
pthread_attr_t *attr*);

Arguments

attr
Thread attributes object whose scheduling policy attribute is obtained.

Description

This routine obtains the scheduling policy of threads created using the thread attributes object specified by the *attr* argument. The default value of the scheduling attribute is SCHED_OTHER.

Return Values

On successful completion, this routine returns the value of the scheduling policy attribute.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Scheduling policy attribute		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.
–1	[ESRCH]	The value specified by *attr* does not refer to an existing thread attributes object.

pthread_attr_getstacksize

Obtains the stacksize attribute of the specified thread attributes object.

Syntax

pthread_attr_getstacksize(
attr);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

C Binding

unsigned long
pthread_attr_getstacksize (
pthread_attr_t *attr*);

Arguments

attr
Thread attributes object whose stacksize attribute is obtained.

Description

This routine obtains the minimum size (in bytes) of the stack for a thread created using the thread attributes object specified by the *attr* argument.

Return Values

On successful completion, this routine returns the stacksize attribute value.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Stacksize attribute		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.

pthread_attr_setguardsize_np

Changes the guardsize attribute of thread creation.

Syntax

pthread_attr_setguardsize_np(
attr,
guardsize);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
guardsize	longword	read

C Binding

int
pthread_attr_setguardsize_np (
pthread_attr_t **attr*,
long *guardsize*);

Arguments

attr
Threads attributes object modified.

guardsize
New value for the guardsize attribute. The *guardsize* argument specifies the mimimum size (in bytes) of the guard area for the stack of a thread.

Description

This routine sets the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas are necessary when threads might allocate large structures on the stack.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.

pthread_attr_setinheritsched

Changes the inherit scheduling attribute of the specified thread attributes object.

Syntax

pthread_attr_setinheritsched(
attr,
inherit);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
inherit	integer	read

C Binding

int
pthread_attr_setinheritsched (
pthread_attr_t **attr*,
int *inherit*);

Arguments

attr

Thread attributes object to be modified.

inherit

New value for the inherit scheduling attribute. Valid values are as follows:

PTHREAD_INHERIT_ SCHED	This is the default value. The created thread inherits the current priority and scheduling policy of the thread calling `pthread_create`.
PTHREAD_DEFAULT_ SCHED	The created thread starts execution with the priority and scheduling policy stored in the thread attributes object.

Description

This routine changes the inherit scheduling attribute of thread creation. The inherit scheduling attribute specifies whether threads created using the specified thread attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the thread attributes object that is passed to `pthread_create`.

The first thread in an application that is not created by an explicit call to `pthread_create` has a scheduling policy of `SCHED_OTHER`. See the `pthread_attr_setprio` and `pthread_attr_setsched` routines for more information on valid priority values and valid scheduling policy values, respectively.

Inheriting scheduling attributes (instead of using the scheduling attributes stored in the attributes object) is useful when a thread is creating several helper threads—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 –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.
–1	[ESRCH]	The value specified by *attr* does not refer to an existing thread attributes object.

pthread_attr_setprio

Changes the scheduling priority attribute of thread creation.

Syntax

```
pthread_attr_setprio(
                    attr,
                    priority );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
priority	integer	read

C Binding

```
int
pthread_attr_setprio (
pthread_attr_t *attr,
int priority);
```

Arguments

attr
Thread attributes object modified.

priority
New value for the priority attribute. The priority attribute is dependent upon scheduling policy. Valid values fall within one of the following three ranges:

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 is the midpoint between `PRI_OTHER_MIN` and `PRI_OTHER_MAX`. (Section 2.8 describes how to specify priorities between the minimum and maximum values.)

Description

This routine sets the execution priority of threads that are created using 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. Call `pthread_attr_setinheritsched` and specify the value `PTHREAD_DEFAULT_SCHED` for the *inherit* argument before calling `pthread_create`.

An application specifies priority only to express the urgency of executing the thread relative to other threads. Priority is not used to control mutual exclusion when accessing shared data. With a sufficient number of processors executing, all ready threads, regardless of priority, execute simultaneously.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *priority* is invalid.
–1	[ERANGE]	One or more arguments supplied have an invalid value.

pthread_attr_setsched

Changes the scheduling policy attribute of thread creation.

Syntax

pthread_attr_setsched(
attr,
scheduler);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
scheduler	integer	read

C Binding

int
pthread_attr_setsched (
pthread_attr_t **attr*,
int *scheduler*);

Arguments

attr
Threads attributes object modified.

scheduler
New value for the scheduling policy attribute. (Policies listed on the same line are equivalent.) Valid values are as follows:

SCHED_FIFO
SCHED_RR
SCHED_FG_NP or SCHED_OTHER
SCHED_BG_NP

See Section 2.3.3.2 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 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. Call pthread_attr_setinheritsched and specify the value PTHREAD_DEFAULT_SCHED for the *inherit* argument before calling pthread_create.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *scheduler* is invalid.

pthread_attr_setstacksize

Changes the stacksize attribute of thread creation.

Syntax

pthread_attr_setstacksize(
attr,
stacksize);

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
stacksize	longword	read

C Binding

int
pthread_attr_setstacksize (
pthread_attr_t **attr*,
long *stacksize*);

Arguments

attr
Threads attributes object modified.

stacksize
New value for the stacksize attribute. The *stacksize* argument specifies the minimum size (in bytes) of stack needed for a thread.

Description

This routine sets the minimum size (in bytes) of the stack needed for a thread created using the attributes object specified by the *attr* argument. Use this routine to adjust the size of the writeable area of the stack.

A thread's stack is fixed at the time of thread creation. Only the main or initial thread can dynamically extend its stack.

Most compilers do not check for stack overflow. Ensure that your thread stack is large enough for anything that you call from the thread.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.

pthread_cancel

Allows a thread to request that it or another thread terminate execution.

Syntax

pthread_cancel(
thread);

Argument	Data Type	Access
thread	opaque pthread_t	read

C Binding

int
pthread_cancel (
pthread_t *thread*);

Arguments

thread
Thread that receives a cancel request.

Description

This routine sends a cancel to the specified thread. A cancel is a mechanism by which a calling thread informs the specified thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread will receive or handle the cancel. The canceled thread can delay processing the cancel after receiving it. For instance, if a cancel arrives during an important operation, the canceled thread can continue if what it is doing cannot be interrupted at the point where the cancel is requested.

Because of communication delays, the calling thread can only rely on the fact that a cancel will eventually become pending in the designated thread (provided that the thread does not terminate beforehand). Furthermore, the calling thread has no guarantee that a pending cancel will be delivered because delivery is controlled by the designated thread.

Termination processing when a cancel is delivered to a thread is similar to `pthread_exit`. Outstanding cleanup routines are executed in the context of the target thread, and a status of –1 is made available to any threads joining with the target thread.

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 –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The specified thread is invalid.
–1	[ESRCH]	*thread* does not specify an existing thread.

pthread_cleanup_pop

Removes the cleanup handler at the top of the cleanup stack and optionally executes it.

Syntax

pthread_cleanup_pop(
execute);

Argument	Data Type	Access
execute	Boolean	read

C Binding

int
pthread_cleanup_pop(
int *execute*);

Arguments

execute
Integer that specifies whether the cleanup routine in `pthread_cleanup_push` is executed when the thread terminates normally (for example, when `pthread_exit` is called). If the value of *execute* is 0, the routine is executed only if the thread terminates abnormally (for example, if the thread is canceled). If the value is 1 or more, the routine is executed regardless of whether the thread terminates normally or abnormally.

Description

This routine removes the routine specified in `pthread_cleanup_push` at the top of the calling thread's cleanup stack and executes it if the value specified in *execute* is nonzero.

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` expanding to a string containing the corresponding right brace (}).

Return Values

If an error is detected, it may be indicated by sending the thread a synchronously generated signal.

pthread_cleanup_push

Establishes a cleanup handler to be executed when the thread exits or is canceled.

Syntax

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

Argument	Data Type	Access
routine	opaque pthread_cleanup_t	read
arg	opaque pthread_addr_t	read

C Binding

```
int
pthread_cleanup_push(
pthread_cleanup_t *routine,
pthread_addr_t arg);
```

Arguments

routine
Routine executed as the cleanup handler.

arg
Argument executed with the cleanup routine.

Description

This routine pushes the specified routine onto the calling thread's cleanup stack. The cleanup routine is popped from the stack and executed with the *arg* argument when any of the following actions occur:

- The thread calls `pthread_exit`.
- The thread is canceled.
- The thread calls `pthread_cleanup_pop` and specifies a nonzero value for the *execute* argument.

This routine and pthread_cleanup_pop are implemented as macros and must appear as statements and in pairs within the same lexical scope. You can think of the pthread_cleanup_push macro as expanding to a string whose first character is a left brace ({) and pthread_cleanup_pop as expanding to a string containing the corresponding right brace (}).

Return Values

If an error is detected, it may be indicated by sending the thread a synchronously generated signal.

pthread_condattr_create

Creates a condition variable attributes object that can be used to specify the attributes of condition variables when they are created.

Syntax

pthread_condattr_create(
attr);

Argument	Data Type	Access
attr	opaque pthread_condattr_t	write

C Binding

int
pthread_condattr_create (
pthread_condattr_t **attr*);

Arguments

attr
Condition variable attributes object that is created.

Description

This routine creates a condition variable attributes object that is used to specify the attributes of condition variables when they are created. The condition variable attributes object is initialized with the default value for all of the attributes defined by a given implementation.

When a condition variable attributes object is used to create a condition variable, the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional arguments to object creation. Changing individual attributes does not affect objects that were previously created using the attributes object.

Return Values

The created condition variable attributes object is returned to the *attr* argument.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[ENOMEM]	Insufficient memory exists to create the condition variable attributes object.

pthread_condattr_delete

Deletes a condition variable attributes object.

Syntax

pthread_condattr_delete(
attr);

Argument	Data Type	Access
attr	opaque pthread_condattr_t	read

C Binding

int
pthread_condattr_delete (
pthread_condattr_t **attr*);

Arguments

attr
Condition variable attributes object deleted.

Description

This routine deletes a condition variable attributes object. Call this routine when a condition variable attributes object created by `pthread_condattr_create` will no longer be referenced.

This routine gives permission to reclaim storage for the condition variable attributes object. Condition variables that are created using this attributes object are not affected by the deletion of the condition variable attributes object.

The results of calling this routine are unpredictable if the handle specified by the *attr* argument refers to an attributes object that does not exist.

Return Values

Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The attributes object specified by *attr* is invalid.

pthread_cond_broadcast

Wakes all threads that are waiting on a condition variable.

Syntax

pthread_cond_broadcast(
cond);

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

C Binding

int
pthread_cond_broadcast (
pthread_cond_t **cond*);

Arguments

cond
Condition variable broadcast.

Description

This routine wakes all threads waiting on a condition variable. 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. If any waiting thread might be able to proceed, call `pthread_cond_signal`.

Call this routine when the associated mutex is either locked or unlocked.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *cond* is invalid.

pthread_cond_destroy

Deletes a condition variable.

Syntax

pthread_cond_destroy(
cond);

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

C Binding

int
pthread_cond_destroy (
pthread_cond_t **cond*);

Arguments

cond
Condition variable deleted.

Description

This routine deletes a condition variable. Call this routine when a condition variable will no longer be referenced. The effect of calling this routine is to give permission to reclaim storage for the condition variable.

The results of this routine are unpredictable if the condition variable specified in *cond* does not exist.

The results of this routine are also unpredictable if there are threads waiting for the specified condition variable to be signaled or broadcast when it is deleted.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *cond* is invalid.
–1	[EBUSY]	A thread is currently executing a `pthread_cond_wait` or `pthread_cond_timedwait` on the condition variable specified in *cond*.

pthread_cond_init

Creates 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

int
pthread_cond_init (
pthread_cond_t **cond*,
pthread_condattr_t *attr*);

Arguments

cond
Condition variable that is created.

attr
Condition variable attributes object that defines the characteristics of the condition variable created. If you specify `pthread_condattr_default`, default attributes are used.

Description

This routine creates and initializes a condition variable. A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data; 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.

Return Values

If an error condition occurs, this routine returns –1, the condition variable is not initialized, and the contents of *cond* are undefined. This routine sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EAGAIN]	The system lacks the necessary resources to initialize another condition variable. The system-imposed limit on the total number of condition variables under execution by a single user is exceeded.
–1	[ENOMEM]	Insufficient memory exists to initialize the condition variable.

pthread_cond_signal

Wakes one thread that is waiting on a condition variable.

Syntax

pthread_cond_signal(
cond);

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

C Binding

int
pthread_cond_signal (
pthread_cond_t **cond*);

Arguments

cond
Condition variable signaled.

Description

This routine wakes one thread waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so 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, but only one thread should proceed.

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.

Call this routine when the associated mutex is either locked or unlocked.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *cond* is invalid.

pthread_cond_signal_int_np

Wakes one thread that is waiting on a condition variable. This routine can only be called from interrupt level.

Syntax

pthread_cond_signal_int_np(
cond);

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

C Binding

int
pthread_cond_signal_int_np(
pthread_cond_t **cond*);

Arguments

cond
Condition variable signaled.

Description

This routine wakes one thread waiting on a condition variable. This routine can only be called from interrupt level. 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.

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 returns.

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

Note

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`.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *cond* is invalid.

pthread_cond_timedwait

Causes a thread to wait for a condition variable to be signaled or broadcast for a specified period of time.

Syntax

pthread_cond_timedwait(
cond,
mutex,
abstime);

Argument	Data Type	Access
cond	opaque pthread_cond_t	read
mutex	opaque pthread_mutex_t	read
abstime	structure timespec	read

C Binding

int
pthread_cond_timedwait (
pthread_cond_t **cond*,
pthread_mutex_t **mutex*,
struct timespec **abstime*);

Arguments

cond
Condition variable waited 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.)

Description

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

- The specified condition variable is signaled or broadcasted.
- The current system clock time is greater than or equal to the time specified by the *abstime* argument.

This routine is identical to `pthread_cond_wait` except that this routine can return before a condition variable is signaled or broadcasted; specifically, when a specified time expires.

If the current time equals or exceeds the expiration time, this routine returns immediately, without causing the current thread to wait. Otherwise, waiting on the condition variable can become a nonblocking loop.

Call this routine after you lock the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

Return Values

If an error condition occurs, this routine returns –1 and *errno* is set to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *cond*, *mutex*, or *abstime* is invalid. Different mutexes are supplied for concurrent `pthread_cond_timedwait` operations or `pthread_cond_wait` operations on the same condition variable.
–1	[EAGAIN]	The time specified by *abstime* expired.
–1	[EDEADLK]	A deadlock condition is detected.

pthread_cond_wait

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

Syntax

```
pthread_cond_wait(
                cond,
                mutex );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read
mutex	opaque pthread_mutex_t	read

C Binding

```
int
pthread_cond_wait (
pthread_cond_t *cond,
pthread_mutex_t *mutex);
```

Arguments

cond
Condition variable waited on.

mutex
Mutex associated with the condition variable specified in *cond*.

Description

This routine causes a thread to wait for a condition variable to be signaled or broadcast. Each condition corresponds to one or more Boolean relations (predicates) based on shared data. The calling thread waits for the data to reach a particular state (for the predicate to become true).

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. If the wait is satisfied as a result of some thread calling `pthread_cond_signal` or `pthread_cond_broadcast`, the mutex is reacquired and the routine returns.

A thread that changes the state of storage protected by the mutex in such a way that a predicate associated with a condition variable might now be true must call either `pthread_cond_signal` or `pthread_cond_broadcast` for that condition variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This routine might (with low probability) return when the condition variable has not been signaled or broadcasted. When this occurs, the mutex is reacquired before the routine returns. (To handle this type of situation, enclose this routine in a loop that checks the predicate.)

Return Values

If an error condition occurs, this routine returns –1 and *errno* is set to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *cond* or *mutex* is invalid.
		Different mutexes are supplied for concurrent `pthread_cond_wait` or `pthread_cond_timedwait` operations.

pthread_create

Creates a thread object and 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	opaque pthread_addr_t	read

C Binding

int
pthread_create (
pthread_t **thread,*
pthread_attr_t *attr,*
pthread_startroutine_t *start_routine,*
pthread_addr_t *arg*);

Arguments

thread
Thread object created.

attr
Thread attributes object that defines the characteristics of the thread being created. If you specify `pthread_attr_default`, 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 object and 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. A thread object defines and controls the executing thread.

Calling this routine sets into motion the following actions:

- An internal thread object is created to describe the thread.
- The associated executable thread is created with attributes specified by the *attr* argument (or with default attributes if `pthread_attr_default` is specified).
- The *thread* argument receives the new thread.
- The *start_routine* function is called.

A thread is created in the ready state and therefore might immediately begin executing the function specified by the *start_routine* argument. The newly created thread may preempt its creator if the new thread follows the `SCHED_RR` or `SCHED_FIFO` scheduling policy or has a priority higher than the creating thread, or both. Otherwise, the new thread begins running at its turn, which might also be before `pthread_create` returns.

The new thread's scheduling policy and priority are, by default, inherited from the creating thread—the scheduling policy and priority set in the attributes object are ignored. To create a thread using the scheduling policy and priority set in the attributes object, you must first disable the inherit scheduling attribute by calling `pthread_attr_setinheritsched`.

The *start_routine* is passed a copy of the *arg* argument. The value of the *arg* argument is specified by the calling application code.

The thread object exists until the `pthread_detach` routine is called or the thread terminates, whichever occurs last.

Synchronization between the caller of `pthread_create` and the newly created thread is done through the use of the `pthread_join` routine (or any other mutexes or condition variables they agree to use).

Return Values

If an error condition occurs, no thread is created, the contents of *thread* are undefined, and this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EAGAIN]	The system lacks the necessary resources to create another thread. The system-imposed limit on the total number of threads under execution by a single user is exceeded.
–1	[ENOMEM]	Insufficient memory exists to create the thread attributes object. This is not a temporary condition.

pthread_delay_np

Causes a thread to wait for a specified period of time before continuing execution.

Syntax

pthread_delay_np(
interval);

Argument	Data Type	Access
interval	struct timespec	read

C Binding

extern int
pthread_delay_np (
struct timespec **interval*);

Arguments

interval
Number of seconds and nanoseconds that the calling thread waits before continuing execution. The value specified must be greater than or equal to 0.

Description

This routine causes a thread to delay execution for a specified period of elapsed wall clock time. The period of time the thread waits is at least as long as the number of seconds and nanoseconds specified in the *interval* argument.

Specifying an interval of 0 seconds and 0 nanoseconds is allowed and can result in the thread giving up the processor or delivering a pending cancel.

The `struct timespec` structure contains the following two fields:

- `tv_sec` is an integer number of seconds
- `tv_nsec` is an integer number of nanoseconds

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[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

int
pthread_detach (
pthread_t **thread*);

Arguments

thread
Thread object marked for deletion.

Description

This routine indicates that storage for the specified thread is reclaimed when the thread terminates. This includes storage for the *thread* argument's return value. If *thread* has not terminated when this routine is called, this routine does not cause it to terminate.

Call this routine when a thread object is no longer referenced. Additionally, call this routine for every thread that is created to ensure that storage for thread objects does not accumulate.

You cannot join with a thread after the thread has been detached.

The results of this routine are unpredictable if the value of *thread* refers to a thread object that does not exist.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *thread* is invalid.
–1	[ESRCH]	The value specified by *thread* does not refer to an existing thread.

pthread_equal

Compares one thread identifier to another thread identifier.

Syntax

equal = pthread_equal (thread1, thread2)

Argument	Data Type	Access
equal	Boolean	write
thread1	opaque pthread_t	read
thread2	opaque pthread_t	read

C Binding

int
pthread_equal (
pthread_t *thread1,*
pthread_t *thread2*);

Arguments

equal
Boolean value that specifies whether *thread1* and *thread2* designate the same object.

thread1
The first thread identifier to be compared.

thread2
The second thread identifier to be compared.

Description

This routine compares one thread identifier to another thread identifier. (This routine does not check whether the objects that correspond to the identifiers currently exist.) If the identifiers have values indicating that they designate the same object, 1 (true) is returned. If the values do not designate the same object, 0 (false) is returned.

This routine is implemented as a C macro.

Return Values

Possible return values are as follows:

Return	Error	Description
0		Values of *thread1* and *thread2* do not designate the same object.
1		Values of *thread1* and *thread2* designate the same object.

pthread_exit

Terminates the calling thread.

Syntax

pthread_exit(
status);

Argument	Data Type	Access
status	opaque pthread_addr_t	read

C Binding

void
pthread_exit (
pthread_addr_t **status*);

Arguments

status
Address value copied and returned to the caller of `pthread_join`.

Description

This routine terminates the calling thread and makes a status value available to any thread that calls `pthread_join` and specifies the terminating thread.

An implicit call to `pthread_exit` is issued when a thread returns from the start routine that was used to create it. The function's return value serves as the thread's exit status. The process exits when the last running thread calls `pthread_exit`.

Return Values

None

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

extern int
pthread_get_expiration_np (
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 that a timed wait expires.

abstime
Value representing the expiration time.

Description

This routine adds a specified interval to the current absolute system time and returns a new absolute time. This new absolute time is used as the expiration time in a call to `pthread_cond_timedwait`.

The `struct timespec` structure contains the following two fields:

- `tv.sec` is an integer number of seconds
- `tv.nsec` is an integer number of nanoseconds

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *delta* is invalid.

pthread_getprio

Obtains the current priority of a thread.

Syntax

pthread_getprio(
thread);

Argument	Data Type	Access
thread	opaque pthread_t	read

C Binding

int
pthread_getprio (
pthread_t *thread*);

Arguments

thread
Thread whose priority is obtained.

Description

This routine obtains the current priority of a thread. The current priority is different from the initial priority of the thread if the `pthread_setprio` routine is called.

The exact effect of different priority values depends upon the scheduling policy assigned to the thread.

Return Values

The current priority value of the thread specified in *thread* is returned. See the description of `pthread_setprio` for valid values.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Priority value		Successful completion.
–1	[EINVAL]	The value specified by *thread* is invalid.

pthread_getscheduler

Obtains the current scheduling policy of a thread.

Syntax

pthread_getscheduler(
thread);

Argument	Data Type	Access
thread	opaque pthread_t	read

C Binding

int
pthread_getscheduler (
pthread_t *thread*);

Arguments

thread
Thread whose scheduling policy is obtained.

Description

This routine obtains the current scheduling policy of a thread. The current scheduling policy of a thread is different from the initial scheduling policy if the `pthread_setscheduler` routine is called.

Return Values

The current scheduling policy value of the thread specified in *thread* is returned. See the description of `pthread_setscheduler` for valid values.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Scheduling policy value		Successful completion.
–1	[EINVAL]	The value specified by *thread* is invalid.
–1	[ESRCH]	The value specified by *thread* does not refer to an existing thread.

pthread_getspecific

Obtains the per-thread context associated with the specified key.

Syntax

pthread_getspecific(
key,
value);

Argument	Data Type	Access
key	opaque pthread_key_t	read
value	opaque pthread_addr_t	write

C Binding

int
pthread_getspecific (
pthread_key_t *key*,
pthread_addr_t **value*);

Arguments

key
Context key value that identifies the context value obtained. This key value must be obtained from `pthread_keycreate`.

value
Address of the current per-thread context value associated with the specified key.

Description

This routine obtains the per-thread context associated with the specified key for the current thread.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The key value is invalid.

pthread_join

Causes the calling thread to wait for the termination of a specified thread.

Syntax

```
pthread_join(
        thread,
        status );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
status	opaque pthread_addr_t	write

C Binding

```
int
pthread_join (
pthread_t thread,
pthread_addr_t *status);
```

Arguments

thread
Thread whose termination is awaited by the caller of this routine.

status
Status value of the terminating thread (in other words, when that thread calls `pthread_exit`.)

Description

This routine causes the calling thread to wait for the termination of a specified thread. A call to this routine returns after the specified thread has terminated.

If the thread exits normally, the status value argument is the address that the specified thread generates as its result. The thread's result is normally returned as the value of the *start_routine* argument in its call to `pthread_create`. If the thread does not exit normally, the value of *status* is –1.

Any number of threads can call this routine. All calling threads are awakened when the specified thread terminates.

If the current thread calls this routine, a deadlock can result (if it is not detected by the implementation).

The results of this routine are unpredictable if the value for *thread* refers to a thread object that no longer exists (i.e., one which has been detached).

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *thread* is invalid.
–1	[ESRCH]	The value specified by *thread* does not refer to an existing thread.

pthread_keycreate

Generates a unique per-thread context key value.

Syntax

pthread_keycreate(
key,
destructor);

Argument	Data Type	Access
key	opaque pthread_key_t	write
destructor	procedure pthread_destructor_t	read

C Binding

int
pthread_keycreate (
pthread_key_t **key*,
pthread_destructor_t *destructor*);

Arguments

key
Value of the new per-thread context key.

destructor
Procedure called to destroy a context value associated with the created key when the thread terminates.

Description

This routine generates a unique per-thread context key value. This key value identifies a per-thread context, which is an address of memory generated by the client containing arbitrary data of any size.

Per-thread context allows client software to associate context information with the current thread. (This mechanism can be thought of as a means for a client to add unique fields to the thread control block.)

For example, per-thread context can be used by a language run-time library that needs to associate a language-specific thread-private data structure with an individual thread. The per-thread context routines also provide a portable means of implementing the class of storage called thread-private static, which is needed to support parallel decomposition in the Fortran language.

This routine generates and returns a new key value. The key provides a cell within each thread. Each call to this routine creates a new cell, and each call within a process returns a key value 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 multiple facilities share access to per-thread context, the facilities must agree on the key value that is associated with the context. The key value must be created once and should be stored in a location known to each facility. (Encapsulate key creation and context value setting within a special facility for that purpose.)

An implementation can choose to predefine some number of keys for favored clients, such as certain compilers, run-time libraries, or the debugger.

When a thread terminates, its per-thread context is automatically destroyed; however, the key value remains. For each per-thread context currently associated with the thread, the destructor routine associated with the key value of that context is called. The order in which per-thread context destructors are called at thread termination is undefined.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[ENOMEM]	An attempt is made to allocate a key when the key name space is exhausted. This is not a temporary condition.
–1	[EAGAIN]	Insufficient memory exists to create the key.
–1	[EINVAL]	Invalid argument.

pthread_lock_global_np

Locks a global mutex if the global mutex is unlocked. If the global mutex is locked, causes the thread to wait for the global mutex to become available.

Syntax

pthread_lock_global_np( );

C Binding

void
pthread_lock_global_np ( );

Arguments

None

Description

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

The thread that has locked the global mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the global mutex in the locked state and with the current thread as the global mutex's current owner.

Use the 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 compatible with multithreading, 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 isn't known to be reentrant uses the same lock. This prevents dependencies among threads calling 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

None

pthread_mutexattr_create

Creates a mutex attributes object that is used to specify the attributes of mutexes when they are created.

Syntax

pthread_mutexattr_create(
attr);

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write

C Binding

int
pthread_mutexattr_create (
pthread_mutexattr_t **attr*);

Arguments

attr
Mutex attributes object created.

Description

This routine creates a mutex attributes object used to specify the attributes of mutexes when they are created. The mutex attributes object is initialized with the default value for all of the attributes defined by a given implementation.

When a mutex attributes object is used to create a mutex, the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional arguments to object creation. Changing individual attributes does not affect any objects that were previously created using the attributes object.

Return Values

The created mutex attributes object is returned to the *attr* argument.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[ENOMEM]	Insufficient memory exists to create the mutex attributes object.

pthread_mutexattr_delete

Deletes a mutex attributes object.

Syntax

pthread_mutexattr_delete(
attr);

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read

C Binding

int
pthread_mutexattr_delete (
pthread_mutexattr_t **attr*);

Arguments

attr
Attributes object deleted.

Description

This routine deletes a mutex attributes object. Call this routine when a mutex attributes object is no longer referenced by the `pthread_mutexattr_create` routine.

This routine gives permission to reclaim storage for the mutex attributes object. Mutexes that were created using this attributes object are not affected by the deletion 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

Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.

pthread_mutexattr_getkind_np

Obtains the mutex type attribute used when a mutex is created.

Syntax

pthread_mutexattr_getkind_np(
attr);

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read

C Binding

int
pthread_mutexattr_getkind_np (
pthread_mutexattr_t *attr*);

Arguments

attr
Mutex attributes object whose mutex kind is obtained.

Description

This routine obtains the mutex type attribute that is used when a mutex is created. See the `pthread_mutexattr_setkind_np` description for information about mutex type attributes.

Return Values

On successful completion, this routine returns the mutex kind.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Mutex type attribute		Successful completion.
–1	[EINVAL]	The value specified by *attr* is invalid.
–1	[ESRCH]	The value specified by *attr* does not refer to an existing mutex attributes object.

pthread_mutexattr_setkind_np

Specifies the mutex type attribute that is used when a mutex is created.

Syntax

pthread_mutexattr_setkind_np(
attr,
kind);

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read
kind	integer	read

C Binding

int
pthread_mutexattr_setkind_np (
pthread_mutexattr_t **attr*,
int *kind*);

Arguments

attr

Mutex attributes object modified.

kind

New value for the mutex type attribute. The *kind* argument specifies the type of mutex that is created. Valid values are `MUTEX_FAST` (default), `MUTEX_RECURSIVE`, and `MUTEX_NONRECURSIVE`.

Description

This routine sets the mutex type attribute that is used when a mutex is created. See Section 2.3.4.1 for information on the types of mutexes.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *attr* or *kind* is invalid.
–1	[ESRCH]	The value specified by *attr* does not refer to an existing mutex attributes object.

pthread_mutex_destroy

Deletes a mutex.

Syntax

pthread_mutex_destroy(
mutex);

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

int
pthread_mutex_destroy (
pthread_mutex_t **mutex*);

Arguments

mutex
Mutex deleted.

Description

This routine deletes a mutex and should be called when a mutex object is longer referenced. This routine reclaims storage for the mutex object.

It is illegal to delete a locked mutex.

The results of this routine are unpredictable if the mutex object specified in the *mutex* argument does not currently exist.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EBUSY]	An attempt is made to destroy a locked mutex.
–1	[EINVAL]	The value specified by *mutex* is invalid.

pthread_mutex_init

Creates 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

int
pthread_mutex_init (
pthread_mutex_t **mutex,*
pthread_mutexattr_t *attr*);

Arguments

mutex
Mutex created.

attr
Mutex attributes object that defines the characteristics of the created mutex. If you specify `pthread_mutexattr_default`, default attributes are used.

Description

This routine creates a mutex. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

The mutex is created and initialized to the unlocked state.

The created mutex is not automatically deallocated because it is considered shared among multiple threads if the thread that called this routine terminates.

Return Values

If an error condition occurs, this routine returns –1, the mutex is not initialized, and the contents of *mutex* are undefined. This routine sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EAGAIN]	The system lacks the necessary resources to initialize another mutex.
		The system-imposed limit on the total number of mutexes under execution by a singled user is exceeded.
–1	[ENOMEM]	Insufficient memory exists to initialize the mutex.

pthread_mutex_lock

Locks an unlocked mutex. If the mutex is locked, causes the thread to wait for the mutex to become available.

Syntax

pthread_mutex_lock(
mutex);

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

int
pthread_mutex_lock (
pthread_mutex_t **mutex*);

Arguments

mutex
Mutex locked.

Description

This routine locks a mutex. If the mutex is locked when a thread calls this routine, the thread waits for the mutex to become available.

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.

If you specified a fast mutex, a deadlock can result if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time. If you specified a recursive mutex, the current owner of a mutex can relock the same mutex without blocking. If the current owner of a nonrecursive mutex calls this routine in an attempt to lock the mutex a second time, the exception `pthread_in_use_e` is raised. See pthread_mutexattr_setkind_np for information about fast, recursive, and nonrecursive mutexes.

Before you lock a mutex, disable asynchronous cancellation.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *mutex* is invalid.
–1	[EDEADLK]	A deadlock condition is detected.

pthread_mutex_trylock

Locks a 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

int
pthread_mutex_trylock (
pthread_mutex_t **mutex*);

Arguments

mutex
Mutex locked.

Description

This routine locks a mutex. If the specified mutex is locked when a thread calls this routine, the calling thread does not wait for the mutex to become available.

When a thread calls this routine, an attempt is made to immediately lock the mutex. If the mutex is successfully locked, 1 is returned and the current thread is then the mutex's current owner.

If the mutex is locked by another thread when this routine is called, 0 is returned and the thread does not wait to acquire the lock. If a fast mutex is owned by the current thread, 0 is returned. If a recursive mutex is owned by the current thread, 1 is returned and the mutex is relocked. (To unlock a recursive mutex, each call to `pthread_mutex_trylock` must be matched by a call to `pthread_mutex_unlock`.)

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
1		Successful completion.
0		The mutex is already locked; therefore, it was not acquired.
–1	[EINVAL]	The value specified by *mutex* is invalid.

pthread_mutex_unlock

Unlocks a mutex.

Syntax

pthread_mutex_unlock(
mutex);

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

C Binding

int
pthread_mutex_unlock (
pthread_mutex_t **mutex*);

Arguments

mutex
Mutex unlocked.

Description

This routine unlocks a mutex. If no threads are waiting for the mutex, the mutex becomes unlocked with no current owner. If one or more threads are waiting to lock the specified mutex, this routine causes one thread to unblock and try to acquire the mutex. The scheduling policy is used to determine which thread acquires the mutex. For the SCHED_FIFO and SCHED_RR policies, a blocked thread is chosen in priority order, using FIFO within priorities.

The results of calling this routine are unpredictable if the mutex specified in *mutex* is unlocked. The results of calling this routine are also unpredictable if the mutex specified in *mutex* is currently owned by a thread other than the calling thread.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The value specified by *mutex* is invalid.

pthread_once

Calls an initialization routine that can be executed by only one thread, a single time.

Syntax

pthread_once(
once_block,
init_routine);

Argument	Data Type	Access
once_block	opaque pthread_once_t	read
init_routine	opaque pthread_initroutine_t	read

C Binding

int
pthread_once (
pthread_once_t **once_block*,
pthread_initroutine_t *init_routine*);

Arguments

once_block

Address of a record that defines the one-time initialization code. Each one-time initialization routine must have its own unique `pthread_once_t`.

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_block* are passed to `pthread_once`.

Description

This routine calls an initialization routine executed by one thread, a single time. This routine allows you to create your own initialization code that is guaranteed to be run only once, even if called simultaneously by multiple threads or multiple times in the same thread.

For example, a mutex or a per-thread context key must be created exactly once. Calling pthread_once prevents the code that creates a mutex or per-thread context from being called by multiple threads. Without this routine, the execution must be serialized so that only one thread performs the initialization. Other threads that reach the same point in the code would be delayed until the first thread is finished.

This routine initializes the control record if it has not been initialized and then determines if the one-time initialization routine has executed once. If it has not executed, this routine calls the initialization routine specified in *init_routine*. If the one-time initialization code has executed once, this routine returns.

Note

If you specify an *init_routine* that directly or indirectly results in a recursive call to pthread_once specifying the same *init_block* argument, the recursive call will result in a deadlock.

The *once_block* must be declared static (for example, either extern or static in the C language), and it must be initialized at compile time. In the C language, using pthread.h or pthread_exc.h, initialize an *once_block* using the pthread_once_init macro. In other languages, you must initialize a pthread_once_t block to a value of three integer zeroes. In C, that corresponds to the following:

```
static pthread_once_t block = {0,0,0};
```

Return Values

If an error occurs, this routine returns –1. No error values have been specified. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	Invalid argument.

pthread_self

Obtains the identifier of the current thread.

Syntax

pthread_self( );

C Binding

pthread_t
pthread_self ( );

Arguments

None

Description

This routine allows a thread to obtain its own identifier. Use this identifier in calls to `pthread_setprio` and `pthread_setscheduler`.

This value becomes meaningless when the thread object is deleted—that is, when the thread has terminated its execution and `pthread_detach` has been called.

Return Values

Returns the identifier of the calling thread to `pthread_t`.

pthread_setasynccancel

Enables or disables the current thread's asynchronous cancelability.

Syntax

old_state = pthread_setasynccancel(
state);

Argument	Data Type	Access
state	integer	read

C Binding

int
pthread_setasynccancel (
int *state*);

Arguments

state
State of asynchronous cancelability to set for the calling thread. Valid values are as follows:

Value	Description
CANCEL_ON	Asynchronous cancelability is enabled.
CANCEL_OFF	Asynchronous cancelability is disabled.

Description

This routine enables or disables the current thread's asynchronous cancelability and returns the previous cancelability state.

When general cancelability is set to `CANCEL_OFF`, a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability is enabled. When general cancelability is set to `CANCEL_ON`, cancelability depends on the state of the thread's asynchronous cancelability. When general cancelability is set to `CANCEL_ON` and asynchronous cancelability is set to `CANCEL_OFF`, the thread can only receive a cancel at specific cancelation points (for example, condition waits, thread joins and calls to `pthread_testcancel`.) If

both general cancelability and asynchronous cancelability are set to CANCEL_ON, the thread can be canceled at any point in its execution.

When a thread is created, the default asynchronous cancelability state is CANCEL_OFF.

If you call this routine to enable asynchronous cancelability, call it in a region of code where asynchronous delivery of cancels has been disabled by a previous call to this routine. Threads routines should not be called in regions of code where asynchronous delivery of cancels is enabled. The previous state of asynchronous delivery can be restored later by another call to this routine.

Return Values

On successful completion, this routine returns the previous state of asynchronous cancelability.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The specified state is not CANCEL_ON or CANCEL_OFF.

pthread_setcancel

Enables or disables the current thread's general cancelability.

Syntax

int pthread_setcancel(
state);

Argument	Data Type	Access
state	integer	read

C Binding

int
pthread_setcancel (
int *state*);

Arguments

state
State of general cancelability to set for the calling thread. Valid values are as follows:

Value	Description
CANCEL_ON	Asynchronous cancelability is enabled.
CANCEL_OFF	Asynchronous cancelability is disabled.

Description

This routine enables or disables the current thread's general cancelability and returns the previous cancelability state.

When general cancelability is set to `CANCEL_OFF`, a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability is enabled.

When a thread is created, the default general cancelability state is `CANCEL_ON`.

Possible Dangers of Disabling Cancelability

The most important use of cancels is to ensure that indefinite wait operations are terminated. For example, a thread waiting on some network connection, which may take days to respond (or may never respond), is normally made cancelable.

However, when cancelability is disabled, no routine is cancelable. Waits must be completed normally before a cancel can be delivered. As a result, the program stops but does not crash and 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 cancels around that particular region of code.

Return Values

On successful completion, this routine returns the previous state of general cancelability.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The specified state is not CANCEL_ON or CANCEL_OFF.

pthread_setprio

Changes the current priority of a thread.

Syntax

pthread_setprio(
thread,
priority);

Argument	Data Type	Access
thread	opaque pthread_t	read
priority	integer	read

C Binding

int
pthread_setprio (
pthread_t *thread*,
int *priority*);

Arguments

thread
Thread whose priority is changed.

priority
New priority value of the thread specified in *thread*. The priority value is dependent upon scheduling policy. Valid values fall within one of the following three ranges.

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

If you create a new thread without specifying a threads attributes object that contains a changed priority attribute, the default priority of the newly

created thread is the midpoint between PRI_OTHER_MIN and PRI_OTHER_MAX (the midpoint between the minimum and the maximum for the SCHED_OTHER policy). (Section 2.8 describes how to specify priorities between the minimum and maximum values.)

Description

This routine changes the current priority of a thread. A thread can change its own priority using the identifier returned by pthread_self.

Changing the priority of a thread can cause it to start executing or be preempted by another thread. The effect of setting different priority values depends on the scheduling priority assigned to the thread. The initial scheduling priority is set by calling the pthread_attr_setprio routine.

An application should specify priority only to express the urgency of executing the thread relative to other threads. Priority should not be used to control mutual exclusion when accessing shared data. With a sufficient number of processors executing, all ready threads, regardless of priority, can be executing simultaneously.

The pthread_attr_setprio routine sets the priority attribute that is used to establish the priority of a new thread when it is created. However, pthread_setprio changes the priority of an existing thread.

Return Values

The previous priority of the thread specified in *thread* is returned.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Priority value		Successful completion.
–1	[EINVAL]	The value specified by *thread* is invalid.
–1	[ENOTSUP]	An attempt is made to set the policy to an unsupported value.
–1	[ESRCH]	The value specified by *thread* does not refer to an existing thread.

Return	Error	Description
–1	[EPERM]	The caller does not have the appropriate privileges to set the priority of the specified thread.

pthread_setscheduler

Changes the current scheduling policy and priority of a thread.

Syntax

pthread_setscheduler(
thread,
scheduler,
priority);

Argument	Data Type	Access
thread	opaque pthread_t	read
scheduler	integer	read
priority	integer	read

C Binding

int
pthread_setscheduler (
pthread_t *thread*,
int *scheduler*,
int *priority*);

Arguments

thread

Thread whose scheduling policy is to be changed.

scheduler

New scheduling policy value for the thread specified in *thread*. (Policies listed on the same line are equivalent.) Valid values are as follows:

SCHED_FIFO
SCHED_RR
SCHED_FG_NP or SCHED_OTHER
SCHED_BG_NP

See Section 2.3.3.2 for a description of the scheduling policies.

priority
New priority value of the thread specified in *thread*. The priority attribute is dependent upon scheduling policy. Valid values fall within one of the following three ranges.

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

If you create a new thread without specifying a threads attributes object that contains a changed priority attribute, the default priority of the newly created thread is the midpoint between `PRI_OTHER_MIN` and `PRI_OTHER_MAX` (the midpoint between the minimum and the maximum for the `SCHED_OTHER` policy). (Section 2.8 describes how to specify priorities between the minimum and maximum values.)

Description

This routine changes the current scheduling policy and priority of a thread. Call this routine to change both the priority and scheduling policy of a thread at the same time. To change only the priority, call the `pthread_setprio` routine.

A thread changes its own scheduling policy and priority by using the identifier returned by `pthread_self`. Changing the scheduling policy or priority, or both, of a thread can cause it to start executing or to be preempted by another thread.

This routine is different from `pthread_attr_setprio` and `pthread_attr_setsched` because those routines set the priority and scheduling policy attributes that are used to establish the priority and scheduling policy of a new thread when it is created. This routine, however, changes the priority and scheduling policy of an existing thread.

Return Values

The previous policy of the thread specified in *thread* is returned.

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Scheduling policy value		Successful completion.
–1	[EINVAL]	The value specified by *thread* is invalid.
–1	[ENOTSUP]	An attempt is made to set the policy to an unsupported value.
–1	[ESRCH]	The value specified by *thread* does not refer to an existing thread.
–1	[EPERM]	The caller does not have the appropriate privileges to set the priority of the specified thread.

pthread_setspecific

Sets the per-thread context associated with the specified key for the current thread.

Syntax

pthread_setspecific(
key,
value);

Argument	Data Type	Access
key	opaque pthread_key_t	read
value	opaque pthread_addr_t	read

C Binding

int
pthread_setspecific (
pthread_key_t *key*,
pthread_addr_t *value*);

Arguments

key

Context key value that uniquely identifies the context cell to receive *value*. This key value must be obtained from `pthread_keycreate`.

value

Address containing data associated with the specified key for the current thread; this is the per-thread context.

Description

This routine sets the per-thread context associated with the specified key for the current thread. If a context is defined for the key in this thread (the current value is not null), the new value is substituted for it.

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.

Return Values

If an error condition occurs, this routine returns –1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
–1	[EINVAL]	The key value is invalid.

pthread_testcancel

Requests delivery of a pending cancel to the current thread.

Syntax

pthread_testcancel( );

C Binding

void
pthread_testcancel ( );

Arguments

None

Description

This routine requests delivery of a pending cancel to the current thread. The cancel is delivered only if a cancel is pending for the current thread and general cancel delivery is not currently disabled. (A thread disables delivery of cancels to itself by calling `pthread_setcancel`.)

This routine, when called within very long loops, ensures that a pending cancel is noticed within a reasonable amount of time.

Return Values

None

pthread_unlock_global_np

Unlocks a global mutex.

Syntax

pthread_unlock_global_np( );

C Binding

void
pthread_unlock_global_np ( );

Arguments

None

Description

This routine unlocks the global mutex 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 try to acquire the mutex. The scheduling policy is used to determine which thread acquires the global mutex. For the policies `SCHED_FIFO` and `SCHED_RR`, a blocked thread is chosen in priority order, using FIFO within priorities.

The results of calling this routine are unpredictable if the global mutex is already unlocked. The results of calling this routine are also unpredictable if the global mutex is owned by a thread other than the calling thread.

Return Values

None

pthread_yield

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( );

C Binding

void
pthread_yield ( );

Arguments

None

Description

This routine notifies the scheduler that the current thread is willing to release its processor to other threads of the same priority. (A thread generally releases its processor to a thread of a higher priority without calling this routine.)

If the current thread's scheduling policy (as specified in a call to `pthread_attr_setsched` or `pthread_setscheduler`) is `SCHED_FIFO`, `SCHED_RR`, or `SCHED_OTHER`, this routine yields the processor to other threads of the same priority. If no threads of the same priority are ready to execute, the thread continues.

This routine allows knowledge of the details of an application to be used to increase fairness. It increases fairness by allowing other threads to access the processor at convenient times, such as when critical resources are free.

Call this routine when a thread is executing code that denies access to other threads on a uniprocessor if the scheduling policy is `SCHED_FIFO`.

Use `pthread_yield` carefully because misuse causes unnecessary context switching, which increases overhead without increasing fairness. For example, it is counter-productive for a thread to yield while a needed resource is locked.

Return Values

None

Part III

Digital Proprietary Interface Reference

Part III provides detailed descriptions of the Digital Proprietary (cma) Interface to DECthreads.

The cma routines ensure a stable, upwardly compatible interface to DECthreads.

To indicate errors, the cma routines raise exceptions. See Appendix C for exception descriptions.

Note

Call `cma_init` before calling any other cma routine.

cma_alert_disable_asynch

Disables asynchronous alert delivery to the current thread.

Syntax

cma_alert_disable_asynch (prior)

Argument	Data Type	Access
prior	opaque cma_t_alert_state	write

C Binding

void
cma_alert_disable_asynch (
cma_t_alert_state *_prior_);

Arguments

prior
Receives the state of asynchronous alert delivery (enabled or disabled) that exists before the call to this routine.

Description

This routine disables asynchronous alert delivery to the current thread. The prior state of asynchronous delivery is stored in the *prior* argument. Call `cma_alert_restore` to restore the setting previous alert delivery setting stored in *prior*.

Asynchronous delivery of alerts is not appropriate over regions of code where resources are being allocated, or when invariants are being modified. It is difficult to determine exactly where an exception was raised within such a region. Usually this makes it very difficult (and often impossible) to properly release resources or restore invariants should an alert be delivered.

External routines often do not function correctly with asynchronous delivery of alerts enabled. Also, most existing code will not be able to cope with asynchronous alerts. It is always safest to disable asynchronous delivery before calling any external routine from a region of code where asynchronous delivery of alerts has been enabled.

Note

An **alertable routine** is one where synchronous alert delivery can occur only at specific, well-defined points. These points are DECthreads routines that can determine whether an alert is pending, and if so, can deliver the alert.

This routine is not alertable.

Exceptions

None

cma_alert_disable_general

Disables general delivery of alerts to the current thread.

Syntax

cma_alert_disable_general (prior)

Argument	Data Type	Access
prior	opaque cma_t_alert_state	write

C Binding

void
cma_alert_disable_general (
cma_t_alert_state *_prior_);

Arguments

prior
Receives the prior state of general alert delivery (enabled or disabled).

Description

This routine disables general delivery of alerts to the current thread and returns the previous state of alert delivery to the *prior* argument.

Exceptions

cma_e_alert_nesting
cma_e_existence
cma_e_use_error

cma_alert_enable_asynch

Enables asynchronous alert delivery to the current thread.

Syntax

cma_alert_enable_asynch

C Binding

```
void
cma_alert_enable_asynch ( );
```

Arguments

None

Description

This routine enables asynchronous alert delivery to the current thread. Unlike `cma_alert_disable_asynch`, this routine does not return the prior state of asynchronous delivery. Before a call to this routine returns, asynchronous delivery is enabled. Any return value would be unreliable. (An alert can occur during the hardware or language procedure linkage.)

To allow restoration of the previous alert state when asynchronous cancelability is no longer needed, call `cma_alert_disable_asynch` to obtain the current alert state prior to calling this routine.

This routine is alertable. If an alert is pending for the current thread, and alert delivery is not currently disabled, then the pending alert is delivered.

Asynchronous delivery of alerts means that the `cma_e_alerted` exception can be raised at any point in the code where an interrupt can occur. The exception could potentially be raised in the middle of a hardware instruction, if that is permitted by the machine.

As a result, it is very difficult to write an exception handler that can restore invariants for a region of code where an asynchronous alert is delivered. Asynchronous delivery should be avoided in any region of code where program invariants can be modified.

The best application for asynchronous alert delivery is when the program needs to perform a long computation that does not affect program invariants, and where adding calls to `cma_alert_test` is impractical or would slow down the computation.

Enabling asynchronous delivery over such a region of code allows the code to be highly responsive to abort requests (for example, the user pressing a Cancel key), without complicating or slowing the computation.

For example, a matrix multiply has only two real states: either the product was created successfully, or it was not. Generally, there is no need to know about the state of the product if it could not be completed. A matrix multiply can also take a very long time and should, therefore, be alertable. But because it involves several nested loops whose limits are parameters, it is not obvious where to place calls to `cma_alert_test` within it, or desirable to do so. To ensure that the matrix multiply is alertable, perform the following steps:

1. Set a flag indicating that the product is not available.
2. Disable asynchronous alerts to get the prior state.
3. Enable asynchronous delivery of alerts.
4. Perform multiplication (which may take a long time but is alertable).
5. Restore prior alert state.
6. Set a flag indicating the product is available.

Exceptions

cma_e_alerted

cma_alert_enable_general

Enables general delivery of alerts to the current thread.

Syntax

cma_alert_enable_general (prior)

Argument	Data Type	Access
prior	opaque cma_t_alert_state	write

C Binding

void
cma_alert_enable_general (
cma_t_alert_state **prior*);

Arguments

prior
Receives the prior state of general alert delivery (enabled or disabled).

Description

This routine enables general delivery of alerts to the current thread and returns the previous state of alert delivery to the *prior* argument. This routine is not alertable. A pending alert will remain pending until the next alert point.

Exceptions

cma_e_alert_nesting
cma_e_existence
cma_e_use_error

cma_alert_restore

Restores the state of general or asynchronous alert delivery.

Syntax

cma_alert_restore (prior)

Argument	Data Type	Access
prior	opaque cma_t_alert_state	read

C Binding

void
cma_alert_restore (
cma_t_alert_state *_prior_);

Arguments

prior
The prior state of alert delivery (enabled or disabled).

Description

This routine restores the alert delivery state (enabled or disabled) that was saved in the *prior* argument in a previous call to `cma_alert_disable_asynch`, `cma_alert_disable_general`, or `cma_alert_enable_general`.

For example, if general delivery of alerts is enabled when you call `cma_alert_disable_general`, the enabled setting is stored in *prior*. Subsequently calling `cma_alert_restore` and passing the *prior* value returned by `cma_alert_disable_general` causes the enabled setting to be restored, and general delivery of alerts is again enabled.

This routine cannot restore the state of alert delivery that existed before a call to `cma_alert_enable_asynch`. When you call `cma_alert_enable_asynch`, asynchronous delivery of alerts is enabled before the call returns. Therefore, return values might not be accurate.

This routine is not alertable. Because this routine is not alertable, an alert that is already pending will remain pending after this call returns. If a pending alert should be delivered immediately, follow a call to this routine with a call to an alertable routine such as `cma_alert_test`.

Exceptions

cma_e_alert_nesting
cma_e_badparam

cma_alert_test

Requests delivery of a pending alert to the current thread.

Syntax

cma_alert_test

C Binding

```
void
cma_alert_test ( );
```

Arguments

None

Description

This routine requests delivery of a pending alert to the current thread. The alert is delivered only if an alert is pending for the current thread and alert delivery is not currently disabled.

This routine is useful when called within very long loops to ensure that a pending alert is noticed within a reasonable amount of time.

This routine is alertable.

Exceptions

cma_e_alerted

cma_attr_create

Creates an attributes object.

Syntax

cma_attr_create (new_attr, attr)

Argument	Data Type	Access
new_attr	opaque cma_t_attr	write
attr	opaque cma_t_attr	read

C Binding

void
cma_attr_create (
cma_t_attr **new_attr*,
cma_t_attr **attr*);

Arguments

new_attr
Variable that receives the new attributes object.

attr
Handle of an attributes object used to specify attributes of the new attributes object. If you specify `cma_c_null` for the *attr* argument, default attributes are used.

Description

This routine creates an attributes object that can be used to specify the attributes of DECthreads objects when they are created.

The individual attributes (internal fields) of the attributes object are set to default values. (The default values of each attribute are discussed in the

descriptions of the following routines.) Use the following routines to change the individual attributes:

```
cma_attr_set_guardsize
cma_attr_set_inherit_sched
cma_attr_set_mutex_kind
cma_attr_set_priority
cma_attr_set_sched
cma_attr_set_stacksize
```

When an attributes object is used to create an object (for example, a thread or mutex), the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional arguments to object creation. Changing individual attributes does not affect any objects that were previously created using the attributes object.

When you set the scheduling policy or priority, or both, in an attributes object, you must disable scheduling inheritance before the scheduling attributes are used.

Exceptions

cma_e_existence
cma_e_use_error

cma_attr_delete

Deletes an attributes object.

Syntax

cma_attr_delete (attr)

Argument	Data Type	Access
attr	opaque cma_t_attr	modify

C Binding

void
cma_attr_delete (
cma_t_attr **attr*);

Arguments

attr
Handle of the attributes object deleted.

Description

This routine deletes an attributes object. Call this routine when an attributes object no longer needs to be referenced through the handle supplied by the `cma_attr_create` routine.

The effect of this routine is to give permission to reclaim storage for the attributes object. The attributes object is marked for deletion, and the *attr* argument is set to the value `cma_c_null`. Specifying `cma_c_null` for the *attr* argument is legal and has no effect. Objects that were created using this attributes object are not affected by the deletion of the attributes object.

The results of calling this routine are unpredictable if the handle specified by the *attr* argument refers to an attributes object that does not exist (unless it is `cma_c_null`.)

Exceptions

cma_e_existence
cma_e_use_error

cma_attr_get_guardsize

Obtains the guardsize attribute of thread creation.

Syntax

cma_attr_get_guardsize (attr, guardsize)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
guardsize	opaque cma_t_natural	write

C Binding

```
void
cma_attr_get_guardsize (
cma_t_attr *attr,
cma_t_natural *guardsize);
```

Arguments

attr

Handle of the attributes object whose guardsize attribute is obtained.

guardsize

Value of the guardsize attribute. The *guardsize* argument specifies the mimimum size (in bytes) of the guard area for the stack of a thread.

Description

This routine obtains the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas are necessary when threads might allocate large structures on the stack.

Exceptions

cma_e_existence
cma_e_use_error

cma_attr_get_inherit_sched

Obtains the inherit scheduling attribute of thread creation.

Syntax

cma_attr_get_inherit_sched (attr, setting)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
setting	opaque cma_t_sched_inherit	write

C Binding

void
cma_attr_get_inherit_sched (
cma_t_attr *_attr_,
cma_t_sched_inherit *_setting_);

Arguments

attr
Handle of the attributes object whose inherit scheduling attribute is obtained.

setting
Receives the value for the inherit scheduling attribute. Refer to the description of `cma_attr_set_inherit_sched` for valid values.

Description

This routine obtains the inherit scheduling attribute of thread creation. 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 `cma_thread_create`.

The default value of the inherit scheduling attribute is `cma_c_sched_inherit`.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_use_error

cma_attr_get_mutex_kind

Obtains the mutex type attribute.

Syntax

cma_attr_get_mutex_kind (attr, kind)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
kind	opaque cma_t_mutex_kind	write

C Binding

void
cma_attr_get_mutex_kind (
cma_t_attr **attr*,
cma_t_mutex_kind **kind*);

Arguments

attr
Handle of the attributes object whose mutex type is obtained.

kind
Value of the mutex type attribute. The *kind* argument specifies the type of mutex that is created. Valid values are `cma_c_mutex_fast` (default), `cma_c_mutex_recursive`, and `cma_c_mutex_nonrecursive`. (See Section 2.4.1 for definitions of the types of mutexes.)

Description

This service obtains the mutex type attribute that is used when a mutex is created. See the `cma_attr_set_mutex_kind` description for information about mutex type attributes.

Exceptions

cma_e_existence
cma_e_use_error

cma_attr_get_priority

Obtains the initial execution priority attribute of thread creation.

Syntax

cma_attr_get_priority (attr, priority)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
priority	opaque cma_t_priority	write

C Binding

void
cma_attr_get_priority (
cma_t_attr *_attr_,
cma_t_priority *_priority_);

Arguments

attr
Handle of the attributes object whose priority attribute is obtained.

priority
Receives the value of the priority attribute. Refer to the description of `cma_attr_set_priority` for valid values.

Description

This routine obtains the initial execution priority of threads created using the attributes object specified by the *attr* argument. The default value of the priority attribute is `cma_c_prio_through_mid`.

Exceptions

cma_e_existence
cma_e_use_error

cma_attr_get_sched

Obtains the scheduling policy attribute of thread creation.

Syntax

cma_attr_get_sched (attr, policy)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
policy	opaque cma_t_sched_policy	write

C Binding

void
cma_attr_get_sched (
cma_t_attr *_attr_,
cma_t_sched_policy *_policy_);

Arguments

attr
Handle of the attributes object whose scheduling policy attribute is obtained.

policy
Receives the value of the scheduling policy attribute. See the description of `cma_attr_set_sched` for valid values.

Description

This routine obtains the scheduling policy of threads created using the attributes object specified by the *attr* argument. The default value of the scheduling attribute is `cma_c_sched_default` (which maps to `cma_c_sched_throughput`).

Exceptions

cma_e_existence
cma_e_use_error

cma_attr_get_stacksize

Obtains the stacksize attribute of thread creation.

Syntax

cma_attr_get_stacksize (attr, stacksize)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
stacksize	opaque cma_t_natural	write

C Binding

void
cma_attr_get_stacksize (
cma_t_attr *_attr_,
cma_t_natural *_stacksize_);

Arguments

attr
Handle of the attributes object whose stacksize attribute is obtained.

stacksize
Value of the stacksize attribute. The *stacksize* argument specifies the mimimum size (in bytes) of the stack needed for a thread.

Description

This routine obtains the minimum size (in bytes) of the stack needed for a thread created using the attributes object specified by the *attr* argument.

Exceptions

cma_e_in_use
cma_e_use_error

cma_attr_set_guardsize

Changes the guardsize attribute of thread creation.

Syntax

cma_attr_set_guardsize (attr, guardsize)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
guardsize	opaque cma_t_natural	read

C Binding

void
cma_attr_set_guardsize (
cma_t_attr *_attr_,
cma_t_natural _guardsize_);

Arguments

attr
Handle of the attributes object modified.

guardsize
New value for the guardsize attribute. The *guardsize* argument specifies the mimimum size (in bytes) of the guard area for the stack of a thread.

Description

This routine sets the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas might be necessary when threads allocate large structures on the stack.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_use_error

cma_attr_set_inherit_sched

Changes the inherit scheduling attribute of thread creation.

Syntax

cma_attr_set_inherit_sched (attr, setting)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
setting	opaque cma_t_sched_inherit	read

C Binding

void
cma_attr_set_inherit_sched (
cma_t_attr **attr*,
cma_t_sched_inherit *setting*);

Arguments

attr
Handle of the attributes object modified.

setting
New value for the inherit priority attribute. Valid values are as follows:

`cma_c_sched_inherit`	This is the default value. The created thread inherits the current priority and scheduling policy of the thread calling `cma_thread_create`.
`cma_c_sched_use_default`	The created thread starts execution with the priority and scheduling policy stored in the attributes object.

Description

This routine changes the inherit scheduling attribute of thread creation. The inherit scheduling attribute specifies whether threads created using this attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the attributes object that is passed to `cma_thread_create`.

The initial thread in an application, which is not created by a call to `cma_thread_create`, has an initial priority of `cma_c_prio_through_mid` and a scheduling policy of `cma_c_sched_other`. See the `cma_attr_set_priority` and `cma_attr_set_sched` routines for more information on valid priority values and valid scheduling policy values, respectively.

Inheriting scheduling attributes (instead of using the scheduling attributes stored in the attributes object) is useful when a thread is creating several helper threads—threads that are intended to work closely with the creating thread to cooperate in solving the same problem. For example, inherited scheduling attributes allow you to ensure that any helper threads created in a sort routine execute with the same priority as the calling thread.

Note

You must set scheduling inheritance to `cma_c_sched_use_default` if you want to create threads with a scheduling policy or priority different from the creating thread.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_use_error

cma_attr_set_mutex_kind

Specifies the mutex type attribute.

Syntax

cma_attr_set_mutex_kind (attr, kind)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
kind	opaque cma_t_mutex_kind	read

C Binding

void
cma_attr_set_mutex_kind (
cma_t_attr **attr*,
cma_t_mutex_kind *kind*);

Arguments

attr

Handle of the attributes object modified.

kind

New value for the mutex type attribute. The *kind* argument specifies the type of mutex that is created. Valid values are `cma_c_mutex_fast` (default), `cma_c_mutex_recursive`, and `cma_c_mutex_nonrecursive`.

Description

This rouine sets the mutex type attribute that is used when a mutex is created. A mutex can be fast, recursive, or nonrecursive. See Section 2.4.1 for more information on the types of mutexes.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_use_error

cma_attr_set_priority

Changes the initial execution priority attribute of thread creation.

Syntax

cma_attr_set_priority (attr, priority)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
priority	opaque cma_t_priority	read

C Binding

void
cma_attr_set_priority (
cma_t_attr *_attr_,
cma_t_priority _priority_);

Arguments

attr
Handle of the attributes object modified.

priority
New value for the priority attribute. The priority attribute is dependent upon scheduling policy. Valid values are as follows:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

The default priority is `cma_c_prio_default_mid`. (This symbol maps to `cma_c_prio_through_mid`.)

Description

This routine sets the initial execution priority of threads that were created using the attributes object specified by the *attr* argument. The default value of the priority attribute is `cma_c_prio_default_mid`.

Note

You must set scheduling inheritance to `cma_c_sched_use_default` if you want to create threads with a scheduling policy or priority different from the creating thread.

An application should specify priority only to express the urgency of executing the thread relative to other threads. Priority should not be used to control mutual exclusion when accessing shared data. With a sufficient number of processors executing, all ready threads, regardless of priority, can be executing simultaneously.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_use_error

cma_attr_set_sched

Changes the scheduling policy attribute of thread creation.

Syntax

cma_attr_set_sched (attr, policy, priority)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
policy	opaque cma_t_sched_policy	read
priority	opaque cma_t_priority	read

C Binding

void
cma_attr_set_sched (
cma_t_attr *_attr_,
cma_t_sched_policy _policy_,
cma_t_priority _priority_);

Arguments

attr
Handle of the attributes object modified.

policy
New value for the scheduling policy attribute. Valid values are as follows:

```
cma_c_sched_fifo
cma_c_sched_rr
cma_c_sched_throughput
cma_c_sched_background
```

See Section 2.3.3.2 for a description of the scheduling policies.

priority
New value for the priority attribute. The priority attribute is dependent upon scheduling policy. Valid values are as follows:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

The default priority is `cma_c_prio_default_mid`. (This symbol maps to `cma_c_prio_through_mid`.)

Description

This routine sets the scheduling policy and priority of a thread that is created using the attributes object specified by the *attr* argument. The default value of the scheduling policy attribute is `cma_c_sched_default`. (This symbol is mapped to the `cma_c_sched_throughput` scheduling policy.) The default scheduling priority is the midrange of the default scheduling policy.

Note

You must set scheduling inheritance to `cma_c_sched_use_default` if you want to create threads with a scheduling policy or priority different from the creating thread.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_use_error

cma_attr_set_stacksize

Changes the stacksize attribute of thread creation.

Syntax

cma_attr_set_stacksize (attr, stacksize)

Argument	Data Type	Access
attr	opaque cma_t_attr	read
stacksize	opaque cma_t_natural	read

C Binding

void
cma_attr_set_stacksize (
cma_t_attr *_attr_,
cma_t_natural _stacksize_);

Arguments

attr
Handle of the attributes object modified.

stacksize
New value for the stacksize attribute. The *stacksize* argument specifies the mimimum size (in bytes) of the stack needed for a thread.

Description

This routine sets the minimum size (in bytes) of the stack needed for a thread created using the attributes object specified by the *attr* argument.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_stackovf
cma_e_use_error

cma_cond_broadcast

Wakes all threads that are waiting on a condition variable.

Syntax

cma_cond_broadcast (condition)

Argument	Data Type	Access
condition	opaque cma_t_cond	read

C Binding

void
cma_cond_broadcast (
cma_t_cond *_condition_);

Arguments

condition
Handle of the condition variable broadcast.

Description

This routine wakes all threads waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so that it might be possible for more than one waiting thread to proceed. If only one waiting thread might be able to proceed, call `cma_cond_signal`.

You can call this routine when the associated mutex is either locked or unlocked.

Exceptions

cma_e_existence
cma_e_use_error

cma_cond_create

Creates a condition variable.

Syntax

cma_cond_create (new_condition, attr)

Argument	Data Type	Access
new_condition	opaque cma_t_cond	write
attr	opaque cma_t_attr	read

C Binding

void
cma_cond_create (
cma_t_cond *_new_condition_,
cma_t_attr *_attr_);

Arguments

new_condition
Variable that receives the handle for the new condition variable.

attr
Handle of the attributes object that defines the characteristics of the condition variable being created.

Description

This routine creates and initializes a condition variable. A condition variable is a synchronization object used in conjunction with a mutex. 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.

A condition variable can be signaled or broadcast to indicate that a predicate might have become true. The broadcast routine indicates that all waiting threads should resume and reevaluate the predicate. The signal routine can be used in the special case where only one waiting thread can continue.

If a thread that holds a mutex determines that the shared data is not in the correct state for it to proceed (the associated predicate is not true), it can wait on a condition variable associated with the desired state. Waiting on the condition variable automatically releases the mutex so that other threads can modify or examine the shared data. When a thread modifies the state of the shared data so that a predicate might be true, it signals or broadcasts on the appropriate condition variable so that threads waiting for that predicate can continue.

It is important that all threads waiting on a particular condition variable at any time hold the *same* mutex. At any time, an arbitrary number of condition variables can be associated with a single mutex, each representing a different predicate of the shared data protected by that mutex.

Condition variables are not owned by a particular thread. Any associated storage is not automatically deallocated when the creating thread terminates.

Exceptions

cma_e_existence
cma_e_use_error

cma_cond_delete

Deletes a condition variable.

Syntax

cma_cond_delete (condition)

Argument	Data Type	Access
condition	opaque cma_t_cond	modify

C Binding

void
cma_cond_delete (
cma_t_cond **condition*);

Arguments

condition
Handle of the condition variable deleted.

Description

This routine deletes a condition variable. Call this routine when a condition variable is no longer referenced. The effect of calling this routine is to give permission to reclaim storage for the condition variable.

When the condition variable is deleted, the *condition* argument is set to the value `cma_c_null`. Specifying `cma_c_null` for *condition* is legal and has no effect.

The results of this routine are unpredictable and the `cma_e_existence` exception is raised if the handle specified in *condition* refers to a condition variable that does not currently exist (unless it is `cma_c_null`).

The results of this routine are unpredictable and the `cma_e_in_use` exception is raised if there are threads waiting for the specified condition variable to be signaled or broadcast when it is deleted.

Exceptions

cma_e_existence
cma_e_in_use
cma_e_use_error

cma_cond_signal

Wakes one thread that is waiting on a condition variable.

Syntax

cma_cond_signal (condition)

Argument	Data Type	Access
condition	opaque cma_t_cond	read

C Binding

void
cma_cond_signal (
cma_t_cond *_condition_);

Arguments

condition
Handle of the condition variable signaled.

Description

This routine wakes one thread waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so 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, but only one thread should proceed.

You can call this routine when the associated mutex is either locked or unlocked.

If you want to signal a thread from interrupt level, use `cma_cond_signal_int`.

Exceptions

cma_e_existence
cma_e_use_error

cma_cond_signal_int

Wakes one thread that is waiting on a condition variable. This routine can only be called from interrupt level.

Syntax

cma_cond_signal (condition)

Argument	Data Type	Access
condition	opaque cma_t_cond	read

C Binding

void
cma_cond_signal_int (
cma_t_cond *_condition_);

Arguments

condition
Handle of the condition variable signaled.

Description

This routine wakes one thread waiting on a condition variable. This routine can only be called from interrupt level. 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.

This routine does not cause a thread blocked on a condition variable to resume execution immediately. A thread resumes execution after the interrupt handler returns.

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

Note

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 `cma_cond_signal`.

Exceptions

cma_e_existence
cma_e_use_error

cma_cond_timed_wait

Causes a thread to wait for a condition variable to be signaled or broadcast for a specified period of time.

Syntax

status = cma_cond_timed_wait (condition, mutex, expiration)

Argument	Data Type	Access
status	opaque cma_t_status	write
condition	opaque cma_t_cond	read
mutex	opaque cma_t_mutex	read
expiration	opaque cma_t_date_time	read

C Binding

cma_t_status
cma_cond_timed_wait (
cma_t_cond *_condition_,
cma_t_mutex *_mutex_,
cma_t_date_time *_expiration_);

Arguments

status
Variable that receives the return status. Valid values are `cma_s_normal` and `cma_s_timed_out`.

condition
Handle of the condition variable on which the thread waits.

mutex
Mutex associated with the condition variable specified in *condition*.

expiration
Absolute time at which the wait should expire, if the condition has not yet been signaled or broadcast. (See the `cma_time_get_expiration` routine, which can be used to obtain a value for this argument.)

Description

This routine causes a thread to wait until:

- A condition variable is signaled or broadcasted.
- The current system clock time is greater than or equal to the time specified by the *expiration* argument.

This routine is identical to `cma_cond_wait` except that this routine can return before a condition variable is signaled or broadcast, specifically, when a specified time expires.

If the wait is completed normally by a signal or broadcast on the condition variable, the return status is `cma_s_normal`. If the wait completes because the expiration time has passed, the return status is `cma_s_timed_out`.

If the current time equals or exceeds the expiration time, this routine returns immediately, without causing the current thread to wait. 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 non-blocking 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.

Exceptions

cma_e_alerted
cma_e_existence
cma_e_use_error

cma_cond_wait

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

Syntax

cma_cond_wait (condition, mutex)

Argument	Data Type	Access
condition	opaque cma_t_cond	read
mutex	opaque cma_t_mutex	read

C Binding

void
cma_cond_wait (
cma_t_cond *_condition_,
cma_t_mutex *_mutex_);

Arguments

condition
Handle of the condition variable on which the thread waits.

mutex
Mutex associated with the condition variable specified in *condition*.

Description

This routine causes a thread to wait for a condition variable to be signaled or broadcasted. Each condition corresponds to one or more predicates upon the data. The calling thread waits for the data to reach a particular state (for a particular predicate to become true).

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. If the wait is satisfied as a result of some thread calling `cma_cond_signal` or `cma_cond_broadcast`, the mutex is reacquired and the routine returns.

As a general rule, a thread that has changed 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 `cma_cond_signal` or `cma_cond_broadcast` for that condition variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This routine is alertable. Alertable means that a pending alert is noticed during the wait on the condition variable. This helps ensure that all long waits can be canceled by alerting the thread.

You must disable asynchronous delivery of alerts in contexts where this routine is used.

If the `cma_e_alerted` exception is raised, the mutex is reacquired before the exception is raised.

This routine might (with low probability) return when the condition variable has not been signaled or broadcast. When a spurious wakeup occurs, the mutex is reacquired before the routine returns. (To handle this type of situation, this routine should always be enclosed in a loop that tests for the desired shared data state.)

Exceptions

cma_e_alerted
cma_e_existence
cma_e_use_error

cma_debug

Invokes the DECthreads internal debugger.

Syntax

cma_debug

C Binding

void
cma_debug (void);

Arguments

None

Description

This routine invokes the DECthreads internal debugger as a callable function. It takes no arguments and does not return a value. It enters the internal debugger parsing loop. Type exit to return to the program.

To pass a list of debugging commands to DECthreads, call `cma_debug_cmd`.

Exceptions

None

cma_debug_cmd

Passes a list of cma_debug commands to DECthreads.

Syntax

cma_debug_cmd (command)

Argument	Data Type	Access
command	character string	read

C Binding

void
cma_debug_cmd (char *cmd);

Arguments

command
cma_debug command string. Null terminated string, commands separated by semicolons.

Description

This routine passes a list of debugging commands to DECthreads. Each command is executed in sequence. Any output is written to standard output. This routine returns when the final command (or Exit command) is executed.

For a list of cma_debug commands, see the appendix specific to your system.

Following are two examples of calling this routine:

```
cma_debug_cmd ("thread -b; mu -lq; cond -wq");
cma_debug_cmd ("att");
```

If you want to invoke the debugger for interactive commands, call cma_debug.

Exceptions

None

cma_delay

Causes a thread to wait for a specified period of time before continuing execution.

Syntax

cma_delay (time_interval)

Argument	Data Type	Access
time_interval	single precision floating point	read

C Binding

void
cma_delay (
cma_t_interval *interval*);

Arguments

interval
Number of seconds that the calling thread waits before continuing execution. Specify a value greater than or equal to 0.

Description

This routine causes a thread to delay execution for a specified period of elapsed time. The period of time the thread waits is at least as long as the number of seconds specified in the *time_interval* argument.

If you specify a value for *time_interval* that is less than 0, the `cma_e_badparam` exception is raised. Specifying 0 for *time_interval* is allowed and can result in the thread giving up the processor or delivering a pending alert.

This routine is alertable.

Exceptions

cma_e_alerted
cma_e_badparam

cma_handle_assign

Assigns a handle to an object.

Syntax

cma_handle_assign (handle1, handle2)

Argument	Data Type	Access
handle1	opaque cma_t_handle	read
handle2	opaque cma_t_handle	write

C Binding

void
cma_handle_assign (
cma_t_handle **handle1,*
cma_t_handle **handle2*);

Arguments

handle1
Handle that is assigned to *handle2*.

handle2
Handle that receives the value from *handle1*.

Description

This routine assigns the value of a handle, or name, to an object. Handles are allocated by the user application. This routine allows you to copy a handle from one object to another.

When an object is created, the storage for the object is allocated and initialized, and a handle for the object is returned. The handle is the only means of referring to an object and performing routines on the object. Because objects are only accessed through handles, you can usually think of the handle as if it were the object itself.

Handles are meaningful only within a single process address space. Attempting to access an object from a process other than the one in which it was created—for example, by means of multiply mapped memory—can result in unpredictable results (it is incorrect, but the error is not necessarily checked).

Exceptions

None

cma_handle_equal

Compares one handle to another handle.

Syntax

equal = cma_handle_equal (handle1, handle2)

Argument	Data Type	Access
equal	Boolean cma_t_boolean	write
handle1	opaque cma_t_handle	read
handle2	opaque cma_t_handle	read

C Binding

cma_t_boolean
cma_handle_equal (
cma_t_handle **handle1,*
cma_t_handle **handle2*);

Arguments

equal

Boolean value that specifies whether *handle1* and *handle2* designate the same object.

handle1

The first handle to be compared.

handle2

The second handle to be compared.

Description

This routine compares one handle to another handle. (This routine does not check whether the objects that correspond to the handles currently exist.) The value `cma_c_true` is returned if the handles have values indicating that they designate the same object. If the values do not designate the same object, the value `cma_c_false` is returned.

Exceptions

None

cma_init

Initializes the DECthreads routines.

Syntax

cma_init

C Binding

```
void
cma_init ( );
```

Arguments

None

Description

This routine initializes the internal storage and process-wide state that is necessary to support DECthreads routines.

Note

Call this routine before calling any other cma routine.

Calling this routine can result in changes to the execution environment that cause a single-threaded process to appear as a DECthreads thread.

If this routine is called more than once, the second and subsequent calls are ignored. If a second call to this routine is made while the first call is still in progress, the exception `cma_e_inialrpro` is raised.

Exceptions

cma_e_inialrpro

cma_key_create

Generates a unique per-thread context key value.

Syntax

cma_key_create (key, attr, destructor)

Argument	Data Type	Access
key	opaque cma_t_key	write
attr	opaque cma_t_attr	read
destructor	procedure cma_t_destructor	read

C Binding

void
cma_key_create (
cma_t_key *_key_,
cma_t_attr *_attr_,
cma_t_destructor _destructor_);

Arguments

key
Receives the value of the new per-thread context key.

attr
Handle of the attributes object that defines the characteristics of the per-thread context key being created.

destructor
Procedure called to destroy a context value associated with this key when the thread terminates.

Description

This routine generates a unique per-thread context key value. This key value identifies a per-thread context, which is an address of memory generated by the client containing arbitrary data of any size.

Per-thread context is a mechanism that allows client software to associate context information with the current thread. (This mechanism can be thought of as a means for a client to add its own unique fields to the thread control block.)

This routine generates and returns a new key value. Each call to this routine within a process returns a key value 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. (Refer to the description of `cma_once` for more information.)

When multiple facilities share access to per-thread context, the facilities must agree on the key value that is associated with the context. The key value must be created only once, and should be stored in a location known to each facility. (It may be desirable to encapsulate the creation of a key, and the setting and getting of context values for that key, within a special facility created for that purpose.)

When a thread terminates, per-thread context is automatically destroyed. For each per-thread context currently associated with the thread, the destructor routine associated with the key value of that context is called.

Exceptions

cma_e_existence
cma_e_use_error

cma_key_get_context

Obtains the per-thread context associated with the specified key.

Syntax

cma_key_get_context (key, context_value)

Argument	Data Type	Access
key	opaque cma_t_key	read
context_value	opaque cma_t_address	write

C Binding

void
cma_key_get_context (
cma_t_key *key*,
cma_t_address **context_value*);

Arguments

key

Context key value that uniquely identifies the context value obtained. This key value must have been obtained from `cma_key_create`.

context_value

Variable that receives the address of the current per-thread context value associated with the specified key.

Description

This routine obtains the per-thread context associated with the specified key for the current thread. If a context has not been defined for the key in this thread, the null pointer `cma_c_null_ptr` is returned in *context_value*.

The exception `cma_e_badparam` is raised if the context key is invalid.

Exceptions

cma_e_badparam

cma_key_set_context

Sets the per-thread context associated with the specified key for the current thread.

Syntax

cma_key_set_context (key, context_value)

Argument	Data Type	Access
key	opaque cma_t_key	read
context_value	opaque cma_t_address	read

C Binding

void
cma_key_set_context (
cma_t_key *key*,
cma_t_address *context_value*);

Arguments

key

Context key value that uniquely identifies the context value specified in *context_value*. This key value must have been obtained from `cma_key_create`.

context_value

Address containing data associated with the specified key for the current thread; this is the per-thread context.

Description

This routine sets the per-thread context associated with the specified key for the current thread. If a context has been defined for the key in this thread (the current value is not null), the new value is substituted for it.

The exception `cma_e_badparam` is raised if the context key is invalid.

Exceptions

cma_e_badparam

cma_lock_global

Locks the global mutex.

Syntax

cma_lock_global

C Binding

void
cma_lock_global ( );

Arguments

None

Description

This routine locks the global mutex. If the global mutex is currently locked by another thread when a thread calls this routine, the calling thread waits for the global mutex to become available.

The thread that has locked the global mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the global mutex in the locked state and with the current thread as the global mutex's current owner.

Use the 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 compatible with multithreading, 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 isn't known to be reentrant uses the same lock. This prevents dependencies among threads calling 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 `cma_unlock_global` as many times as it called this routine to allow another thread to lock the global mutex.)

Exceptions

cma_e_existence
cma_e_use_error

cma_mutex_create

Creates a mutex.

Syntax

cma_mutex_create (new_mutex, attr)

Argument	Data Type	Access
new_mutex	opaque cma_t_mutex	write
attr	opaque cma_t_attr	read

C Binding

void
cma_mutex_create (
cma_t_mutex *_new_mutex,_
cma_t_attr *_attr_);

Arguments

new_mutex
Receives a handle for the mutex.

attr
Handle of the attributes object that defines the characteristics of the mutex created. If you specify `cma_c_null`, default attributes are used.

Description

This routine creates a mutex. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

The mutex is created and initialized to the unlocked state.

If the thread that called this routine terminates, the created mutex is not automatically deallocated because it is considered to be shared among multiple threads.

Exceptions

cma_e_existence
cma_e_use_error

cma_mutex_delete

Deletes a mutex.

Syntax

cma_mutex_delete (mutex)

Argument	Data Type	Access
mutex	opaque cma_t_mutex	modify

C Binding

void
cma_mutex_delete (
cma_t_mutex *mutex);

Arguments

mutex
Handle of the mutex deleted. After the call to this routine, the *mutex* argument is set to the value `cma_c_null`.

Description

This routine deletes a mutex and should be called when a mutex is no longer referenced. Calling this routine reclaims storage for the mutex object.

After the mutex is deleted, the *mutex* argument is set to the value `cma_c_null`. Calling this routine and specifying a value of `cma_c_null` for the *mutex* argument is legal and has no effect.

Do not delete a mutex that has a current owner (in other words, is locked). If you try to delete a mutex that is locked, the `cma_e_in_use` exception is raised.

The results of this routine are unpredictable if the handle specified in the *mutex* argument refers to a mutex object that does not currently exist (unless it is `cma_c_null`).

Exceptions

cma_e_existence
cma_e_in_use
cma_e_use_error

cma_mutex_lock

Locks a mutex if the mutex is unlocked. If the mutex is locked, causes the thread to wait for the mutex to become available.

Syntax

cma_mutex_lock (mutex)

Argument	Data Type	Access
mutex	opaque cma_t_mutex	read

C Binding

void
cma_mutex_lock (
cma_t_mutex *_mutex_);

Arguments

mutex
Handle of the mutex locked.

Description

This routine locks a mutex. If the specified mutex is already locked when a thread calls this routine, the thread waits for the mutex to become available.

The thread that has locked a mutex becomes its current owner and remains the owner until the same thread (and only that 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.

If you specified a fast mutex, a deadlock can result if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time. If you specified a recursive mutex, the current owner of a mutex can relock the same mutex without blocking. If the current owner of a nonrecursive mutex calls this routine in an attempt to lock the mutex a second time, the exception `cma_e_in_use` is raised. See cma_attr_set_mutex_kind for information about fast, recursive, and nonrecursive mutexes.

Before you lock a mutex, disable asynchronous alert delivery.

Exceptions

cma_e_existence
cma_e_in_use
cma_e_use_error

cma_mutex_try_lock

Locks a mutex. If the mutex is already locked, the calling thread does not wait for the mutex to become available.

Syntax

status = cma_mutex_try_lock (mutex)

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
mutex	opaque cma_t_mutex	read

C Binding

cma_t_boolean
cma_mutex_try_lock (
cma_t_mutex *_mutex_);

Arguments

status
Boolean value that specifies whether the element was successfully locked. Valid values are `cma_c_true` and `cma_c_false`.

mutex
Handle of the mutex to be locked.

Description

This routine locks a mutex. If the specified mutex is already locked when a thread calls this routine, the calling thread does not wait for the mutex to become available.

When a thread calls this routine, an attempt is made to immediately lock the mutex. If the mutex is successfully locked, the Boolean value `cma_c_true` is returned. The current thread is then the mutex's current owner.

If the mutex is already locked when this routine is called (even if it was previously locked by the current thread), the Boolean value `cma_c_false` is returned and the thread does not wait to acquire the lock.

Before you lock a mutex, disable asynchronous alert delivery.

Exceptions

cma_e_existence
cma_e_use_error

cma_mutex_unlock

Unlocks a mutex.

Syntax

cma_mutex_unlock (mutex)

Argument	Data Type	Access
mutex	opaque cma_t_mutex	read

C Binding

void
cma_mutex_unlock (
cma_t_mutex *_mutex_);

Arguments

mutex
Handle of the mutex unlocked.

Description

This routine unlocks a mutex. If there are no threads waiting for the mutex, the mutex becomes unlocked with no current owner. If one or more threads are waiting to lock the specified mutex, calling this routine causes one thread to unblock and try to acquire the mutex .

The results of calling this routine are unpredictable if the mutex specified in *mutex* is already unlocked. In that case, the exception `cma_e_use_error` is raised.

The results of calling this routine are also unpredictable if the mutex specified in *mutex* is currently owned by a thread other than the calling thread. In that case, the exception `cma_e_in_use` is raised.

Exceptions

cma_e_existence
cma_e_in_use
cma_e_use_error

cma_once

Calls an initialization routine that can be executed by only one thread, a single time.

Syntax

cma_once (init_block, init_routine, arg)

Argument	Data Type	Access
init_block	opaque cma_t_once	read
init_routine	opaque cma_t_init_routine	read
arg	opaque cma_t_address	read

C Binding

void
cma_once (
cma_t_once **init_block*,
cma_t_init_routine **init_routine*,
cma_t_address *arg*);

Arguments

init_block
Address of a record that defines the one-time initialization code. Each one-time initialization routine must have its own unique *cma_t_once*.

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 *init_block* are passed to cma_once.

arg
Argument passed to the *init_routine*.

Description

This routine calls an initialization routine that can be executed by only one thread, a single time. This routine allows you to create your own initialization code that is guaranteed to be run only once, even if called simultaneously by multiple threads.

For example, a mutex or a per-thread context key must be created exactly once. Calling `cma_once` prevents the problem that occurs when the code that creates a mutex or per-thread context can be called by multiple threads. Without this routine, the execution must be serialized so that only one thread performs the initialization. Other threads that reach the same point in the code would be delayed until the first thread is finished.

This routine initializes the control record if it has not already been initialized, and then determines if the client one-time initialization routine has already executed once. If it has not executed, then this routine calls the initialization routine specified in *init_routine*. If the client one-time initialization code has already executed once, then this routine returns.

Because the *init_routine* accepts an argument (*arg*), a single initialization routine can be used to initialize any number of objects. For example, an initialization routine that creates a global mutex might take the address of a *cma_t_mutex* variable, which receives the handle of a new mutex. Note that you cannot make every call to the initialization routine using the same control block; it would only be called once. Effectively, each value of *arg* must be associated with its own contol block.

Note

If you specify an *init_routine* that directly or indirectly results in a recursive call to `cma_once` specifying the same *init_block* argument, the recursive call will result in a deadlock.

The *init_block* must be declared static (for example, either `extern` or `static` in the C language), and it must be initialized at compile time. In the C language, using `cma.h`, initialize an *init_block* using the `cma_once_init` macro. In other languages, you must initialize a `cma_t_once` block to a value of three integer zeroes. In C, that corresponds to the following:

```
static cma_t_once block = {0,0,0};
```

The result of this routine is unpredictable and the `cma_e_badparam` exception is raised if the *init_block* is not a properly initialized one-time initialization block.

Exceptions

cma_e_badparam

Example

The following C code segment declares a one-time initialization section that creates a mutex for later use:

```
#include <CMA.H>
static cma_t_once make_my_mutex = cma_once_init; ❶
static cma_t_mutex        my_mutex; ❷
      .
      .
      .
void initialize_mutex (cma_t_address_arg) {  ❸
    cma_mutex_create (&my_mutex, &cma_c_null);
    }
    .
    .
    .
    cma_once (&make_my_mutex, initialize mutex, 0); ❹
```

❶ Declare the cma_t_once that defines the particular one-time initialization code.

❷ Declare the mutex to be initialized.

❸ Declare the initialization routine that will create the mutex.

❹ Call cma_once with the initialization control block and routine. If no thread has already executed the initialization routine, it will be called. Otherwise, cma_once returns.

cma_stack_check_limit_np

Determines whether sufficient space exists on the current thread's stack to allocate the requested number of bytes of local storage.

Syntax

status = cma_stack_check_limit_np (size)

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
size	opaque cma_t_integer	read

C Binding

cma_t_boolean
cma_stack_check_limit_np (
cma_t_integer *size*);

Arguments

status
Boolean value that specifies whether the requested size can be allocated without overflowing the current thread's stack.

size
Number of bytes requested. DECthreads determines whether a stack allocation of the specified size extends beyond the end of the thread's stack.

Description

This routine determines whether sufficient space exists on the current thread's stack to allocate the requested number of bytes of local storage. If the requested size fits (if it does not extend beyond the current thread's stack), `cma_c_true` is returned. If the requested size extends beyond the end of the stack, `cma_c_false` is returned.

A DECthreads stack consists of the following three parts:

- A **green zone**, which is the normal area where procedure activation frames and stack automatic variables are allocated.
- A **reserved zone**, which is available for allocation but indicates that the thread has almost reached the end of the stack. (A thread should not use the reserved zone; it has been set aside for use by error handling mechanisms.)
- A **guard zone**, which is normally protected with no access so that an attempt by the thread to read or write will fail with a hardware error.

If the thread does not use the `cma_stack_check_limit_np` routine, it is possible for a thread to skip over the guard zone of its stack by, for example, allocating a very large array on the stack. If the thread writes to the part that overlaps the other stack before attempting to access the part in its own guard zone, it could corrupt another thread's stack. This results in unpredictable behavior of the application and is difficult to debug. The `cma_stack_check_limit_np` routine recognizes that situation and returns `cma_c_false`.

When `cma_stack_check_limit_np` is called from the main thread that is running on the default process stack, `cma_stack_check_limit_np` attempts to access each page that would fall within the requested allocation. It returns `cma_c_true` unless the system is unable to expand the process stack to the needed size.

Exceptions

cma_e_existence
cma_e_use_error

cma_thread_alert

Cancels thread execution.

Syntax

cma_thread_alert (thread)

Argument	Data Type	Access
thread	opaque cma_t_thread	read

C Binding

void
cma_thread_alert (
cma_t_thread *_thread_);

Arguments

thread
Handle of the thread that receives an alert.

Description

This routine sends an alert to the specified thread. Issuing an alert does not guarantee that the alerted thread will receive or handle the alert. The alerted thread can delay processing the alert after receiving it. For instance, if an alert arrives during an important operation, the alerted thread can continue if what it is doing cannot be interrupted at the point where the alert is requested.

Because of communication delays, the calling thread can only rely on the fact that an alert will eventually become pending in the designated thread (provided that the thread does not terminate beforehand). Furthermore, the calling thread has no guarantee that a pending alert will be delivered because delivery is controlled by the designated thread.

The results of this routine are unpredictable, and the `cma_e_existence` exception is raised if the value specified in *thread* is `cma_c_null`, or if it refers to a thread that does not currently exist.

This routine is not alertable.

Exceptions

cma_e_existence
cma_e_use_error

cma_thread_create

Creates a thread object and thread.

Syntax

cma_thread_create (new_thread, attr, start_routine, arg)

Argument	Data Type	Access
new_thread	opaque cma_t_thread	write
attr	opaque cma_t_attr	read
start_routine	cma_t_start_routine	read
arg	pointer	read

C Binding

void
cma_thread_create (
cma_t_thread **new_thread,*
cma_t_attr **attr,*
cma_t_start_routine *start_routine,*
cma_t_address *arg*);

Arguments

new_thread
Variable that receives a handle for the thread object.

attr
Handle of the attributes object that defines the characteristics of the thread being created. If you specify `cma_c_null`, default attributes are used.

start_routine
Function executed as the new thread's start routine. This argument is the address of a routine that takes one argument of type `cma_t_address`, and returns a value of type `cma_t_address`.

arg
Address value that is copied and passed to the thread's start routine.

Description

This routine creates a thread object and a thread. The thread routine is a function of type cma_t_start_routine. The function accepts a single argument of type cma_t_address and returns a function value of type cma_t_address. For example, the following routine coded in Ada, is compatible with the cma_t_start_routine type:

```
function START_ROUTINE (
            ARG             : in CMA_T_ADDRESS) return CMA_T_ADDRESS;
```

The same example coded in C, is as follows:

```
cma_t_address
start_routine (
    cma_t_address           arg);
```

Calling this routine sets into motion the following actions:

- An internal thread object is created to describe the thread.
- The associated executable thread is created with attributes specified by the *attr* argument (or with default attributes if cma_c_null is specified.)
- The *new_thread* argument receives the handle of the new thread.
- The *start_routine* function is called.

The thread is created in the ready state and therefore might immediately begin executing the function specified by the *start_routine* argument. The newly created thread will begin running before cma_thread_create completes if the new thread follows the cma_c_sched_rr or cma_c_sched_fifo scheduling policy or has a priority higher than the creating thread, or both. Otherwise, the new thread begins running at its turn, which might also be before cma_thread_create returns.

The *start_routine* is passed a copy of the *arg* argument. The value of the *arg* argument is specified by the calling application code.

The thread object exists until the cma_thread_detach routine is called and the thread terminates, whichever occurs last.

Synchronization between the caller of cma_thread_create and the newly created thread is done through the use of the cma_thread_join routine (or any other mutexes or condition variables they agree to use).

Exceptions

cma_e_existence
cma_e_use_error

cma_thread_detach

Marks a thread object for deletion.

Syntax

cma_thread_detach (thread)

Argument	Data Type	Access
thread	opaque cma_t_thread	modify

C Binding

void
cma_thread_detach (
cma_t_thread **thread*);

Arguments

thread
Handle of the thread object marked for deletion.

Description

This routine indicates that storage for the specified thread can be reclaimed when the thread terminates. If the thread object is no longer needed by the thread, then the thread object is deallocated immediately. The *thread* argument is set to the value `cma_c_null`.

Call this routine when no other threads are interested in joining with the thread. Call this routine where appropriate for every thread that is created to ensure that storage for thread objects does not accumulate.

Once this routine has been called, other threads cannot join with the detached thread.

Calling this routine for a value for *thread* of `cma_c_null` is legal and has no effect.

The results of this routine are unpredictable if the value of *thread* refers to a thread object that does not currently exist.

Exceptions

cma_e_existence
cma_e_use_error

cma_thread_exit_error

Terminates the current thread when an error occurs.

Syntax

cma_thread_exit_error

C Binding

```
void
cma_thread_exit_error ( );
```

Arguments

None

Description

This routine terminates execution of the current thread within an arbitrary routine when an error occurs. Normally, a thread terminates when the *start_routine* argument to `cma_thread_create` returns.

Call this routine only when an error occurs that requires thread termination and you do not want to signify the error by raising an exception. (Raising an exception is the preferred means of indicating errors however, an unhandled exception will terminate the program.)

Exceptions

None

cma_thread_exit_normal

Terminates the current thread when successful completion occurs prematurely.

Syntax

cma_thread_exit_normal (result)

Argument	Data Type	Access
result	opaque cma_t_address	read

C Binding

void
cma_thread_exit_normal (
cma_t_address *result*);

Arguments

result
Address value that is copied and returned to the caller of `cma_thread_join`.

Description

This routine terminates execution of the current thread within an arbitrary routine when successful completion occurs prematurely.

Normally, a thread terminates when the *start_routine* argument to `cma_thread_create` returns. Call this routine when it is not necessary or convenient to allow the thread's start routine to return normally to its caller.

Exceptions

None

cma_thread_get_priority

Obtains the current priority of a thread.

Syntax

cma_thread_get_priority (thread, priority)

Argument	Data Type	Access
thread	opaque cma_t_thread	read
priority	opaque cma_t_priority	write

C Binding

void
cma_thread_get_priority (
cma_t_thread *thread*,
cma_t_priority **priority*);

Arguments

thread
Handle of the thread whose priority is obtained.

priority
Variable that receives the current priority value of the thread specified in *thread*. Refer to the description of `cma_thread_set_priority` for valid values.

Description

This routine obtains the current priority of a thread. The current priority can be different from the initial priority of the thread if the `cma_thread_set_priority` routine has been called, or if the thread's scheduling policy dynamically modifies thread priorities.

The exact effect of different priority values is dependent upon the scheduling policy assigned to the thread.

Exceptions

cma_e_existence
cma_e_use_error

cma_thread_get_sched

Obtains the current scheduling policy of a thread.

Syntax

cma_thread_get_sched (thread, policy)

Argument	Data Type	Access
thread	opaque cma_t_thread	read
policy	opaque cma_t_sched_policy	write

C Binding

void
cma_thread_get_sched (
cma_t_thread *_thread_,
cma_t_sched_policy *_policy_);

Arguments

thread
Handle of the thread whose scheduling policy is obtained.

policy
Variable that receives the current scheduling policy value of the thread specified in *thread*. Refer to the description of `cma_thread_set_sched` for valid values.

Description

This routine obtains the current scheduling policy of a thread. The current scheduling policy of a thread can be different from the initial scheduling policy if the `cma_thread_set_sched` routine has been called.

Exceptions

cma_e_existence
cma_e_use_error

cma_thread_get_self

Obtains the handle of the current thread.

Syntax

cma_thread_get_self (thread)

Argument	Data Type	Access
thread	opaque cma_t_thread	write

C Binding

void
cma_thread_get_self (
cma_t_thread *_thread_);

Arguments

thread
Variable that receives the handle of the current thread.

Description

This routine allows a thread to obtain its own handle. This value becomes meaningless when the thread object has been deleted—that is, when the thread has terminated its execution and `cma_thread_detach` has been called.

Exceptions

None

cma_thread_join

Causes the calling thread to wait for the termination of a specified thread.

Syntax

cma_thread_join (thread, exit_status, result)

Argument	Data Type	Access
thread	opaque cma_t_thread	read
exit_status	opaque cma_t_exit_status	write
result	pointer	write

C Binding

void
cma_thread_join (
cma_t_thread *_thread_,
cma_t_exit_status *_exit_status_,
cma_t_address *_result_);

Arguments

thread

Handle of the thread whose termination is awaited by the caller of this routine.

exit_status

Variable that receives a value indicating whether the thread specified by *thread* successfully terminated. Valid values are as follows:

Value	Description
`cma_c_term_normal`	Normal termination
`cma_c_term_error`	Error termination (result of calling `cma_thread_exit_error( )`)
`cma_c_term_alert`	Alert termination

result

Address value that is optionally returned by the *start_routine* of the thread specified by the *thread* argument in its call to `cma_thread_create`.

Description

This routine causes the calling thread to wait for the termination of a specified thread. A call to this routine returns after the specified thread has terminated.

The value returned as the *exit_status* argument indicates whether the thread terminated normally, because of an error, or because of an alert.

The value returned as the *result* argument is the address that the specified thread generates as its result. The thread's result is normally returned as the value of the *start_routine* argument in its call to `cma_thread_create`. In order for the *result* argument to be valid, the following must occur:

- The *exit_status* argument must have the value `cma_c_term_normal`.
- The *start-routine* function for the specified thread must return a value. (Returning a value is optional for the start routine.)

Any number of threads can call this routine. All threads are awakened when the specified thread terminates. If the thread is already terminated, this routine returns immediately.

If the current thread calls this routine, a deadlock results if it is detected by the implementation.

The results of this routine are unpredictable if the value for *thread* refers to `cma_c_null` or a thread that has been detached.

This routine is alertable.

Exceptions

cma_e_alerted
cma_e_existence
cma_e_use_error

cma_thread_set_priority

Changes the current priority of a thread.

Syntax

cma_thread_set_priority (thread, priority)

Argument	Data Type	Access
thread	opaque cma_t_thread	read
priority	opaque cma_t_priority	read

C Binding

void
cma_thread_set_priority (
cma_t_thread **thread*,
cma_t_priority *priority*);

Arguments

thread
Handle of the thread whose priority is changed.

priority
New value of the priority attribute. The priority attribute is dependent upon scheduling policy. Valid values are as follows:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

The default priority is `cma_c_prio_default_mid`. (This symbol maps to `cma_c_prio_through_mid`.)

Description

This routine changes the current priority of a thread. A thread can change its own priority.

Changing the priority of a thread can cause it to start executing or to be preempted by another thread. The exact effect of setting different priority values depends on the scheduling priority assigned to the thread. The scheduling priority is set by calling the `cma_attr_set_priority` routine.

An application should specify priority only to express the urgency of executing the thread relative to other threads. Priority should not be used to control mutual exclusion when accessing shared data. With a sufficient number of processors executing, all ready threads, regardless of priority, can be executing simultaneously.

This routine is different from `cma_attr_set_priority` in that `cma_attr_set_priority` sets the priority attribute that is used to establish the priority of a new thread when it is created. However, this routine changes the priority of an existing thread.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_unimp
cma_e_use_error
exc_e_nopriv

cma_thread_set_sched

Changes the current scheduling policy and priority of a thread.

Syntax

cma_thread_set_sched (thread, policy, priority)

Argument	Data Type	Access
thread	opaque cma_t_thread	read
policy	opaque cma_t_sched_policy	read
priority	opaque cma_t_priority	read

C Binding

void
cma_thread_set_sched (
cma_t_thread **thread*,
cma_t_sched_policy *policy*,
cma_t_priority *priority*);

Arguments

thread

Handle of the thread whose scheduling policy is changed.

policy

New scheduling policy value of the thread specified in *thread*. Valid values are as follows:

```
cma_c_sched_fifo
cma_c_sched_rr
cma_c_sched_throughput
cma_c_sched_background
```

See Section 2.3.3.2 for a description of the scheduling policies.

priority
New priority value of the thread specified in *thread*. The priority value is dependent upon scheduling policy. Valid values are as follows:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

The default priority is `cma_c_prio_default_mid`. (This symbol maps to `cma_c_prio_through_mid`.)

Description

This routine changes the current scheduling policy and priority of a thread. You can call this routine to change both the priority and scheduling policy of a thread at the same time. To change only the priority, call the `cma_thread_set_priority` routine.

A thread can change its own scheduling policy and priority. Changing the scheduling policy or priority, or both, of a thread can cause it to start executing or to be preempted by another thread.

This routine is different from `cma_attr_set_priority` and `cma_attr_set_sched` in that those routines set the priority and scheduling policy attributes that are used to establish the priority and scheduling policy of a new thread when it is created. This routine, however, changes the priority and scheduling policy of an existing thread.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_unimp
cma_e_use_error
exc_e_nopriv

cma_time_get_expiration

Obtains a *cma_t_date_time* value representing a desired expiration time.

Syntax

cma_time_get_expiration (expiration, interval)

Argument	Data Type	Access
expiration	opaque cma_t_date_time	write
interval	opaque cma_t_interval	read

C Binding

void
cma_time_get_expiration (
cma_t_date_time **expiration*,
cma_t_interval *interval*);

Arguments

expiration
Variable that receives the `cma_t_date_time` value representing the expiration time.

interval
Number of seconds to add to the current system time. The result is the time that the timed wait should expire.

Description

This routine adds a specified interval to the current absolute system time and returns a new absolute time. This new absolute time can be used as the expiration time in a call to `cma_cond_timed_wait`.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_use_error

cma_unlock_global

Unlocks a global mutex.

Syntax

cma_unlock_global

C Binding

void
cma_unlock_global

Arguments

None

Description

This service unlocks the global mutex when each call to `cma_lock_global` has been matched by a call to this routine. For example, if you called `cma_lock_global` three times, `cma_unlock_global` 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, one thread is unblocked and trys to acquire the global lock again. The scheduling policy is used to determine which thread acquires the global mutex.

The results of calling this routine are unpredictable if the global mutex is already unlocked. The results of calling this service are also unpredictable if the global mutex is owned by a thread other than the calling thread.

Exceptions

cma_e_existence
cma_e_use_error

cma_yield

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

Syntax

cma_yield

C Binding

```
void
cma_yield ( );
```

Arguments

None

Description

This routine notifies the scheduler that the current thread is willing to release its processor to other threads of the same priority. (A thread generally will release its processor to a thread of a higher priority without calling this routine.)

If the current thread's scheduling policy (as specified in a call to `cma_attr_set_sched` or `cma_thread_set_sched`) is `cma_c_sched_rr` or `cma_c_sched_fifo`, this routine yields the processor to other threads of the same priority. If no threads of the same priority are ready to execute, the thread continues.

This routine allows knowledge of the details of an application to be used to increase performance. It can increase performance of access to the processor by removing the current thread from the processor. It can also increase performance of access to shared resources by removing the current thread from the processor as soon as it is finished with the resource.

As a general guideline, consider calling this routine when a thread is executing code that could deny access to other threads on a uniprocessor if the scheduling policy is "run current thread until blocked" (`cma_c_sched_fifo`).

Use this routine carefully and sparingly, because misuse can cause unnecessary context switching, which in turn can increase overhead without increasing fairness. For example, it is counter-productive for a thread to yield while it has a needed resource locked.

Exceptions

None

Part IV

DECthreads Library Reference (OpenVMS and DEC OSF/1 Systems Only)

Part IV provides detailed descriptions of the DECthreads Library routines, which are part of the Digital Proprietary Interface to DECthreads.

The DECthreads Library consists of routines that allow you to create and control higher-level objects.

To indicate errors, the DECthreads Library routines raise exceptions. See Appendix C for exception descriptions.

Note

You can call the DECthreads Library routines from programs running on OpenVMS and DEC OSF/1 systems only.

cma_lib_attr_create

Creates a library attributes object.

Syntax

cma_lib_attr_create (new_attr, attr)

Argument	Data Type	Access
new_attr	opaque cma_lib_t_attr	write
attr	opaque cma_lib_t_attr	read

C Binding

void
cma_lib_attr_create (
cma_lib_t_attr *_new_attr_,
cma_lib_t_attr *_attr_);

Arguments

new_attr
Variable that receives a handle for the new attributes object.

attr
Handle of the attributes object used to control attributes of the new attributes object. If you specify `cma_c_null` for the *attr* argument, default attributes are used.

Description

This routine creates an attributes object that is used to specify the attributes of objects when they are created in routines with the `cma_lib_` prefix.

The queuesize attribute is the only currently defined DECthreads library routines attribute. Use the `cma_lib_attr_set_queuesize` routine to change the queuesize attribute.

Delete an attributes object by calling the `cma_lib_attr_delete` routine when it is no longer needed to create objects.

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_attr_delete

Deletes a library attributes object.

Syntax

cma_lib_attr_delete (attr)

Argument	Data Type	Access
attr	opaque cma_lib_t_attr	read, write

C Binding

void
cma_lib_attr_delete (
cma_lib_t_attr *attr);

Arguments

attr
Handle of the attributes object deleted.

Description

This routine deletes a library attributes object. The attributes object is marked for deletion, and the *attr* argument is set to the value `cma_c_null`. Specifying `cma_c_null` for the *attr* argument is legal and has no effect. Objects that were created using this attributes object are not affected by the deletion of the attributes object.

The results of calling this routine are unpredictable if the handle specified by the *attr* argument refers to an attributes object that does not exist (unless it is `cma_c_null`.)

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_attr_get_queuesize

Obtains the maximum number of elements available on an atomic queue attribute that is used when a queue is created.

Syntax

cma_lib_attr_get_queuesize (attr, queuesize)

Argument	Data Type	Access
attr	opaque cma_lib_t_attr	read
queuesize	opaque cma_t_natural	write

C Binding

```
void
cma_lib_attr_get_queuesize (
cma_lib_t_attr *attr,
cma_t_natural *queuesize);
```

Arguments

attr
Handle of the library attributes object whose queuesize is obtained. This value is returned by `cma_lib_attr_create`.

queuesize
Variable that receives the current value of the queuesize attribute.

Description

This routine obtains the queuesize attribute that is used when a queue is created. The queuesize attribute specifies the maximum number of elements allowed on a queue. The default queuesize attribute is 128.

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_attr_set_queuesize

Specifies the attribute for the maximum number of elements allowed on an atomic queue that is used when a queue is created.

Syntax

cma_lib_attr_set_queuesize (attr, queuesize)

Argument	Data Type	Access
attr	opaque cma_lib_t_attr	read
queuesize	opaque cma_t_natural	read

C Binding

void
cma_lib_attr_set_queuesize (
cma_lib_t_attr **attr*,
cma_t_natural *queuesize*);

Arguments

attr
Handle of the library attributes object to be modified. This value is returned by `cma_lib_attr_create`.

queuesize
New value for the queue size attribute. The default value is 128.

Description

This routine sets the queuesize attribute that is used when a queue is created. The queuesize attribute specifies the maximum number of elements allowed on a queue.

A queue contains a fixed number of available queue items. Call this routine if you want to increase or decrease the queue size of new queues when they are created.

Exceptions

cma_e_badparam
cma_e_existence
cma_e_use_error

cma_lib_queue_create

Creates an atomic queue.

Syntax

cma_lib_queue_create (new_queue, attr)

Argument	Data Type	Access
new_queue	opaque cma_lib_t_queue	write
attr	opaque cma_lib_t_attr	read

C Binding

void
cma_lib_queue_create (
cma_lib_t_queue *_new_queue_,
cma_lib_t_attr *_attr_);

Arguments

new_queue
Variable that receives the handle of the atomic queue created.

attr
Library attributes object used when creating the queue.

Description

This routine creates an atomic queue. Unlike mutexes and condition variables, an atomic queue implements interthread communication that is not explictly dependent on synchronization. A queue can communicate information among threads or within a single thread (for example, between an interrupt routine and the normal thread code).

The primary characteristic of an atomic queue is that any operation on the queue is guaranteed to complete before any other operation on that same queue can begin.

The created queue is not affected by termination of the thread that created it. It remains valid until explicitly deleted by `cma_lib_queue_delete`.

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_queue_delete

Deletes an atomic queue.

Syntax

cma_lib_queue_delete (queue)

Argument	Data Type	Access
queue	opaque cma_lib_t_queue	read, write

C Binding

void
cma_lib_queue_delete (
cma_lib_t_queue *_queue_);

Arguments

queue
Handle of the queue to be deleted.

Description

This routine deletes the specified atomic queue. After deletion, the handle is set to `cma_c_null`. A queue remains valid until explicitly deleted by `cma_lib_queue_delete`.

A queue must be empty for it to be deleted. If the queue is not empty when you call this routine, the exception `cma_e_in_use` is raised.

Exceptions

cma_e_existence
cma_e_in_use
cma_e_use_error

cma_lib_queue_dequeue

Removes the first element from an atomic queue.

Syntax

cma_lib_queue_dequeue (queue, element)

Argument	Data Type	Access
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	write

C Binding

void
cma_lib_queue_dequeue (
cma_lib_t_queue **queue*,
cma_lib_t_address **element*);

Arguments

queue
Handle of the queue from which the element is removed.

element
Variable that receives the address of the removed queue element.

Description

This routine removes the first element from an atomic queue. If the queue is empty, the calling thread is blocked until an element is inserted into the queue. When the element is enqueued the calling thread resumes, the new element is removed, and this routine returns.

Call `cma_lib_queue_try_dequeue` to remove an element from a queue and return a status code (instead of blocking) if the queue is empty.

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_queue_enqueue

Inserts an element at the end of an atomic queue.

Syntax

cma_lib_queue_enqueue (queue, element)

Argument	Data Type	Access
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

C Binding

void
cma_lib_queue_enqueue (
cma_lib_t_queue **queue*,
cma_lib_t_address *element*);

Arguments

queue
Handle of the queue to which the element is inserted.

element
Address of the queue element inserted.

Description

This routine inserts an element at the end of a queue. If the queue is full, the calling thread is blocked until an element is removed from the queue. When the element is dequeued the calling thread resumes, the new element is inserted, and this routine returns.

Call `cma_lib_queue_try_enqueue` to insert an element into a queue and return a status code (instead of blocking) if the queue is full.

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_queue_requeue

Inserts an element at the front of an atomic queue.

Syntax

cma_lib_queue_requeue (queue, element)

Argument	Data Type	Access
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

C Binding

void
cma_lib_queue_requeue (
cma_lib_t_queue **queue*,
cma_lib_t_address *element*);

Arguments

queue
Handle of the queue to which the element is inserted.

element
Address of the queue element inserted.

Description

This routine inserts an element at the front of a queue. If the queue is full, the calling thread is blocked until an element is removed from the queue. When the element is dequeued the calling thread resumes, the new element is inserted, and this routine returns.

Call `cma_lib_queue_try_requeue` to insert an element at the front of a queue and return a status code (instead of blocking) if the queue is full.

This routine allows you to replace an element that was erroneously removed from a queue. For example, a queue might hold information of various types. In that case a thread can remove the oldest (first) element of the queue, check its type, and requeue the element if it is not the desired type (rather than enqueuing it, which would place the element at the end of the queue).

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_queue_try_dequeue

Removes the first element from an atomic queue.

Syntax

status = cma_lib_queue_try_dequeue (queue, element)

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	write

C Binding

cma_t_boolean
cma_lib_queue_dequeue (
cma_lib_t_queue **queue*,
cma_lib_t_address **element*);

Arguments

status
Boolean value that specifies whether the element was dequeued.

queue
Handle of the queue from which the element is removed.

element
Variable that receives the address of the removed queue element.

Description

This routine removes the first element from an atomic queue. If the queue is empty, the calling thread does not wait for an element to be enqueued. Instead, the routine returns with the status `cma_c_false`. If the queue is not empty, the first element value is returned to *element*, and the routine returns with the status `cma_c_true`.

Call `cma_lib_queue_dequeue` if you want to remove an element from a queue but cause the calling thread to block if the queue is empty.

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_queue_try_enqueue

Inserts an element at the end of an atomic queue.

Syntax

status = cma_lib_queue_try_enqueue (queue, element)

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

C Binding

cma_t_boolean
cma_lib_queue_try_enqueue (
cma_lib_t_queue **queue*,
cma_lib_t_address *element*);

Arguments

status
Boolean value that specifies whether the element was enqueued.

queue
Handle of the queue to which the element is inserted.

element
Address of the queue element inserted.

Description

This routine inserts an element at the end of a queue. If the queue is full, the calling thread does not wait for an element to be removed. Instead, the routine returns with the status `cma_c_false`. If the queue is not full, the element is inserted at the end of the queue and the routine returns with the status `cma_c_true`.

Call cma_lib_queue_enqueue if you want to insert an element into a queue but cause the calling thread to block if the queue is full. Call cma_lib_queue_try_enqueue_int if you want to insert an element into a queue from interrupt level.

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_queue_try_requeue

Inserts an element at the front of an atomic queue.

Syntax

status = cma_lib_queue_try_requeue (queue, element)

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

C Binding

cma_t_boolean
cma_lib_queue_requeue (
cma_lib_t_queue **queue*,
cma_lib_t_address *element*);

Arguments

status
Boolean value that specifies whether the element was requeued.

queue
Handle of the queue to which the element is inserted.

element
Address of the queue element inserted.

Description

This routine inserts an element at the front of a queue. If the queue is full, the calling thread does not wait until an element is removed from the queue. Instead, the routine returns with the status `cma_c_false`. If the queue is not full, the element is added to the front of the queue and the routine returns with the status `cma_c_true`.

Call `cma_lib_queue_requeue` if you want to insert an element into a queue but cause the calling thread to block if the queue is full.

This routine allows you to replace an element that was erroneously removed from a queue. For example, a queue might hold information of various types. In that case a thread can remove the oldest (first) element of the queue, check its type, and requeue the element if it is not the desired type (rather than enqueuing it, which would place the element at the end of the queue).

Exceptions

cma_e_existence
cma_e_use_error

cma_lib_queue_try_enqueue_int

Inserts an element at the end of an atomic queue from interrupt level.

Syntax

status = cma_lib_queue_try_enqueue_int (queue, element)

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

C Binding

cma_t_boolean
cma_lib_queue_try_enqueue_int (
cma_lib_t_queue **queue*,
cma_lib_t_address *element*);

Arguments

status
Boolean value that specifies whether the element was enqueued.

queue
Handle of the queue to which the element is inserted.

element
Address of the queue element inserted.

Description

This routine inserts an element at the end of a queue from interrupt level. If the queue is full, the calling thread does not wait for an element to be removed. Instead, the routine returns with the status `cma_c_false`. If the queue is not full, the element is inserted at the end of the queue and the routine returns with the status `cma_c_true`.

Note

This routine allows you to add elements to a queue from a software interrupt handler. Do not call this routine from non-interrupt code. If you want to add elements to a queue from the normal noninterrupt level without blocking, use `cma_lib_queue_try_enqueue`.

Exceptions

cma_e_existence
cma_e_use_error

A

Considerations for Systems Based on UNIX Software

This appendix discusses DECthreads issues specific to systems based on the UNIX operating system.

A.1 DEC OSF/1 Systems

The DEC OSF/1 operating system supports multiple concurrent streams of execution within a process using the Mach kernel. DECthreads utilizes these kernel execution contexts to implement user threads.

Because all services provided directly by the kernel are automatically thread-synchronous and reentrant, programming is easier than in other DECthreads implementations based on user-mode thread context switching. For example, unlike on ULTRIX systems, files remain in normal blocking mode unless you set them to nonblocking mode.

Many language run-time routines are provided in thread-reentrant versions. For example, reentrant versions of C run-time routines are provided in the `libc_r` package. In general, any entry point provided in `libc_r` is reentrant. However, you still need to provide your own mutual exclusion using the DECthreads global lock for many library routines that are not provided by `libc_r`. (See Section 3.2 for information on how to accomplish this.)

DECthreads provides jacket routines for the kernel routines `fork(2)` and `sigaction(2)`. DECthreads also provides compile time jackets implementing the `atfork(2)` routine and the `sigwait(2)` routine specified by the POSIX 1003.4a draft standard. These jackets are described in Section A.3 in more detail. DECthreads also provides (at load time) thread-reentrant versions of the `malloc(3)`, `free(3)`, `realloc(3)`, `calloc(3)`, and `cfree(3)` routines.

A.1.1 Compiling Multithreaded Applications: Static or Shared Libraries

Multithreaded applications can be compiled using either static or shared libraries. For a discussion about static and shared libraries, see the *DEC OSF/1 Programmer's Guide*.

Table A–1 contains the libraries supported for multithreaded programming.

Table A–1 DEC OSF/1 Static and Shared Libraries for Multithreaded Programs

libmach.a	Static version of the base system threads interface. Not recommended to be called directly
libmach.so	Shared version of the base system threads interface. Not recommended to be called directly
libpthreads.a	Static version of the base pthreads package. Requires libmach.a and libc_r.a
libpthreads.so	Shared version of the base pthreads package. Requires libmach.so and libc_r.so
libc_r.a	Static version of reentrant libc routines. Contains reentrant versions of libc routines
libc_r.so	Shared version of reentrant libc routines. Contains reentrant versions of libc routines

Compile a multithreaded application using static versions of `libmach`, `libpthreads`, and `libc_r` as follows:

```
% cc -non_shared -o myprog myprog.c -threads
```

Compile a multithreaded application using shared versions of `libmach`, `libpthreads`, and `libc_r` as follows:

```
% cc -o myprog myprog.c -threads
```

A.1.2 Support for the DEC OSF/1 Realtime Options

DECthreads supports the DEC OSF/1 real-time kernel. This allows you to set the scheduling policy and priority of threads. In order to do this, the DEC OSF/1 Realtime Options must be installed on the system and a real-time kernel must be configured. See the *DEC OSF/1 Installation Guide* for more information on installing the Realtime Options.

Programs that use threads with real-time policy or that use priorities higher than the default process priority must be executed with root privileges. For more information, see the *DEC OSF/1 Realtime Programming Guide*.

If you are not running the real-time kernel or your program lacks necessary privileges, attempting to call the following routines returns –1 and results in *errno* being set to the EPERM or ENOSYS error value:

pthread_attr_setprio	(Error returned by pthread_create at thread creation)
pthread_attr_setsched	(Error returned by pthread_create at thread creation)
pthread_setprio	
pthread_setscheduler	

If you are not running the real-time kernel or your program lacks necessary privileges, attempting to call the following routines raises the `cma_e_unimp` or `exc_e_nopriv` exceptions:

cma_attr_set_priority	(Exception raised by cma_thread_create at thread creation)
cma_attr_set_sched	(Exception raised by cma_thread_create at thread creation)
cma_thread_set_priority	
cma_thread_set_sched	

A.2 ULTRIX Systems

On the ULTRIX operating system, DECthreads emulates thread-synchronous versions of the kernel I/O routines, using compile-time jacket routines. It accomplishes this by setting the files into nonblocking mode and transparently blocking threads on condition variables until the I/O operation can be completed. This has the following important implications for threaded programs:

- There may be small delays (latency) between the time when the file becomes ready and the time DECthreads actually retries a blocked I/O. This is because DECthreads cannot continually monitor the state of all files.
- All modules performing I/O should include the `cma.h` or `pthread.h` header file, even if they perform no explicit DECthreads operations; this is because the header enables the compile-time jacket routines. I/O operations from modules not compiled with a DECthreads header file may not work correctly.

DECthreads also provides jacket routines for `fork(2)` and `sigaction(2)` and provides the routines `sigwait(2)` and `atfork(2)`.

On ULTRIX systems, the DECthreads library is called `libcma.a`. Because DECthreads uses the internationalized message catalog routines, it requires `libi.a`. To build a program using DECthreads, you need to specify both of these libraries to `ld` or `cc`, as follows:

```
% cc -o myprog myprog.c -lcma -li
```

A.3 Jacketed UNIX Routines

The following sections discuss the jacketed UNIX routines provided by DECthreads.

A.3.1 Thread-Synchronous I/O

On ULTRIX systems, DECthreads supplies compile-time jacket routines that emulate thread-synchronous I/O (only the thread performing the I/O operation is blocked). These routines have the same calling interface as the standard UNIX routines. Several important differences are detailed in Section A.2, but, in general, the use of these jackets is transparent to the calling program.

A.3.2 Forking a New Process from a Thread

Because the child process created by `fork(2)` inherits the address space of the parent but not all of the threads, DECthreads must perform some initialization within the child. This is accomplished through the `fork` jacket routine. Any module within a threaded program that calls `fork(2)` must include the `cma.h` or `pthread.h` header file.

A.3.2.1 The fork() Jacket Routine

A jacket is provided for the `fork( )` system call. A specific thread environment must exist in the forked process when it resumes (begins) execution. This jacket routine allows code to be executed in the context of the new process before the user code resumes execution in it.

The jacket routine initializes the internal DECthreads state in the child process. In the child process, all mutexes are unlocked, there are no threads waiting on condition variables, and all user threads except the thread that called `fork( )` have been destroyed.

A.3.2.2 The atfork() Jacket Routine

The `atfork( )` jacket routine helps to address unpredictability with the semantics of the `fork( )` routine in a multithreaded environment. Using `fork( )` from a threaded application or from an application that uses threaded libraries, can be unpredictable. For example, one thread has a mutex locked and the state covered by that mutex is inconsistent while another thread calls `fork( )`. In the child, the mutex will be in the locked state locked by a (nonexistent) thread and thus can never be unlocked. Having the child simply reinitialize the mutex is unsatisfactory since this approach does not resolve the question of how to correct the inconsistent state in the child.

The `atfork( )` jacket routine provides a means for threaded applications or libraries to protect themselves when a `fork( )` occurs. The `atfork( )` routine allows you to set up routines that will run at the following times:

- Prior to the `fork( )` in the parent process
- After the `fork( )` in the child process
- After the `fork( )` in the parent process

Within these routines you can ensure that all locks are locked prior to `fork( )` and that they are unlocked after `fork( )`—thereby protecting any data or resources associated with the mutexes. You can register any number of sets of `atfork( )` routines; that is, any number of libraries or user programs can set up `atfork( )` routines and they will all execute at `fork( )` time.

Note

Using `atfork( )` can potentially cause a deadlock if two applications, subsystems, or libraries call into one another using calls that require locking. Specifically, when these component's routines use `atfork( )` to run prior to the `fork( )` in the parent process, a deadlock may occur when these routines are executing.

A.3.2.3 Compiling Code with Jacketed System Calls

You do not have to rename your system calls to take advantage of the jacket routines. The jacket routines are put into place when you compile your program by macros that rename instances of the jacketed system calls to the name of the DECthreads jacket routine. Therefore, a reference to a DECthreads jacket routine is compiled into your code instead of a reference to the system call. When the code is executed, it calls the jacket routine, which then calls the system on your code's behalf.

If you do not want to use any of the jacket routines, you can add the following line to your program before any of the thread header files:

```
#define _CMA_NOWRAPPERS_
```

By adding this definition, you prevent the jacket routines from being substituted for the real routines.

If you want to use most of the jacket routines but do not want to use a specific jacket routine, you can undefine a specific jacket routine by adding the following definition after the thread header files:

```
#undef routine_name
```

For example, to prevent the `fork( )` jacket routine from being substituted for the real `fork( )` routine, add the following definition after the thread header files:

```
#undef fork
```

Digital does not recommend undefining jacket routines, but in some cases it may be necessary. Note that most jacket routines are not implemented on the DEC OSF/1 operating system and might not be implemented on future versions of the ULTRIX operating system.

A.4 Using Signals

This section discusses the types of signals, DECthreads signal handling, and alternatives to using signals.

A.4.1 Types of Signals

Signals are delivered as a result of some event. UNIX signals are grouped into the following four categories of pairs that are orthogonal to each other:

- Terminating and synchronous
- Nonterminating and asynchronous
- Nonterminating and synchronous
- Terminating and asynchronous

The action that DECthreads takes when a particular signal is delivered depends on the characteristics of that signal.

A.4.1.1 Terminating Signals

Terminating signals result in the termination of the process by default. Whether a particular signal is terminating or not is independent of whether it is synchronously or asynchronously delivered.

A.4.1.2 Nonterminating Signals

Nonterminating signals do not result in the termination of the process by default.

Nonterminating signals represent events that can be either internal or external to the process. The process might desire notification about or ignore these events. When a nonterminating asynchronous signal is delivered to the process, DECthreads awakens any threads that are waiting for the signal. This is the only action that DECthreads takes, because, by default, the signal has no effect.

A.4.1.3 Synchronous Signals

Synchronous signals are the result of an event that occurs inside a process and are delivered synchronously with respect to that event. For example, if a floating point calculation results in an overflow, then a `SIGFPE` (floating point exception signal) is delivered to the process immediately following the instruction that resulted in the overflow.

Synchronous, terminating signals represent an error that has occurred in the currently executing thread.

A.4.1.4 Asynchronous Signals

Asynchronous signals are the result of an event that is external to the process and are delivered at any point in a thread's execution when such an event occurs. For example, when a user running a program types the interrupt character at the terminal (generally Ctrl/C), a `SIGINT` (interrupt signal) is delivered to the process.

Asynchronous, terminating signals represent an occurrence of an event that is external to the process, and, if unhandled, results in the termination of the process. When an asynchronous terminating signal is delivered, DECthreads catches it and checks to see if any threads are waiting for it. If threads are waiting, they are awakened, and the signal is considered handled and is dismissed. If there are no waiting threads, then DECthreads terminates the process in a manner similar to the unhandled behavior of the signal.

A.4.2 DECthreads Signal Handling

DECthreads provides the POSIX 1003.4a `sigwait(2)` and `sigaction(2)` services to allow threads to perform activities similar to signal handling without having to deal with signals directly.

DECthreads does not support handlers for the following UNIX signals:

Signal	Reason Handler Is Not Provided
SIGKILL and SIGSTOP	These signals cannot be caught by user mode code.
SIGTRAP	Catching this signal interferes with debugging.

A.4.2.1 The POSIX sigwait Service

DECthreads implementation of the POSIX 1003.4a `sigwait` service allows any thread to block until one of a specified set of signals is delivered. A thread waits for any of the asynchronous signals except for `SIGKILL` and `SIGSTOP`.

A thread cannot 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 is it waiting, and so the thread will wait forever.

POSIX stipulates that the thread must block the signals it will wait for before calling `sigwait`. When any thread issues a `sigwait` request for a signal, DECthreads may declare its own handler for that signal. This handler remains current until no threads are waiting for the signal.

A.4.2.2 The POSIX sigaction Service

The DECthreads implementation of the POSIX sigaction service allows for per-thread handlers to be installed for catching synchronous signals. The `sigaction(2)` service only modifies behavior for individual threads and will only work for synchronous signals. Setting the signal action to `SIG_DFL` for a specific signal will restore the thread's default behavior for that signal. Attempting to set a signal action for an asynchronous signal is an error.

A.4.3 Signal Alternatives Using the sigwait Routine

Avoid dealing with UNIX signals directly by using signal handler routines in multithreaded programs. DECthreads provides alternatives to signal handling.

One alternative to using asynchronous signals directly is to use the `sigwait( )` routine defined by the POSIX 1003.4a draft standard. The `sigwait( )` routine takes a signal mask (POSIX.1 `sigset_t` type) as an argument and returns the number of a signal (`int`). The `sigwait( )` routine causes the calling thread to block (without affecting other threads) until one of the signals in the `sigset_t` is received. The routine will then return with the number of that signal.

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 alert (cancel) other threads to cause the program to shut down the current activities.

The prototype for `sigwait( )` is as follows:

```
int sigwait (sigset_t *set);
```

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, signals cannot be used in a modular way because, on most current implementations of UNIX, signals are inherently a process construct. There is only one instantiation of each signal and of each signal handler routine for all of the threads in an application. If one thread handles a particular signal in one way, and a different thread handles the same signal in a different way, then the thread that installs its signal handler last handles the signal.

Do not use asynchronous programming techniques in conjunction with threads, particularly those that increase parallelism such as using timer signals and I/O signals. These techniques are complicated and error-prone; they are also unnecessary because threads provide a mechanism for parallel execution that is simpler and less error-prone. Furthermore, most of the threads services are not supported for use in interrupt routines (such as signal handlers), and portions of run-time libraries cannot be used reliably inside a signal handler.

A.5 Software Interrupts and Exceptions

From a portable point of view, it is difficult to determine in which thread a software interrupt handler runs. It is also difficult to determine what happens if an exception propagates out of a software interrupt handler.

A software interrupt handler must not allow an exception to propagate out of it. If this happens, the exception might be caught by another thread's exception handler and results in unpredictable behavior.

Therefore, it is best to avoid complicated coding in a software interrupt routine. If you write a software interrupt handler, you should release a waiting thread or enqueue an action item.

This has the advantage of minimizing the code in the software interrupt, which benefits the application by reducing the latency and increasing the throughput for such interrupts. For more information on exceptions, see Chapter 4.

A.6 Signals Reported as Exceptions

The following sections list UNIX signals that are reported as DECthreads exceptions.

A.6.1 Synchronous Terminating Signals

Table A–2 shows the UNIX signals that are considered synchronous terminating on common UNIX platforms and the DECthreads exceptions that are associated with those signals.

Table A–2 Synchronous Terminating Signals

Signal	Exception
SIGILL	exc_illinstr_e
SIGIOT	exc_SIGIOT_e
SIGEMT	exc_SIGEMT_e
SIGFPE	exc_aritherr_e
SIGBUS	exc_illaddr_e
SIGSEGV	exc_illaddr_e
SIGSYS	exc_SIGSYS_e
SIGPIPE	exc_SIGPIPE_e
SIGXCPU	exc_SIGXCPU_e
SIGXFSZ	exc_SIGXFSZ_e

A.6.2 Arithmetic Error Signals

The signal code for arithmetic errors (`SIGFPE`) is decoded by the exception package handlers so that each specific type of arithmetic error can be reported as a distinct exception. The `exc_aritherr_e` exception is raised only if the code is not one of these known values. The DECthreads Exception Package does not distinguish between faults and traps; such a distinction would be of no value to an application, since exceptions are terminating.

Table A–3 shows the arithmetic error signals and the DECthreads exceptions that are associated with those signals.

Table A–3 Arithmetic Error Signals

Signal	Exception
FPE_INTOVF_TRAP	exc_intovf_e
FPE_INTDIV_TRAP	exc_intdiv_e
FPE_FLTOVF_TRAP	exc_fltovf_e
FPE_FLTDIV_TRAP	exc_fltdiv_e
FPE_FLTUND_TRAP	exc_fltund_e
FPE_DECOVF_TRAP	exc_decovf_e
FPE_SUBRNG_TRAP	exc_subrng_e
FPE_FLTOVF_FAULT	exc_fltovf_e
FPE_FLTDIV_FAULT	exc_fltdiv_e
FPE_FLTUND_FAULT	exc_fltund_e

A.6.3 Illegal Instruction Signals

The signal code for illegal instruction errors (`SIGILL`) is decoded so that each specific type of instruction error can be reported as a distinct exception. The `exc_illinstr_e` exception is raised only when none of these known codes apply.

Table A–4 shows the illegal instruction signals and the DECthreads exceptions that are associated with those signals.

Table A–4 Illegal Instruction Signals

Signal	Exception
ILL_RESAD_FAULT	exc_resaddr_e
ILL_PRIVIN_FAULT	exc_privinst_e
ILL_RESOP_FAULT	exc_resoper_e

A.7 Debugging Threads

At the dbx prompt, enter the following command:

```
print cma_debug()
```

The following prompt appears:

```
DECthreads debug>
```

Note

Some versions of dbx also allow you to enter `call cma_debug( )`.

Enter any of the DECthreads debugging commands and qualifiers listed in Table A–5.

Table A–5 DECthreads Debugging Commands and Qualifiers

Command	Qualifier	Function
threads		List known threads
	-c	Currently running thread
	-r	Ready queue (in order)
	-a	All threads (DECthreads internal threads are otherwise omitted)
	-b	Blocked threads
	-t	Terminated threads
	-i	An alternate view of threads with more detailed information
conditions		List known condition variables
	-w	List conditions with waiters
	-q	List waiting thread sequence numbers
mutexes		List known mutexes
	-l	List locked mutexes
	-q	List waiting thread sequence numbers

(continued on next page)

Table A–5 (Cont.) DECthreads Debugging Commands and Qualifiers

Command	Qualifier	Function
attributes		List known attributes objects
help		List all thread debugging commands and qualifiers
exit		Exit cma_debug

You can abbreviate the `cma_debug` commands. All commands accept a numeric argument (for example, `t 1`) as the sequence number of an object to be listed. Specifying a sequence number causes DECthreads to ignore any selection switches on the command. For example:

```
m -l 3
```

Mutex 3 is listed even if it is not locked.

Type Exit or enter Ctrl/D to exit.

B

Considerations for OpenVMS Systems

This appendix discusses DECthreads issues and restrictions specific to the OpenVMS operating system.

B.1 Using DECthreads with Asynchronous System Trap (AST) Routines

An **AST** is an OpenVMS mechanism for reporting an asynchronous event to a process. Following are three restrictions concerning the use of ASTs with DECthreads:

- Avoid blocking ASTs for an extended period of time. DECthreads depends on ASTs internally for several functions; for example, blocking them for an extended period of time can disrupt thread scheduling.
- Do not call any DECthreads routines with ASTs blocked; this may result in unexpected behavior.
- Do not call DECthreads routines, except those that have the `_int` (interrupt) suffix in their names, from within an AST routine. Calling any other DECthreads routines from code running in an AST can be unreliable or cause unexpected behavior.

B.2 Declaring an OpenVMS Condition Handler

This section discusses a restriction on declaring an OpenVMS condition handler while using DECthreads exceptions and DECthreads behavior when a condition occurs.

Following are three ways to declare an OpenVMS condition handler:

- Calling `VAXC$ESTABLISH` (from a program written in C)
- Calling `LIB$ESTABLISH` (from a program written in MACRO or BLISS)
- Placing the address of the condition handler directly into the stack frame (from a program written in VAX MACRO or VAX BLISS)

Do not declare an OpenVMS condition handler within a DECthreads TRY /ENDTRY exception block. Doing so deletes without notification any handler that exists for the current procedure. If your code declares a condition handler within the TRY/ENDTRY block, DECthreads exceptions fail until the next TRY statement is executed. The TRY statement restores the DECthreads condition handler.

You can declare an OpenVMS VAX condition handler outside of a TRY/ENDTRY block with no restrictions.

If an OpenVMS VAX condition handler is declared when you execute a TRY statement, DECthreads saves the handler address and reestablishes it when the TRY block exits. When DECthreads receives a condition it does not handle (including SS$_UNWIND, SS$_DEBUG or a condition code that does not have a *SEVERE* severity), DECthreads invokes the saved OpenVMS VAX condition handler.

B.3 Linking OpenVMS Images

The DECthreads OpenVMS calling standard interface (routine names that begin with cma$, such as cma$mutex_lock or cma$lib_queue_enqueue) are resolved by the linker automatically and do not require explicit shareable image references in an OpenVMS Linker options file. However, if your image references the DECthreads open interface (routine names that begin with cma_ or pthread_, such as cma_mutex_lock, cma_lib_queue_enqueue or pthread_mutex_lock) you must link against the appropriate images, using an OpenVMS Linker options file. Function names beginning with cma_lib_ (DECthreads Library Services routines, including the atomic queue operations) are defined in the image SYS$SHARE:CMA$OPEN_LIB_SHR.EXE. Other functions beginning with cma_, as well as the P1003.4a and P1003.4a exception interfaces, are defined in the image SYS$SHARE:CMA$OPEN_RTL.EXE.

DECthreads is supplied only as shareable images. It is not supplied as object libraries.

B.4 Including DECthreads Header Files

Include one of the following DECthreads header files in your program to call the appropriate DECthreads library. These header files and their associated routine libraries are listed in Table B–1.

Table B–1 DECthreads Header Files

Header File	Routine Library
cma.h	cma routines (cma_)
cma$defs.h	cma routines (cma$)
pthread.h	pthread routines with status-returning interface
pthread_exc.h	pthread routines with exception-returning interface

B.5 Debugging Threads

Following are two methods you can use to debug your multithreaded program, and they can be used separately or in combination:

- Calling the cma_debug routine from the OpenVMS Debugger
- Using OpenVMS Debugger tasking support

B.5.1 Calling the cma_debug Routine

If you are programming using the open interface to DECthreads (cma_ routine names), at the OpenVMS Debugger prompt, enter the following command:

```
DBG> call @cma_g_debug
```

If you are programming using the OpenVMS calling standard interface to DECthreads ((cma$ routine names), at the OpenVMS Debugger prompt, enter the following command:

```
DBG> call @cma$g_debug
```

The DECthreads debugging commands are listed in Table B–2.

Table B–2 DECthreads Debugging Commands and Qualifiers

Command	Qualifier	Function
Threads		List known threads
	-c	Currently running thread
	-r	Ready queue (in order)
	-a	All threads (null thread is otherwise omitted)

(continued on next page)

Table B–2 (Cont.) DECthreads Debugging Commands and Qualifiers

Command	Qualifier	Function
	-b	Blocked threads
	-t	Terminated threads
Conditions		List known condition variables
	-w	List conditions with waiters
	-q	List waiting thread sequence numbers
Mutexes		List known mutexes
	-l	List locked mutexes
	-q	List waiting thread sequence numbers
Attributes		List known attributes objects
Help		List all thread debugging commands and qualifiers
Exit		Exit cma_debug

You can abbreviate the `cma_debug` commands. All commands accept a numeric argument (for example, `t 1`) as the sequence number of an object to be listed. Specifying a sequence number causes DECthreads to ignore any selection switches on the command. For example:

```
m -l 3
```

Mutex 3 is listed even if it is not locked.

Type Exit or enter Ctrl/Z to exit.

B.5.2 Using OpenVMS Debugger Tasking Support

The OpenVMS Debugger includes support for multithreaded programs. This support enables you to perform functions such as the following:

- Display thread information
- Modify thread characteristics to control thread execution, priority, state transitions, and so on
- Monitor thread-specific events and state transitions

The OpenVMS Debugger contains no new commands or qualifiers for multithreaded (tasking) support. However, the following commands, which are task-related, have been enhanced to provide the new support:

- SET TASK, SHOW TASK
- SET EVENT_FACILITY (specify THREADS)
- SHOW EVENT_FACILITY
- SET BREAK/EVENT, SET TRACE/EVENT (THREADS events are defined)

See the *OpenVMS Debugger Manual* for complete information about these commands and qualifiers.

C

DECthreads Exceptions

Table C–1 lists the DECthreads exceptions and gives an explanation of each exception. It also contains the recommended action you should take if an exception occurs.

Table C–1 DECthreads Exceptions

Exception	Explanation and User Action
cma_e_alerted	Thread execution was alerted Explanation: A thread was requested to terminate by either the `cma_thread_alert` or `pthread_cancel` routine. DECthreads uses an alert to request that a thread terminate after first performing cleanup and shutdown operations. User Action: If you do not want threads to terminate at the point where this alert is being delivered, you can use several routines (`cma_alert_disable_general`, `cma_alert_disable_asynch`, `cma_alert_restore`, `pthread_setcancel`, and `pthread_setasynccancel`) to specify points in the thread process where alerts cannot be delivered to the thread.
cma_e_alert_nesting	Improper nesting of alert scope Explanation: An attempt was made to restore an inner scope after an enclosing outer scope had already been restored.

(continued on next page)

Table C–1 (Cont.) DECthreads Exceptions

Exception	Explanation and User Action
	User Action: Examine the code to determine where the incorrect alert state variable was passed to the `cma_alert_restore` routine.
cma_e_badparam	Parameter to DECthreads operation is invalid
	Explanation: A parameter passed to a DECthreads routine is improper; for example, the value is of the wrong type or is out of range.
	User Action: Determine which routine raised the exception. Then consult the documentation to determine the correct parameters and value ranges. Update your code accordingly and retry the operation.
	If you continue to have problems, report it to Digital, including a small test program that reproduces the problem.
cma_e_exccop	Exception raised; OpenVMS condition code follows
	Explanation: The EXC_HANDLING.H package, which provides portable exceptions for the C language, raised an exception. The accompanying OpenVMS condition code identifies the error.
	User Action: See the documentation for the software that your program is calling to determine the reason for this exception. Correct the problem or use the EXC_ HANDLING.H package to provide an exception handler.
cma_e_exccoplos	Exception raised; some information lost

(continued on next page)

Table C–1 (Cont.) DECthreads Exceptions

Exception	Explanation and User Action
	Explanation: An exception was raised using an old mechanism that does not preserve all exception information. The exception message displays, but argument values are not substituted in the message text. User Action: None required. However, you can recompile the code that raised the exception to allow it to use the new mechanism that preserves the exception information.
cma_e_exception	Exception raised; address of exception object is object-address Explanation: The EXC_HANDLING.H package, which provides portable exceptions for the C language, was used to raise an exception. User Action: See the documentation for the software that your program is calling to determine the reason for this exception. Correct the problem or use the EXC_HANDLING.H package to provide an exception handler.
cma_e_existence	Object referenced does not currently exist Explanation: A DECthreads routine was requested to operate on an object that does not exist. User Action: Consult the documentation for the DECthreads routine that issued this message to determine the conditions that caused it. Also check the program where the call is issued to determine which object or objects that are being passed as parameters do not currently exist.
cma_e_exit_thread	Current thread was requested to exit

(continued on next page)

Table C–1 (Cont.) DECthreads Exceptions

Exception	Explanation and User Action
	Explanation: The `cma_thread_exit` routine was called to force the thread to shut down in an orderly fashion. This message notifies all active exception handlers to perform any necessary cleanup activities. User Action: None
cma_e_inialrpro	DECthreads initialization is already in progress Explanation: A call was made to the DECthreads initialization routine `cma_init` while DECthreads was still trying to initialize itself on a prior call. DECthreads initialization must be complete before any DECthreads routines are used. Once DECthreads is fully initialized, all calls to `cma_init` complete successfully. User Action: Remove the offending concurrent call to the `cma_init` routine or delay it until the first call to `cma_init` has completed.
cma_e_in_use	Object referenced is already in use Explanation: The DECthreads operation cannot be performed on the specified object because it is already in use; for example, the routine is attempting to delete a mutex that is locked. User Action: Determine which routine caused the error and make sure the object is in an appropriate state before attempting the operation.
cma_e_nostackmem	No space is currently available to create a new stack

(continued on next page)

Table C–1 (Cont.) DECthreads Exceptions

Exception	Explanation and User Action
	Explanation: A call to `cma_thread_create` or another DECthreads routine requires that a new stack be created, but there is insufficient space to create it. User Action: Reduce the size of thread stacks previously created, so that additional stacks may be created. Alternatively, adjust system or user quotas to allow the allocation of more virtual memory.
cma_e_stackovf	Attempted stack overflow was detected Explanation: A thread overflowed its stack. User Action: Recreate the thread with a larger stack or redesign the code to require less stack space; for example, nest your calls less deeply or allocate less storage on the stack.
cma_e_unimp	The specified DECthreads feature is not implemented Explanation: You attempted to use a feature that is not implemented in the version of DECthreads that you are running. This error can occur when a program developed on a system running a higher version of DECthreads is executed on a system that is running a lower version of DECthreads. User Action: Use a higher version of DECthreads that supports the feature or do not attempt to use the feature with a lower version of DECthreads.

(continued on next page)

Table C–1 (Cont.) DECthreads Exceptions

Exception	Explanation and User Action
cma_e_uninitexc	Uninitialized exception raised Explanation: Code using the EXC_HANDLING.H package, which provides portable exceptions for the C language, attempted to raise an exception that has not been initialized. User Action: Check the error messages to determine the program location where the uninitialized exception is being raised. Use the EXCEPTION_INIT macro defined in the EXC_HANDLING.H package to initialize the exception.
cma_e_unkstatus	Unknown exception reported Explanation: A status exception was raised for which DECthreads is unable to provide a meaningful text translation. User Action: Check the value of the status with which the exception object was initialized to make sure it has a proper text translation.
cma_e_use_error	Requested operation is inappropriate for the specified object Explanation: The state or type of an object is inappropriate for the operation; for example, the operation attempts to unlock a mutex that is not locked. User Action: Determine which routine caused the error and consult the documentation to learn which object states are appropriate for the routine.
cma_e_wrongmutex	Wrong mutex specified in condition wait

(continued on next page)

Table C–1 (Cont.) DECthreads Exceptions

Exception	Explanation and User Action
	Explanation: A thread attempted to wait for a condition variable that already has at least one thread waiting and that thread has specified a different mutex. DECthreads requires that all threads concurrently waiting for a condition variable specify the same mutex. User Action: Design your code so that each condition variable represents a particular state of shared data that is protected by a given mutex.

See Table 4–1 for a list of pthread exceptions. Most pthread exceptions correspond directly to a cma interface exception.

D

DECthreads Nonstandard Types for the POSIX 1003.4a Interface

Table D–1 lists the DECthreads nonstandard data types that correspond to the types specified by the P1003.4a standard, with an explanation of each type.

The types you choose depend on your goals. If you want to strictly code to the P1003.4a standard, use the P1003.4a types. If you want to run your program on a C compiler that is not fully ANSI C-compliant, using the DECthreads types may be the better choice. (The DECthreads types are designed to select the appropriate existing types provided by the C compiler.)

Table D–1 DECthreads Nonstandard Data Types

DECthreads Type	P1003.4a Type	Explanation
pthread_addr_t or any_t	`"void *"`	Unspecified type used for passing thread arguments, return values, and thread-specific data values.
pthread_startroutine_t or pthread_func_t	`"void *(*)(void *arg)"`	The type of a thread start routine (which is passed to `pthread_create`).
pthread_cleanup_t	`"void (*)(void *arg)"`	The type of a routine specified for a cleanup routine (which is passed to `pthread_cleanup_push`).
pthread_destructor_t	`"void (*)(void *arg)"`	The type of a routine used for a thread-specific data destructor function.

(continued on next page)

Table D–1 (Cont.) DECthreads Nonstandard Data Types

DECthreads Type	P1003.4a Type	Explanation
pthread_initroutine_t	`"void (*)(void)"`	The type of a routine used to initialize data with `pthread_once`.

Glossary

alert

See cancel.

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.

atomic queue

DECthreads Library object that can be used to communicate information among threads or among routines in a single thread.

attributes

Individual components of the attributes object. Attributes specify detailed properties about the objects to be created. *See also* attributes object.

attributes object

Object used to describe DECthreads objects (thread, mutex, condition variable, queue, or attributes object). This description consists of the individual attribute values that are used to create an object. *See also* attributes.

cancel

Mechanism by which one thread requests termination of another thread (or itself).

cancelable routine

Routine where general (synchronous) cancelability can occur only at specific, well-defined points.

cancelation point

DECthreads routine that, when called by a routine, can determine whether a cancel is pending for that routine, and if so, can deliver the cancel.

condition variable

Object that allows a thread to block its own execution until some shared data reaches a particular state.

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.

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.

exception

Object that describes an error condition.

exception scope

Block of code where exceptions are handled.

fast mutex

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

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 lock used by all threads in a program when calling routines or code that is not thread-safe to ensure serialization and exclusivity of the unsafe code.

guard area

Reserved area designed to help prevent or detect, or both, overflow of the thread's stack.

guardsize attribute

Minimum size (in bytes) of the guard area for the stack of a thread.

handle

Storage, similar to a pointer, that refers to a specific DECthreads object.

inherit scheduling attribute

Attribute that specifies whether a newly created thread inherits the scheduling attributes (scheduling priority and policy) of the creating thread or uses the scheduling attributes stored in the attributes object. *See also* thread attributes object.

lifetime

Length of time memory exists.

multithreaded programming

Division of a program into multiple threads that execute concurrently.

mutex

Meaning 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* fast mutex, nonrecursive mutex, and recursive mutex.

mutex attributes object

Attribute that allows you to specify values other than the defaults for mutex attributes when you create a mutex.

mutex type attribute

Attribute that specifies whether a mutex is fast, recursive, or nonrecursive.

nonrecursive mutex

Mutex that can be locked exactly once by a thread, like a fast mutex. If a thread tries to lock the mutex again without first unlocking it, the thread receives an error instead of deadlocking. *See also* mutex.

nonterminating signal

Signal that does not result in the termination of the process by default. *See also* terminating signal.

per-thread context

See thread-specific data.

predicate

Boolean expression that defines a state; threads wait on a condition variable for 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.

queuesize attribute

Number of elements allowed on a queue. *See also* atomic queue.

race condition

Occurs when two or more threads perform an operation, and the result of the operation depends on unpredictable timing factors.

recursive mutex

Mutex that can be locked more than once by a given thread without causing a deadlock. *See also* mutex and deadlock.

scheduling policy attribute

Attribute that describes how the thread is scheduled for execution relative to the other threads in the program. *See also* thread attributes object.

scheduling priority attribute

Attribute 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.

stacksize attribute

Minimum size (in bytes) of the memory required for a thread's stack.

stack memory

Memory that is allocated by code generated by the language compiler at run time, generally when a routine is initially called. *See also* dynamic memory and static memory.

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.

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.

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 other than the defaults when you create a thread.

thread-reentrant

Routine that functions normally despite being called simultaneously or sequentially in different threads.

thread-safe

Routine that can be called simultaneously from multiple threads without risk of corruption.

thread-specific data

User-specified fields of arbitrary data that can be added to a thread's context.

timeslicing

Mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.

Index

C

D

E

F

G

H

I

J

K

Q

R

S

T

U

W

Y

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[a]	————	SSB Order Processing – NQO/V19 *or* U. S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

DEC OSF/1
Guide to DECthreads
AA-Q2DPA-TK

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- Do Not Tear - Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

--- Do Not Tear - Fold Here ----

Cut
Along
Dotte
Line

Reader's Comments

DEC OSF/1
Guide to DECthreads
AA-Q2DPA-TK

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- Do Not Tear - Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut
Alon
Dott
Line