Threads and the Single UNIX® Specification, Version 2

Extracted from Go Solo 2

The Open Group

The Open Group gratefully acknowledges:

- the Space and Naval Warfare Systems Command, who gave permission for use of their original paper on POSIX threads.
- members of the Aspen Group who contributed to the development of the Aspen threads extensions
- the authors of these papers, Karen Gordon, Joseph M. Gwinn, Jim Oblinger, Frank Prindle, Finnbarr P. Murphy, Dave Butenhof

Threads and the Single UNIX® Specification, Version 2

Extracted from Go Solo 2

ISBN: X909
Document Number: 0-13-575689-8

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire RG1 1AX

Or by email to:

a.josey@opengroup.org

Chapter 10

POSIX Threads

POSIX.1c, the POSIX Threads Extension, introduces a robust set of thread facilities to the POSIX family of standards. These thread facilities enable programmers to create and manage multiple threads of control within a single POSIX process.

This chapter offers a programmer's overview of POSIX.1c, which is included in the Single UNIX Specification, Version 2. It describes the application programming interfaces specified under the eight threads-related options, and it briefly explains why and how the interfaces may be used. For each option, an indication of its status within the Single UNIX Specification is given.

10.1 Introduction

Many applications can be designed as a set of cooperating sequential tasks, or threads of control, each assigned to some specific aspect of the problem being solved. For example, in realtime control applications, a separate task may be assigned to each sensor and actuator. In producer-consumer applications, some tasks may be producers, and others may be consumers. In a file server, each request for service may be handled by a newly spawned task. In many applications, dedicated tasks may handle background processing, while the main task does the foreground processing. Other dedicated tasks may await asynchronous events, so that the main task does not have to be interrupted when the events occur. In parallel-programming applications, the parallel threads of execution correspond to cooperating sequential tasks, which may run simultaneously on different processors in a multiprocessor system. In all cases, effective management of the cooperating sequential tasks is vital to the success of the application.

In traditional POSIX and UNIX operating systems, a sequential task — that is, thread of control — corresponds to the process. The process is the basic programming abstraction in these operating systems. Each process has a distinct address space — access to specific files, I/O devices, and other computer system resources — and a single thread of control. Resources are not shared among processes, because the processes, in general, represent different programs that must be protected from one another. In other words, the process serves not only as a unit of concurrency, but also as a unit of resource allocation (that is, protection boundary). In these operating systems, the cooperating sequential task programming paradigm is implemented by dividing the application into multiple processes.

The problem with the POSIX process model is that the burden of being a unit of resource allocation makes the process a *heavyweight* unit of concurrency.[1] POSIX.1c addresses this problem by adding a second level of concurrency to the POSIX process model. In particular, an

1. Processes are relatively *heavyweight* in terms of the overhead incurred at process creation and at context switches, due to the amount of process context that must be established at creation and saved and restored at context switches. Processes are also heavyweight with respect to communication, since processes can share data only through the explicit use of interprocess communication mechanisms such as message passing.

application is allowed to establish concurrent threads of control within a single process. The threads of control — variously referred to as threads, C threads, pthreads, tasks, or lightweight processes — share process memory and other resources. Thus, they incur minimal overhead and can offer efficient, high-performance concurrency. The thread mechanism has been identified as being particularly important to realtime systems, multiprocessor systems, and Ada programs using the Ada tasking facility.

POSIX.1c specifies a robust set of thread facilities. In doing so, it introduces new application program interfaces in the following functional areas:

- Thread management (ISO/IEC 9945-1: 1996 (POSIX-1), §16)
- Thread-specific data (ISO/IEC 9945-1: 1996 (POSIX-1), §17)
- Thread cancellation (ISO/IEC 9945-1: 1996 (POSIX-1), §18)
- Thread synchronization (ISO/IEC 9945-1: 1996 (POSIX-1), §11)
- Thread execution scheduling (ISO/IEC 9945-1: 1996 (POSIX-1), §13.1-13.5)
- Thread synchronization scheduling (ISO/IEC 9945-1: 1996 (POSIX-1), §13.6).

POSIX.1c also amends certain interfaces in POSIX.1 and POSIX.1b:

- Process creation interfaces are amended to work in the presence of threads (ISO/IEC 9945-1: 1996 (POSIX-1), §3.1).
- Signal interfaces are amended to work in the presence of threads (ISO/IEC 9945-1: 1996 (POSIX-1), §3.3).
- Thread creation is introduced as an event notification mechanism (ISO/IEC 9945-1: 1996 (POSIX-1), §3.3).
- All POSIX.1 and POSIX.1b blocking functions are amended to suspend only the calling thread, instead of the entire process in which the thread is embedded (ISO/IEC 9945-1: 1996 (POSIX-1), §3.5, 6, 11, 14).
- The POSIX.1b blocking functions (that is, semaphore locking, message sending, message receiving) that were defined in POSIX.1b to use process priorities to resolve contention are amended to take thread priorities into account (ISO/IEC 9945-1: 1996 (POSIX-1), §11, 15).
- POSIX.1c establishes thread-safe versions of POSIX.1 and C-language functions (ISO/IEC 9945-1: 1996 (POSIX-1), §4, 5, 8, 9).
- POSIX.1c redefines *errno* in order to make it meaningful in the presence of threads (ISO/IEC 9945-1: 1996 (POSIX-1), §2.4).

POSIX.1c thread facilities are specified as a series of eight options to POSIX.1, as amended by POSIX.1b. The core facilities are part of the *threads* option, whose presence in an operating system implementation is indicated by definition of the symbol _POSIX_THREADS. The remaining facilities are placed under seven subsidiary options, as illustrated below.

Option	Mandatory POSIX	Mandatory UNIX 98
_POSIX_THREAD_SAFE_FUNCTIONS	Yes	Yes
_POSIX_THREAD_ATTR_STACKADDR	No	Yes
_POSIX_THREAD_ATTR_STACKSIZE	No	Yes
_POSIX_THREAD_PROCESS_SHARED	No	Yes
_POSIX_THREAD_PRIORITY_SCHEDULING	No	Realtime Threads FG*
_POSIX_THREAD_PRIO_INHERIT	No	Realtime Threads FG
_POSIX_THREAD_PRIO_PROTECT	No	Realtime Threads FG

One of these — _POSIX_THREAD_SAFE_FUNCTIONS — is required to be supported whenever the threads option is supported; the others need not be for POSIX threads conformance. For conformance to the Single UNIX Specification, Version 2, the threads options are split so that non-realtime functionality is mandatory, and realtime functionality is grouped into a single option: the Realtime Threads Feature Group. The facilities described in the following sections are part of the threads option, unless explicitly described as being part of one of the above options.

10.2 Thread Management

POSIX.1c introduces interfaces for creating, managing, and terminating threads, where a thread is defined as follows ISO/IEC 9945-1: 1996 (POSIX-1), §2.2.2):

> ‘‘A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy (if the thread priority scheduling option is used), errno value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread ... shall be accessible to all threads in the same process.’’

Thus, threads share process state, including process address space, as well as other resources such as files. Threads are identified by thread IDs, which are guaranteed to be unique only within a process.

Threads have properties referred to as *thread attributes* associated with them. There is one mandatory attribute — *detachstate* — and several optional attributes. The *detachstate* attribute governs the lifetime of a thread ID. That is, if a thread is created in a *detached* state, then its thread ID loses significance upon termination of the thread; if a thread is created in a *joinable* state, then its thread ID can be used as a parameter of the *pthread_join*( ) function, discussed below in the list of interfaces.

POSIX.1c specifies two classes of optional thread attributes. The first class is associated with the POSIX.1c stack options _POSIX_THREAD_ATTR_STACKSIZE and _POSIX_THREAD_ATTR_STACKADDR, and the second class is associated with the POSIX.1c thread priority scheduling option _POSIX_THREAD_PRIORITY_SCHEDULING. The stack attributes are *stacksize*, which specifies a minimum size for the thread’s stack, and *stackaddr*, which specifies the address to be used for the thread’s stack. The stack attributes are mandatory for conformance to the Single UNIX Specification, Version 2. The scheduling attributes are discussed in Section 10.6 on page 92.

* FG is an abbreviation for Feature Group.

At thread creation, a thread can be assigned default attribute values, or it can be assigned attribute values as specified in a designated *thread attributes object*. Thread attributes objects, which hold the values of thread attributes, are of an *opaque* data type.[2] Accordingly, the standard defines functions for setting and getting the values of each of the attributes held in threads attributes objects. In addition, the standard defines functions for setting and getting the values of selected attributes dynamically. For example, there are functions defined to set and get the value of the thread *detachstate* attribute in a specified thread attributes object, but not to dynamically change the value of the attribute in an existing thread. On the other hand, the priority of a thread can be assigned a value at thread creation through a thread attributes object, and, in addition, the priority can be dynamically changed during the execution of a thread.

The *attributes object* concept is used not only for threads, but also for mutexes and condition variables. It is intended to support extensibility. Implementations, or future standards, can add attributes without modifying the interfaces defined in POSIX.1c. They simply add functions for setting and getting the values of the new attributes in the thread attributes object.

POSIX.1c defines the following specific functions for thread management:

- Initializing a thread attributes object (*pthread_attr_init*( )); that is, setting all attributes to default values.
- Destroying a thread attributes object (*pthread_attr_destroy*( )); that is, rendering it invalid for use until it is re-initialized. In the POSIX.1c model, a thread attributes object must be destroyed before it can be re-initialized. The effect of re-initializing a thread attributes object that has not been destroyed (and is thus still initialized) is undefined.
- Setting or getting the values of the following thread attributes in a specified thread attributes object: *detachstate*, *stacksize* and *stackaddr*. The functions are *pthread_attr_setdetachstate*( ), *pthread_attr_getdetachstate*( ), *pthread_attr_setstacksize*( ), *pthread_attr_getstacksize*( ), *pthread_attr_setstackaddr*( ), and *pthread_attr_getstackaddr*( ).
- Creating a new thread (*pthread_create*( )), with attributes having either default values or values given in a specified thread attributes object. Upon creation, the thread starts executing a specified start routine, with a specified argument. The signal mask of the newly created thread is inherited from the thread that created it.

  The same thread attributes object can be used multiple times to create multiple threads. An application programmer could also set up multiple thread attributes objects, each with different attribute values, to be used in creating different classes of threads.
- Obtaining the calling thread's ID (*pthread_self*( )).
- Comparing two specified thread IDs (*pthread_equal*( )).
- Waiting for another thread, specified by its thread ID, to terminate (*pthread_join*( )). This function represents one mechanism through which threads can synchronize their execution. Other synchronization mechanisms include mutexes, condition variables, and signals,

2. This means that the internal structure of the thread attributes object is purposely unspecified in the standard and, in effect, hidden from applications; an application can access individual attributes held in the attributes object in a portable way only through operating system functions. The standard specifies separate operating system functions for each attribute. Therefore, the addition of a new attribute entails the addition of new operating system functions for accessing the attribute in the thread attributes object, rather than the re-specification of the data type of the thread attributes object. From this viewpoint, the opaqueness of the data type is an extensibility mechanism: old applications that can safely ignore the new attribute are unaffected by the addition of the new functions, whereas they would be affected by a re-specification of the data type of the thread attributes object (that is, they would have to be recompiled).

described in Section 10.5 on page 90 and Section 10.9 on page 96.

- Terminating the calling thread (*pthread_exit*( )).

In addition, POSIX.1c defines a function, *pthread_once*( ), in support of dynamic package initialization. The function enables a package, such as a C library, to execute a specified initialization routine only once within a multi-threaded user process, upon the first invocation of the package by the process. The function, which would be called from within the package doing the initialization, is as follows.

Executing a specified initialization routine (*pthread_once*( )) only once within a process, at its first invocation. The *pthread_once*( ) function has two arguments: the initialization routine and a special-purpose flag. The flag is initialized to a special value in accordance with the standard's specification of the *pthread_once*( ) function.

The motivation, as described in ISO/IEC 9945-1: 1996 (POSIX-1), §B.16.2.8, is that some C-library routines are set up to execute an initialization routine upon the first invocation of the library routine by a process. The library routine uses a flag, declared as a static variable, to indicate whether or not the initialization routine has been executed. The flag is initially set to a value indicating non-initialized. Then, when the library routine is first invoked by the process, the library routine checks the flag, finds it set to the non-initialized value, executes the initialization routine, and finally sets the flag to the initialized value. In subsequent invocations, the library routine finds the flag set to the initialized value and bypasses the call to the initialization routine.

The problem that arises when a multi-threaded process attempts to use a library routine set up in this way is that two threads might call the library routine at about the same time and both find it set to the non-initialized value. That is, the second thread might check the flag before the first has a chance to change the value to indicate initialized. The solution to this problem entails providing mutually exclusive access to the code that deals with the flag and the initialization routine. The *pthread_once*( ) function, through the use of a special-purpose flag, in effect implements this solution.

10.3 Thread-specific Data

POSIX.1c provides a mechanism that enables applications to maintain specified data on a per-thread basis. The mechanism is motivated by the need of some modules (that is, groups of related functions) to maintain selected data across function invocations (that is, to maintain static data in the C programming language). If such a module is being used by multiple threads of a multi-threaded process, then the module may need to maintain such data separately for each calling thread, depending on the particular application.

For example, consider a module consisting of push, pop, and clear stack functions. Suppose that the module declares the stack and stack pointer as static data, instead of as function parameters. If the module is being used in a multi-threaded process, in which each thread needs its own stack and stack pointer, the module must have a mechanism for maintaining per-thread stacks and stack pointers.

In the POSIX.1c model, per-thread data is maintained through a key/value mechanism, an approach designed for efficiency and ease of use.[3] A *key* is an opaque object of type

3. An application process could maintain thread-specific data in other ways. For example, it could use a hash function on thread ID as a means of access to an area of thread-specific data. Then the application process would have to manage the use of sub-areas of the thread-specific data. The POSIX.1c thread-specific data interfaces, on the other hand, provide threads more direct access to sub-areas of their thread-specific data. Moreover, the sub-areas are independent; different parts of the application process can use different sub-areas without any need for process-wide cooperation.

pthread_key_t. The *value* of a key is thread-specific; that is, each key that has been established for a multi-threaded process has a distinct value for each thread of the process. For this reason, the thread-specific data of a process is sometimes thought of as a matrix, with rows corresponding to keys and columns to threads, although implementations need not work this way. A process can have up to PTHREAD_KEYS_MAX keys (or rows), where PTHREAD_KEYS_MAX is at least 128.

Keys are of opaque data type so that operating system implementations can have freedom in setting them up to offer efficient access to thread-specific data. Instead of holding thread-specific values directly, keys may hold means of accessing thread-specific values. Conceptually, a key isolates a row of the thread-specific data matrix, and then the key uses the thread ID of the calling thread (the thread calling *pthread_getspecific*( ) or *pthread_setspecific*( )) to isolate an entry in the row, thus obtaining the desired key value.

Typically, the value associated with a given key for a given thread is a pointer to memory dynamically allocated for the exclusive use of the given thread (for example, per-thread stack and stack pointer). The scenario for establishment and use of thread-specific data can be described as follows. A module that needs to maintain static data on a per-thread basis creates a new thread-specific data key as a part of its initialization routine. At initialization, all threads of the process are assigned null values for the new key. Then, upon each thread's first invocation of the module (which can be determined by checking for a null key value), the module dynamically allocates memory for the exclusive use of the calling thread, and stores a pointer to the memory as the calling thread's value of the new key. Upon later invocations of the same module by the same thread, the module can access the thread's data through the new key (that is, the thread's value for the key). Other modules can independently create other thread-specific data keys for other per-thread data for their own use.

The interfaces for managing thread-specific data are as follows:

- Creating or deleting a process-wide key to thread-specific data (*pthread_key_create*( ), *pthread_key_delete*( )). At creation, the value of the key is initialized to NULL for all active threads in the process in which the calling thread is embedded.

  At creation, a *destructor function* optionally can be specified for the key. Upon thread termination, if the key has a non-NULL value for the terminating thread, the destructor function is invoked with the key value as its argument. It is envisioned that the destructor function would be used to release the memory identified by the key value.

- Setting or getting the calling thread's value of a specified key (*pthread_setspecific*( ), *pthread_getspecific*( )). Note that a thread can access only its own values of any keys, because there is no thread ID parameter in these functions. When a thread accesses a key, it implicitly accesses its own value of the key.

10.4 Thread Cancellation

POSIX.1c provides facilities for canceling threads. The facilities enable a thread to cancel another specified thread within the same process. The facilities are aimed at allowing applications to cancel *cooperating* threads.

A thread that is the target of cancellation must be cooperative in the following sense. Each thread controls its own cancelability state and type. The cancelability state can be *enabled* or *disabled*. For the enabled state, the type can be *asynchronous* (cancellation requests accepted at any time) or *deferred* (cancellation accepted only at designated cancellation points). By default, threads are initialized with state enabled and type deferred. An uncooperative thread could disable cancellation, and thus thwart cancellation requests.

An application is expected to make cancellation graceful by specifying cancellation cleanup handlers. The cleanup handlers perform actions such as unlocking mutexes owned by the target thread (the thread being canceled), signaling conditions that the target thread may have caused to become true, releasing resources, and so on. These actions enable other threads in the application to make progress after the cancellation occurs.

The thread cancellation facilities specified in POSIX.1c include the following functions:

- Canceling execution of a specified thread (*pthread_cancel*( )). The cancellation occurs in accordance with the target thread's cancelability state and type. If and when cancellation does occur, cancellation cleanup handlers are invoked in last-in-first-out (LIFO) order, followed by thread-specific data destructor functions in unspecified order. Then the thread is terminated.
- Setting the cancelability state of the calling thread (*pthread_setcancelstate*( )) to either enabled or disabled.
- Setting the cancelability type of the calling thread (*pthread_setcanceltype*( )) to either asynchronous or deferred.
- Creating a cancellation point (*pthread_testcancel*( )). To designate a point in its execution as a cancellation point, a thread simply makes a call to the *pthread_testcancel*( ) function at that point.
- Establishing cancellation cleanup handlers (*pthread_cleanup_push*( ) and *pthread_cleanup_pop*( )). The *pthread_cleanup_push*( ) function is used to identify a specified routine as a cleanup handler. Conceptually, the routine is pushed onto the top of the calling thread's cancellation cleanup stack. The *pthread_cleanup_pop*( ) function is used to remove and optionally execute the cleanup handler at the top of the cancellation cleanup stack.

  The *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions are used in pairs within the same lexical scope. For example, a given routine of an application can begin by pushing a cleanup handler written for the routine onto the cancellation cleanup stack of a calling thread and end by popping the cleanup handler from the stack. If the routine calls another routine, the second routine can also begin by pushing a different cleanup handler onto the stack and end by popping the cleanup handler from the stack. Thus, the most recently called subroutines have their cleanup handlers at the top of the cancellation cleanup stack.

10.5 Thread Synchronization

POSIX.1c introduces mutexes and condition variables as inter-thread synchronization mechanisms. While POSIX.1b semaphores could be used,[4] mutexes and condition variables offer higher performance and enhanced robustness in many cases. As noted in POSIX.1c, semaphores have two distinct uses: locking and waiting. Mutexes take over the locking role of semaphores, while condition variables take over the waiting role. That is, the mutex (a term derived from ‘‘mutual exclusion’’) is a locking mechanism used by threads to ensure mutually exclusive use of shared data and critical sections of code.[5] The condition variable is a waiting mechanism used to synchronize the executions of threads with respect to higher level actions of one another.

A condition variable always has a mutex associated with it. The mutex provides mutually exclusive access to some shared data (for example, a bounded buffer). In a typical scenario, a thread locks the mutex and then uses a condition variable to wait for some condition (for example, buffer not full) to become true with respect to the shared data. After using the shared data, the thread uses another condition variable to signal that it has caused some other condition (for example, buffer not empty) to become true with respect to the shared data, and then unlocks the mutex.

Mutexes and condition variables can be applied to the producer-consumer problem as follows. Suppose a producer and consumer need to coordinate use of a bounded buffer. They can do so through the use of two condition variables, non-empty and non-full, and a mutex. The mutex protects the shared resources, namely, the buffer and its attributes (that is, count of unconsumed characters, producer’s index, consumer’s index). The producer’s execution cycle is as follows:

1. Produce a data item.
2. Lock the mutex.
3. Check the condition that the count of unconsumed data items in the buffer is less than the number of slots in the buffer. If the condition is false, wait on the condition variable non-full (which unlocks the mutex for the duration of the wait) and then repeat the check upon return from the wait. When the condition is determined to be true, go to step 4.
4. Put a data item in the buffer, and increment both the producer’s index into the buffer and the count of unconsumed data items in the buffer.
5. If the count of unconsumed data items is now equal to one, inform the consumer by signaling via the condition variable non-empty. If the consumer was blocked on non-empty, this signal unblocks it.
6. Unlock the mutex.

The consumer’s execution cycle is complementary.

---

4. Mutexes and condition variables can be used only by entities that share memory. While they are introduced primarily as an inter-thread synchronization mechanism, they can also be used among processes that share memory. Semaphores, on the other hand, are primarily an interprocess synchronization mechanism, even though they can be used by threads within a process. Semaphores get around the need for shared memory through global names, which make them accessible to all processes.
5. Unlike semaphores, mutexes are always unlocked by the entity that locked them. Thus, during the time a mutex is locked, it is said to be owned by the entity that locked it. Because a mutex has an owner, priority inheritance can be applied to the use of the mutex as a means of overcoming priority inversion (see Section 10.7 on page 94).

Both mutexes and condition variables are opaque objects (of types **pthread_mutex_t** and **pthread_cond_t**, respectively), and they use the attributes object concept described above in Section 10.2 on page 85. Accordingly, the POSIX.1c mutex and condition variable facilities include the following functions:

- Initializing or destroying a mutex attributes object (*pthread_mutexattr_init*( ), *pthread_mutexattr_destroy*( )). These functions are similar to their counterparts for threads attributes objects (see Section 10.2 on page 85).
- Setting or getting the value of the mutex attribute *pshared* in a specified mutex attributes object. These functions (*pthread_mutexattr_setpshared*( ), and *pthread_mutexattr_getpshared*( )) and the attribute itself are optional. Their presence in an implementation is indicated by definition of the symbol _POSIX_THREAD_PROCESS_SHARED. They extend the boundaries within which a mutex can be used beyond the threads of a single process. In particular, they enable a mutex to be used by any threads of any processes sharing the memory in which the mutex resides. This functionality is mandatory for conformance to the Single UNIX Specification, Version 2.
- Initializing a specified mutex (*pthread_mutex_init*( )), with attributes having either default values or values given in a specified mutex attributes object. Upon initialization, the mutex is in an unlocked state.
- Destroying a specified mutex (*pthread_mutex_destroy*( )); that is, rendering it invalid for subsequent use in mutex operations other than mutex initialization.
- Initializing or destroying a condition variable attributes object. These functions (*pthread_condattr_init*( ), *pthread_condattr_destroy*( )) are similar to their counterparts for threads attributes objects and mutexes.
- Setting or getting the value of the condition variable attribute *pshared* in a specified condition variable attributes object. These optional functions (*pthread_condattr_setpshared*( ), *pthread_condattr_getpshared*( )) are similar to their counterparts for mutexes. This functionality is mandatory for conformance to the Single UNIX Specification, Version 2.
- Initializing or destroying a specified condition variable. These functions (*pthread_cond_init*( ), *pthread_cond_destroy*( )) are similar to their counterparts for mutexes.

The mutex functions defined by POSIX.1c are as follows:

- Locking a specified mutex. Two forms of interfaces are provided. The first, *pthread_mutex_lock*( ), causes the calling thread to block if the mutex is already locked. The second, *pthread_mutex_trylock*( ), is a conditional form. That is, it locks the mutex only if the mutex is currently unlocked; otherwise, it simply returns control to the calling thread. In no case does *pthread_mutex_trylock*( ) block the caller. In other words, *pthread_mutex_trylock*( ) *polls* the mutex.

  Contention for mutexes is resolved according to the scheduling attributes of the calling threads, described in Section 10.6 on page 92.

  When a thread acquires a lock on a mutex, it is said to become the owner of the mutex.
- Unlocking a specified mutex. This function (*pthread_mutex_unlock*( )) is called by the owner of a mutex to release the mutex.

The condition variable functions defined by POSIX.1c are:

- Blocking on a specified condition variable, under the protection of a specified mutex (*pthread_cond_wait*( )). Typically, prior to invoking the wait operation, the calling thread shall have locked the mutex, examined the condition associated with the condition variable by

querying shared data, and found the condition to be false. The invocation of the wait operation then causes the calling thread both to release the mutex and to block on the condition variable, the release and block being done atomically.

Upon return from the wait operation, the calling thread owns the mutex and can check the condition. If the condition is true, the thread can unlock the mutex and proceed. If the condition is false, the thread can simply repeat the wait operation.

A timeout can be associated with the wait operation through the use of the *pthread_cond_timedwait*( ) function.

- Unblocking a thread or threads blocked on a specified condition variable. Two functions are provided: *pthread_cond_signal*( ) for unblocking at least one blocked thread (if any are blocked), and *pthread_cond_broadcast*( ) for unblocking all blocked threads.

  The order in which threads are unblocked depends on the scheduling attributes of the threads, particularly priority.

POSIX.1c also amends the POSIX.1b semaphore interfaces. As indicated in Section 10.11 on page 98, it amends the *sem_wait*( ) function to suspend only the calling thread, and it amends the *sem_wait*( ) and *sem_post*( ) functions to take thread priorities into account in the resolution of contention for semaphores. In addition, it introduces an optimization feature: if a semaphore is initialized with a *pshared* argument of zero, then the semaphore is to be used as an inter-thread (*versus* interprocess) synchronization mechanism, and the operating system implementation can potentially optimize its performance. If an application tries to use a semaphore whose *pshared* argument is zero for interprocess synchronization, the effect is undefined in the standard.

10.6 Thread Execution Scheduling

POSIX.1c defines thread priority scheduling facilities as an option, designated by the symbol _POSIX_THREAD_PRIORITY_SCHEDULING, modeled on the POSIX.1b process priority scheduling option. However, the thread scheduling facilities, unlike the process scheduling facilities, use the attributes object concept, previously described in Section 10.2 on page 85. For conformance to the Single UNIX Specification, Version 2, this functionality is supported if the implementation provides the Realtime Threads Feature Group.

In particular, the thread scheduling facilities add the following attributes to the thread attributes object:

- Scheduling contention scope (*contentionscope*). The contention scope may be *process-wide* or *system-wide*.[6] If a thread has process-wide contention scope, it directly contends for processor resources only with other threads in its same process. If a thread has system-wide contention scope, it contends for processor resources with all threads in the system, according to their system-level scheduling attributes.

  For threads with system-wide contention scope, their system-level scheduling attributes are equal to their thread scheduling attributes. For threads with process-wide contention scope, their system-level scheduling attributes are derived from their thread and process scheduling

6. System-wide means across an implementation of a POSIX operating system, which is generally assumed to reside on a uniprocessor or a multiprocessor. (System-wide does not mean across autonomous systems inter-connected by a local area network.)

attributes using an implementation-dependent mapping. Thus, a thread with process-wide contention scope indirectly contends for processor resources with threads of system-wide scheduling scope.

- Scheduling policy (*schedpolicy*). The policies are the same as those specified for processes in POSIX.1b:

  1. preemptive, dynamic-priority-driven, using FIFO contention resolution within a priority level (SCHED_FIFO)[7]
  2. preemptive, dynamic-priority-driven, using round robin contention resolution within a priority level (SCHED_RR)
  3. implementation-dependent (SCHED_OTHER).

  The SCHED_FIFO and SCHED_RR policies are affected by the scheduling allocation domains of threads, where the scheduling allocation domain (also known as *processor affinity*) of a thread is defined to be the set of processors on which the thread can be scheduled at any given time. In particular, these policies are guaranteed to resolve contention as described in the preceding paragraph only in the case of allocation domains of size one. For allocation domains of size greater than one, POSIX.1c does not mandate a standard definition of the operation of these policies; instead, these policies operate in an implementation-dependent manner.

- Scheduling parameters (*schedparam*, a pointer to a structure). The SCHED_FIFO and SCHED_RR policies have a single parameter (that is, priority); the SCHED_OTHER policy has implementation-dependent parameters.

- Scheduling attributes inheritance flag (*inheritsched*). This attribute indicates whether the other scheduling attributes of a thread should be inherited[8] from the creating thread, or set to the values specified in the thread attributes object being used to create the thread.

For each of the attributes, POSIX.1c defines functions for setting and getting the value of the attribute in a specified thread attributes object (that is, *pthread_attr_setscope*( ), *pthread_attr_getscope*( ), *pthread_attr_setschedpolicy*( ), *pthread_attr_getschedpolicy*( ), *pthread_attr_setschedparam*( ), *pthread_attr_getschedparam*( ), *pthread_attr_setinheritsched*( ), *pthread_attr_getinheritsched*( )). For selected attributes — that is, the scheduling policy and associated parameters — the standard also defines functions for dynamically setting and getting the value of the attribute in a specified thread (*pthread_setschedparam*( ), *pthread_getschedparam*( )).[9] The values of the other attributes (*contentionscope* and *inheritsched*) cannot be changed after thread creation. POSIX.1c also introduces a function modeled on the POSIX.1b *sched_yield*( ) function. The function, *pthread_yield*( ), causes the calling thread to relinquish control of the processor. Additionally, the *sched_yield*( ) function itself is amended to refer to the calling thread, as opposed to the calling process. The

---

7. The rate monotonic scheduling policy [Liu and Layland 73; Sha and Goodenough 90; Sha and Sathaye 93] can be used as the method of assigning priorities to periodic threads in a periodic system. It assigns higher priorities to threads with shorter periods.
8. Here, the priority of a thread can be inherited from its creating thread. It should be noted that the term *priority inheritance* is not used in reference to this type of inheritance. As described in Section 10.7 on page 94, priority inheritance refers to the case in which a given thread inherits the priority of a higher-priority thread that the given thread is blocking by holding a resource such as a mutex, the intention being to prevent priority inversion.
9. This function changes the scheduling policy and parameters atomically, so that threads always maintain a consistent state with respect to scheduling. Changing the policy alone, or changing some but not all parameters, would leave the thread in an incomplete, incoherent, or undesired state. For example, suppose that a (valid) priority scheduling policy is specified, but an invalid scheduling priority is specified. Then, the *pthread_setschedparam*( ) function should fail.

pthread_yield( ) function is added, because the _POSIX_THREAD_PRIORITY_SCHEDULING option may be supported in the absence of _POSIX_PRIORITY_SCHEDULING, in which case the *sched_yield*( ) function would not be available.

10.7 Thread Synchronization Scheduling

POSIX.1c addresses priority inversion (a thread of high priority being blocked by one or more threads of lower priority), through optional synchronization scheduling facilities.[10] For conformance to the Single UNIX Specification, Version 2, this functionality is supported if the implementation provides the Realtime Threads Feature Group.

The prototypical example of priority inversion is the following scenario:

1. A low priority thread locks a mutex *m*.
2. A high priority thread preempts the low priority thread and attempts to lock the mutex *m*, but blocks because the mutex is already locked by the low priority thread.
3. The low priority thread resumes execution but is immediately preempted by a medium priority thread. This causes priority inversion: the high priority thread is being delayed by lower priority threads. The priority inversion can be unbounded if, for example, medium priority threads continue to arrive.

A proven mechanism for bounding priority inversion is priority inheritance, which causes a thread blocking higher priority threads to execute at the priority of the highest-priority blocked thread. In the above example, priority inheritance would cause the low priority thread to inherit the high priority of the thread being blocked on the mutex lock operation, and would consequently reduce the period of priority inversion caused by the low priority thread to the duration of the critical section that the low priority thread is executing under the protection of the mutex. However, other low priority threads could be holding other mutexes required by the high priority thread, leading to an extended period of priority inversion, namely, a chain of blocking of duration equal to the sum of the lengths of all the critical sections in the chain. Basic priority inheritance can be extended, through priority ceiling protocols, to address this problem (Sha and Goodenough 90; Sha et al. 90).

POSIX.1c specifies the basic priority inheritance protocol under the option _POSIX_THREAD_PRIO_INHERIT. It specifies the *priority ceiling protocol emulation*, an efficiently implementable variant of the original priority ceiling protocol, under the option _POSIX_THREAD_PRIO_PROTECT. The facilities use the mutex attributes object, adding the following attributes:

- Mutex protocol (*protocol*). The protocol indicates whether the mutex should be used with:
    1. the basic priority inheritance protocol
    2. the priority ceiling protocol emulation

10. Priority inversion is a problem associated with realtime systems. It is not generally regarded as a serious problem in non-realtime systems for two main reasons. First, non-realtime systems have fairness, throughput, and average response time as measures of merit. The lengthening of a single response time is not considered to be a disaster in non-realtime systems, whereas in realtime systems it can lead to a missed deadline and ultimately failure of the mission. Second, fairness-based schedulers typically employ various mechanisms to ensure the forward progress of all processes and threads, albeit slowly, so the inversion eventually resolves itself.

3. no priority inheritance protocol.

- Mutex priority ceiling (*prioceiling*). If the priority ceiling protocol emulation is specified, then a priority ceiling needs to be specified. The priority ceiling of a mutex is the priority of the highest-priority thread that ever locks it. (The application developer knows which thread, because the developer assigns priorities to threads and also determines which threads use which mutexes for synchronization). Under the priority ceiling protocol emulation, a thread executes a critical section at a priority equal to the highest priority ceiling of all the mutexes it owns at the time of entry to the critical section. The use of the priority ceiling protocol emulation eliminates chained blocking, reducing the period of priority inversion to the length of at most one critical section.

For each of these attributes, POSIX.1c defines functions for setting and getting the value of the attribute in a specified mutex attributes object (*pthread_mutexattr_setprotocol*( ), *pthread_mutexattr_getprotocol*( ), *pthread_mutexattr_setprioceiling*( ), *pthread_mutexattr_getprioceiling*( )). The mutex attributes object can be used at mutex initialization time to initialize the attributes of a mutex to specified values. Furthermore, the standard defines functions for dynamically changing and examining the priority ceiling of a specified mutex (*pthread_mutex_setprioceiling*( ) and *pthread_mutex_getprioceiling*( )). These functions can be used to alter the priority ceilings of mutexes when the priorities of the threads that use them are changed (for example, when a realtime system changes mode, to move from one phase of a mission to another).

10.8 Process Creation

POSIX.1c amends the process creation facilities of POSIX.1. In particular, it specifies how the process creation functions behave in the case of multi-threaded processes.

It should be recalled that in POSIX.1, process creation is achieved through the *fork*( ) and *exec*( ) functions. The *fork*( ) function is used to create a child process, whose process image is identical to the parent process image, with the exception of a few attributes, such as process ID, parent process ID, and accumulated user and system CPU times. Upon successful completion of the *fork*( ) function, both the parent process and the child process resume execution at the point following the return from *fork*( ). Typically, a conditional statement follows the call to *fork*( ); the conditional statement specifies one action for the parent and another for the child. If the child process is intended to run a different program, its action is to call one of the *exec*( ) functions to replace its current process image with a new process image from a specified executable file. On the other hand, if the child process is intended to serve as a new thread of control within the same program, then it simply continues executing its original process image.

POSIX.1c amends the *fork*( ) and *exec*( ) functions as follows:

- The *fork*( ) function is required to create a single-threaded process; when called from within a multi-threaded process, only one thread — the calling thread — is replicated in the child process.
- When called from within a multi-threaded process, the *exec*( ) functions cause all threads of the calling process to be terminated, and the specified executable file to be loaded and made ready for execution.

In addition, POSIX.1c introduces the following function:

- Registration of fork handlers (*pthread_atfork*( )). The fork handlers are routines that are to be executed in association with calls to the *fork*( ) function. There are three classes of fork handlers: *prepare*, *parent*, and *child*. Prepare fork handlers are executed prior to *fork*( )

processing, in the context of the calling thread. Parent fork handlers are executed upon completion of *fork*( ) processing in the parent, again in the context of the calling thread. Child fork handlers are executed upon completion of fork() processing in the child, in the context of the single thread initially existing in the child process.

Fork handlers are envisioned as a mechanism for dealing with the problem of *orphaned mutexes* that can occur when a multi-threaded process calls *fork*( ). The problem arises when threads other than the calling thread own mutexes at the time of the call to *fork*( ). Since the non-calling threads are not replicated in the child process, the child process is created with mutexes locked by non-existent threads. These mutexes can therefore never be unlocked.

Fork handlers are intended to resolve the problem of orphaned mutexes in the following way. Prepare fork handlers can be written to lock all mutexes. In this way, orphaned mutexes are avoided, and the resources protected by the mutexes are not left in inconsistent states. This is due to the fact that the calling thread itself, which is replicated in the child process, has locked all mutexes. Thus, both the parent and child processes have all mutexes locked upon completion of *fork*( ) processing, at which time the parent and child fork handlers execute. The parent and child fork handlers unlock mutexes locked by the prepare fork handler.

Fork handlers are especially useful in enabling independently-developed libraries and application programs to protect themselves from one another. A multi-threaded library can protect itself from application programs that issue *fork*( ) operations, possibly without even knowing that the library is multi-threaded, by providing fork handlers. Similarly, an application program can protect itself from *fork*( ) operations issued by library functions.

10.9 Signal Interfaces

Signals are the mechanism for notifying processes of POSIX.1 events such as keyboard interrupts, timer expirations, floating point overflows, invalid hardware instructions and invalid memory references, as well as POSIX.1b events such as asynchronous I/O completion, timer expiration, and message arrival. Signals are also used for limited interprocess communication; a process can send a specified signal (for example, terminate, stop, continue, or application-defined in the case of POSIX.1b realtime signals) to a specified process.

POSIX.1c specifies how signals behave in a multi-threaded process. In particular, it amends the POSIX.1 signal facilities, as amended by POSIX.1b, as follows:

- At generation, a signal is associated with either:
  1. the process, or
  2. a specific thread within the process.

  As stated in ISO/IEC 9945-1: 1996 (POSIX-1), §3.3.1.2:

  > ‘‘Signals that are generated by some action attributable to a particular thread, such as a hardware fault, shall be generated for the thread that caused the signal to be generated. Signals that are generated in association with a process ID or process group ID or an asynchronous event such as terminal activity shall be generated for the process.’’

  Timer expirations represent another class of asynchronous events.[11] In keeping with the

POSIX.1c signal model quoted above, in which signals denoting asynchronous events are generated for the process, the signals stemming from POSIX.1b timer expirations are left intact by POSIX.1c. In other words, the signals are generated for the process, and not for the thread that created the timer. Moreover, POSIX.1c clarifies the POSIX.1 *alarm*( ) function by specifying that the SIGALRM signal is generated for the process, and not for the thread that called *alarm*( ).[12]

- Signal masks are maintained on a per-thread basis. Thus, each thread of a multi-threaded process can independently specify which signals are to be blocked from delivery to it. A new function (*pthread_sigmask*( )) is defined for examining and changing the signal mask of a thread within a multi-threaded process. The interface is modeled on the POSIX.1 interface *sigprocmask*( ), whose behavior is declared by POSIX.1c to be unspecified in a multi-threaded process.
- However, in the interest of minimality and efficiency, the action to be taken upon delivery of a signal is specified on a process-wide basis as being one of the following: take the default action associated with the signal, ignore the signal, or execute a specified signal-catching function.
- Also in the interest of minimality and efficiency, signals are not delivered to multiple threads. POSIX.1 signals are delivered to at most one thread; POSIX.1b signals are delivered to exactly one thread.[13] If a signal was generated for a specific thread, then it is sent to that thread. If a signal was generated for the process, then it is sent to one thread of the process. The choice of thread is unspecified in POSIX.1c; implementations are free to deliver the signal to any eligible thread. An application can reduce the set of eligible threads to one specific thread through signal masks. The thread that receives the signal can then distribute the signal to the pertinent thread(s); for example, via condition variables or the *pthread_kill*( ) function (described below in the list of new functions).
- Whenever a signal with a specified signal action of terminate, stop, or continue is delivered to a thread, the signal action applies not just to the target thread, but to the entire process in which the thread is embedded. That is, all the threads of the process are terminated, stopped, or continued, in accordance with the specified signal action. This means that single-threaded programs may be rewritten as multi-threaded programs without their externally visible signal behavior changing.
- The POSIX.1 function *sigpending*( ) is amended to apply to signals pending for either the process or the calling thread.
- The POSIX.1 function *sigsuspend*( ) is amended to suspend only the calling thread.
- The POSIX.1b functions *sigwaitinfo*( ) and *sigtimedwait*( ) are amended to suspend only the calling thread. POSIX.1c also introduces the following functions:

---

11. An asynchronous event is simply an event whose time of occurrence is not tied to a specific point in the instruction sequence of a program.
12. Signals resulting from multiple invocations of the *alarm*( ) function are indistinguishable. Thus, the *alarm*( ) function cannot easily be used in a multi-threaded process, unless only one of the threads uses the *alarm*( ) function. On the other hand, signals indicating the expirations of POSIX.1b timers can be distinguished — when used in conjunction with POSIX.1b realtime signals — through the assignment of unique signal numbers or unique signal values to the realtime signals associated with the timers.
13. Recall that in the case of a single-threaded process, POSIX.1 signals are delivered *at most once* (some may be lost), whereas POSIX.1b realtime signals are delivered *exactly once* (they are queued, rather than overwritten).

— Examining and changing the signal mask of a thread within a multi-threaded process (*pthread_sigmask*( ), which was mentioned above in the discussion on signal masks).

— Waiting for a signal from a specified set of signals to become pending (*sigwait*( )). The idea is that a thread would use its signal mask to block asynchronous delivery of signals and then use this function to poll for occurrences of the events represented by the signals.

When more than one thread is using *sigwait*( ) to wait for the same signal, only one thread returns from *sigwait*( ) when the signal becomes pending. The choice of thread is unspecified in POSIX.1c.

— Sending a specified signal to a specified thread (*pthread_kill*( )).[14]

10.10 Thread Creation

POSIX.1c introduces a new event notification mechanism, based on thread creation. It does so by extending the POSIX.1b realtime signal facilities to enable a process to designate thread creation (in particular, execution of a specified notification function within a newly created thread) as a notification mechanism for a specified event (designated by a realtime signal number). The notification function is invoked with the signal value generated at the time of the event as an input parameter.

10.11 Blocking Functions

POSIX.1c amends POSIX.1 and POSIX.1b to ensure that all blocking functions suspend only the calling thread, and not the entire process in which the thread is embedded. The functions that are explicitly amended include the following:

- POSIX.1 functions *wait*( ) and *waitpid*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §3.2)
- POSIX.1 function *sigsuspend*( ) and POSIX.1b functions *sigwaitinfo*( ) and *sigtimedwait*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §3.3)[15]
- POSIX.1 functions *pause*( ) and *sleep*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §3.4)
- POSIX.1 function *open*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §5)
- POSIX.1 functions *read*( ), *write*( ), *fcntl*( ) (when the command argument F_SETLKW is used), and POSIX.1b functions *lio_listio*( ) (when the synchronization option LIO_WAIT is used) and *aio_suspend*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §6)
- POSIX.1b function *sem_wait*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §11)
- POSIX.1b function *nanosleep*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §14).

The IEEE PASC Realtime Working Group[16] did not find it necessary to explicitly amend all

14. The name *pthread_kill*( ) is used to denote the similarity of this function to the POSIX.1 *kill*( ) function, which is the general interface used to send a signal to a process. Both are misnamed, because only certain signals, sent under certain conditions, cause the termination of the thread or process.

15. Previously noted in Section 10.9 on page 96. Repeated here for completeness.

16. This working group was originally designated the IEEE P1003.4 Working Group, and is now a subgroup of the larger IEEE PASC System Services Working Group (SSWG). It is sometimes referred to as SSWG-RT.

POSIX.1 and POSIX.1b blocking functions, due to the wording used in the descriptions of some of the functions. That is, some of the descriptions do not state that the calling process blocks; instead, they state that the function call blocks. For example, the *mqsend*( ) function specification states the following (POSIX.1b, §5.2.4.2):

> ‘‘If the specified message queue is full ... *mq_send*( ) shall block until space becomes available to enqueue the message ...’’

POSIX.1c amends selected POSIX.1b functions to use thread priorities, if applicable, in the resolution of resource contention. The selected functions are those that were defined in POSIX.1b to use process priorities for resolving resource contention:

- *sem_wait*( ) and *sem_post*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §11.2.6-11.2.7)
- *mq_send*( ) and *mq_receive*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §15.2.4-15.2.5).

As in POSIX.1b, the focus of POSIX.1c is on processor scheduling, in part to minimize the impact on existing standards (that is, POSIX.1), as well as existing implementations, which typically do not depend on priorities for resolving all resource contention. This is why only selected functions are mandated to use process and/or thread priorities in resolving resource contention.

10.12 Thread-safe POSIX.1 and C-language Functions

POSIX.1 and C-language functions were written to work in an environment of single-threaded processes. Reentrancy was not an issue in their design: the possibility of a process attempting to *re-enter* a function through concurrent invocations was not considered, because threads — the enabler of concurrency within a process — were not anticipated.

So, as it turns out, some POSIX.1 and C-language functions are inherently non-reentrant with respect to threads; that is, their interface specifications preclude reentrancy.[17] For example, some functions (such as *asctime*( )) return a pointer to a result stored in memory space allocated by the function on a per-process basis. Such a function is non-reentrant, because its result can be overwritten by successive invocations. Other POSIX.1 and C-language functions, while not inherently non-reentrant, may be implemented in ways that lead to non-reentrancy. For example, some functions (such as *rand*( )) store state information (such as a seed value, which survives multiple function invocations) in memory space allocated by the function on a per-process basis. The implementation of such a function is non-reentrant if the implementation fails to synchronize invocations of the function and thus fails to protect the state information. The problem is that when the state information is not protected, concurrent invocations can interfere with one another (for example, see the same seed value).

Functions must be reentrant in an environment of multi-threaded processes, in order to ensure that they can be safely invoked by concurrently executing threads. POSIX.1c takes three actions in the pursuit of reentrancy. First, POSIX.1c imposes reentrancy as a general rule: all functions, unless explicitly singled out as exceptions to the rule, must be implemented in a way that preserves reentrancy. Second, POSIX.1c redefines *errno*, as described below in Section 10.13 on page 102. Third, for those functions whose interface specifications preclude

17. In POSIX.1c, a *reentrant function* is defined as a ‘‘function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved” (ISO/IEC 9945-1: 1996 (POSIX-1), §2.2.2).

reentrancy, POSIX.1c defines alternative reentrant versions as follows:

- As previously noted, some functions are non-reentrant because they return results in per-process library-allocated structures that may be static and thus subject to overwriting by successive calls. These include:

  — The POSIX.1 process environment functions *getlogin*( ) and *ttyname*( ) (see ISO/IEC 9945-1: 1996 (POSIX-1), §4.2.4 and 4.7.2)

  — The C-language functions *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) (see ISO/IEC 9945-1: 1996 (POSIX-1), §8.3.4 to 8.3.7)

  — The POSIX.1 system database functions *getgrgid*( ), *getgrnam*( ), *getpwuid*( ), and *getpwnam*( ) (see ISO/IEC 9945-1: 1996 (POSIX-1), §9.2.1 and 9.2.2).

  POSIX.1c defines reentrant versions of these functions; the new functions have _r appended to the function names (for example, *asctime_r*( )). To achieve reentrancy, the new _r functions replace library-allocated structures with application-allocated structures that are passed as arguments to the functions at invocation.

- Some functions can be reentrant or non-reentrant, depending on their arguments. These include the C-language function *tmpnam*( ) and the POSIX.1 process environment function *ctermid*( ). These functions have pointers to character strings as arguments. If the pointers are not NULL, the functions store their results in the character string; however, if the pointers are NULL, the functions store their results in an area that may be static and thus subject to overwriting by successive calls.

  To ensure reentrancy of these functions, POSIX.1c simply restricts their arguments to non-NULL (ISO/IEC 9945-1: 1996 (POSIX-1), §4.7.1 and 8.2.5).

- As previously noted, some functions are non-reentrant because they communicate across multiple function invocations by maintaining state information in static library-allocated storage, which is shared by all the threads of a process, possibly without the benefit of synchronization. These include the C-language function *rand*( ), which is used to generate a process-wide pseudorandom number sequence. The function *rand*( ), which is called with no arguments, returns the next pseudorandom number in a sequence determined by an initial seed value (set via the function *srand*( )). As a side effect, the function *rand*( ) updates the seed value, enabling the sequence to progress. The seed value is held in a library-allocated static memory location. In a multi-threaded process, two or more threads might concurrently invoke *rand*( ), read the same seed value, and thus acquire the same pseudorandom number.

  POSIX.1c defines a reentrant version, *rand_r*( ), of this function (ISO/IEC 9945-1: 1996 (POSIX-1), §8.3.3). To ensure reentrancy, the *rand_r*( ) function is required to synchronize (that is, serialize) calls to itself, so that a thread is forced to finish acquiring one pseudorandom number in a sequence before another thread can begin to acquire the next number in the sequence.

  In addition to reentrancy, the *rand_r*( ) function offers applications flexibility in generating pseudorandom number sequences. It does so through the introduction of an argument: a pointer to an application-supplied memory location that is used to hold the seed value. As indicated above, an application can use *rand_r*( ) to generate a reliable process-wide pseudorandom number sequence (that is, a sequence without replicates). Alternatively, an application can use *rand_r*( ) to generate per-thread pseudorandom number sequences, by having each thread use a distinct seed as its *rand_r*( ) argument. In fact, an application can use *rand_r*( ) to generate an arbitrary number of uncorrelated sequences of pseudorandom numbers (each sequence governed by a distinct seed), which could prove to be useful in

Monte Carlo simulations and other similar applications.

Other functions in this class include:

— The C-language function *strtok*( ) (see ISO/IEC 9945-1: 1996 (POSIX-1), §8.3.3), which is used to find the next token in a string.

— The POSIX.1 file and directory function *readdir*( ), which is used to read the next entry in a directory stream. Note that this function also suffers from the problem of returning its result in a library-allocated structure. Both deficiencies are resolved in the reentrant version *readdir_r*( ) (ISO/IEC 9945-1: 1996 (POSIX-1), §5.1.2).

- The POSIX.1 and C-language functions that operate on character streams (represented by pointers to objects of type FILE) are required by POSIX.1c to be implemented in such a way that reentrancy is achieved (see ISO/IEC 9945-1: 1996 (POSIX-1), §8.2). This requirement has a drawback; it imposes substantial performance penalties because of the synchronization that must be built into the implementations of the functions for the sake of reentrancy. POSIX.1c addresses this tradeoff between reentrancy (safety) and performance by introducing high-performance, but non-reentrant (potentially unsafe), versions of the following C-language standard I/O functions: *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ). The non-reentrant versions are named *getc_unlocked*( ), and so on, to stress their unsafeness.

  To make it possible for multi-threaded applications to use the non-reentrant versions of the standard I/O functions safely, POSIX.1c introduces the following character stream locking functions: *flockfile*( ), *ftrylockfile*( ), and *funlockfile*( ). An application thread can use these functions to ensure that a sequence of I/O operations on a given character stream is executed as a unit (without interference from other threads).[18]

  As stated in the description of the character stream locking functions, all standard I/O functions that reference character streams shall behave as if they use *flockfile*( ) and *funlockfile*( ) internally to obtain ownership of the character streams. Thus, when an application thread locks a character stream, the standard I/O functions cannot be used by other threads to operate on the character stream until the thread holding the lock releases it.

The specifications introduced by POSIX.1c for the purpose of ensuring reentrancy of POSIX.1 and C-language functions are mandatory for operating system implementations that support threads. They are optional for implementations that do not support threads. This is accomplished in the standard by associating the reentrancy specifications with a separate option, _POSIX_THREAD_SAFE_FUNCTIONS, which is declared to be mandatory for implementations supporting the threads option. Accordingly, this option is mandatory for conformance to the Single UNIX Specification, Version 2.

18. It should be noted that the *flockfile*( ) function, like the *pthread_mutex_lock*( ) function, can lead to priority inversion. The application developer should take this into account when designing an application and analyzing its performance.

10.13 Redefinition of errno

In POSIX.1, *errno* is defined as an external global variable. But this definition is unacceptable in a multi-threaded environment, because its use can result in non-deterministic results. The problem is that two or more threads can encounter errors, all causing the same *errno* to be set. Under these circumstances, a thread might end up checking *errno* after it has already been updated by another thread.

To circumvent the resulting non-determinism, POSIX.1c redefines *errno* as a service that can access the per-thread error number as follows (ISO/IEC 9945-1: 1996 (POSIX-1), §2.4):

> ‘‘Some functions may provide the error number in a variable accessed through the symbol *errno*. The symbol *errno* is defined by including the header **<errno.h>**, as specified by the ISO C standard ... For each thread of a process, the value of *errno* shall not be affected by function calls or assignments to *errno* by other threads.’’

In addition, all POSIX.1c functions avoid using *errno* and, instead, return the error number directly as the function return value, with a return value of zero indicating that no error was detected. This strategy is, in fact, being followed on a POSIX-wide basis for all new functions.

Chapter 11

X/Open Threads

System Interfaces and Headers, Issue 5 (XSH) includes the threads model and interfaces defined in IEEE Std 1003.1c-1995 together with a number of extensions. These extensions, based on widely accepted existing industry practice, were developed by the Aspen Group and submitted to X/Open. This chapter is a brief introduction to these extensions. It assumes a working knowledge of the threads model specified in POSIX.1c and threads programming concepts in general.

11.1 Introduction

The X/Open Threads Extension is built upon the threads model and interfaces defined in IEEE Std 1003.1c-1995, commonly known as POSIX.1c or Pthreads. POSIX.1c contains much optional functionality. When POSIX.1c was incorporated into XSH, Issue 5, the majority of the POSIX.1c optional functionality was made mandatory and additional functionality, known as the Aspen threads extensions, was incorporated at the request of the Aspen Group.

11.2 The Aspen Group

Over the past few years almost all UNIX system vendors implemented some flavor of a threads package based on the POSIX.1c interfaces. Each vendor found that the POSIX.1c interfaces were not complete in solving all their threads requirements. Consequently, each vendor implemented extensions to their thread packages to meet those requirements.

Unfortunately for application developers, not all vendors implemented the same exact set of extensions. To make things worse, the same functionality was added, but used different interface names or parameter sets. In short, this resulted in proprietary threads interfaces that are not portable across implementations, yet certain applications, such as database engines, were making heavy use of these proprietary interfaces.

Fortunately, many of the threads extensions developed were general enough that they are easily supported on any UNIX system threads implementation. In late 1995, the Aspen Group formed a subgroup to standardize the interfaces and functionality of the common thread extensions that various UNIX system vendors had implemented. The threads extensions that came out of this work by the Aspen Group comprise extensions that were made for OSF DCE 1.0 as well as others by Sun, HP, and Digital. The Aspen Group handed the completed work over to X/Open in 1996 as input to the next issue of XSH.

The following extensions to POSIX.1c were agreed by the Aspen Group:

- extended mutex attribute types
- read-write locks and attributes
- thread concurrency level

- thread stack guard size
- parallel I/O.

A total of 19 new functions were specified.

The Aspen Group carefully followed the threads programming model specified in POSIX.1c when developing these extensions. As with POSIX.1c all the new functions return zero if successful, otherwise an error number is returned to indicate the error.

The concept of attribute objects was introduced in POSIX.1c to allow implementations to extend the standard without changing the existing interfaces. Attribute objects were defined for threads, mutexes, and condition variables. Attributes objects are defined as implementation-dependent opaque types to aid extensibility, and functions are defined to allow attributes to be set or retrieved. The Aspen Group followed this model when adding the new type attribute of **pthread_mutexattr_t** or the new read-write lock attributes object **pthread_rwlockattr_t**.

11.3 Extended Mutex Attributes

POSIX.1c defines a mutex attributes object as an implementation-dependent opaque object of type **pthread_mutexattr_t**, and specifies a number of attributes which this object must have and a number of functions which manipulate these attributes. These attributes include *detachstate*, *inheritsched*, *schedparm*, *schedpolicy*, *contentionscope*, *stackaddr*, and *stacksize*.

XSH, Issue 5 specifies another mutex attribute called *type*. The *type* attribute allows applications to specify the behavior of mutex locking operations in situations where the POSIX.1c behavior is undefined. The OSF DCE threads implementation, based on Draft 4 of POSIX.1c, specified a similar attribute. Note that the names of the attributes have changed somewhat from the OSF DCE threads implementation.

XSH, Issue 5 also extends the specification of the following POSIX.1c functions which manipulate mutexes:

pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )

to take account of the new mutex attribute type and to specify behavior which was declared as undefined in POSIX.1c. How a calling thread acquires or releases a mutex now depends upon the mutex *type* attribute.

The *type* attribute can have the following values:

Value	Definition
PTHREAD_MUTEX_NORMAL	Basic mutex with no specific error checking built in. Does not report a deadlock error.
PTHREAD_MUTEX_RECURSIVE	Allows any thread to recursively lock a mutex. The mutex must be unlocked an equal number of times to release the mutex.
PTHREAD_MUTEX_ERRORCHECK	Detects and reports simple usage errors; that is, an attempt to unlock a mutex that is not locked by the calling thread or that is not locked at all, or an attempt to relock a mutex the thread

Value	Definition
PTHREAD_MUTEX_DEFAULT	already owns. The default mutex type. May be mapped to any of the above mutex types or may be an implementation-dependent type.

Normal mutexes do not detect deadlock conditions; for example, a thread will hang if it tries to relock a normal mutex that it already owns. Attempting to unlock a mutex locked by another thread, or unlocking an unlocked mutex, results in undefined behavior. Normal mutexes will usually be the fastest type of mutex available on a platform but provide the least error checking.

Recursive mutexes are useful for converting old code where it is difficult to establish clear boundaries of synchronization. A thread can relock a recursive mutex without first unlocking it. The relocking deadlock which can occur with normal mutexes cannot occur with this type of mutex. However, multiple locks of a recursive mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. Furthermore, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock a recursive mutex which another thread has locked returns with an error. A thread attempting to unlock a recursive mutex that is not locked shall return with an error. Never use a recursive mutex with condition variables because the implicit unlock performed by *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) will not actually release the mutex if it had been locked multiple times.

Errorcheck mutexes provide error checking and are useful primarily as a debugging aid. A thread attempting to relock an errorcheck mutex without first unlocking it returns with an error. Again, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock an errorcheck mutex which another thread has locked returns with an error. A thread attempting to unlock an errorcheck mutex that is not locked also returns with an error. It should be noted that errorcheck mutexes will almost always be much slower than normal mutexes due to the extra state checks performed.

The *default* mutex type provides implementation-dependent error checking. The default mutex may be mapped to one of the other defined types or may be something entirely different. This enables each vendor to provide the mutex semantics which the vendor feels will be most useful to their target users. Most vendors will probably choose to make normal mutexes the default so as to give applications the benefit of the fastest type of mutexes available on their platform. Check your implementation's documentation.

An application developer can use any of the mutex types almost interchangeably as long as the application does not depend upon the implementation detecting (or failing to detect) any particular errors. Note that a recursive mutex can be used with condition variable waits as long as the application never recursively locks the mutex.

Two functions are provided in XSH, Issue 5 for manipulating the *type* attribute of a mutex attributes object. This attribute is set or returned in the *type* parameter of these functions. The *pthread_mutexattr_settype*( ) function is used to set a specific type value while *pthread_mutexattr_gettype*( ) is used to return the type of the mutex. Setting the *type* attribute of a mutex attributes object affects only mutexes initialized using that mutex attributes object. Changing the *type* attribute does not affect mutexes previously initialized using that mutex attributes object.

11.4 Read-Write Locks and Attributes

Read-write locks (also known as readers-writer locks) allow a thread to exclusively lock some shared data while updating that data, or allow any number of threads to have simultaneous read-only access to the data.

Unlike a mutex, a read-write lock distinguishes between reading data and writing data. A mutex excludes all other threads. A read-write lock allows other threads access to the data, providing no thread is modifying the data. Thus, a read-write lock is less primitive than either a mutex-condition variable pair or a semaphore.

Application developers should consider using a read-write lock rather than a mutex to protect data that is frequently referenced but seldom modified. Most threads (readers) will be able to read the data without waiting and will only have to block when some other thread (a writer) is in the process of modifying the data. Conversely a thread that wants to change the data is forced to wait until there are no readers. This type of lock is often used to facilitate parallel access to data on multiprocessor platforms or to avoid context switches on single processor platforms where multiple threads access the same data.

If a read-write lock becomes unlocked and there are multiple threads waiting to acquire the write lock, the implementation's scheduling policy determines which thread shall acquire the read-write lock for writing. If there are multiple threads blocked on a read-write lock for both read locks and write locks, it is unspecified whether the readers or a writer acquire the lock first. However, for performance reasons, implementations often favor writers over readers to avoid potential writer starvation.

A read-write lock object is an implementation-dependent opaque object of type **pthread_rwlock_t** as defined in **<pthread.h>**. There are two different sorts of locks associated with a read-write lock — a *read lock* and a *write lock*.

The *pthread_rwlockattr_init*( ) function initializes a read-write lock attributes object with the default value for all the attributes defined in the implementation. After a read-write lock attributes object has been used to initialize one or more read-write locks, changes to the read-write lock attributes object, including destruction, do not affect previously initialized read-write locks.

Implementations must provide at least the read-write lock attribute process-shared. This attribute can have the following values:

Value	Definition
PTHREAD_PROCESS_SHARED	Any thread of any process that has access to the memory where the read-write lock resides can manipulate the read-write lock.
PTHREAD_PROCESS_PRIVATE	Only threads created within the same process as the thread that initialized the read-write lock can manipulate the read-write lock. This is the default value.

The *pthread_rwlockattr_setpshared*( ) function is used to set the process-shared attribute of an initialized read-write lock attributes object while the function *pthread_rwlockattr_getpshared*( ) obtains the current value of the process-shared attribute.

A read-write lock attributes object is destroyed using the *pthread_rwlockattr_destroy*( ) function. The effect of subsequent use of the read-write lock attributes object is undefined.

A thread creates a read-write lock using the *pthread_rwlock_init*( ) function. The attributes of the read-write lock can be specified by the application developer, otherwise the default

implementation-dependent read-write lock attributes are used if the pointer to the read-write lock attributes object is NULL. In cases where the default attributes are appropriate, the PTHREAD_RWLOCK_INITIALIZER macro can be used to initialize statically allocated read-write locks.

A thread which wants to apply a read lock to the read-write lock can use either *pthread_rwlock_rdlock*( ) or *pthread_rwlock_tryrdlock*( ). If *pthread_rwlock_rdlock*( ) is used, the thread acquires a read lock if a writer does not hold the write lock and there are no writers blocked on the write lock. If a read lock is not acquired, the calling thread blocks until it can acquire a lock. However, if *pthread_rwlock_tryrdlock*( ) is used, the function returns immediately with the error EBUSY if any thread holds a write lock or there are blocked writers waiting for the write lock.

A thread which wants to apply a write lock to the read-write lock can use either of two functions: *pthread_rwlock_wrlock*( ) or *pthread_rwlock_trywrlock*( ). If *pthread_rwlock_wrlock*( ) is used, the thread acquires the write lock if no other reader or writer threads hold the read-write lock. If the write lock is not acquired, the thread blocks until it can acquire the write lock. However, if *pthread_rwlock_trywrlock*( ) is used, the function returns immediately with the error EBUSY if any thread is holding either a read or a write lock.

The *pthread_rwlock_unlock*( ) function is used to unlock a read-write lock object held by the calling thread. Results are undefined if the read-write lock is not held by the calling thread. If there are other read locks currently held on the read-write lock object, the read-write lock object shall remain in the read locked state but without the current thread as one of its owners. If this function releases the last read lock for this read-write lock object, the read-write lock object shall be put in the unlocked read state. If this function is called to release a write lock for this read-write lock object, the read-write lock object shall be put in the unlocked state.

The same POSIX working group which developed POSIX.1b and POSIX.1c is currently developing IEEE PASC P1003.1j draft standard, which specifies a set of extensions for realtime and threaded programming. This includes readers-writer locks which are nearly identical to the XSH, Issue 5 read-write locks. The Aspen Group was aware of this draft standard, but felt that there was an immediate and urgent need for standardization in the area of read-write locks.

The following table maps the XSH, Issue 5 read-write lock functions to their equivalent IEEE PASC P1003.1j draft 5 functions:

XSH, Issue 5	IEEE PASC P1003.1j
pthread_rwlock_init( )	*rwlock_init*( )
pthread_rwlock_destroy( )	*rwlock_destroy*( )
pthread_rwlock_rdlock( )	*rwlock_rlock*( )
pthread_rwlock_tryrdlock( )	*rwlock_tryrlock*( )
pthread_rwlock_wrlock( )	*rwlock_wlock*( )
pthread_rwlock_trywrlock( )	*rwlock_trywlock*( )
pthread_rwlock_unlock( )	*rwlock_unlock*( )
pthread_rwlockattr_init( )	*rwlock_attr_init*( )
pthread_rwlockattr_destroy( )	*rwlock_attr_destroy*( )
pthread_rwlockattr_setpshared( )	*rwlock_attr_setpshared*( )
pthread_rwlockattr_getpshared( )	*rwlock_attr_getpshared*( )

The Aspen Group chose function names which are different from those used in the IEEE PASC P1003.1j draft standard to avoid name space conflicts with those interfaces. Note that draft 5 requires the header **<semaphore.h>** while XSH, Issue 5 requires the **<pthread.h>** header. However, it is hoped that the final POSIX.1j standard will adopt the Aspen functions names and headers instead of the current ones.

11.5 Thread Concurrency Level

On threads implementations that multiplex user threads onto a smaller set of kernel execution entities, the system attempts to create a reasonable number of kernel execution entities for the application upon application startup.

On some implementations, these kernel entities are retained by user threads that block in the kernel. Other implementations do not *timeslice* user threads so that multiple compute-bound user threads can share a kernel thread. On such implementations, some applications may use up all the available kernel execution entities before its user-space threads are used up. The process may be left with user threads capable of doing work for the application but with no way to schedule them.

The *pthread_setconcurrency*( ) function enables an application to request more kernel entities; that is, specify a desired concurrency level. However, this function merely provides a hint to the implementation. The implementation is free to ignore this request or to provide some other number of kernel entities. If an implementation does not multiplex user threads onto a smaller number of kernel execution entities, the *pthread_setconcurrency*( ) function has no effect.

The *pthread_setconcurrency*( ) function may also have an effect on implementations where the kernel mode and user mode schedulers cooperate to ensure that ready user threads are not prevented from running by other threads blocked in the kernel.

The *pthread_getconcurrency*( ) function always returns the value set by a previous call to *pthread_setconcurrency*( ). However, if *pthread_setconcurrency*( ) was not previously called, this function shall return zero to indicate that the threads implementation is maintaining the concurrency level.

11.6 Thread Stack Guard Size

DCE threads introduced the concept of a *thread stack guard size*. Most thread implementations add a region of protected memory to a thread's stack, commonly known as a *guard region*, as a safety measure to prevent stack pointer overflow in one thread from corrupting the contents of another thread's stack. The default size of the guard regions attribute is PAGESIZE bytes and is implementation-dependent.

Some application developers may wish to change the stack guard size. When an application creates a large number of threads, the extra page allocated for each stack may strain system resources. In addition to the extra page of memory, the kernel's memory manager has to keep track of the different protections on adjoining pages. When this is a problem, the application developer may request a guard size of 0 bytes to conserve system resources by eliminating stack overflow protection.

Conversely an application that allocates large data structures such as arrays on the stack may wish to increase the default guard size in order to detect stack overflow. If a thread allocates two pages for a data array, a single guard page provides little protection against thread stack overflows since the thread can corrupt adjoining memory beyond the guard page.

XSH, Issue 5 defines a new attribute of a thread attributes object; that is, the *guardsize* attribute which allows applications to specify the size of the guard region of a thread's stack.

Two functions are provided for manipulating a thread's stack guard size. The *pthread_attr_setguardsize*( ) function sets the thread *guardsize* attribute, and the *pthread_attr_getguardsize*( ) function retrieves the current value.

An implementation may round up the requested guard size to a multiple of the configurable system variable PAGESIZE. In this case, *pthread_attr_getguardsize*( ) returns the guard size specified by the previous *pthread_attr_setguardsize*( ) function call and not the rounded up value.

If an application is managing its own thread stacks using the *stackaddr* attribute, the *guardsize* attribute is ignored and no stack overflow protection is provided. In this case, it is the responsibility of the application to manage stack overflow along with stack allocation.

11.7 Parallel I/O

Many I/O intensive applications, such as database engines, attempt to improve performance through the use of parallel I/O. However, POSIX.1 does not support parallel I/O very well because the current offset of a file is an attribute of the file descriptor.

Suppose two or more threads independently issue read requests on the same file. To read specific data from a file, a thread must first call *lseek*( ) to seek to the proper offset in the file, and then call *read*( ) to retrieve the required data. If more than one thread does this at the same time, the first thread may complete its seek call, but before it gets a chance to issue its read call a second thread may complete its seek call, resulting in the first thread accessing incorrect data when it issues its read call. One workaround is to lock the file descriptor while seeking and reading or writing, but this reduces parallelism and adds overhead.

Instead, XSH, Issue 5 provides two functions to make seek/read and seek/write operations atomic. The file descriptor's current offset is unchanged, thus allowing multiple read and write operations to proceed in parallel. This improves the I/O performance of threaded applications. The *pread*( ) function is used to do an atomic read of data from a file into a buffer. Conversely, the *pwrite*( ) function does an atomic write of data from a buffer to a file.

11.8 Functional Overview

The **<pthread.h>** header defines the following new types:

- **pthread_rwlock_t**

  Read-write lock object.

- **pthread_rwlockattr_t**

  Read-write lock attributes object.

The **<pthread.h>** header defines the following new macros:

- PTHREAD_RWLOCK_INITIALIZER

  Statically initialize a read-write lock object.

All of the following functions have their prototypes defined in **<pthread.h>**:

- *pthread_mutexattr_gettype*( )

  Get the value of the *type* attribute of the specified mutex attribute object *attr*.

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr,
    int *type);
```

- *pthread_mutexattr_settype*()

  Set the value of the *type* attribute of the specified mutex attribute object *attr*.

  ```
  int pthread_mutexattr_settype(pthread_mutexattr_t *attr,
      int *type);
  ```

- *pthread_rwlock_init*()

  Initialize the read-write lock object *rwlock*.

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

- *pthread_rwlock_rdlock*()

  Lock the read-write lock object *rwlock* for reading.

  ```
  int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
  ```

- *pthread_rwlock_tryrdlock*()

  Lock the read-write lock object *rwlock* for reading unless there is an existing write lock or blocked writers.

  ```
  int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
  ```

- *pthread_rwlock_wrlock*()

  Lock the read-write lock object *rwlock* for writing. Block, if necessary, until the write lock becomes available.

  ```
  int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
  ```

- *pthread_rwlock_trywrlock*()

  Lock the read-write lock object *rwlock* for writing unless there are any existing read or write locks.

  ```
  int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
  ```

- *pthread_rwlock_unlock*()

  Unlock the read-write lock object *rwlock*.

  ```
  int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
  ```

- *pthread_rwlock_destroy*()

  Destroy the read-write lock object *rwlock*.

  ```
  int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
  ```

- *pthread_rwlockattr_init*()

  Initialize the read-write lock attributes object *rwlockattr*.

  ```
  int pthread_rwlockattr_init(pthread_rwlock_t *rwlockattr);
  ```

- *pthread_rwlockattr_getpshared*()

  Get the value of the process-shared attribute of the read-write lock attributes object *rwlockattr*.

  ```
  int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t
      *rwlockattr, int *pshared);
  ```

- *pthread_rwlockattr_setpshared*()

  Set the value of the process-shared attribute of the read-write lock attributes object *rwlockattr*.

  ```
  int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *rwlockattr,
      int *pshared);
  ```

- *thread_rwlockattr_destroy*()

  Destroy the read-write lock attributes object *rwlockattr*.

  ```
  int pthread_rwlockattr_destroy(pthread_rwlock_t *rwlockattr);
  ```

- *pthread_getconcurrency*()

  Get the level of thread concurrency.

  ```
  int pthread_getconcurrency(void);
  ```

- *pthread_setconcurrency*()

  Set the level of thread concurrency.

  ```
  int pthread_setconcurrency(int new_level);
  ```

- *pthread_attr_getguardsize*()

  Get the value of the *guardsize* attribute of the thread attributes object *attr*.

  ```
  int pthread_attr_getguardsize(const pthread_attr_t *attr,
      size_t *guardsize);
  ```

- *pthread_attr_setguardsize*()

  Set the value of the *guardsize* attribute of the thread attributes object *attr*.

  ```
  int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
  ```

- *pread*()

  Read *nbyte* bytes from offset *offset* in the file opened on file descriptor *filedes*.

  ```
  size_t pread(int filedes, void *buf, size_t nbyte, off_t offset);
  ```

- *pwrite*()

  Write *nbyte* bytes from offset *offset* in the file opened on file descriptor *filedes*.

  ```
  size_t pwrite(int filedes, void *buf, size_t nbyte, off_t offset);
  ```