Chapter 13

Thread−Level Parallelism

IRIX 6.5 conforms to ISO/IEC 9945−1:1996 and UNIX 98; that is, it supports POSIX threads, or pthreads.

This chapter contains the following main topics:

"Overview of POSIX Threads" summarizes the similarities and differences of pthreads and processes.

"Compiling and Debugging a Pthread Application" covers compiling and debugging tools.

"Creating Pthreads" covers the process of creating a pthread with the desired attributes.

"Executing and Terminating Pthreads" discusses how threads initialize themselves and how you synchronize on thread termination.

"Using Thread−Unique Data"tells how to define variables that have a unique value in each thread.

"Pthreads and Signals" discusses the pthread−specific details of signal handling (see"Signals" for the general information).

"Scheduling Pthreads" covers scheduling priorities and policies.

"Synchronizing Pthreads" details the use of mutexes and condition variables.

Overview of POSIX Threads

A *thread* is an independent execution state; that is, a set of machine registers, a call stack, and the ability to execute code. When IRIX creates a process, it also creates one thread to execute that process. However, you can write a program that creates many more threads to execute in the same address space. For a comparison of pthreads to processes, see "Thread−Level Parallelism".

POSIX threads are similar in some ways to IRIX lightweight processes made with *sproc()*. You use pthreads in preference to lightweight processes for two main reasons: portability and performance. A program based on pthreads is normally easier to port from another vendor's equipment than a program that depends on a unique facility such as *sproc()*. Table 13−1summarizes some of the differences between pthreads and lightweight processes.

Table 13−1 Comparison of Pthreads and Processes

Attribute	POSIX Threads	Lightweight Processes	UNIX Processes
Source portability	Standard interface, portable between vendors	*sproc()* is unique to IRIX	*fork()* is a UNIX standard
Creation overhead	Relatively small	Moderately large	Quite large
Block/Unblock (Dispatch) Overhead	Few microseconds	Many microseconds	Many microseconds
Address space	Shared	Shared, or copy on write, or separate	Separate
Memory−mapped files and arenas	Shared	Shared, or copy on write, or separate	Explicit sharing only
Mutual exclusion objects	Mutexes, condition variables, and read−write	IRIX semaphores and locks; POSIX	IRIX semaphores and locks; POSIX

	locks; POSIX semaphores; IRIX semaphores and locks	semaphores	semaphores
Files, pipes, and I/O streams	Shared single−process file table	Shared or separate file table	Separate file table
Signal masks and signal handlers	Each thread has a mask but handlers are shared	Each process has a mask and its own handlers	Each process has a mask and its own handlers
Resource limits	Single−process limits	Single−process limits	Limits apply to each process separately
Process ID	One PID applies to all threads	PID per process plus share−group PID	PID per process

It takes relatively little time to create or destroy a pthread, as compared to creating a lightweight process. Threads share all resources and attributes of a single process (except for the signal mask; see "Pthreads and Signals"). If you want each executing entity to have its own set of file descriptors, or if you want to make sure that one entity cannot modify data shared with another entity, you must use lightweight processes or normal processes.

Compiling and Debugging a Pthread Application

A pthread application is a C or a C++ program that uses some of the POSIX pthreads functions. In order to use these functions, and in order to access the thread−safe versions of the standard I/O macros, you must include the proper header files and link with the pthreads library. You can debug and analyze the compiled program using some of the tools available for IRIX.

Compiling Pthread Source

The header files related to pthreads functions are summarized in Table 13−2

Table 13−2 Header Files Related to Pthreads

Header	Primary Contents
errno.h	System error codes returned by pthreads functions.
pthread.h	Pthread functions and special pthread data types.
sched.h	The *sched_param* structure and related functions used in setting thread priorities.
stdio.h	Standard stream I/O macros, including thread−safe versions.
sys/types.h	IRIX and standard data types.
limits.h	Some POSIX constants such as _POSIX_THREAD_THREADS_MAX.
unistd.h	Constants used when calling *sysconf()* to query POSIX limits (see the sysconf(3) reference page).

It is recommended that the thread−safe options be enabled at compile time using the feature test macro, _POSIX_C_SOURCE (see intro(3) for details). For example, to compile these options, use this command:

```
cc −D_POSIX_C_SOURCE=199506L app.c −llib0 −llib1 ... −lpthread
```

You can use pthreads with a program compiled to any of the supported execution models: *−32*for compatibility with older systems, *−n32*for 64−bit data and 32−bit addressing, or*−64*for 64−bit addressing.

The pthreads functions are defined in the library *libpthread.so*. Link with this library using the *−lpthread*compiler option, which should be the last library on the command line. The compiler chooses the correct library based on the execution model: */usr/lib/libpthread.so*,

/usr/lib32/libpthread.so, and */usr/lib64/libpthread.so.*

Note: A pthread program is a program that links with *libpthread*. Do not link with *libpthread* unless you intend to use the pthread interface, because *libpthread* replaces many standard library functions.

Debugging Pthread Programs

The *dbx* debugger and Workshop Debugger have been extended for use with threaded programs. See the dbx(1M) reference page and the documentation for Workshop Debugger for more details.

Creating Pthreads

You create a pthread by calling *pthread_create()*. One argument to this function is a thread attribute object of type *pthread_attr_t*. You pass a null address to request a thread having default attributes, or you prepare an attribute object to reflect the features you want the thread to have. You can use one attribute object to create many pthreads.

Functions related to attribute objects and pthread creation are summarized in Table 13−3and described in the following sections:

"Initial Detach State"

"Initial Scheduling Scope, Priority, and Policy"

"Thread Stack Allocation"

Table 13−3 Functions for Creating Pthreads

Function	Purpose
pthread_attr_init(3P)	Initialize a *pthread_attr_t* object to default settings.
pthread_attr_setdetachstate(3P)	Set the automatic−detach attribute.
pthread_attr_setinheritsched(3P)	Specify whether scheduling attributes come from the attribute object or are inherited from the creating thread.
pthread_attr_setschedparam(3P)	Set the starting thread priority.
pthread_attr_setschedpolicy(3P)	Set the scheduling policy.
pthread_attr_setscope(3P)	Set the scheduling scope.
pthread_attr_setstacksize(3P)	Set the stack size attribute.
pthread_attr_setguardsize(3P)	Set the stack guard size attribute.
pthread_attr_setstackaddr(3P)	Set the address of memory to use as a stack (when you allocate the stack for the new thread).
pthread_attr_destroy(3P)	Uninitialize a *pthread_attr_t* object.
pthread_create(3P)	Create a new thread based on an attribute object, or with default attributes.

Initial Detach State

Detaching means that the pthreads library frees up resources held by the thread after it terminates (see "Joining and Detaching"). There are three ways to detach a thread:

automatically when the thread terminates

explicitly by calling *pthread_join()*

explicitly by calling *pthread_detach()*

You can use *pthread_attr_setdetachstate()* to specify that a thread should be detached automatically when it terminates. Do this when you know that the thread will not be joined or detached by an explicit function call.

Initial Scheduling Scope, Priority, and Policy

You can specify an initial thread scheduling scope by calling *pthread_attr_setscope()* and passing one of the scope constants (PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS) in the *pthread_attr_t* object. By default, process scope is selected and scheduling is performed by the thread runtime, but thread scheduling by the kernel is provided with the system scope attribute. System scope threads run at real−time policy and priority and may be created only by privileged users.

You can specify an initial thread priority in a *struct sched_param* object in memory (the structure is declared in *sched.h*). Set the desired priority in the *sched_priority* field. Pass the structure to *pthread_attr_setschedparam()*.

You can specify an initial scheduling policy by calling *pthread_attr_setschedpolicy()*, passing one of the policy constants SCHED_FIFO or SCHED_RR.

The *pthread_attr_setinheritsched()* function is used to specify, in the attribute object, whether a new thread's scheduling policy and priority should be taken from the attribute object, or whether they should be inherited from the thread that creates the new thread. When you set an attribute object for inheritance, the scheduling policy and priority in the attribute object are ignored.

Scheduling scope, priorities, and policies are described in "Scheduling Pthreads".

Thread Stack Allocation

Each pthread has an execution stack area in memory. By default, *pthread_create()* allocates stack space from dynamic memory, and automatically releases it when the thread terminates.

You use *pthread_attr_setstacksize()* to specify the size of this stack area. You cannot specify a stack size less than a minimum. A pthread process can find the minimum by calling *sysconf()* with _SC_THREAD_STACK_MIN (see the sysconf(3C) reference page).

Threads may overrun their stack area. By default, a thread's stack is created with guard protection, and extra memory is allocated at the overflow end of the stack as a buffer. If an application overflows into this buffer, an exception results (a SIGSEGV signal is delivered to the thread).

The *guardsize* attribute controls the size of the guard area for the created thread's stack and protects against overflow of the stack pointer. The *guardsize* attribute is set using *pthread_attr_setguardsize()*.

Note: Because thread stack space is taken from dynamic memory, the allocation is charged against the process virtual memory limit, not the process stack size limit as you might expect.

Executing and Terminating Pthreads

The functions for managing the progress of a thread are summarized in Table 13−4and described in the following sections:

"Getting the Thread ID"

"Initializing Static Data"

"Setting Event Handlers"

"Terminating a Thread"

"Joining and Detaching"

Table 13−4 Functions for Managing Thread Execution

Function	Purpose
pthread_atfork(3P)	Register functions to handle the event of a *fork()*.
pthread_cancel(3P)	Request cancellation of a specified thread.
pthread_cleanup_push(3P)	Register function to handle the event of thread termination.
pthread_cleanup_pop(3P)	Unregister and optionally call termination handler.
pthread_detach(3P)	Detach a terminated thread.
pthread_exit(3P)	Explicitly terminate the calling thread.
pthread_join(3P)	Wait for a thread to terminate and receive its return value.
pthread_once(3P)	Execute initialization function once only.
pthread_self(3P)	Return the calling thread's ID.
pthread_equal(3P)	Compare two thread IDs for equality.
pthread_setcancelstate(3P)	Permit or block cancellation of the calling thread.
pthread_setcanceltype(3P)	Specify deferred or asynchronous cancellation.
pthread_testcancel(3P)	Permit cancellation to take place, if it is pending.

Getting the Thread ID

Call *pthread_self()* to get the thread ID of the calling thread. A thread can use this thread ID when changing its own scheduling priority, for example (see "Scheduling Pthreads").

Initializing Static Data

Your program may use static data that should be initialized exactly once. The code can be entered by multiple threads, and might be entered concurrently. How can you ensure that only one thread will perform the initialization?

One answer is to create a variable of type *pthread_once_t*, statically initialized to the value PTHREAD_ONCE_INIT. Call *pthread_once(),* passing the addresses of the variable and of an initialization function. The pthreads library ensures that the initialization function is called only once, and that any other threads calling *pthread_once()* for this variable wait until the first thread completes the initialization function. See Example 13−1

Example 13−1 One−Time Initialization

```
pthread_once_t first_time_flag = PTHREAD_ONCE_INIT;
elaborate_struct_t uninitialized; /* thing to initialize */
void elaborate_initializer(void); /* function to do it */
int subroutine(...)
{
   ...
   pthread_once(&first_time_flag, elaborate_initializer);
   ...
}
```

Setting Event Handlers

A thread can establish functions that are called when it terminates and when the process forks.

Call *pthread_cleanup_push()* to register a function that is to be called in the event that the current thread terminates, either by exiting or by cancellation. Call *pthread_cleanup_pop()* to retract this registration and, optionally, to call the handler. These functions are often used in library code, with the push operation done on entry to the library and the pop done upon exit from the library. The push and pop operations are in fact implemented partly as macro code. For this reason, calls to them must be strictly balanced—a pop for each push—and each push/pop pair must appear in a single C lexical scope. A nonstructured jump such as a longjmp (see the setjmp(3) reference page) or goto can cause unexpected results.

Call *pthread_atfork()* to register three handlers related to a UNIX *fork()* call. The first handler executes just before the *fork()* takes place; the second executes just after the *fork()* in the parent process; the third executes just after the *fork()* in the child process.

The *fork()* operation creates a new process with a copy of the calling process's address space, including any locked mutexes or semaphores. Typically, the new process immediately calls *exec()* to replace the address space with a new program. When this is the case, there is no need for *pthread_atfork()* (see the exec(2) and fork(2) reference pages). However, if the new process continues to execute with the inherited address space, including perhaps calls to library code that uses pthreads, it may be necessary for the library code to reinitialize data in the address space of the child process. You can do this in the fork event handlers.

Terminating a Thread

A thread begins execution in the function that is named in the *pthread_create()* call. When it returns from that function, the thread terminates. A thread can terminate earlier by calling *pthread_exit()*. In either case, the thread returns a value of type *void**.

One thread can request early termination of another by calling *pthread_cancel()*, passing the thread ID of the target thread. A thread can protect itself against cancellation using two built−in switches:

> The *pthread_setcancelstate()* function lets you postpone cancellation indefinitely (PTHREAD_CANCEL_DISABLE) or permit cancellation (PTHREAD_CANCEL_ENABLE).
>
> The *pthread_setcanceltype()* function lets you decide when cancellation will take place, if it is allowed at all. Cancellation can happen whenever it is requested (PTHREAD_CANCEL_ASYNCHRONOUS) or only at defined points (PTHREAD_CANCEL_DEFERRED).

When you prevent cancellation by setting PTHREAD_CANCEL_DISABLE, a cancellation request is blocked but remains pending until the thread terminates or changes its cancellation state.

The initial cancellation state of a thread is PTHREAD_CANCEL_ENABLE and the type is PTHREAD_CANCEL_DEFERRED. In this state, a cancellation request is blocked until the thread calls a function that is a defined cancellation point. The functions that are cancellation points are listed in the pthread_setcanceltype(3P) reference page. A thread can explicitly permit cancellation by calling *pthread_testcancel()*.

Joining and Detaching

Sometimes you do not care when threads terminate—your program starts a set of threads, and they continue until the entire program terminates.

In other cases, threads are created and terminated as the program runs. One thread can wait for another to terminated by calling *pthread_join()*, specifying the thread ID. The function does not return until the specified thread terminates. The value the specified thread passed to *pthread_exit()* is returned. At this time, your program can release any resources that you associate with the thread, for example, stack space (see "Thread Stack Allocation").

The *pthread_join()* function also detaches the terminated thread. If your program does not use *pthread_join()*, you must arrange for terminated threads to be detached in some other way. One way is by specifying automatic detachment when the threads are created (see "Initial Detach State"). Another is to call *pthread_detach()* at any time after creating the thread, including after it has terminated.

If your program creates threads and lets them terminate, but does not detach them, resources will be used up and eventually an error will occur when trying to create a thread.

Using Thread–Unique Data

In some designs, especially modules of library code, you need to store data that is both

- unique to the calling thread
- persistent from one function call to another

Normally, the only data that is unique to a thread is the contents of its local variables on the stack, and these do not persist between calls. However, the pthreads library provides a way to create persistent, thread–unique data. The functions for this are summarized inTable 13–5.

Table 13–5 Functions for Thread–Unique Data

Function	Purpose
pthread_key_create(3P)	Create a key.
pthread_key_delete(3P)	Delete a key.
pthread_getspecific(3P)	Retrieve this thread's value for a key.
pthread_setspecific(3P)	Set this thread's value for a key.

Your program calls *pthread_key_create()* to define a new storage *key*. Once created, a key may be used by all threads to identify a unique key *value.*

Any thread can use *pthread_getspecific()* to retrieve that thread's unique value stored under a key. A thread can fetch only its own value, which is the value stored by this same thread using *pthread_setspecific()*. The initial stored value is NULL.

When you create a key, you can specify a destructor function that is called automatically when a thread terminates. The destructor is called while the key is valid and the key value for the terminating thread is not NULL. The destructor receives the thread's key value as its argument.

Pthreads and Signals

For a general overview of signal concepts and numbers, see "Signals" and the signal(5) reference page. IRIX supports three different signal facilities: BSD signals, SVR4 signals, and POSIX signals.

When you are writing a pthreads program, you should use only the POSIX signal facilities (see "POSIX Signal Facility").

Setting Signal Masks

Each thread has a signal mask that specifies the signals it is willing to receive (see "Signal Blocking and Signal Masks"). In a program that is linked with the pthreads library, this should be changed using *pthread_sigmask()*. Each thread inherits the signal mask of the thread that calls *pthread_create()*. Typically you set an initial mask in the first thread, so that it can be inherited by all other threads.

Note: In IRIX, you can use *sigprocmask()* instead of *pthread_sigmask()*, but it may not be portable to other systems.

When a signal is directed to a specific thread that is blocking the signal, the signal remains pending on the thread until that thread unblocks it. When a signal is directed to a process, it is delivered to the first thread that is not blocking that signal. If all threads block that signal, the signal remains pending on the process until some thread unblocks it or the process terminates.

A thread can find out which signals are pending by calling *sigpending()*. This function returns a mask showing the set of signals pending on the process as a whole or for the calling thread; that is, the signals that could be delivered to the calling thread if they were not blocked.

Setting Signal Actions

When a signal is delivered, some action is taken. You specify what that action should be using the *sigaction()* function. These actions are set on a process−wide basis,*not* individually for each thread. Although each thread has a private signal mask, signal actions are shared with all threads in the process. See "Signal Handling Policies" for details.

Receiving Signals Synchronously

You can design a program to receive signals in a synchronous manner instead of asynchronously. To do this, set a mask that blocks all the signals that are to be received synchronously. Then call one of the following three functions:

sigwait(3)	Suspend until one of a specified set of signals is generated, then return the signal number.
sigwaitinfo(3)	Like *sigwait()*, but returns additional information about the signal.
sigtimedwait(3)	Like *sigwaitinfo()*, but also returns after a specified time has elapsed if no signal is received.

Using these functions you can write a thread that treats signals as a stream of events to be processed. This is generally the safest program model, much easier to work with than the asynchronous model of signal delivery.

Scheduling Pthreads

The pthreads scheduling algorithm is controlled by three variables: a scope, policy, and priority for each thread. These variables are set initially when the thread is created (see "Initial Scheduling Scope, Priority, and Policy"), but policy and priority can be modified while the thread is running.

Contention Scope

The scheduling contention scope of a pthread (see pthread_attr_setscope(3P)) determines the set of threads that it competes against for resources.

System scope threads compete with all other threads on the system and can be created only by privileged users. These threads are used in programs when some form of guaranteed (that is, real−time) response is required. Their scheduling parameters directly affect how the system treats them. In addition to the usual scheduling attributes, they can select a CPU on which to run using the *pthread_setrunon_np()* call.

Process scope threads compete within the process and their scheduling attributes are used by the pthread library to select which threads to run on a pool of kernel entities. The size of the pool is determined dynamically, but may be influenced using the *pthread_setconcurrency()* call.

Process scope threads generally require fewer resources than system scope threads because they can share kernel resources. The kernel entities themselves share a common set of scheduling attributes which privileged users can change using the process scheduling interfaces (see sched_setscheduler(2) and sched_setparam(2)). For further details, see the pthreads(5) reference page.

The functions used in scheduling are summarized in Table 13−6and described in the following sections:

"Scheduling Policy"

"Scheduling Priority"

Table 13−6 Functions for Schedule Management

Function	Purpose
pthread_getschedparam(3P)	Get a thread's policy and priority.
pthread_setschedparam(3P)	Set a thread's policy and priority.
sched_get_priority_max(3C)	Return the maximum priority value.
sched_get_priority_min(3C)	Return the minimum priority value.
sched_yield(2)	Relinquish the processor.
pthread_setconcurrency(3P)	Modify concurrency level.
pthread_getconcurrency(3P)	Check the concurrency level.
pthread_setrunon_np(3P)	Select a CPU to run a system scope thread.
pthread_getrunon_np(3P)	Query a named CPU's affinity.

Scheduling Policy

There are two scheduling policies in this implementation: first−in−first−out (SCHED_FIFO) and the default round−robin (SCHED_RR). SCHED_FIFO and SCHED_RR are similar. The round−robin scheduler ensures that after a thread has used a certain maximum amount of time, it is moved to the end of the queue of threads of the same priority, and can be preempted by other threads.

The details of scheduling are discussed in the pthread_attr_setschedpolicy(3P) reference page.

Scheduling Priority

Threads are ordered by priority values, with a small number representing a low priority, and a larger number representing a higher priority. Threads with higher priorities are chosen to execute before threads with lower priorities.

The *sched_get_priority_max()* and *sched_get_priority_min()* functions return the highest and lowest priority numbers for a given policy. There are at least 32 priority values and the lowest is greater than or equal to 0.

A thread can set another's priority and scheduling policy, using *pthread_setschedparam()*. A simple function to set a specified priority on the current thread is shown in Example 13−2

Example 13−2 Function to Set Own Priority

```
#include <sched.h> /* struct sched_param */
void setMyPriority(int newP)
{
   pthread_t myTid = pthread_self();
   int policy;
   struct sched_param sp;
   (void) pthread_getschedparam(myTID,&policy,&sp);
   sp.sched_priority = newP;
   (void) pthread_setschedparam(myTID,policy,&sp);
}
```

Synchronizing Pthreads

Threads using a common address space must cooperate and coordinate their use of shared variables. IRIX provides many mechanisms for coordinating threads, including:

- POSIX semaphores for general coordination and resource management (see "POSIX Facilities for Mutual Exclusion").
- POSIX or SVR4 message queues (see Chapter 6, "Message Queues").
- POSIX mutex objects, which allow threads to gain exclusive use of a shared variable (see "Mutexes").
- POSIX condition variables, which allow a thread to wait when a controlling predicate is false (see "Condition Variables").
- POSIX read−write locks, which allow one thread exclusive access to locked data to write it or read access to locked data for several threads (see "Read−Write Locks").
- IRIX semaphores and locks.
- SVR4 semaphores.

Tip: Synchronization between processes (such as POSIX process−shared mechanisms, IRIX IPC, and SVR4 IPC) is more costly than synchronization between threads (POSIX process−private mechanisms). So where possible, use the process−private mechanisms.

Mutexes

A mutex is a software object that arbitrates the right to modify some shared variable, or the right to execute a critical section of code. A mutex can be owned by only one thread at a time; other threads trying to acquire it wait. Mutexes are intended to be lightweight and owned only for a short time.

Preparing Mutex Objects

When a thread wants to modify a variable that it shares with other threads, or execute a critical section, the thread claims the associated mutex. This can cause the thread to wait until it can acquire the mutex. When the thread has finished using the shared variable or critical code, it releases the mutex. If two or more threads claim the mutex at once, one acquires the mutex and continues, while the others are blocked until the mutex is released.

A mutex has attributes that control its behavior. The pthreads library contains several functions used to prepare a mutex for use. These functions are summarized in Table 13−7.

Table 13−7 Functions for Preparing Mutex Objects

Function	Purpose
pthread_mutexattr_init(3P)	Initialize a *pthread_mutexattr_t* with default attributes.
pthread_mutexattr_destroy(3P)	Uninitialize a *pthread_mutexattr_t*.
pthread_mutexattr_getprotocol(3P)	Query the priority protocol.
pthread_mutexattr_setprotocol(3P)	Set the priority protocol choice.
pthread_mutexattr_getprioceiling(3P)	Query the minimum priority.
pthread_mutexattr_setprioceiling(3P)	Set the minimum priority.
pthread_mutexattr_getpshared(3P)	Query the process−shared attribute.
pthread_mutexattr_setpshared(3P)	Set the process−shared attribute.
pthread_mutexattr_gettype(3P)	Get the mutex type.
pthread_mutexattr_settype(3P)	Set the mutex type.
pthread_mutex_init(3P)	Initialize a mutex object.
pthread_mutex_destroy(3P)	Uninitialize a mutex object.

A mutex must be initialized before use. You can do this in one of three ways:

Static assignment of the constant PTHREAD_MUTEX_INITIALIZER.

Calling *pthread_mutex_init()* passing NULL instead of the address of a mutex attribute object.

Calling *pthread_mutex_init()* passing a *pthread_mutexattr_t* object that you have set up with attribute values.

The first two methods initialize the mutex to default attributes.

Four attributes can be set in a *pthread_mutexattr_t*. You can set the priority inheritance protocol using *pthread_mutexattr_setprotocol()* to one of three values:

PTHREAD_PRIO_NONE — The mutex has no effect on the thread that acquires it. This is the default.

PTHREAD_PRIO_PROTECT — The thread holding the mutex runs at a priority at least as high as th highest priority of any mutex that it currently holds.

PTHREAD_PRIO_INHERIT — The thread holding the mutex runs at a priority at least as high as th highest priority of any thread blocked on that mutex.

If a thread acquires a mutex and then is suspended (for example, because its time slice is up), other threads can be blocked waiting for the mutex. The PTHREAD_PRIO_PROTECT protocol prevents this. Using *pthread_mutexattr_setprioceiling(),* you set a priority higher than normal for the mutex. A thread that acquires the mutex runs at this higher priority while it holds the mutex.

Another problem is that when a low−priority thread has acquired a mutex, and a thread with higher priority claims the mutex and is blocked, a "priority inversion" takes place—a higher−priority thread is forced to wait for one of lower priority. The PTHREAD_PRIO_INHERIT protocol prevents this—when a thread of higher priority blocks, the thread holding the mutex has its priority boosted during the time it holds the mutex.

Tip: PTHREAD_PRIO_NONE uses a faster code path than the other two priority options for mutexes.

By default, only threads within a process share a mutex. Using *pthread_mutexattr_setpshared()*, you can allow any thread (from any process) with access to the mutex memory location to use the mutex. Enable mutex sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Note: The PTHREAD_PRIO_INHERIT attribute is not available with *pthread_mutexattr_setpshared()*.

By default, no error checking is performed on threads that attempt to use a mutex. For example, a thread that attempts to lock a mutex that it already owns deadlocks. Using *pthread_mutexattr_settype()* with PTHREAD_MUTEX_ERRORCHECK allows you to have the lock call return an error instead. If recursive mutexes are required, PTHREAD_MUTEX_RECURSIVE enables recursive mutexes.

Using Mutexes

The functions for claiming, releasing, and using mutexes are summarized in Table 13−8.

Table 13−8 Functions for Using Mutexes

Function	Purpose
pthread_mutex_lock(3P)	Claim a mutex, blocking until it is available.
pthread_mutex_trylock(3P)	Test a mutex and acquire it if it is available, else return an error.
pthread_mutex_unlock(3P)	Release a mutex.
pthread_mutex_getprioceiling(3P)	Query the minimum priority of a mutex.
pthread_mutex_setprioceiling(3P)	Set the minimum priority of a mutex.

To determine where mutexes should be used, examine the memory variables and other objects (such as files) that can be accessed from multiple threads. Create a mutex for each set of shared objects that are used together. Ensure that the code acquires the proper mutex before it modifies the shared objects. You acquire a mutex by calling *pthread_mutex_lock()*, and release it with *pthread_mutex_unlock()*. When a thread must not be blocked, it can use *pthread_mutex_trylock()* to test the mutex and lock it only if it is available.

Condition Variables

A condition variable provides a way in which a thread can wait for an event (or condition) defined by the program, to be satisfied. Condition variables use mutexes to synchronize the wait and wakeup operations.

Preparing Condition Variables

Like mutexes and threads themselves, condition variables are supplied with a mechanism of attribute objects (*pthread_condattr_t* objects) and static and dynamic initializers. (Only the condition variable for the process−shared attribute can be initialized in this implementation.) The functions for initializing one are summarized in Table 13−9.

Table 13−9 Functions for Preparing Condition Variables

Function	Purpose
pthread_condattr_init(3P)	Initialize a *pthread_condattr_t* to default attributes.
pthread_condattr_destroy(3P)	Uninitialize a *pthread_condattr_t*.
pthread_condattr_getpshared(3P)	Get the process−shared attribute.
pthread_condattr_setpshared(3P)	Set the process−shared attribute.
pthread_cond_init(3P)	Initialize a condition variable based on an attribute object.
pthread_cond_destroy(3P)	Uninitialize a condition variable.

A condition variable must be initialized before use. You can do this in one of three ways:

Static assignment of the constant PTHREAD_COND_INITIALIZER.

Calling *pthread_cond_init()* passing NULL instead of the address of an attribute object.

Calling *pthread_cond_init()* passing a *pthread_condattr_t* object that you have set up with attribute values.

The first two methods initialize the variable to default attributes.

By default, only threads within a process share a condition variable. Using *pthread_condattr_setpshared()*, you can allow any thread (from any process) with access to the condition variable memory location to use the condition variable. Enable condition variable sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Using Condition Variables

A condition variable is a software object that represents a test of a Boolean condition. Typically the condition changes because of a software event such as "other thread has supplied data." A thread establishes that it needs to wait by first evaluating the condition. The thread that satisfies the condition signals the condition variable, releasing one or all threads that are waiting.

For example, a thread might acquire a mutex that represents a shared resource. While holding the mutex, the thread finds that the shared resource is not complete. The thread does three things:

Wait, giving up the mutex so that some other thread can renew the shared resource.

Wait until the condition is signalled.

Wake−up, re−acquiring the mutex for the shared resource and rechecking the condition.

These three actions are combined into one using a condition variable. The functions used with condition variables are summarized in Table 13−10.

Table 13−10 Functions for Using Condition Variables

Function	Purpose
pthread_cond_wait(3P)	Wait on a condition variable.
pthread_cond_timedwait(3P)	Wait on a condition variable, returning with an error after a time limit expires.
pthread_cond_signal(3P)	Signal that an awaited event has occurred, releasing at least one waiting thread.
pthread_cond_broadcast(3P)	Signal that an awaited event has occurred, releasing all waiting threads.

The *pthread_cond_wait()* and *pthread_cond_timedwait()* functions require both a condition variable and a mutex that is owned by the calling thread. The mutex is released and the wait begins. When the event is signalled (or the time limit expires), the mutex is reacquired, as if by a call to *pthread_mutex_lock()*.

The POSIX standard explicitly warns that it is possible in some cases for a conditional wait to return before the event has been signalled. For this reason, a conditional wait should always be coded in a loop that tests the shared resource for the needed status. These principles are suggested in the code in Example 13−3 which is modeled after an example in the POSIX 1003.1c standard.

Example 13−3 Use of Condition Variables

```
#include <assert.h>
#include <pthread.h>
typedef int listKey_t;
typedef struct element_s { /* list element */
   listKey_t key;
   struct element_s *next;
   int busyFlag;
   pthread_cond_t notBusy; /* event of no−longer−in−use */
} element_t;
typedef struct listHead_s { /* list head and mutex */
   pthread_mutex_t mutList; /* right to modify the list */
   element_t *head;
} listHead_t;
/*
|| Internal function to find an element in a list, returning NULL
|| if the key is not in the list.
|| A returned element could be in use by another thread (busy).
|| The caller is assumed to hold the list mutex, otherwise
|| the returned value could be made invalid at any time.
*/
static element_t *scanList(listHead_t* lp, listKey_t key)
{
   element_t *ep;
   for (ep=lp->head; (ep) ; ep=ep->next)
   {
      if (ep->key == key) break;
   }
   return ep;
}
/*
|| Public function to find a key in a list, wait until the element
|| is no longer busy, mark it busy, and return it.
*/
element_t *getFromList(listHead_t* lp, listKey_t key)
{
   element_t *ep;
```

```
   pthread_mutex_lock(&lp->mutList); /* lock list against changes */
   while ((ep=scanList(lp,key)) && (ep->busyFlag))
   {
      pthread_cond_wait(&ep->notBusy, &lp->mutList); /* (A) */
   }
   if (ep) ep->busyFlag = 1;
   pthread_mutex_unlock(&lp->mutList);
   return ep;
}
/*
|| Public function to release an element returned by getFromList().
*/
void freeInList(listHead_t* lp, element_t *ep)
{
   assert(ep->busyFlag);
   pthread_mutex_lock(&lp->mutList); /* lock list to prevent races *
/
   ep->busyFlag = 0;
   pthread_cond_signal(&ep->notBusy);
   pthread_mutex_unlock(&lp->mutList);
}
/*
|| Public function to delete a list element returned by getFromList(
).
*/
void deleteInList(listHead_t* lp, element_t *ep)
{
   element_t **epp;
   assert(ep->busyFlag);
   pthread_mutex_lock(&lp->mutList);
   for (epp = &lp->head; ep != *epp; epp = &((*epp)->next))
   { /* finding anchor of *ep in list */ }
   *epp = ep->next; /* remove *ep from list */
   ep->busyFlag = 0;
   pthread_cond_broadcast(&ep->notBusy);
   pthread_mutex_unlock(&lp->mutList);
   pthread_cond_destroy(&ep->notBusy);
   free(ep);
}
```

The functions in Example 13−3implement part of a simple library for managing lists. In a list head, *mutList* is a mutex object that represents the right to modify any part of the list. The elements of a list can be "busy," that is, in use by some thread. An element that is busy has a nonzero *busyFlag* field.

The *getFromList()* function looks up an element in a specified list, makes that element busy, and returns it. The function begins by acquiring the list mutex. This ensures that the list cannot change while the function is searching the list, and makes it legitimate for the function to change the busy flag in an element.

When it finds the element, the function might discover that the element is already busy. In this case, it must wait for the event "element is no longer busy," which is represented by the condition variable *notBusy* in the element. In order to wait for this event, *getFromList()* calls *pthread_cond_wait()* passing its list mutex and the condition variable (point "(A)" in the code). This releases the list mutex so that other threads can acquire the list and do their work on other elements.

When any thread wants to release the use of a list element, it calls *freeInList().* After clearing the busy flag in the list element, *freeInList()* announces that the event "element is no longer busy" has occurred, by calling *pthread_cond_signal()*.

This call releases a thread that is waiting at point "(A)." If there is more than one thread waiting for the same element, the first in priority order is released. The released thread re−acquires the list mutex and resumes execution. The first thing it does is repeat its search of the list for the desired key and, on finding the element again, test it again for busyness. This repetition is needed because it is possible to get spurious returns from a condition variable.

When a thread wants to delete a list element, it gets the list element by calling *getFromList()*. This ensures that the element is busy, so no other thread is using it. Then the thread calls *deleteInList()*. This function changes the list, so it begins by acquiring the list mutex. Then it can safely modify the list pointers. It scans up the list looking for the pointer that points to the target element. It removes the target element from the list by copying its *next* field to replace the pointer to the target element.

With the element removed from the list, *deleteInList()* calls *pthread_cond_broadcast()* to wake up all threads—not just the first thread—that might be waiting for the element to become nonbusy. Each of these threads resumes execution at point "(A)" by attempting to re−acquire the list mutex. However, *deleteInList()* is still holding the list mutex. The mutex is released; then the other threads can resume execution following point "(A)," but this time when they search the list, the desired key is no longer found.

Meanwhile, *deleteInList()* uses *pthread_cond_destroy()* to release any memory that the pthreads library might have associated with the condition variable, before releasing the list element object itself.

Read−Write Locks

A read−write lock is a software object that gives one thread the right to modify some data, or multiple threads the right to read that data. A read−write lock can be owned for write or for read. If acquired for write, only one thread can own it and other threads must wait. If acquired for read, other threads wishing to acquire it for write must wait, but multiple readers can own the lock at the same time.

Preparing Read−Write Locks

When a thread wants to modify or read data shared by several threads, the thread claims the associated lock. This can cause the thread to wait until it can acquire the lock. When the thread has finished reading or writing the shared data, it releases the lock.

A read−write lock has attributes that control its behavior. The pthreads library contains several functions used to prepare a lock for use. These functions are summarized in Table 13−11.

Table 13−11 Functions for Preparing Read−Write Locks

Function	Purpose
pthread_rwlockattr_init(3P)	Initialize a *pthread_rwlockattr_t* with default

	attributes.
pthread_rwlockattr_destroy(3P)	Uninitialize a *pthread_rwlockattr_t*.
pthread_rwlockattr_getpshared(3P)	Query the process−shared attribute.
pthread_rwlockattr_setpshared(3P)	Set the process−shared attribute.
pthread_rwlock_init(3P)	Initialize a rwlock object based on a *pthread_rwlockattr_t*.
pthread_rwlock_destroy(3P)	Uninitialize a read−write lock object.

A read−write lock must be initialized before use. You can do this in one of three ways:

Static assignment of the constant PTHREAD_RWLOCK_INITIALIZER.

Calling *pthread_rwlock_init()* passing NULL instead of the address of a read−write lock attribute object.

Calling *pthread_rwlock_init()* passing a *pthread_rwlockattr_t* object that you have set up with attribute values.

The first two methods initialize the read−write lock to default attributes.

By default, only threads within a process share a read−write lock. Using *pthread_rwlockattr_setpshared()*, you can allow any thread (from any process) with access to the read−write lock memory location to claim the read−write lock. Enable read−write lock sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Using Read−Write Locks

The functions for claiming, releasing, and using read−write locks are summarized inTable 13−12

Table 13−12 Functions for Using Read−Write Locks

Function	Purpose
pthread_rwlock_wrlock(3P)	Apply a write lock, blocking until it is available.
pthread_rwlock_trywrlock(3P)	Test a write lock and acquire it if it is available, else return an error.
pthread_rwlock_rdlock(3P)	Apply a read lock, blocking until it is available.
pthread_rwlock_tryrdlock(3P)	Test a read lock and acquire it if it is available, else return an error.
pthread_rwlock_unlock(3P)	Release a read or a write lock.

To determine where read−write locks should be used, examine the memory variables and other objects (such as files) that can be accessed from multiple threads. Create a read lock for each set of shared objects that are used together. Ensure that the code acquires the write lock before it modifies the shared objects. You acquire a write lock by calling *pthread_rwlock_wrlock()*, and release it with *pthread_rwlock_unlock()*. A read lock is acquired by calling *pthread_rwlock_rdlock()*, and released with *pthread_rwlock_unlock()*. When a thread must not be blocked, it can use *pthread_rwlock_trywrlock()* or *pthread_rwlock_tryrdlock()* to test the lock and apply it only if it is available.