Multithreaded programming

This section provides guidelines for writing multithreaded programs using the threads library (**libpthreads.a**).
The AIX® threads library is based on the X/Open Portability Guide Issue 5 standard. For this reason, the following information presents the threads library as the AIX implementation of the XPG5 standard.
Parallel programming uses the benefits of multiprocessor systems, while maintaining a full binary compatibility with existing uniprocessor systems. The parallel programming facilities are based on the concept of threads.
The advantages of using parallel programming instead of serial programming techniques are as follows:
- Parallel programming can improve the performance of a program.
- Some common software models are well-suited to parallel-programming techniques.

Traditionally, multiple single-threaded processes have been used to achieve parallelism, but some programs can benefit from a finer level of parallelism. Multithreaded processes offer parallelism within a process and share many of the concepts involved in programming multiple single-threaded processes.
The following information introduces threads and the associated programming facilities. It also discusses general topics concerning parallel programming:
Note: In this topic collection, the word *thread* used alone refers to *user threads*. This also applies to user-mode environment programming references, but not to topics related to kernel programming.

- Understanding threads and processes
  A *thread* is an independent flow of control that operates within the same address space as other independent flows of controls within a process.
- Threadsafe and threaded libraries in AIX
  This section describes the thread libraries in the AIX.
- Creating threads
  Thread creation differs from process creation in that no parent-child relation exists between threads.
- Terminating threads
  A thread automatically terminates when it returns from its entry-point routine.
- Synchronization overview
  One main benefit of using threads is the ease of using synchronization facilities.
- Using mutexes
  A *mutex* is a mutual exclusion lock. Only one thread can hold the lock.
- Using condition variables
  Condition variables allow threads to wait until some event or condition has occurred.
- Using read-write locks
  In many situations, data is read more often than it is modified or written.
- Joining threads
  Joining a thread means waiting for it to terminate, which can be seen as a specific usage of condition variables.

- Scheduling threads
  Threads can be scheduled, and the threads library provides several facilities to handle and control the scheduling of threads.
- Contention scope and concurrency level
  The *contention scope* of a user thread defines how it is mapped to a kernel thread
- Synchronization scheduling
  Programmers can control the execution scheduling of threads when there are constraints, especially time constraints, that require certain threads to be executed faster than other ones.
- One-time initializations
  Some C libraries are designed for dynamic initialization, in which the global initialization for the library is performed when the first procedure in the library is called.
- Thread-specific data
  Many applications require that certain data be maintained on a per-thread basis across function calls.
- Creating complex synchronization objects
  The subroutines provided in the threads library can be used as primitives to build more complex synchronization objects.
- Signal management
  Signals in multithreaded processes are an extension of signals in traditional single-threaded programs.
- Process duplication and termination
  Because all processes have at least one thread, creating (that is, duplicating) and terminating a process implies the creation and the termination of threads.
- Threads library options
  This section describes special attributes of threads, mutexes, and condition variables.
- Writing reentrant and threadsafe code
  In single-threaded processes, only one flow of control exists. The code executed by these processes thus need not be reentrant or threadsafe. In multithreaded programs, the same functions and the same resources may be accessed concurrently by several flows of control.
- Developing multithreaded programs
  Developing multithreaded programs is similar to developing programs with multiple processes. Developing programs also consists of compiling and debugging the code.
- Developing multithreaded programs to examine and modify pthread library objects
  The pthread debug library (**libpthdebug.a**) provides a set of functions that enable application developers to examine and modify pthread library objects.
- Developing multithreaded program debuggers
  The pthread debug library (**libpthdebug.a**) provides a set of functions that allows developers to provide debug capabilities for applications that use the pthread library.
- Benefits of threads
  Multithreaded programs can improve performance compared to traditional parallel programs that use multiple processes. Furthermore, improved performance can be obtained on multiprocessor systems using threads.

Parent topic: General programming concepts

Understanding threads and processes

A *thread* is an independent flow of control that operates within the same address space as other independent flows of controls within a process.

Traditionally, thread and process characteristics are grouped into a single entity called a *process*. In other operating systems, threads are sometimes called *lightweight processes*, or the meaning of the word *thread* is sometimes slightly different.

The following sections discuss the differences between a thread and a process.

In traditional single-threaded process systems, a process has a set of properties. In multithreaded systems, these properties are divided between processes and threads.

Threads have some limitations and cannot be used for some special purposes that require multi-processed programs.

- Process properties
  A process in a multithreaded system is the changeable entity.
- Thread properties
  A thread is the schedulable entity. A thread is the schedulable entity.
- Initial thread
  When a process is created, one thread is automatically created. This thread is called the *initial thread*.
- Modularity
  Programs are often modeled as a number of distinct parts interacting with each other to produce a desired result or service.
- Software models
  This section describes the different software models.
- Kernel Threads and User Threads
  A kernel thread is the schedulable entity, which means that the system scheduler handles kernel threads.
- Thread models and virtual processors
  User threads are mapped to kernel threads by the threads library. The way this mapping is done is called the *thread model*.
- Threads Library API
  This section provides general information about the threads library API.
- Object-Oriented Interface
  The threads library API provides an object-oriented interface. The programmer manipulates opaque objects using pointers or other universal identifiers.
- Naming Convention for the Threads Library
  The identifiers used by the threads library follow a strict naming convention. All identifiers of the threads library begin with **pthread_**.
- pthread implementation files
  This section describes the **pthread** implementation files.

Parent topic: Multithreaded programming

Related concepts:

Contention scope and concurrency level

Process duplication and termination

Process properties

A process in a multithreaded system is the changeable entity.
It must be considered as an execution frame. It has traditional process attributes, such as:
- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory

A process also provides a common address space and common system resources, as follows:
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory)

Parent topic: Understanding threads and processes

Thread properties

A thread is the schedulable entity. A thread is the schedulable entity.
It has only those properties that are required to ensure its independent control of flow. These include the following properties:

- Stack
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Some thread-specific data

An example of thread-specific data is the **errno** error indicator. In multithreaded systems, **errno** is no longer a global variable, but usually a subroutine returning a thread-specific **errno** value. Some other systems may provide other implementations of **errno**.
Threads within a process must not be considered as a group of processes. All threads share the same address space. This means that two pointers having the same value in two threads refer to the same data. Also, if any thread changes one of the shared system resources, all threads within the process are affected. For example, if a thread closes a file, the file is closed for all threads.

Parent topic: Understanding threads and processes

Initial thread

When a process is created, one thread is automatically created. This thread is called the *initial thread*.

It ensures the compatibility between the old processes with a unique implicit thread and the new multithreaded processes. The initial thread has some special properties, not visible to the programmer, that ensure binary compatibility between the old single-threaded programs and the multithreaded operating system. It is also the initial thread that executes the **main** routine in multithreaded programs.

Parent topic: Understanding threads and processes

Modularity

Programs are often modeled as a number of distinct parts interacting with each other to produce a desired result or service.

A program can be implemented as a single, complex entity that performs multiple functions among the different parts of the program. A more simple solution consists of implementing several entities, each entity performing a part of the program and sharing resources with other entities.

By using multiple entities, a program can be separated according to its distinct activities, each having an associated entity. These entities do not have to know anything about the other parts of the program except when they exchange information. In these cases, they must synchronize with each other to ensure data integrity.

Threads are well-suited entities for modular programming. Threads provide simple data sharing (all threads within a process share the same address space) and powerful synchronization facilities, such as mutexes (mutual exclusion locks) and condition variables.

Parent topic: Understanding threads and processes

Software models

This section describes the different software models.
All these models lead to modular programs. Models may also be combined to efficiently solve complex tasks.
These models can apply to either traditional multi-process solutions, or to single process multi-thread solutions, on multithreaded systems. In the following descriptions, the word *entity* refers to either a single-threaded *process* or to a single *thread* in a multithreaded process.
The following common software models can easily be implemented with threads:

- Master/Slave Model
  In the master/slave (sometimes called boss/worker) model, a master entity receives one or more requests, then creates slave entities to execute them. Typically, the master controls the number of slaves and what each slave does. A slave runs independently of other slaves.
- Divide-and-Conquer Models
  In the divide-and-conquer (sometimes called *simultaneous computation* or *work crew*) model, one or more entities perform the same tasks in parallel. There is no master entity; all entities run in parallel independently.
- Producer/Consumer Models
  The producer/consumer (sometimes called *pipelining*) model is typified by a production line. An item proceeds from raw components to a final item in a series of stages.

Parent topic: Understanding threads and processes

Master/Slave Model

In the master/slave (sometimes called boss/worker) model, a master entity receives one or more requests, then creates slave entities to execute them. Typically, the master controls the number of slaves and what each slave does. A slave runs independently of other slaves.

An example of this model is a print job spooler controlling a set of printers. The spooler's role is to ensure that the print requests received are handled in a timely fashion. When the spooler receives a request, the master entity chooses a printer and causes a slave to print the job on the printer. Each slave prints one job at a time on a printer, while it also handles flow control and other printing details. The spooler may support job cancelation or other features that require the master to cancel slave entities or reassign jobs.

Parent topic: Software models

Divide-and-Conquer Models

In the divide-and-conquer (sometimes called *simultaneous computation* or *work crew*) model, one or more entities perform the same tasks in parallel. There is no master entity; all entities run in parallel independently.

An example of a divide-and-conquer model is a parallelized **grep** command implementation, which could be done as follows. The **grep** command first establishes a pool of files to be scanned. It then creates a number of entities. Each entity takes a different file from the pool and searches for the pattern, sending the results to a common output device. When an entity completes its file search, it obtains another file from the pool or stops if the pool is empty.

Parent topic: Software models

Producer/Consumer Models

The producer/consumer (sometimes called *pipelining*) model is typified by a production line. An item proceeds from raw components to a final item in a series of stages.

Usually a single worker at each stage modifies the item and passes it on to the next stage. In software terms, an AIX® command pipe, such as the **cpio** command, is an example of this model.
For example, a reader entity reads raw data from standard input and passes it to the processor entity, which processes the data and passes it to the writer entity, which writes it to standard output. Parallel programming allows the activities to be performed concurrently: the writer entity may output some processed data while the reader entity gets more raw data.

Parent topic: Software models

Kernel Threads and User Threads

A kernel thread is the schedulable entity, which means that the system scheduler handles kernel threads.

These threads, known by the system scheduler, are strongly implementation-dependent. To facilitate the writing of portable programs, libraries provide *user* threads.

A *kernel thread* is a kernel entity, like processes and interrupt handlers; it is the entity handled by the system scheduler. A kernel thread runs within a process, but can be referenced by any other thread in the system. The programmer has no direct control over these threads, unless you are writing kernel extensions or device drivers. For more information about kernel programming, see *Kernel Extensions and Device Support Programming Concepts*.

A *user thread* is an entity used by programmers to handle multiple flows of controls within a program. The API for handling user threads is provided by the *threads library*. A user thread only exists within a process; a user thread in process *A* cannot reference a user thread in process *B*. The library uses a proprietary interface to handle kernel threads for executing user threads. The user threads API, unlike the kernel threads interface, is part of a POSIX-standards compliant portable-programming model. Thus, a multithreaded program developed on an AIX® system can easily be ported to other systems.

On other systems, user threads are simply called *threads*, and *lightweight process* refers to kernel threads.

Parent topic: Understanding threads and processes

Thread models and virtual processors

User threads are mapped to kernel threads by the threads library. The way this mapping is done is called the *thread model*.

There are three possible thread models, corresponding to three different ways to map user threads to kernel threads.

- M:1 model
- 1:1 model
- M:N model

The mapping of user threads to kernel threads is done using *virtual processors*. A virtual processor (VP) is a library entity that is usually implicit. For a user thread, the VP behaves like a CPU. In the library, the VP is a kernel thread or a structure bound to a kernel thread.

In the M:1 model, all user threads are mapped to one kernel thread; all user threads run on one VP. The mapping is handled by a library scheduler. All user-threads programming facilities are completely handled by the library. This model can be used on any system, especially on traditional single-threaded systems.

In the 1:1 model, each user thread is mapped to one kernel thread; each user thread runs on one VP. Most of the user threads programming facilities are directly handled by the kernel threads. This model is the default model.

In the M:N model, all user threads are mapped to a pool of kernel threads; all user threads run on a pool of virtual processors. A user thread may be bound to a specific VP, as in the 1:1 model. All unbound user threads share the remaining VPs. This is the most efficient and most complex thread model; the user threads programming facilities are shared between the threads library and the kernel threads. This model can be set by setting the AIXTHREAD_SCOPE environment variable to **P**.

Parent topic: Understanding threads and processes

Threads Library API

This section provides general information about the threads library API.
Although the following information is not required for writing multithreaded programs, it can help the programmer understand the threads library API.

Parent topic: Understanding threads and processes

Object-Oriented Interface

The threads library API provides an object-oriented interface. The programmer manipulates opaque objects using pointers or other universal identifiers.
This ensures the portability of multithreaded programs between systems that implement the threads library and also allows implementation changes between two releases of AIX®, necessitating only that programs be recompiled. Although some definitions of data types may be found in the threads library header file (**pthread.h**), programs should not rely on these implementation-dependent definitions to directly handle the contents of structures. The regular threads library subroutines must always be used to manipulate the objects.
The threads library essentially uses the following kinds of objects (opaque data types): threads, mutexes, rwlocks, and condition variables. These objects have attributes that specify the object properties. When creating an object, the attributes must be specified. In the threads library, these creation attributes are themselves objects, called *threads attributes objects*.
The following pairs of objects are manipulated by the threads library:
- Threads and thread-attributes objects
- Mutexes and mutex-attributes objects
- Condition variables and condition-attributes objects
- Read-write locks

An attributes object is created with attributes having default values. Attributes can then be individually modified by using subroutines. This ensures that a multithreaded program will not be affected by the introduction of new attributes or by changes in the implementation of an attribute. An attributes object can thus be used to create one or several objects, and then destroyed without affecting objects created with the attributes object.
Using an attributes object also allows the use of object classes. One attributes object may be defined for each object class. Creating an instance of an object class is done by creating the object using the class attributes object.

Parent topic: Understanding threads and processes

Naming Convention for the Threads Library

The identifiers used by the threads library follow a strict naming convention. All identifiers of the threads library begin with **pthread_**.
User programs should not use this prefix for private identifiers. This prefix is followed by a component name. The following components are defined in the threads library:

Component	Description
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr	Thread attributes objects
pthread_cond	Condition variables
pthread_condattr	Condition attributes objects
pthread_key	Thread-specific data keys
pthread_mutex	Mutexes
pthread_mutexattr	Mutex attributes objects

Data type identifiers end with **_t**. Subroutine and macro names end with an _ (underscore), followed by a name identifying the action performed by the subroutine or the macro. For example, **pthread_attr_init** is a threads library identifier (**pthread_**) that concerns thread attributes objects (**attr**) and is an initialization subroutine (**_init**).
Explicit macro identifiers are in uppercase letters. Some subroutines may, however, be implemented as macros, although their names are in lowercase letters.

Parent topic: Understanding threads and processes

pthread implementation files

This section describes the **pthread** implementation files.
The following AIX® files provide the implementation of pthreads:

Implementaion	Description
/usr/include/pthread.h	C/C++ header with most pthread definitions.
/usr/include/sched.h	C/C++ header with some scheduling definitions.
/usr/include/unistd.h	C/C++ header with **pthread_atfork()** definition.
/usr/include/sys/limits.h	C/C++ header with some pthread definitions.
/usr/include/sys/pthdebug.h	C/C++ header with most pthread debug definitions.
/usr/include/sys/sched.h	C/C++ header with some scheduling definitions.
/usr/include/sys/signal.h	C/C++ header with **pthread_kill()** and **pthread_sigmask()** definitions.
/usr/include/sys/types.h	C/C++ header with some pthread definitions.
/usr/lib/libpthreads.a	32-bit/64-bit library providing UNIX98 and POSIX 1003.1c pthreads.
/usr/lib/libpthreads_compat.a	32-bit only library providing POSIX 1003.1c Draft 7 pthreads.
/usr/lib/profiled/libpthreads.a	Profiled 32-bit/64-bit library providing UNIX98 and POSIX 1003.1c pthreads.
/usr/lib/profiled/libpthreads_compat.a	Profiled 32-bit only library providing POSIX 1003.1c Draft 7 pthreads.

Parent topic: Understanding threads and processes

Threadsafe and threaded libraries in AIX

This section describes the thread libraries in the AIX®.
By default, all applications are now considered "threaded," even though most are of the case "single threaded." These threadsafe libraries are as follows:

Threadsafe libraries		
libbsd.a	libc.a	libm.a
libsvid.a	libtli.a	libxti.a
libnetsvc.a		

POSIX threads libraries
The following POSIX threads libraries are available:
- libpthreads.a POSIX threads library
- The **libpthreads.a** library is based on the POSIX 1003.1c industry standard for a portable user threads API. Any program written for use with a POSIX thread library can be ported for use with another POSIX threads library; only the performance and very few subroutines of the threads library are implementation-dependent. To enhance the portability of the threads library, the POSIX standard made the implementation of several programming facilities optional. For more information about checking the POSIX options, see Threads Library Options.

- libpthreads_compat.a POSIX draft 7 threads library
- AIX provides binary compatibility for existing multi-threads applications that were coded to Draft 7 of the POSIX thread standard. These applications will run without relinking. The **libpthreads_compat.a** library is provided only for compatibility with earlier versions of applications written by using the Draft 7 of the POSIX Thread Standard. All new applications must use the **libpthreads.a** library, which supports both 32-bit and 64-bit applications. The **libpthreads_compat.a** library supports only 32-bit applications. Beginning with AIX 5.1, the **libpthreads.a** library supports the Single UNIX Specification, Version 2, which includes the final POSIX 1003.1c Pthread Standard.

Parent topic: Multithreaded programming

Related concepts:
Benefits of threads
Threads library options
Developing multithreaded programs

Creating threads

Thread creation differs from process creation in that no parent-child relation exists between threads.

All threads, except the *initial thread* automatically created when a process is created, are on the same hierarchical level. A thread does not maintain a list of created threads, nor does it know the thread that created it.

When creating a thread, an entry-point routine and an argument must be specified. Every thread has an entry-point routine with one argument. The same entry-point routine may be used by several threads.

A thread has attributes, which specify the characteristics of the thread. To control thread attributes, a thread attributes object must be defined before creating the thread.

Thread attributes Oobject

The thread attributes are stored in an opaque object, the *thread attributes object*, used when creating the thread. It contains several attributes, depending on the implementation of POSIX options. The object is accessed through a variable of type **pthread_attr_t**. In AIX®, the **pthread_attr_t** data type is a pointer to a structure; on other systems, it may be a structure or another data type.

Creating and destroying the thread attributes object

o

The thread attributes object is initialized to default values by the **pthread_attr_init** subroutine. The attributes are handled by subroutines. The thread attributes object is destroyed by the **pthread_attr_destroy** subroutine. This subroutine can release storage dynamically allocated by the **pthread_attr_init** subroutine, depending on the implementation of the threads library.

In the following example, a thread attributes object is created and initialized with default values, then used and finally destroyed:

```
 pthread_attr_t attributes;
                /* the attributes object is created */
...
if (!pthread_attr_init(&attributes)) {
                /* the attributes object is initialized */
        ...
                /* using the attributes object */
        ...
        pthread_attr_destroy(&attributes);
                /* the attributes object is destroyed */
}
```

The same attributes object can be used to create several threads. It can also be modified between two thread creations. When the threads are created, the attributes object can be destroyed without affecting the threads created with it.

Detachstate attribute

The following attribute is always defined:

- Detachstate

- Specifies the detached state of a thread.

The value of the attribute is returned by the **pthread_attr_getdetachstate** subroutine; it can be set by the **pthread_attr_setdetachstate** subroutine. Possible

values for this attributes are the following symbolic constants:

- **PTHREAD_CREATE_DETACHED**
  - Specifies that the thread will be created in the detached state
- **PTHREAD_CREATE_JOINABLE**
  - Specifies that the thread will be created in the joinable state

The default value is **PTHREAD_CREATE_JOINABLE**.

If you create a thread in the joinable state, you must call the **pthread_join** subroutine with the thread. Otherwise, you may run out of storage space when creating new threads, because each thread takes up a significant amount of memory. For more information on the pthread_join subroutine, see Calling the pthread_join Subroutine.

Other threads attributes

AIX also defines the following attributes, which are intended for advanced programs and may require special execution privilege to take effect. Most programs operate correctly with the default settings. The use of the following attributes is explained in Using the inheritsched Attribute.

- **Contention Scope**
  - Specifies the contention scope of a thread
- **Inheritsched**
  - Specifies the inheritance of scheduling properties of a thread
- **Schedparam**
  - Specifies the scheduling parameters of a thread
- **Schedpolicy**
  - Specifies the scheduling policy of a thread

The use of the following stack attributes is explained in Stack Attributes.

- **Stacksize**
  - Specifies the size of the thread's stack
- **Stackaddr**
  - Specifies the address of the thread's stack
- **Guardsize**
  - Specifies the size of the guard area of the thread's stack

Creating a thread using the pthread_create subroutine

A thread is created by calling the **pthread_create** subroutine. This subroutine creates a new thread and makes it runnable.

Using the thread attributes object

When calling the **pthread_create** subroutine, you may specify a thread attributes object. If you specify a **NULL** pointer, the created thread will have the default attributes. Thus, the following code fragment:

```
 pthread_t thread;
pthread_attr_t attr;
...
pthread_attr_init(&attr);
pthread_create(&thread, &attr, init_routine, NULL);
pthread_attr_destroy(&attr);
```

is equivalent to the following:

```
 pthread_t thread;
...
pthread_create(&thread, NULL, init_routine, NULL);
```

Entry point routine
When calling the **pthread_create** subroutine, you must specify an entry-point routine. This routine, provided by your program, is similar to the **main** routine for the process. It is the first user routine executed by the new thread. When the thread returns from this routine, the thread is automatically terminated.
The entry-point routine has one parameter, a void pointer, specified when calling the **pthread_create** subroutine. You may specify a pointer to some data, such as a string or a structure. The creating thread (the one calling the **pthread_create** subroutine) and the created thread must agree upon the actual type of this pointer.
The entry-point routine returns a void pointer. After the thread termination, this pointer is stored by the threads library unless the thread is detached. For more information about using this pointer, see Returning Information from a Thread.
Returned information
The **pthread_create** subroutine returns the thread ID of the new thread. The caller can use this thread ID to perform various operations on the thread.
Depending on the scheduling parameters of both threads, the new thread may start running before the call to the **pthread_create** subroutine returns. It may even happen that, when the **pthread_create** subroutine returns, the new thread has already terminated. The thread ID returned by the **pthread_create** subroutine through the *thread* parameter is then already invalid. It is, therefore, important to check for the **ESRCH** error code returned by threads library subroutines using a thread ID as a parameter.
If the **pthread_create** subroutine is unsuccessful, no new thread is created, the thread ID in the *thread* parameter is invalid, and the appropriate error code is returned. For more information, see Example of a Multi-Threaded Program.
Handling Thread IDs
The thread ID of a newly created thread is returned to the creating thread through the *thread* parameter. The current thread ID is returned by the **pthread_self** subroutine.
A thread ID is an opaque object; its type is **pthread_t**. In AIX, the **pthread_t** data type is an integer. On other systems, it may be a structure, a pointer, or any other data type.
To enhance the portability of programs using the threads library, the thread ID should always be handled as an opaque object. For this reason, thread IDs should be compared using the **pthread_equal** subroutine. Never use the C equality operator (**==**), because the **pthread_t** data type may be neither an arithmetic type nor a pointer.

Parent topic: Multithreaded programming

Related concepts:
Developing multithreaded programs

Terminating threads

A thread automatically terminates when it returns from its entry-point routine.
A thread can also explicitly terminate itself or terminate any other thread in the process, using a mechanism called *cancelation*. Because all threads share the same data space, a thread must perform cleanup operations at termination time; the threads library provides cleanup handlers for this purpose.

Exiting a thread

A process can exit at any time when a thread calls the **exit** subroutine. Similarly, a thread can exit at any time by calling the **pthread_exit** subroutine.
Calling the **exit** subroutine terminates the entire process, including all its threads. In a multithreaded program, the **exit** subroutine should only be used when the entire process needs to be terminated; for example, in the case of an unrecoverable error. The **pthread_exit** subroutine should be preferred, even for exiting the initial thread.
Calling the **pthread_exit** subroutine terminates the calling thread. The *status* parameter is saved by the library and can be further used when joining the terminated thread. Calling the **pthread_exit** subroutine is similar, but not identical, to returning from the thread's initial routine. The result of returning from the thread's initial routine depends on the thread:

- Returning from the initial thread implicitly calls the **exit** subroutine, thus terminating all the threads in the process.
- Returning from another thread implicitly calls the **pthread_exit** subroutine. The return value has the same role as the *status* parameter of the **pthread_exit** subroutine.

To avoid implicitly calling the **exit** subroutine, to use the **pthread_exit** subroutine to exit a thread.
Exiting the initial thread (for example, by calling the **pthread_exit** subroutine from the **main** routine) does not terminate the process. It terminates only the initial thread. If the initial thread is terminated, the process will be terminated when the last thread in it terminates. In this case, the process return code is 0.
The following program displays exactly 10 messages in each language. This is accomplished by calling the **pthread_exit** subroutine in the **main** routine after creating the two threads, and creating a loop in the **Thread** routine.

```
 #include <pthread.h>    /* include file for pthreads - the 1st */
#include <stdio.h>      /* include file for printf()           */


void *Thread(void *string)


{
       int i;


       for (i=0; i<10; i++)
               printf("%s\n", (char *)string);
       pthread_exit(NULL);
}


int main()
{
       char *e_str = "Hello!";
```

```
        char *f_str = "Bonjour !";

        pthread_t e_th;
        pthread_t f_th;

        int rc;

        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        pthread_exit(NULL);
}
```

The **pthread_exit** subroutine releases any thread-specific data, including the thread's stack. Any data allocated on the stack becomes invalid, because the stack is freed and the corresponding memory may be reused by another thread. Therefore, thread synchronization objects (mutexes and condition variables) allocated on a thread's stack must be destroyed before the thread calls the **pthread_exit** subroutine.
Unlike the **exit** subroutine, the **pthread_exit** subroutine does not clean up system resources shared among threads. For example, files are not closed by the **pthread_exit** subroutine, because they may be used by other threads.

Canceling a thread

The thread cancelation mechanism allows a thread to terminate the execution of any other thread in the process in a controlled manner. The target thread (that is, the one that is being canceled) can hold cancelation requests pending in a number of ways and perform application-specific cleanup processing when the notice of cancelation is acted upon. When canceled, the thread implicitly calls the **pthread_exit((void *)-1)** subroutine.
The cancelation of a thread is requested by calling the **pthread_cancel** subroutine. When the call returns, the request has been registered, but the thread may still be running. The call to the **pthread_cancel** subroutine is unsuccessful only when the specified thread ID is not valid.

Cancelability state and type

The cancelability state and type of a thread determines the action taken upon receipt of a cancelation request. Each thread controls its own cancelability state and type with the **pthread_setcancelstate** and **pthread_setcanceltype** subroutines. The following possible cancelability states and cancelability types lead to three possible cases, as shown in the following table.

Cancelability State	Cancelability Type	Resulting Case
Disabled	Any (the type is ignored)	Disabled cancelability
Enabled	Deferred	Deferred cancelability
Enabled	Asynchronous	Asynchronous cancelability

The possible cases are described as follows:
- *Disabled cancelability*. Any cancelation request is set pending, until the cancelability state is changed or the thread is terminated in another way. A thread should disable cancelability only when performing operations that cannot be interrupted. For example, if a thread is performing some complex file-save operations (such as an indexed database) and is canceled during the operation, the files may be left in an inconsistent state. To avoid this, the thread should disable cancelability during the file save operations.
- *Deferred cancelability*. Any cancelation request is set pending, until the thread reaches the next cancelation point. It is the default cancelability state. This cancelability state ensures that a thread can be cancelled, but limits the cancelation to specific moments in the thread's execution, called *cancelation points* . A thread canceled on a cancelation point leaves the system in a safe state; however, user data may be inconsistent or locks may be held by the canceled thread. To avoid these situations, use cleanup handlers or disable cancelability within critical regions. For more information, see Using Cleanup Handlers .
- *Asynchronous cancelability*. Any cancelation request is acted upon immediately. A thread that is asynchronously canceled while holding resources may leave the process, or even the system, in a state from which it is difficult or impossible to recover. For more information about async-cancel safety, see Async-Cancel Safety .

Async-cancel safety

A function is said to be *async-cancel safe* if it is written so that calling the function with asynchronous cancelability enabled does not cause any resource to be corrupted, even if a cancelation request is delivered at any arbitrary instruction. Any function that gets a resource as a side effect cannot be made async-cancel safe. For example, if the **malloc** subroutine is called with asynchronous cancelability enabled, it might acquire the resource successfully, but as it was returning to the caller, it could act on a cancelation request. In such a case, the program would have no way of knowing whether the resource was acquired or not.

For this reason, most library routines cannot be considered async-cancel safe. It is recommended that you use asynchronous cancelability only if you are sure only to perform operations that do not hold resources and only to call library routines that are async-cancel safe.

The following subroutines are async-cancel safe; they ensure that cancelation will be handled correctly, even if asynchronous cancelability is enabled:
- **pthread_cancel**
- **pthread_setcancelstate**
- **pthread_setcanceltype**

An alternative to asynchronous cancelability is to use deferred cancelability and to add explicit cancelation points by calling the **pthread_testcancel** subroutine.

Cancelation points

Cancelation points are points inside of certain subroutines where a thread must act on any pending cancelation request if deferred cancelability is enabled. All of these subroutines may block the calling thread or compute indefinitely.

An explicit cancelation point can also be created by calling the **pthread_testcancel** subroutine. This subroutine simply creates a cancelation point. If deferred cancelability is enabled, and if a cancelation request is pending, the request is acted upon and the thread is terminated. Otherwise, the subroutine simply returns.
Other cancelation points occur when calling the following subroutines:
- **pthread_cond_wait**
- **pthread_cond_timedwait**
- **pthread_join**

The **pthread_mutex_lock** and **pthread_mutex_trylock** subroutines do not provide a cancelation point. If they did, all functions calling these subroutines (and many functions do) would provide a cancelation point. Having too many cancelation points makes programming very difficult, requiring either lots of disabling and restoring of cancelability or extra effort in trying to arrange for reliable cleanup at every possible place. For more information about these subroutines, see Using Mutexes.
Cancelation points occur when a thread is executing the following functions:

Function	
aio_suspend	**close**
creat	**fcntl**
fsync	**getmsg**
getpmsg	**lockf**
mq_receive	**mq_send**
msgrcv	**msgsnd**
msync	**nanosleep**
open	**pause**
poll	**pread**
pthread_cond_timedwait	**pthread_cond_wait**
pthread_join	**pthread_testcancel**
putpmsg	**pwrite**
read	**readv**
select	**sem_wait**
sigpause	**sigsuspend**
sigtimedwait	**sigwait**
sigwaitinfo	**sleep**
system	**tcdrain**
usleep	**wait**
wait3	**waitid**
waitpid	**write**
writev	

A cancelation point can also occur when a thread is executing the following functions:

Function Function Function

catclose	**catgets**	**catopen**
closedir	**closelog**	**ctermid**
dbm_close	**dbm_delete**	**dbm_fetch**
dbm_nextkey	**dbm_open**	**dbm_store**
dlclose	**dlopen**	**endgrent**
endpwent	**fwprintf**	**fwrite**
fwscanf	**getc**	**getc_unlocked**
getchar	**getchar_unlocked**	**getcwd**
getdate	**getgrent**	**getgrgid**
getgrgid_r	**getgrnam**	**getgrnam_r**
getlogin	**getlogin_r**	**popen**
printf	**putc**	**putc_unlocked**
putchar	**putchar_unlocked**	**puts**
pututxline	**putw**	**putwc**
putwchar	**readdir**	**readdir_r**
remove	**rename**	**rewind**
endutxent	**fclose**	**fcntl**
fflush	**fgetc**	**fgetpos**
fgets	**fgetwc**	**fgetws**
fopen	**fprintf**	**fputc**
fputs	**getpwent**	**getpwnam**
getpwnam_r	**getpwuid**	**getpwuid_r**
gets	**getutxent**	**getutxid**
getutxline	**getw**	**getwc**
getwchar	**getwd**	**rewinddir**
scanf	**seekdir**	**semop**
setgrent	**setpwent**	**setutxent**
strerror	**syslog**	**tmpfile**
tmpnam	**ttyname**	**ttyname_r**
fputwc	**fputws**	**fread**
freopen	**fscanf**	**fseek**
fseeko	**fsetpos**	**ftell**
ftello	**ftw**	**glob**
iconv_close	**iconv_open**	**ioctl**
lseek	**mkstemp**	**nftw**
opendir	**openlog**	**pclose**
perror	**ungetc**	**ungetwc**
unlink	**vfprintf**	**vfwprintf**
vprintf	**vwprintf**	**wprintf**
wscanf		

The side effects of acting upon a cancelation request while suspended during a call of a function is the same as the side effects that may be seen in a single-threaded

program when a call to a function is interrupted by a signal and the given function returns [EINTR]. Any such side effects occur before any cancelation cleanup handlers are called.
Whenever a thread has cancelability enabled and a cancelation request has been made with that thread as the target and the thread calls the **pthread_testcancel** subroutine, the cancelation request is acted upon before the **pthread_testcancel** subroutine returns. If a thread has cancelability enabled and the thread has an asynchronous cancelation request pending and the thread is suspended at a cancelation point waiting for an event to occur, the cancelation request will be acted upon. However, if the thread is suspended at a cancelation point and the event that it is waiting for occurs before the cancelation request is acted upon, the sequence of events determines whether the cancelation request is acted upon or whether the request remains pending and the thread resumes normal execution.

Cancelation example

In the following example, both "writer" threads are canceled after 10 seconds, and after they have written their message at least five times.

```
 #include <pthread.h>    /* include file for pthreads - the 1st */
#include <stdio.h>      /* include file for printf()           */
#include <unistd.h>     /* include file for sleep()            */
 void *Thread(void *string)
{
       int i;
       int o_state;

       /* disables cancelability */
       pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &o_state);

       /* writes five messages */
       for (i=0; i<5; i++)
               printf("%s\n", (char *)string);

       /* restores cancelability */
       pthread_setcancelstate(o_state, &o_state);

       /* writes further */
       while (1)
               printf("%s\n", (char *)string);
       pthread_exit(NULL);
}
 int main()
{
       char *e_str = "Hello!";
       char *f_str = "Bonjour !";

       pthread_t e_th;
       pthread_t f_th;

       int rc;
```

```
        /* creates both threads */
        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                return -1;
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                return -1;

          /* sleeps a while */
        sleep(10);

        /* requests cancelation */
        pthread_cancel(e_th);
        pthread_cancel(f_th);

        /* sleeps a bit more */
        sleep(10);
        pthread_exit(NULL);
}
```

Timer and sleep subroutines

Timer routines execute in the context of the calling thread. Thus, if a timer expires, the watchdog timer function is called in the thread's context. When a process or thread goes to sleep, it relinquishes the processor. In a multithreaded process, only the calling thread is put to sleep.

Using cleanup handlers

Cleanup handlers provide a portable mechanism for releasing resources and restoring invariants when a thread terminates.

Calling cleanup handlers

Cleanup handlers are specific to each thread. A thread can have several cleanup handlers; they are stored in a thread-specific LIFO (last-in, first-out) stack. Cleanup handlers are all called in the following cases:

- The thread returns from its entry-point routine.
- The thread calls the **pthread_exit** subroutine.
- The thread acts on a cancelation request.

A cleanup handler is pushed onto the cleanup stack by the **pthread_cleanup_push** subroutine. The **pthread_cleanup_pop** subroutine pops the topmost cleanup handler from the stack and optionally executes it. Use this subroutine when the cleanup handler is no longer needed.

The cleanup handler is a user-defined routine. It has one parameter, a void pointer, specified when calling the **pthread_cleanup_push** subroutine. You can specify a pointer to some data that the cleanup handler needs to perform its operation.

In the following example, a buffer is allocated for performing some operation. With deferred cancelability enabled, the operation can be stopped at any cancelation point. In that case, a cleanup handler is established to release the buffer.

```
 /* the cleanup handler */
```

```
cleaner(void *buffer)

{
       free(buffer);
}

/* fragment of another routine */
...
myBuf = malloc(1000);
if (myBuf != NULL) {

       pthread_cleanup_push(cleaner, myBuf);

       /*
        *       perform any operation using the buffer,
        *       including calls to other functions
        *       and cancelation points
        */

       /* pops the handler and frees the buffer in one call */
       pthread_cleanup_pop(1);
}
```

Using deferred cancelability ensures that the thread will not act on any cancelation request between the buffer allocation and the registration of the cleanup handler, because neither the **malloc** subroutine nor the **pthread_cleanup_push** subroutine provides any cancelation point. When popping the cleanup handler, the handler is executed, releasing the buffer. More complex programs may not execute the handler when popping it, because the cleanup handler should be thought of as an "emergency exit" for the protected portion of code.

Balancing the push and pop operations

The **pthread_cleanup_push** and **pthread_cleanup_pop** subroutines should always appear in pairs within the same lexical scope; that is, within the same function and the same statement block. They can be thought of as left and right parentheses enclosing a protected portion of code.

The reason for this rule is that on some systems these subroutines are implemented as macros. The **pthread_cleanup_push** subroutine is implemented as a left brace, followed by other statements: `#define pthread_cleanup_push(rtm,arg) { \`

```
       /* other statements */
```

The **pthread_cleanup_pop** subroutine is implemented as a right brace, following other statements: `#define pthread_cleanup_pop(ex) \`

```
       /* other statements */  \
}
```

Adhere to the balancing rule for the **pthread_cleanup_push** and **pthread_cleanup_pop** subroutines to avoid compiler errors or unexpected behavior of your programs when porting to other systems.

In AIX®, the **pthread_cleanup_push** and **pthread_cleanup_pop** subroutines are library routines, and can be unbalanced within the same statement block. However, they must be balanced in the program, because the cleanup handlers are stacked.

Subroutine	Description
pthread_attr_destroy	Deletes a thread attributes object.
pthread_attr_getdetachstate	Returns the value of the detachstate attribute of a thread attributes object.
pthread_attr_init	Creates a thread attributes object and initializes it with default values.
pthread_cancel	Requests the cancelation of a thread.
pthread_cleanup_pop	Removes, and optionally executes, the routine at the top of the calling thread's cleanup stack.
pthread_cleanup_push	Pushes a routine onto the calling thread's cleanup stack.
pthread_create	Creates a new thread, initializes its attributes, and makes it runnable.
pthread_equal	Compares two thread IDs.
pthread_exit	Terminates the calling thread.
pthread_self	Returns the calling thread's ID.
pthread_setcancelstate	Sets the calling thread's cancelability state.
pthread_setcanceltype	Sets the calling thread's cancelability type.
pthread_testcancel	Creates a cancelation point in the calling thread.

Parent topic: Multithreaded programming

Related concepts:
One-time initializations

Synchronization overview

One main benefit of using threads is the ease of using synchronization facilities. To effectively interact, threads must synchronize their activities. This includes:
- Implicit communication through the modification of shared data
- Explicit communication by informing each other of events that have occurred

More complex synchronization objects can be built using the primitive objects. The threads library provides the following synchronization mechanisms:Although primitive, these powerful mechanisms can be used to build more complex mechanisms.
The threads library provides the following synchronization mechanisms:
- Mutexes (See Using Mutexes)
- Condition variables (See Using Condition Variables)
- Read-write locks (See Using Read-Write Locks)
- Joins (See Joining Threads)

Although primitive, these powerful mechanisms can be used to build more complex mechanisms.

Parent topic: Multithreaded programming

Related concepts:
Creating complex synchronization objects

Using mutexes

A *mutex* is a mutual exclusion lock. Only one thread can hold the lock. Mutexes are used to protect data or other resources from concurrent access. A mutex has attributes, which specify the characteristics of the mutex.

Mutex attributes object

Like threads, mutexes are created with the help of an attributes object. The mutex attributes object is an abstract object, containing several attributes, depending on the implementation of POSIX options. It is accessed through a variable of type **pthread_mutexattr_t**. In AIX®, the **pthread_mutexattr_t** data type is a pointer; on other systems, it may be a structure or another data type.

Creating and destroying the mutex attributes object

The mutex attributes object is initialized to default values by the **pthread_mutexattr_init** subroutine. The attributes are handled by subroutines. The thread attributes object is destroyed by the **pthread_mutexattr_destroy** subroutine. This subroutine may release storage dynamically allocated by the **pthread_mutexattr_init** subroutine, depending on the implementation of the threads library.

In the following example, a mutex attributes object is created and initialized with default values, then used and finally destroyed:

```
pthread_mutexattr_t attributes;
                /* the attributes object is created */
...
if (!pthread_mutexattr_init(&attributes)) {
                /* the attributes object is initialized */
        ...
                /* using the attributes object */
        ...
        pthread_mutexattr_destroy(&attributes);
                /* the attributes object is destroyed */
}
```

The same attributes object can be used to create several mutexes. It can also be modified between mutex creations. When the mutexes are created, the attributes object can be destroyed without affecting the mutexes created with it.

Mutex attributes

The following mutex attributes are defined:

Attribute	Description
Protocol	Specifies the protocol used to prevent priority inversions for a mutex. This attribute depends on either the priority inheritance or the priority protection POSIX option.
Process-shared	Specifies the process sharing of a mutex. This attribute depends on the process sharing POSIX option.

For more information on these attributes, see Threads Library Options and Synchronization Scheduling.

Creating and destroying mutexes

A mutex is created by calling the **pthread_mutex_init** subroutine. You may specify a mutex attributes object. If you specify a **NULL** pointer, the mutex will have the default attributes. Thus, the following code fragment:

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;
...
pthread_mutexattr_init(&attr);
pthread_mutex_init(&mutex, &attr);
pthread_mutexattr_destroy(&attr);
```

is equivalent to the following:

```
pthread_mutex_t mutex;
...
pthread_mutex_init(&mutex, NULL);
```

The ID of the created mutex is returned to the calling thread through the *mutex* parameter. The mutex ID is an opaque object; its type is **pthread_mutex_t**. In AIX, the **pthread_mutex_t** data type is a structure; on other systems, it might be a pointer or another data type.

A mutex must be created once. However, avoid calling the **pthread_mutex_init** subroutine more than once with the same *mutex* parameter (for example, in two threads concurrently executing the same code). Ensuring the uniqueness of a mutex creation can be done in the following ways:

- Calling the **pthread_mutex_init** subroutine prior to the creation of other threads that will use this mutex; for example, in the initial thread.
- Calling the **pthread_mutex_init** subroutine within a one time initialization routine. For more information, see One-Time Initializations.
- Using a static mutex initialized by the **PTHREAD_MUTEX_INITIALIZER** static initialization macro; the mutex will have default attributes.

After the mutex is no longer needed, destroy it by calling the **pthread_mutex_destroy** subroutine. This subroutine may reclaim any storage allocated by the **pthread_mutex_init** subroutine. After having destroyed a mutex, the same **pthread_mutex_t** variable can be reused to create another mutex. For example, the following code fragment is valid, although not very practical:

```
pthread_mutex_t mutex;
...
for (i = 0; i < 10; i++) {

        /* creates a mutex */
        pthread_mutex_init(&mutex, NULL);

        /* uses the mutex */

        /* destroys the mutex */
        pthread_mutex_destroy(&mutex);
}
```

Like any system resource that can be shared among threads, a mutex allocated on a thread's stack must be destroyed before the thread is terminated. The threads

library maintains a linked list of mutexes. Thus, if the stack where a mutex is allocated is freed, the list will be corrupted.

Types of mutexes

The type of mutex determines how the mutex behaves when it is operated on. The following types of mutexes exist:

- **PTHREAD_MUTEX_DEFAULT or PTHREAD_MUTEX_NORMAL**
  - Results in a deadlock if the same pthread tries to lock it a second time using the **pthread_mutex_lock** subroutine without first unlocking it. This is the default type.
- **PTHREAD_MUTEX_ERRORCHECK**
  - Avoids deadlocks by returning a non-zero value if the same thread attempts to lock the same mutex more than once without first unlocking the mutex.
- **PTHREAD_MUTEX_RECURSIVE**
  - Allows the same pthread to recursively lock the mutex using the **pthread_mutex_lock** subroutine without resulting in a deadlock or getting a non-zero return value from **pthread_mutex_lock**. The same pthread has to call the **pthread_mutex_unlock** subroutine the same number of times as it called **pthread_mutex_lock** subroutine in order to unlock the mutex for other pthreads to use.

When a mutex attribute is first created, it has a default type of **PTHREAD_MUTEX_NORMAL**. After creating the mutex, the type can be changed using the **pthread_mutexattr_settype** API library call.

The following is an example of creating and using a recursive mutex type:

```
pthread_mutexattr_t    attr;
pthread_mutex_t         mutex;


pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&mutex, &attr);


struct {
       int a;
       int b;
       int c;
} A;


f()
{
       pthread_mutex_lock(&mutex);
       A.a++;
       g();
       A.c = 0;
       pthread_mutex_unlock(&mutex);
}


g()
{
       pthread_mutex_lock(&mutex);
```

```
        A.b += A.a;
        pthread_mutex_unlock(&mutex);
}
```

Locking and unlocking mutexes

A mutex is a simple lock, having two states: locked and unlocked. When it is created, a mutex is unlocked. The **pthread_mutex_lock** subroutine locks the specified mutex under the following conditions:

- If the mutex is unlocked, the subroutine locks it.
- If the mutex is already locked by another thread, the subroutine blocks the calling thread until the mutex is unlocked.
- If the mutex is already locked by the calling thread, the subroutine might block forever or return an error depending on the type of mutex.

The **pthread_mutex_trylock** subroutine acts like the **pthread_mutex_lock** subroutine without blocking the calling thread under the following conditions:

- If the mutex is unlocked, the subroutine locks it.
- If the mutex is already locked by any thread, the subroutine returns an error.

The thread that locked a mutex is often called the *owner* of the mutex.

The **pthread_mutex_unlock** subroutine resets the specified mutex to the unlocked state if it is owned by the calling mutex under the following conditions:

- If the mutex was already unlocked, the subroutine returns an error.
- If the mutex was owned by the calling thread, the subroutine unlocks the mutex.
- If the mutex was owned by another thread, the subroutine might return an error or unlock the mutex depending on the type of mutex. Unlocking the mutex is not recommended because mutexes are usually locked and unlocked by the same pthread.

Because locking does not provide a cancelation point, a thread blocked while waiting for a mutex cannot be canceled. Therefore, it is recommended that you use mutexes only for short periods of time, as in instances where you are protecting data from concurrent access. For more information, see Cancelation Points and Canceling a Thread.

Protecting data with mutexes

Mutexes are intended to serve either as a low-level primitive from which other thread synchronization functions can be built or as a data protection lock. For more information about implementing long locks and writer-priority readers/writers locks see Using mutexes.

Mutex usage example

Mutexes can be used to protect data from concurrent access. For example, a database application may create several threads to handle several requests concurrently. The database itself is protected by a mutex called **db_mutex**. For example:

```
/* the initial thread */
pthread_mutex_t mutex;
int i;
...
pthread_mutex_init(&mutex, NULL);    /* creates the mutex      */
for (i = 0; i < num_req; i++)        /* loop to create threads */
        pthread_create(th + i, NULL, rtn, &mutex);
...                                  /* waits end of session   */
```

```
pthread_mutex_destroy(&mutex);       /* destroys the mutex     */
...

/* the request handling thread */
...                                  /* waits for a request  */
pthread_mutex_lock(&db_mutex);       /* locks the database   */
...                                  /* handles the request  */
pthread_mutex_unlock(&db_mutex);     /* unlocks the database */
...
```

The initial thread creates the mutex and all the request-handling threads. The mutex is passed to the thread using the parameter of the thread's entry point routine. In a real program, the address of the mutex may be a field of a more complex data structure passed to the created thread.

Avoiding Deadlocks

There are a number of ways that a multithreaded application can deadlock. Following are some examples:

- A mutex created with the default type, **PTHREAD_MUTEX_NORMAL**, cannot be relocked by the same pthread without resulting in a deadlock.
- An application can deadlock when locking mutexes in reverse order. For example, the following code fragment can produce a deadlock between threads A and B.

```
/* Thread A */
pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);

/* Thread B */
pthread_mutex_lock(&mutex2);
pthread_mutex_lock(&mutex1);
```

- An application can deadlock in what is called *resource* deadlock. For example:

```
struct {
        pthread_mutex_t mutex;
        char *buf;
} A;

struct {
        pthread_mutex_t mutex;
        char *buf;
} B;

struct {
        pthread_mutex_t mutex;
        char *buf;
} C;

use_all_buffers()
{
        pthread_mutex_lock(&A.mutex);
        /* use buffer A */
```

```
        pthread_mutex_lock(&B.mutex);
        /* use buffers B */

        pthread_mutex_lock(&C.mutex);
        /* use buffer C */

        /* All done */
        pthread_mutex_unlock(&C.mutex);
        pthread_mutex_unlock(&B.mutex);
        pthread_mutex_unlock(&A.mutex);
}

use_buffer_a()
{
        pthread_mutex_lock(&A.mutex);
        /* use buffer A */
        pthread_mutex_unlock(&A.mutex);
}

functionB()
{
        pthread_mutex_lock(&B.mutex);
        /* use buffer B */
        if (..some condition)
        {
          use_buffer_a();
        }
        pthread_mutex_unlock(&B.mutex);
}

/* Thread A */
use_all_buffers();

/* Thread B */
functionB();
```

This application has two threads, thread A and thread B. Thread B starts to run first, then thread A starts shortly thereafter. If thread A executes **use_all_buffers()** and successfully locks **A.mutex**, it will then block when it tries to lock **B.mutex**, because thread B has already locked it. While thread B executes **functionB** and some_condition occurs while thread A is blocked, thread B will now also block trying to acquire **A.mutex**, which is already locked by thread A. This results in a deadlock. The solution to this deadlock is for each thread to acquire all the resource locks that it needs before using the resources. If it cannot acquire the locks, it must release them and start again.

Mutexes and race conditions

Mutual exclusion locks (mutexes) can prevent data inconsistencies due to race

conditions. A race condition often occurs when two or more threads must perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.
Consider, for example, a single counter, *X*, that is incremented by two threads, A and B. If *X* is originally 1, then by the time threads A and B increment the counter, *X* should be 3. Both threads are independent entities and have no synchronization between them. Although the C statement X++ looks simple enough to be atomic, the generated assembly code may not be, as shown in the following pseudo-assembler code:

```
move    X, REG
inc     REG
move    REG, X
```

If both threads in the previous example are executed concurrently on two CPUs, or if the scheduling makes the threads alternatively execute on each instruction, the following steps may occur:

1. Thread A executes the first instruction and puts *X*, which is 1, into the thread A register. Then thread B executes and puts *X*, which is 1, into the thread B register. The following example illustrates the resulting registers and the contents of memory *X*.
   ```
   Thread A Register = 1
   Thread B Register = 1
   Memory X          = 1
   ```
2. Thread A executes the second instruction and increments the content of its register to 2. Then thread B increments its register to 2. Nothing is moved to memory *X*, so memory *X* stays the same. The following example illustrates the resulting registers and the contents of memory *X*.
   ```
   Thread A Register = 2
   Thread B Register = 2
   Memory X          = 1
   ```
3. Thread A moves the content of its register, which is now 2, into memory *X*. Then thread B moves the content of its register, which is also 2, into memory *X*, overwriting thread A's value. The following example illustrates the resulting registers and the contents of memory *X*.
   ```
   Thread A Register = 2
   Thread B Register = 2
   Memory X          = 2
   ```

In most cases, thread A and thread B execute the three instructions one after the other, and the result would be 3, as expected. Race conditions are usually difficult to discover, because they occur intermittently.
To avoid this race condition, each thread should lock the data before accessing the counter and updating memory *X*. For example, if thread A takes a lock and updates the counter, it leaves memory *X* with a value of 2. After thread A releases the lock, thread B takes the lock and updates the counter, taking 2 as its initial value for *X* and increment it to 3, the expected result.

Parent topic: Multithreaded programming

Using condition variables

Condition variables allow threads to wait until some event or condition has occurred.

A condition variable has attributes that specify the characteristics of the condition. Typically, a program uses the following objects:
- A boolean variable, indicating whether the condition is met
- A mutex to serialize the access to the boolean variable
- A condition variable to wait for the condition

Using a condition variable requires some effort from the programmer. However, condition variables allow the implementation of powerful and efficient synchronization mechanisms. For more information about implementing long locks and semaphores with condition variables, see Creating Complex Synchronization Objects.
When a thread is terminated, its storage may not be reclaimed, depending on an attribute of the thread. Such threads can be joined by other threads and return information to them. A thread that wants to join another thread is blocked until the target thread terminates. This joint mechanism is a specific case of condition-variable usage, the condition is the thread termination.

Condition attributes object

Like threads and mutexes, condition variables are created with the help of an attributes object. The *condition attributes object* is an abstract object, containing at most one attribute, depending on the implementation of POSIX options. It is accessed through a variable of type **pthread_condattr_t**. In AIX®, the **pthread_condattr_t** data type is a pointer; on other systems, it may be a structure or another data type.

Creating and destroying the condition attributes object

The condition attributes object is initialized to default values by the **pthread_condattr_init** subroutine. The attribute is handled by subroutines. The thread attributes object is destroyed by the **pthread_condattr_destroy** subroutine. This subroutine can release storage dynamically allocated by the **pthread_condattr_init** subroutine, depending on the implementation of the threads library.
In the following example, a condition attributes object is created and initialized with default values, then used and finally destroyed:

```
pthread_condattr_t attributes;
                /* the attributes object is created */
...
if (!pthread_condattr_init(&attributes)) {
                /* the attributes object is initialized */
        ...
                /* using the attributes object */
        ...
        pthread_condattr_destroy(&attributes);
                        /* the attributes object is destroyed */
}
```

The same attributes object can be used to create several condition variables. It can also be modified between two condition variable creations. When the condition

variables are created, the attributes object can be destroyed without affecting the condition variables created with it.

Condition attribute

The following condition attribute is supported:

- Process-shared

- Specifies the process sharing of a condition variable. This attribute depends on the process sharing POSIX option.

Creating and destroying condition variables

A condition variable is created by calling the **pthread_cond_init** subroutine. You may specify a condition attributes object. If you specify a **NULL** pointer, the condition variable will have the default attributes. Thus, the following code fragment:

```
 pthread_cond_t cond;
pthread_condattr_t attr;
...
pthread_condattr_init(&attr);
pthread_cond_init(&cond, &attr);
pthread_condattr_destroy(&attr);
```

is equivalent to the following:

```
pthread_cond_t cond;
...
pthread_cond_init(&cond, NULL);
```

The ID of the created condition variable is returned to the calling thread through the *condition* parameter. The condition ID is an opaque object; its type is **pthread_cond_t**. In AIX, the **pthread_cond_t** data type is a structure; on other systems, it may be a pointer or another data type.

A condition variable must be created once. Avoid calling the **pthread_cond_init** subroutine more than once with the same *condition* parameter (for example, in two threads concurrently executing the same code). Ensuring the uniqueness of a newly created condition variable can be done in the following ways:

- Calling the **pthread_cond_init** subroutine prior to the creation of other threads that will use this variable; for example, in the initial thread.
- Calling the **pthread_cond_init** subroutine within a one-time initialization routine. For more information, see One-Time Initializations.
- Using a static condition variable initialized by the **PTHREAD_COND_INITIALIZER** static initialization macro; the condition variable will have default attributes.

After the condition variable is no longer needed, destroy it by calling the **pthread_cond_destroy** subroutine. This subroutine may reclaim any storage allocated by the **pthread_cond_init** subroutine. After having destroyed a condition variable, the same **pthread_cond_t** variable can be reused to create another condition. For example, the following code fragment is valid, although not very practical:

```
 pthread_cond_t cond;
...
for (i = 0; i < 10; i++) {

      /* creates a condition variable */
```

```
        pthread_cond_init(&cond, NULL);

        /* uses the condition variable */

        /* destroys the condition */
        pthread_cond_destroy(&cond);
}
```

Like any system resource that can be shared among threads, a condition variable allocated on a thread's stack must be destroyed before the thread is terminated. The threads library maintains a linked list of condition variables; thus, if the stack where a mutex is allocated is freed, the list will be corrupted.

Using condition variables

A condition variable must always be used together with a mutex. A given condition variable can have only one mutex associated with it, but a mutex can be used for more than one condition variable. It is possible to bundle into a structure the condition, the mutex, and the condition variable, as shown in the following code fragment:

```
struct condition_bundle_t {
        int             condition_predicate;
        pthread_mutex_t condition_lock;
        pthread_cond_t  condition_variable;
};
```

Waiting for a condition

The mutex protecting the condition must be locked before waiting for the condition. A thread can wait for a condition to be signaled by calling the **pthread_cond_wait** or **pthread_cond_timedwait** subroutine. The subroutine atomically unlocks the mutex and blocks the calling thread until the condition is signaled. When the call returns, the mutex is locked again.

The **pthread_cond_wait** subroutine blocks the thread indefinitely. If the condition is never signaled, the thread never wakes up. Because the **pthread_cond_wait** subroutine provides a cancelation point, the only way to exit this deadlock is to cancel the blocked thread, if cancelability is enabled. For more information, see Canceling a Thread.

The **pthread_cond_timedwait** subroutine blocks the thread only for a given period of time. This subroutine has an extra parameter, *timeout*, specifying an absolute date where the sleep must end. The *timeout* parameter is a pointer to a **timespec** structure. This data type is also called **timestruc_t**. It contains the following fields:

- tv_sec
  - A long unsigned integer, specifying seconds

- tv_nsec
  - A long integer, specifying nanoseconds

Typically, the **pthread_cond_timedwait** subroutine is used in the following manner:

```
 struct timespec timeout;
...
time(&timeout.tv_sec);
timeout.tv_sec += MAXIMUM_SLEEP_DURATION;
pthread_cond_timedwait(&cond, &mutex, &timeout);
```

The *timeout* parameter specifies an absolute date. The previous code fragment shows how to specify a duration rather than an absolute date.
To use the **pthread_cond_timedwait** subroutine with an absolute date, you can use the **mktime** subroutine to calculate the value of the tv_sec field of the **timespec** structure. In the following example, the thread waits for the condition until 08:00 January 1, 2001, local time:

```
struct tm      date;
time_t         seconds;
struct timespec timeout;
...

date.tm_sec = 0;
date.tm_min = 0;
date.tm_hour = 8;
date.tm_mday = 1;
date.tm_mon = 0;         /* the range is 0-11 */
date.tm_year = 101;      /* 0 is 1900 */
date.tm_wday = 1;        /* this field can be omitted -
                            but it will really be a Monday! */
date.tm_yday = 0;        /* first day of the year */
date.tm_isdst = daylight;
       /* daylight is an external variable - we are assuming
          that daylight savings time will still be used... */

seconds = mktime(&date);

timeout.tv_sec = (unsigned long)seconds;
timeout.tv_nsec = 0L;

pthread_cond_timedwait(&cond, &mutex, &timeout);
```

The **pthread_cond_timedwait** subroutine also provides a cancelation point, although the sleep is not indefinite. Thus, a sleeping thread can be canceled, whether or not the sleep has a timeout.

Signaling a condition

A condition can be signaled by calling either the **pthread_cond_signal** or the **pthread_cond_broadcast** subroutine.
The **pthread_cond_signal** subroutine wakes up at least one thread that is currently blocked on the specified condition. The awoken thread is chosen according to the scheduling policy; it is the thread with the most-favored scheduling priority (see Scheduling Policy and Priority). It may happen on multiprocessor systems, or some non-AIX systems, that more than one thread is awakened. Do not assume that this subroutine wakes up exactly one thread.
The **pthread_cond_broadcast** subroutine wakes up every thread that is currently blocked on the specified condition. However, a thread can start waiting on the same condition just after the call to the subroutine returns.
A call to these routines always succeeds, unless an invalid *cond* parameter is specified. This does not mean that a thread has been awakened. Furthermore,

signaling a condition is not remembered by the library. For example, consider a condition C. No thread is waiting on this condition. At time t, thread 1 signals the condition C. The call is successful although no thread is awakened. At time t+1, thread 2 calls the **pthread_cond_wait** subroutine with C as *cond* parameter. Thread 2 is blocked. If no other thread signals C, thread 2 may wait until the process terminates.
You can avoid this kind of deadlock by checking the **EBUSY** error code returned by the **pthread_cond_destroy** subroutine when destroying the condition variable, as in the following code fragment:
The **pthread_yield** subroutine gives the opportunity to another thread to be scheduled; for example, one of the awoken threads. For more information about the **pthread_yield** subroutine.
The **pthread_cond_wait** and the **pthread_cond_broadcast** subroutines must not be used within a signal handler. To provide a convenient way for a thread to await a signal, the threads library provides the **sigwait** subroutine. For more information about the **sigwait** subroutine. For more information about the **sigwait** subroutine, see Signal Management.

Synchronizing threads with condition variables

```
 while (pthread_cond_destroy(&cond) == EBUSY) {
       pthread_cond_broadcast(&cond);
       pthread_yield();
}
```

Condition variables are used to wait until a particular condition predicate becomes true. This condition predicate is set by another thread, usually the one that signals the condition.

Condition wait semantics

A condition predicate must be protected by a mutex. When waiting for a condition, the wait subroutine (either the **pthread_cond_wait** or **pthread_cond_timedwait** subroutine) atomically unlocks the mutex and blocks the thread. When the condition is signaled, the mutex is relocked and the wait subroutine returns. It is important to note that when the subroutine returns without error, the predicate may still be false. The reason is that more than one thread may be awoken: either a thread called the **pthread_cond_broadcast** subroutine, or an unavoidable race between two processors simultaneously woke two threads. The first thread locking the mutex will block all other awoken threads in the wait subroutine until the mutex is unlocked by the program. Thus, the predicate may have changed when the second thread gets the mutex and returns from the wait subroutine.
In general, whenever a condition wait returns, the thread should reevaluate the predicate to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait subroutine does not imply that the predicate is either true or false.
It is recommended that a condition wait be enclosed in a "while loop" that checks the predicate. Basic implementation of a condition wait is shown in the following code fragment:

```
 pthread_mutex_lock(&condition_lock);
while (condition_predicate == 0)
       pthread_cond_wait(&condition_variable, &condition_lock);
...
pthread_mutex_unlock(&condition_lock);
```

Timed wait semantics

When the **pthread_cond_timedwait** subroutine returns with the timeout error, the predicate may be true, due to another unavoidable race between the expiration of the timeout and the predicate state change.

Just as for non-timed wait, the thread should reevaluate the predicate when a timeout occurred to determine whether it should declare a timeout or should proceed anyway. It is recommended that you carefully check all possible cases when the **pthread_cond_timedwait** subroutine returns. The following code fragment shows how such checking could be implemented in a robust program: `int result = CONTINUE_LOOP;`

```
pthread_mutex_lock(&condition_lock);
while (result == CONTINUE_LOOP) {
       switch (pthread_cond_timedwait(&condition_variable,
               &condition_lock, &timeout)) {

               case 0:
               if (condition_predicate)
                       result = PROCEED;
               break;

               case ETIMEDOUT:
               result = condition_predicate ? PROCEED : TIMEOUT;
               break;

               default:
               result = ERROR;
               break;
       }
}

...
pthread_mutex_unlock(&condition_lock);
```

The **result** variable can be used to choose an action. The statements preceding the unlocking of the mutex should be done as soon as possible, because a mutex should not be held for long periods of time.

Specifying an absolute date in the *timeout* parameter allows easy implementation of real-time behavior. An absolute timeout need not be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait. For cases where the system clock is advanced discontinuously by an operator, using an absolute timeout ensures that the timed wait will end as soon as the system time specifies a date later than the *timeout* parameter.

Condition variables usage example

The following example provides the source code for a synchronization point routine. A *synchronization point* is a given point in a program where different threads must wait until all threads (or at least a certain number of threads) have reached that point.

A synchronization point can simply be implemented by a counter, which is protected by a lock, and a condition variable. Each thread takes the lock, increments the counter, and waits for the condition to be signaled if the counter did not reach its maximum. Otherwise, the condition is broadcast, and all threads can proceed. The last thread that calls the routine broadcasts the condition. `#define SYNC_MAX_COUNT  10`

```
void SynchronizationPoint()
{
	/* use static variables to ensure initialization */
	static mutex_t sync_lock = PTHREAD_MUTEX_INITIALIZER;
	static cond_t  sync_cond = PTHREAD_COND_INITIALIZER;
	static int sync_count = 0;

	/* lock the access to the count */
	pthread_mutex_lock(&sync_lock);

	/* increment the counter */
	sync_count++;

	/* check if we should wait or not */
	if (sync_count < SYNC_MAX_COUNT)

	     /* wait for the others */
	     pthread_cond_wait(&sync_cond, &sync_lock);

	else

	    /* broadcast that everybody reached the point */
	    pthread_cond_broadcast(&sync_cond);

	/* unlocks the mutex - otherwise only one thread
	        will be able to return from the routine! */
	pthread_mutex_unlock(&sync_lock);
}
```

This routine has some limitations: it can be used only once, and the number of threads that will call the routine is coded by a symbolic constant. However, this example shows a basic usage of condition variables. For more complex usage examples. For more complex usage examples, see Creating Complex Synchronization Objects.

Parent topic: Multithreaded programming

Related concepts:
Joining threads

Using read-write locks

In many situations, data is read more often than it is modified or written.
In these cases, you can allow threads to read concurrently while holding the lock and allow only one thread to hold the lock when data is modified. A multiple-reader single-writer lock (or read-write lock) does this. A read-write lock is acquired either for reading or writing, and then is released. The thread that acquires the read-write lock must be the one that releases it.

Read-write attributes object

The **pthread_rwlockattr_init** subroutine initializes a read-write lock attributes object (**attr**). The default value for all attributes is defined by the implementation.
Unexpected results can occur if the **pthread_rwlockattr_init** subroutine specifies an already-initialized read-write lock attributes object.
The following examples illustrate how to call the **pthread_rwlockattr_init** subroutine with the **attr** object: `pthread_rwlockattr_t attr;`
and: `pthread_rwlockattr_init(&attr);`

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) does not affect any previously initialized read-write locks.
The **pthread_rwlockattr_destroy** subroutine destroys a read-write lock attributes object. Unexpected results can occur if the object is used before it is reinitialized by another call to the **pthread_rwlockattr_init** subroutine. An implementation can cause the **pthread_rwlockattr_destroy** subroutine to set the object referenced by the **attr** object to an invalid value.

Creating and Destroying Read-Write Locks

The **pthread_rwlock_init** subroutine initializes the read-write lock referenced by the **rwlock** object with the attributes referenced by the **attr** object. If the **attr** object is NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked. After initialized, the lock can be used any number of times without being reinitialized.
Unexpected results can occur if the call to the **pthread_rwlock_init** subroutine is called specifying an already initialized read-write lock, or if a read-write lock is used without first being initialized.
If the **pthread_rwlock_init** subroutine fails, the **rwlock** object is not initialized and the contents are undefined.
The **pthread_rwlock_destroy** subroutine destroys the read-write lock object referenced by the **rwlock** object and releases any resources used by the lock.
Unexpected results can occur in any of the following situations:

- If the lock is used before it is reinitialized by another call to the **pthread_rwlock_init** subroutine.
- An implementation can cause the **pthread_rwlock_destroy** subroutine to set the object referenced by the **rwlock** object to an invalid value. Unexpected results can occur if **pthread_rwlock_destroy** is called when any thread holds the **rwlock** object.
- Attempting to destroy an uninitialized read-write lock results in unexpected results. A destroyed read-write lock object can be reinitialized using the **pthread_rwlock_init** subroutine. Unexpected results can occur if the read-write

lock object is referenced after it has been destroyed.

In cases where default read-write lock attributes are appropriate, use the **PTHREAD_RWLOCK_INITIALIZER** macro to initialize read-write locks that are statically allocated. For example: `pthread_rwlock_t rwlock1 = PTHREAD_RWLOCK_INITIALIZER;` The effect is similar to dynamic initialization using a call to the **pthread_rwlock_init** subroutine with the *attr* parameter specified as NULL, except that no error checks are performed. For example: `pthread_rwlock_init(&rwlock2, NULL);`

The following example illustrates how to use the **pthread_rwlock_init** subroutine with the attr parameter initialized. For an example of how to initialize the *attr* parameter, see Read-Write Attributes Object. `pthread_rwlock_init(&rwlock, &attr);`

Locking a read write lock object for reading

The **pthread_rwlock_rdlock** subroutine applies a read lock to the read-write lock referenced by the **rwlock** object. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread does not return from the **pthread_rwlock_rdlock** call until it can acquire the lock. Results are undefined if the calling thread holds a write lock on the **rwlock** object at the time the call is made.

A thread may hold multiple concurrent read locks on the **rwlock** object (that is, successfully call the **pthread_rwlock_rdlock** subroutine *n* times). If so, the thread must perform matching unlocks (that is, it must call the **pthread_rwlock_unlock** subroutine *n* times).

The **pthread_rwlock_tryrdlock** subroutine applies a read lock similar to the **pthread_rwlock_rdlock** subroutine with the exception that the subroutine fails if any thread holds a write lock on the **rwlock** object or there are writers blocked on the **rwlock** object. Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler, the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

Locking a read-write lock object for writing

The **pthread_rwlock_wrlock** subroutine applies a write lock to the read-write lock referenced by the **rwlock** object. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock on the **rwlock** object. Otherwise, the thread does not return from the **pthread_rwlock_wrlock** call until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

The **pthread_rwlock_trywrlock** subroutine applies a write lock similar to the **pthread_rwlock_wrlock** subroutine, with the exception that the function fails if any thread currently holds **rwlock** for reading or writing. Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler, the thread resumes waiting for the read-write lock for

writing as if it was not interrupted.

Sample read-write lock programs

The following sample programs demonstrate how to use locking subroutines. To run these programs, you need the **check.h** file and **makefile**.

check.h file: `#include stdio.h`

```
#include stdio.h
#include stdio.h
#include stdio.h

/* Simple function to check the return code and exit the program
  if the function call failed
  */
static void compResults(char *string, int rc) {
 if (rc) {
   printf("Error on : %s, rc=%d",
          string, rc);
   exit(EXIT_FAILURE);
 }
 return;
}
```

Make file: `cc_R = xlc_r`

```
TARGETS = test01 test02 test03

OBJS = test01.o test02.o test03.o

SRCS = $(OBJS:.o=.c)

$(TARGETS): $(OBJS)
   $(CC_R) -o $@ $@.o

clean:
   rm $(OBJS) $(TARGETS)

run:
   test01
   test02
   test03
```

Single-thread example

The following example uses the **pthread_rwlock_tryrdlock** subroutine with a single thread. For an example of using the **pthread_rwlock_tryrdlock** subroutine with multiple threads, see Multiple-Thread Example.

`Example: test01.c`

```
#define _MULTI_THREADED
#include pthread.h
```

```
#include stdio.h
#include "check.h"

pthread_rwlock_t       rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *rdlockThread(void *arg)
{
 int             rc;
 int             count=0;

 printf("Entered thread, getting read lock with mp wait\n");
 Retry:
 rc = pthread_rwlock_tryrdlock(&rwlock);
 if (rc == EBUSY) {
   if (count >= 10) {
     printf("Retried too many times, failure!\n");

     exit(EXIT_FAILURE);
   }
   ++count;
   printf("Could not get lock, do other work, then RETRY...\n");
   sleep(1);
   goto Retry;
 }
 compResults("pthread_rwlock_tryrdlock() 1\n", rc);

 sleep(2);

 printf("unlock the read lock\n");
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);

 printf("Secondary thread complete\n");
 return NULL;
}

int main(int argc, char **argv)
{
 int                   rc=0;
 pthread_t             thread;

 printf("Enter test case - %s\n", argv[0]);

 printf("Main, get the write lock\n");
 rc = pthread_rwlock_wrlock(&rwlock);
 compResults("pthread_rwlock_wrlock()\n", rc);

 printf("Main, create the try read lock thread\n");
```

```
 rc = pthread_create(&thread, NULL, rdlockThread, NULL);
 compResults("pthread_create\n", rc);

 printf("Main, wait a bit holding the write lock\n");
 sleep(5);

 printf("Main, Now unlock the write lock\n");
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);

 printf("Main, wait for the thread to end\n");
 rc = pthread_join(thread, NULL);
 compResults("pthread_join\n", rc);

 rc = pthread_rwlock_destroy(&rwlock);
 compResults("pthread_rwlock_destroy()\n", rc);
 printf("Main completed\n");
 return 0;
}
```

The output for this sample program will be similar to the following:

```
Enter test case - ./test01
Main, get the write lock
Main, create the try read lock thread
Main, wait a bit holding the write lock

Entered thread, getting read lock with mp wait
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Main, Now unlock the write lock
Main, wait for the thread to end
unlock the read lock
Secondary thread complete
Main completed
```

Multiple-thread example

The following example uses the **pthread_rwlock_tryrdlock** subroutine with multiple threads. For an example of using the **pthread_rwlock_tryrdlock** subroutine with a single thread, see Single-Thread Example.

```
Example: test02.c

#define _MULTI_THREADED
#include pthread.h
#include stdio.h
#include "check.h"

pthread_rwlock_t       rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

```
void *wrlockThread(void *arg)
{
 int             rc;
 int             count=0;

 printf("%.8x: Entered thread, getting write lock\n",
        pthread_self());
 Retry:
 rc = pthread_rwlock_trywrlock(&rwlock);
 if (rc == EBUSY) {
   if (count >= 10) {
     printf("%.8x: Retried too many times, failure!\n",
            pthread_self());
     exit(EXIT_FAILURE);
   }

   ++count;
   printf("%.8x: Go off an do other work, then RETRY...\n",
          pthread_self());
   sleep(1);
   goto Retry;
 }
 compResults("pthread_rwlock_trywrlock() 1\n", rc);
 printf("%.8x: Got the write lock\n", pthread_self());

 sleep(2);

 printf("%.8x: Unlock the write lock\n",
        pthread_self());
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);

 printf("%.8x: Secondary thread complete\n",
        pthread_self());
 return NULL;
}

int main(int argc, char **argv)
{
 int                   rc=0;
 pthread_t             thread, thread2;

 printf("Enter test case - %s\n", argv[0]);

 printf("Main, get the write lock\n");
 rc = pthread_rwlock_wrlock(&rwlock);
 compResults("pthread_rwlock_wrlock()\n", rc);
```

```
 printf("Main, create the timed write lock threads\n");
 rc = pthread_create(&thread, NULL, wrlockThread, NULL);
 compResults("pthread_create\n", rc);

 rc = pthread_create(&thread2, NULL, wrlockThread, NULL);
 compResults("pthread_create\n", rc);

 printf("Main, wait a bit holding this write lock\n");
 sleep(1);

 printf("Main, Now unlock the write lock\n");
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);

 printf("Main, wait for the threads to end\n");
 rc = pthread_join(thread, NULL);
 compResults("pthread_join\n", rc);

 rc = pthread_join(thread2, NULL);
 compResults("pthread_join\n", rc);

 rc = pthread_rwlock_destroy(&rwlock);
 compResults("pthread_rwlock_destroy()\n", rc);
 printf("Main completed\n");
 return 0;
}
```

The output for this sample program will be similar to the following:

```
Enter test case - ./test02
Main, get the write lock
Main, create the timed write lock threads
Main, wait a bit holding this write lock
00000102: Entered thread, getting write lock
00000102: Go off an do other work, then RETRY...
00000203: Entered thread, getting write lock
00000203: Go off an do other work, then RETRY...
Main, Now unlock the write lock
Main, wait for the threads to end
00000102: Got the write lock
00000203: Go off an do other work, then RETRY...
00000203: Go off an do other work, then RETRY...
00000102: Unlock the write lock
00000102: Secondary thread complete
00000203: Got the write lock
00000203: Unlock the write lock
00000203: Secondary thread complete
Main completed
```

Read-write read-lock example

The following example uses the **pthread_rwlock_rdlock** subroutine to implement read-write read locks:

Example: test03.c

```
#define _MULTI_THREADED
#include pthread.h
#include stdio.h
#include "check.h"

pthread_rwlock_t       rwlock;

void *rdlockThread(void *arg)
{
 int rc;

 printf("Entered thread, getting read lock\n");
 rc = pthread_rwlock_rdlock(&rwlock);
 compResults("pthread_rwlock_rdlock()\n", rc);
 printf("got the rwlock read lock\n");

 sleep(5);

 printf("unlock the read lock\n");
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);
 printf("Secondary thread unlocked\n");
 return NULL;
}

void *wrlockThread(void *arg)
{
 int rc;

 printf("Entered thread, getting write lock\n");
 rc = pthread_rwlock_wrlock(&rwlock);
 compResults("pthread_rwlock_wrlock()\n", rc);

 printf("Got the rwlock write lock, now unlock\n");
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);
 printf("Secondary thread unlocked\n");
 return NULL;
}




int main(int argc, char **argv)
{
```

```
 int                   rc=0;
 pthread_t             thread, thread1;

 printf("Enter test case - %s\n", argv[0]);

 printf("Main, initialize the read write lock\n");
 rc = pthread_rwlock_init(&rwlock, NULL);
 compResults("pthread_rwlock_init()\n", rc);

 printf("Main, grab a read lock\n");
 rc = pthread_rwlock_rdlock(&rwlock);
 compResults("pthread_rwlock_rdlock()\n",rc);

 printf("Main, grab the same read lock again\n");
 rc = pthread_rwlock_rdlock(&rwlock);
 compResults("pthread_rwlock_rdlock() second\n", rc);

 printf("Main, create the read lock thread\n");
 rc = pthread_create(&thread, NULL, rdlockThread, NULL);
 compResults("pthread_create\n", rc);

 printf("Main - unlock the first read lock\n");
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);

 printf("Main, create the write lock thread\n");
 rc = pthread_create(&thread1, NULL, wrlockThread, NULL);
 compResults("pthread_create\n", rc);

 sleep(5);
 printf("Main - unlock the second read lock\n");
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);

 printf("Main, wait for the threads\n");
 rc = pthread_join(thread, NULL);
 compResults("pthread_join\n", rc);

 rc = pthread_join(thread1, NULL);
 compResults("pthread_join\n", rc);

 rc = pthread_rwlock_destroy(&rwlock);
 compResults("pthread_rwlock_destroy()\n", rc);

 printf("Main completed\n");
 return 0;
}
```

The output for this sample program will be similar to the following:

```
$ ./test03
Enter test case - ./test03
Main, initialize the read write lock
Main, grab a read lock
Main, grab the same read lock again
Main, create the read lock thread
Main - unlock the first read lock
Main, create the write lock thread
Entered thread, getting read lock
got the rwlock read lock
Entered thread, getting write lock
Main - unlock the second read lock
Main, wait for the threads
unlock the read lock
Secondary thread unlocked
Got the rwlock write lock, now unlock
Secondary thread unlocked
Main completed
```

Parent topic: Multithreaded programming

Joining threads

Joining a thread means waiting for it to terminate, which can be seen as a specific usage of condition variables.

Waiting for a thread

Using the **pthread_join** subroutine alows a thread to wait for another thread to terminate. More complex conditions, such as waiting for multiple threads to terminate, can be implemented by using condition variables.

Calling the pthread_join subroutine

The **pthread_join** subroutine blocks the calling thread until the specified thread terminates. The target thread (the thread whose termination is awaited) must not be detached. If the target thread is already terminated, but not detached, the **pthread_join** subroutine returns immediately. After a target thread has been joined, it is automatically detached, and its storage can be reclaimed.

The following table indicates the possible cases when a thread calls the **pthread_join** subroutine, depending on the **state** and the **detachstate** attribute of the target thread.

Target State	Undetached target	Detached target
Target is still running	The caller is blocked until the target is terminated.	The call returns immediately, indicating an error.
Target is terminated	The call returns immediately, indicating a successful completion.	

Multiple joins

Several threads can join the same target thread, if the target is not detached. The success of this operation depends on the order of the calls to the **pthread_join** subroutine and the moment when the target thread terminates.

- Any call to the **pthread_join** subroutine occurring before the target thread's termination blocks the calling thread.
- When the target thread terminates, all blocked threads are awoken, and the target thread is automatically detached.
- Any call to the **pthread_join** subroutine occurring after the target thread's termination will fail, because the thread is detached by the previous join.
- If no thread called the **pthread_join** subroutine before the target thread's termination, the first call to the **pthread_join** subroutine will return immediately, indicating a successful completion, and any further call will fail.

Join example

In the following example, the program ends after exactly five messages display in each language. This is done by blocking the initial thread until the "writer" threads exit.

```
#include <pthread.h>    /* include file for pthreads - the 1st */
#include <stdio.h>      /* include file for printf()           */


void *Thread(void *string)
{
       int i;
```

```
        /* writes five messages and exits */
        for (i=0; i<5; i++)
                printf("%s\n", (char *)string);
        pthread_exit(NULL);
}

int main()
{
        char *e_str = "Hello!";
        char *f_str = "Bonjour !";

        pthread_attr_t attr;
        pthread_t e_th;
        pthread_t f_th;

        int rc;

        /* creates the right attribute */
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,
                PTHREAD_CREATE_UNDETACHED);

        /* creates both threads */
        rc = pthread_create(&e_th, &attr, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, &attr, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        pthread_attr_destroy(&attr);

        /* joins the threads */
        pthread_join(e_th, NULL);
        pthread_join(f_th, NULL);

        pthread_exit(NULL);
}
```

A thread cannot join itself because a deadlock would occur and it is detected by the library. However, two threads may try to join each other. They will deadlock, but this situation is not detected by the library.

Returning information from a thread

The **pthread_join** subroutine also allows a thread to return information to another thread. When a thread calls the **pthread_exit** subroutine or when it returns from its entry-point routine, it returns a pointer (see Exiting a Thread). This pointer is stored as long as the thread is not detached, and the **pthread_join** subroutine can return it. For example, a multithreaded **grep** command may choose the implementation in the following example. In this example, the initial thread creates one thread per file to

scan, each thread having the same entry point routine. The initial thread then waits for all threads to be terminated. Each "scanning" thread stores the found lines in a dynamically allocated buffer and returns a pointer to this buffer. The initial thread prints each buffer and releases it.

```
/* "scanning" thread */
...
buffer = malloc(...);
       /* finds the search pattern in the file
          and stores the lines in the buffer   */
return (buffer);


/* initial thread */
...
for (/* each created thread */) {
       void *buf;
       pthread_join(thread, &buf);
       if (buf != NULL) {
              /* print all the lines in the buffer,
                      preceded by the filename of the thread */
              free(buf);
       }
}
...
```

If the target thread is canceled, the **pthread_join** subroutine returns a value of -1 cast into a pointer (see Canceling a Thread). Because -1 cannot be a pointer value, getting -1 as returned pointer from a thread means that the thread was canceled. The returned pointer can point to any kind of data. The pointer must still be valid after the thread was terminated and its storage reclaimed. Therefore, avoid returning a value, because the destructor routine is called when the thread's storage is reclaimed.
Returning a pointer to dynamically allocated storage to several threads needs special consideration. Consider the following code fragment:

```
void *returned_data;
...
pthread_join(target_thread, &returned_data);
/* retrieves information from returned_data */
free(returned_data);
```

The **returned_data** pointer is freed when it is executed by only one thread. If several threads execute the above code fragment concurrently, the **returned_data** pointer is freed several times; a situation that must be avoided. To prevent this, use a mutex-protected flag to signal that the **returned_data** pointer was freed. The following line from the previous example:

```
free(returned_data);
```

would be replaced by the following lines, where a mutex can be used for locking the access to the critical region (assuming the *flag* variable is initially 0):

```
/* lock - entering a critical region, no other thread should
              run this portion of code concurrently */
if (!flag) {
```

```
        free(returned_data);
        flag = 1;
}
/* unlock - exiting the critical region */
```

Locking access to the critical region ensures that the **returned_data** pointer is freed only once.
When returning a pointer to dynamically allocated storage to several threads all executing different code, you must ensure that exactly one thread frees the pointer.

Parent topic: Multithreaded programming

Related concepts:
Using condition variables
Thread-specific data

Scheduling threads

Threads can be scheduled, and the threads library provides several facilities to handle and control the scheduling of threads.
It also provides facilities to control the scheduling of threads during synchronization operations such as locking a mutex. Each thread has its own set of scheduling parameters. These parameters can be set using the thread attributes object before the thread is created. The parameters can also be dynamically set during the thread's execution.
Controlling the scheduling of a thread can be a complicated task. Because the scheduler handles all threads system wide, the scheduling parameters of a thread interact with those of all other threads in the process and in the other processes.
The following facilities are the first to be used if you want to control the scheduling of a thread.
The threads library allows the programmer to control the execution scheduling of the threads in the following ways:
- By setting scheduling attributes when creating a thread
- By dynamically changing the scheduling attributes of a created thread
- By defining the effect of a mutex on the thread's scheduling when creating a mutex (known as *synchronization scheduling*)
- By dynamically changing the scheduling of a thread during synchronization operations (known as *synchronization scheduling*)

Scheduling parameters

A thread has the following scheduling parameters:

Parameter	Description
scope	The contention scope of a thread is defined by the thread model used in the threads library.
policy	The scheduling policy of a thread defines how the scheduler treats the thread after it gains control of the CPU.
priority	The scheduling priority of a thread defines the relative importance of the work being done by each thread.

The scheduling parameters can be set before the thread's creation or during the thread's execution. In general, controlling the scheduling parameters of threads is important only for threads that are CPU-intensive. Thus, the threads library provides default values that are sufficient for most cases.

Using the inheritsched attribute

The **inheritsched** attribute of the thread attributes object specifies how the thread's scheduling attributes will be defined. The following values are valid:

Values	Description

PTHREAD_INHERIT_SCHED	Specifies that the new thread will get the scheduling attributes (**schedpolicy** and **schedparam** attributes) of its creating thread. Scheduling attributes defined in the attributes object are ignored.
PTHREAD_EXPLICIT_SCHED	Specifies that the new thread will get the scheduling attributes defined in this attributes object.

The default value of the **inheritsched** attribute is PTHREAD_INHERIT_SCHED. The attribute is set by calling the **pthread_attr_setinheritsched** subroutine. The current value of the attribute is returned by calling the **pthread_attr_getinheritsched** subroutine.
To set the scheduling attributes of a thread in the thread attributes object, the **inheritsched** attribute must first be set to PTHREAD_EXPLICIT_SCHED. Otherwise, the attributes-object scheduling attributes are ignored.

Scheduling policy and priority

The threads library provides the following scheduling policies:

Library	Description
SCHED_FIFO	First-in first-out (FIFO) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run to completion in FIFO order.
SCHED_RR	Round-robin (RR) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run for a fixed time slice in FIFO order.
SCHED_OTHER	Default AIX® scheduling. Each thread has an initial priority that is dynamically modified by the scheduler, according to the thread's activity; thread execution is time-sliced. On other systems, this scheduling policy may be different.

In versions of AIX prior to 5.3, changing the priority of a thread when setting its scheduling policy to **SCHED_OTHER** is not permitted. In this case, the kernel directly manages the priority, and the only legal value that can be passed to the **pthread_setschedparam** subroutine is the **DEFAULT_PRIO** value. The **DEFAULT_PRIO** value is defined in **pthread.h** file as 1, and any other passed values are ignored.
Beginning with AIX 5.3, you can change the priority of a thread when you set its scheduling policy to **SCHED_OTHER**. The legal values that can be passed to the **pthread_setschedparam** subroutine are from 40 to 80, however, only privileged users can set a priority greater than 60. A priority in the range of 1 to 39 provides the same priority as that of 40, and a priority in the range of 81 to 127 provides the same priority as that of 80.
Note: In AIX, the kernel inverts the priority levels. For the AIX kernel, the priority is in the range from 0 to 127, where 0 is the most favored priority and 127 the least

favored priority. Commands, such as the **ps** command, report the kernel priority. The threads library handles the priority through a **sched_param** structure, defined in the **sys/sched.h** header file. This structure contains the following fields:

Fields	Description
sched_priority	Specifies the priority.
sched_policy	This field is ignored by the threads library. Do not use.

Setting the scheduling policy and priority at creation time

The scheduling policy can be set when creating a thread by setting the **schedpolicy** attribute of the thread attributes object. The **pthread_attr_setschedpolicy** subroutine sets the scheduling policy to one of the previously defined scheduling policies. The current value of the **schedpolicy** attribute of a thread attributes object can be obtained by using the **pthread_attr_getschedpolicy** subroutine.

The scheduling priority can be set at creation time of a thread by setting the **schedparam** attribute of the thread attributes object. The **pthread_attr_setschedparam** subroutine sets the value of the **schedparam** attribute, copying the value of the specified structure. The **pthread_attr_getschedparam** subroutine gets the **schedparam** attribute.

In the following code fragment, a thread is created with the round-robin scheduling policy, using a priority level of 3:

```
sched_param schedparam;

schedparam.sched_priority = 3;

pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_RR);
pthread_attr_setschedparam(&attr, &schedparam);

pthread_create(&thread, &attr, &start_routine, &args);
pthread_attr_destroy(&attr);
```

For more information about the **inheritsched** attribute, see Using the inheritsched Attribute.

Setting the scheduling attributes at execution time

The **pthread_getschedparam** subroutine returns the **schedpolicy** and **schedparam** attributes of a thread. These attributes can be set by calling the **pthread_setschedparam** subroutine. If the target thread is currently running on a processor, the new scheduling policy and priority will be implemented the next time the thread is scheduled. If the target thread is not running, it can be scheduled immediately at the end of the subroutine call.

For example, consider a thread T that is currently running with round-robin policy at the moment the **schedpolicy** attribute of T is changed to FIFO. T will run until the end of its time slice, at which time its scheduling attributes are then re-evaluated. If no threads have higher priority, T will be rescheduled, even before other threads having the same priority. Consider a second example where a low-priority thread is

not running. If this thread's priority is raised by another thread calling the **pthread_setschedparam** subroutine, the target thread will be scheduled immediately if it is the highest priority runnable thread.
Note: Both subroutines use a *policy* parameter and a **sched_param** structure. Although this structure contains a sched_policy field, programs should not use it. The subroutines use the *policy* parameter to pass the scheduling policy, and the subroutines then ignore the sched_policy field.

Scheduling-policy considerations

Applications should use the default scheduling policy, unless a specific application requires the use of a fixed-priority scheduling policy. Consider the following points about using the nondefault policies:

- Using the round-robin policy ensures that all threads having the same priority level will be scheduled equally, regardless of their activity. This can be useful in programs where threads must read sensors or write actuators.
- Using the FIFO policy should be done with great care. A thread running with FIFO policy runs to completion, unless it is blocked by some calls, such as performing input and output operations. A high-priority FIFO thread may not be preempted and can affect the global performance of the system. For example, threads doing intensive calculations, such as inverting a large matrix, should never run with FIFO policy.

The setting of scheduling policy and priority is also influenced by the contention scope of threads. Using the FIFO or the round-robin policy may not always be allowed.

sched_yield subroutine

The **sched_yield** subroutine is the equivalent for threads of the **yield** subroutine. The **sched_yield** subroutine forces the calling thread to relinquish the use of its processor and gives other threads an opportunity to be scheduled. The next scheduled thread may belong to the same process as the calling thread or to another process. Do not use the **yield** subroutine in a multithreaded program. The interface **pthread_yield** subroutine is not available in Single UNIX Specification, Version 2.

- List of scheduling subroutines
  This section lists scheduling subroutines.

Parent topic: Multithreaded programming

Related concepts:
Synchronization scheduling
List of scheduling subroutines
Developing multithreaded programs

List of scheduling subroutines

This section lists scheduling subroutines.

Subroutine	Description
pthread_attr_getschedparam	Returns the value of the **schedparam** attribute of a thread attributes object.
pthread_attr_setschedparam	Sets the value of the **schedparam** attribute of a thread attributes object.
pthread_getschedparam	Returns the value of the **schedpolicy** and **schedparam** attributes of a thread.
sched_yield	Forces the calling thread to relinquish use of its processor.

Parent topic: Scheduling threads

Related concepts:
Scheduling threads

Contention scope and concurrency level

The *contention scope* of a user thread defines how it is mapped to a kernel thread . The threads library defines the following contention scopes:

- PTHREAD_SCOPE_PROCESS

- Process contention scope, sometimes called *local contention scope*. Specifies that the thread will be scheduled against all other local contention scope threads in the process. A process-contention-scope user thread is a user thread that shares a kernel thread with other process-contention-scope user threads in the process. All user threads in an M:1 thread model have process contention scope.

- PTHREAD_SCOPE_SYSTEM

- System contention scope, sometimes called *global contention scope*. Specifies that the thread will be scheduled against all other threads in the system and is directly mapped to one kernel thread. All user threads in a 1:1 thread model have system contention scope.

In an M:N thread model, user threads can have either system or process contention scope. Therefore, an M:N thread model is often referred as a *mixed-scope* model. The *concurrency level* is a property of M:N threads libraries. It defines the number of virtual processors used to run the process-contention scope user threads. This number cannot exceed the number of process-contention-scope user threads and is usually dynamically set by the threads library. The system also sets a limit to the number of available kernel threads.

Setting the contention scope

The contention scope can only be set before a thread is created by setting the contention-scope attribute of a thread attributes object. The **pthread_attr_setscope** subroutine sets the value of the attribute; the **pthread_attr_getscope** returns it. The contention scope is only meaningful in a mixed-scope M:N library implementation. A **TestImplementation** routine could be written as follows:

```
int TestImplementation()
{
	pthread_attr_t a;
	int result;

	pthread_attr_init(&a);
	switch (pthread_attr_setscope(&a, PTHREAD_SCOPE_PROCESS))
	{
		case 0:		result = LIB_MN; break;
		case ENOTSUP:	result = LIB_11; break;
		case ENOSYS:	result = NO_PRIO_OPTION; break;
		default:	result = ERROR; break;
	}

	pthread_attr_destroy(&a);
	return result;
}
```

Impacts of contention scope on scheduling

The contention scope of a thread influences its scheduling. Each contention-scope thread is bound to one kernel thread. Thus, changing the scheduling policy and priority of a global user thread results in changing the scheduling policy and priority of the underlying kernel thread.
In AIX®, only kernel threads with root authority can use a fixed-priority scheduling policy (FIFO or round-robin). The following code will always return the **EPERM** error code if the calling thread has system contention scope but does not have root authority. This code would not fail, if the calling thread had process contention scope.

```
schedparam.sched_priority = 3;
pthread_setschedparam(pthread_self(), SCHED_FIFO, schedparam);
```

Note: Root authority is not required to control the scheduling parameters of user threads having process contention scope.
Local user threads can set any scheduling policy and priority, within the valid range of values. However, two threads having the same scheduling policy and priority but having different contention scope will not be scheduled in the same way. Threads having process contention scope are executed by kernel threads whose scheduling parameters are set by the library.

Parent topic: Multithreaded programming

Related concepts:
Understanding threads and processes

Synchronization scheduling

Programmers can control the execution scheduling of threads when there are constraints, especially time constraints, that require certain threads to be executed faster than other ones.

Synchronization objects, such as mutexes, may block even high-priority threads. In some cases, undesirable behavior, known as *priority inversion*, may occur. The threads library provides the *mutex protocols* facility to avoid priority inversions.

Synchronization scheduling defines how the execution scheduling, especially the priority, of a thread is modified by holding a mutex. This allows custom-defined behavior and avoids priority inversions. It is useful when using complex locking schemes. Some implementations of the threads library do not provide synchronization scheduling.

Priority inversion

Priority inversion occurs when a low-priority thread holds a mutex, blocking a high-priority thread. Due to its low priority, the mutex owner may hold the mutex for an unbounded duration. As a result, it becomes impossible to guarantee thread deadlines.

The following example illustrates a typical priority inversion. In this example, the case of a uniprocessor system is considered. Priority inversions also occur on multiprocessor systems in a similar way.

In our example, a mutex *M* is used to protect some common data. Thread *A* has a priority level of 100 and is scheduled very often. Thread *B* has a priority level of 20 and is a background thread. Other threads in the process have priority levels near 60. A code fragment from thread *A* is as follows:

```
pthread_mutex_lock(&M);            /* 1 */
...
pthread_mutex_unlock(&M);
```

A code fragment from thread *B* is as follows:

```
pthread_mutex_lock(&M);          /* 2 */
...
fprintf(...);                    /* 3 */
...
pthread_mutex_unlock(&M);
```

Consider the following execution chronology. Thread *B* is scheduled and executes line 2. While executing line 3, thread *B* is preempted by thread *A*. Thread *A* executes line 1 and is blocked, because the mutex *M* is held by thread *B*. Thus, other threads in the process are scheduled. Because thread *B* has a very low priority, it may not be rescheduled for a long period, blocking thread *A*, although thread *A* has a very high priority.

Mutex protocols

To avoid priority inversions, the following mutex protocols are provided by the threads library:

- Priority inheritance protocol

- Sometimes called *basic priority inheritance protocol*. In the priority inheritance protocol, the mutex holder inherits the priority of the highest-priority blocked thread. When a thread tries to lock a mutex using this protocol and is blocked, the mutex owner temporarily receives the blocked thread's priority, if that priority is higher than the owner's. It recovers its original priority when it unlocks

the mutex.

- Priority protection protocol

- Sometimes called *priority ceiling protocol emulation*. In the priority protection protocol, each mutex has a *priority ceiling*. It is a priority level within the valid range of priorities. When a thread owns a mutex, it temporarily receives the mutex priority ceiling, if the ceiling is higher than its own priority. It recovers its original priority when it unlocks the mutex. The priority ceiling should have the value of the highest priority of all threads that may lock the mutex. Otherwise, priority inversions or even deadlocks may occur, and the protocol would be inefficient.

Both protocols increase the priority of a thread holding a specific mutex, so that deadlines can be guaranteed. Furthermore, when correctly used, mutex protocols can prevent mutual deadlocks. Mutex protocols are individually assigned to mutexes.

Choosing a mutex protocol

The choice of a mutex protocol is made by setting attributes when creating a mutex. The mutex protocol is controlled through the protocol attribute. This attribute can be set in the mutex attributes object by using the **pthread_mutexattr_getprotocol** and **pthread_mutexattr_setprotocol** subroutines. The protocol attribute can have one of the following values:

Value	Description
PTHREAD_PRIO_DEFAULT	No value
PTHREAD_PRIO_NONE	Denotes no protocol.
PTHREAD_PRIO_INHERIT	Denotes the priority inheritance protocol.
PTHREAD_PRIO_PROTECT	Denotes the priority protection protocol.

Note: The behavior of **PTHREAD_PRIO_DEFAULT** is the same as the **PTHREAD_PRIO_INHERIT** attribute. With reference to the mutex locking, the threads acting with the default attribute will temporarily boost the priority of a mutex holder when a user is locked and has a higher priority than the owner. Therefore, there are only three behaviors that are possible, although there are four values for the possible priority in the attribute structure.

The priority protection protocol uses one additional attribute: the prioceiling attribute. This attribute contains the priority ceiling of the mutex. The prioceiling attribute can be controlled in the mutex attributes object, by using the **pthread_mutexattr_getprioceiling** and **pthread_mutexattr_setprioceiling** subroutines.

The prioceiling attribute of a mutex can also be dynamically controlled by using the **pthread_mutex_getprioceiling** and **pthread_mutex_setprioceiling** subroutines. When dynamically changing the priority ceiling of a mutex, the mutex is locked by the library; it should not be held by the thread calling the **pthread_mutex_setprioceiling** subroutine to avoid a deadlock. Dynamically setting the priority ceiling of a mutex can be useful when increasing the priority of a thread. The implementation of mutex protocols is optional. Each protocol is a POSIX option.

Inheritance or protection
Both protocols are similar and result in promoting the priority of the thread holding the mutex. If both protocols are available, programmers must choose a protocol. The choice depends on whether the priorities of the threads that will lock the mutex are available to the programmer who is creating the mutex. Typically, mutexes defined by a library and used by application threads will use the inheritance protocol, whereas mutexes created within the application program will use the protection protocol.
In performance-critical programs, performance considerations may also influence the choice. In most implementations, especially in AIX®, changing the priority of a thread results in making a system call. Therefore, the two mutex protocols differ in the amount of system calls they generate, as follows:
- Using the inheritance protocol, a system call is made each time a thread is blocked when trying to lock the mutex.
- Using the protection protocol, one system call is always made each time the mutex is locked by a thread.

In most performance-critical programs, the inheritance protocol should be chosen, because mutexes are low contention objects. Mutexes are not held for long periods of time; thus, it is not likely that threads are blocked when trying to lock them.

- List of synchronization subroutines
  This section lists synchronization subroutines.

Parent topic: Multithreaded programming

Related concepts:
Scheduling threads

List of synchronization subroutines

This section lists synchronization subroutines.

- **pthread_mutex_destroy**
  - Deletes a mutex.
- **pthread_mutex_init**
  - Initializes a mutex and sets its attributes.
- **PTHREAD_MUTEX_INITIALIZER**
  - Initializes a static mutex with default attributes.
- **pthread_mutex_lock** or **pthread_mutex_trylock**
  - Locks a mutex.
- **pthread_mutex_unlock**
  - Unlocks a mutex.
- **pthread_mutexattr_destroy**
  - Deletes a mutex attributes object.
- **pthread_mutexattr_init**
  - Creates a mutex attributes object and initializes it with default values.
- **pthread_cond_destroy**
  - Deletes a condition variable.
- **pthread_cond_init**
  - Initializes a condition variable and sets its attributes.
- **PTHREAD_COND_INITIALIZER**
  - Initializes a static condition variable with default attributes.
- **pthread_cond_signal** or **pthread_cond_broadcast**
  - Unblocks one or more threads blocked on a condition.
- **pthread_cond_wait** or **pthread_cond_timedwait**
  - Blocks the calling thread on a condition.
- **pthread_condattr_destroy**
  - Deletes a condition attributes object.
- **pthread_condattr_init**
  - Creates a condition attributes object and initializes it with default values.

Parent topic: Synchronization scheduling

One-time initializations

Some C libraries are designed for dynamic initialization, in which the global initialization for the library is performed when the first procedure in the library is called.
In a single-threaded program, this is usually implemented using a static variable whose value is checked on entry to each routine, as in the following code fragment:

```
 static int isInitialized = 0;
extern void Initialize();


int function()
{
       if (isInitialized == 0) {
               Initialize();
               isInitialized = 1;
       }
       ...
}
```

For dynamic library initialization in a multithreaded program, a simple initialization flag is not sufficient. This flag must be protected against modification by multiple threads simultaneously calling a library function. Protecting the flag requires the use of a mutex; however, mutexes must be initialized before they are used. Ensuring that the mutex is only initialized once requires a recursive solution to this problem.
To keep the same structure in a multithreaded program, use the **pthread_once** subroutine. Otherwise, library initialization must be accomplished by an explicit call to a library exported initialization function prior to any use of the library. The **pthread_once** subroutine also provides an alternative for initializing mutexes and condition variables.
Read the following to learn more about one-time initializations:

One-time initialization object

The uniqueness of the initialization is ensured by the one-time initialization object. It is a variable having the **pthread_once_t** data type. In AIX® and most other implementations of the threads library, the **pthread_once_t** data type is a structure. A one-time initialization object is typically a global variable. It must be initialized with the **PTHREAD_ONCE_INIT** macro, as in the following example:

```
 static pthread_once_t once_block = PTHREAD_ONCE_INIT;
```

The initialization can also be done in the initial thread or in any other thread. Several one-time initialization objects can be used in the same program. The only requirement is that the one-time initialization object be initialized with the macro.

One-time initialization routine

The **pthread_once** subroutine calls the specified initialization routine associated with the specified one-time initialization object if it is the first time it is called; otherwise, it does nothing. The same initialization routine must always be used with the same one-time initialization object. The initialization routine must have the following prototype: `void init_routine();`

The **pthread_once** subroutine does not provide a cancelation point. However, the

initialization routine may provide cancelation points, and, if cancelability is enabled, the first thread calling the **pthread_once** subroutine may be canceled during the execution of the initialization routine. In this case, the routine is not considered as executed, and the next call to the **pthread_once** subroutine would result in recalling the initialization routine.
It is recommended to use cleanup handlers in one-time initialization routines, especially when performing non-idempotent operations, such as opening a file, locking a mutex, or allocating memory.
One-time initialization routines can be used for initializing mutexes or condition variables or to perform dynamic initialization. In a multithreaded library, the code fragment shown above (void init_routine();) would be written as follows:

```
static
pthread_once_t once_block = PTHREAD_ONCE_INIT;

extern void Initialize();


int function()

{

       pthread_once(&once_block, Initialize);

       ...

}
```

Parent topic: Multithreaded programming

Related concepts:
Terminating threads
Thread-specific data
Writing reentrant and threadsafe code

Thread-specific data

Many applications require that certain data be maintained on a per-thread basis across function calls.

For example, a multithreaded **grep** command using one thread for each file must have thread-specific file handlers and list of found strings. The thread-specific data interface is provided by the threads library to meet these needs.

Thread-specific data may be viewed as a two-dimensional array of values, with keys serving as the row index and thread IDs as the column index. A thread-specific data *key* is an opaque object, of the **pthread_key_t** data type. The same key can be used by all threads in a process. Although all threads use the same key, they set and access different thread-specific data values associated with that key. Thread-specific data are void pointers, which allows referencing any kind of data, such as dynamically allocated strings or structures.

In the following figure, thread T2 has a thread-specific data value of 12 associated with the key K3. Thread T4 has the value of 2 associated with the same key.

Keys	**T1** Thread	**T2** Thread	**T3** Thread	**T4** Thread
K1	6	56	4	1
K2	87	21	0	9
K3	23	**12**	61	**2**
K4	11	76	47	88

Creating and destroying keys

Thread-specific data keys must be created before being used. Their values can be automatically destroyed when the corresponding threads terminate. A key can also be destroyed upon request to reclaim its storage.

Key creation

A thread-specific data key is created by calling the **pthread_key_create** subroutine. This subroutine returns a key. The thread-specific data is set to a value of **NULL** for all threads, including threads not yet created.

For example, consider two threads *A* and *B*. Thread *A* performs the following operations in chronological order:

1. Create a thread-specific data key *K*. Threads *A* and *B* can use the key *K*. The value for both threads is **NULL**.
2. Create a thread *C*. Thread *C* can also use the key *K*. The value for thread *C* is **NULL.**

The number of thread-specific data keys is limited to 450 per process. This number can be retrieved by the **PTHREAD_KEYS_MAX** symbolic constant.

The **pthread_key_create** subroutine must be called only once. Otherwise, two different keys are created. For example, consider the following code fragment:

```
 /* a global variable */
static pthread_key_t theKey;


/* thread A */
...
pthread_key_create(&theKey, NULL);   /* call 1 */
```

```
...

/* thread B */
...
pthread_key_create(&theKey, NULL);   /* call 2 */
...
```

In our example, threads *A* and *B* run concurrently, but call 1 happens before call 2. Call 1 will create a key *K1* and store it in the **theKey** variable. Call 2 will create another key *K2*, and store it also in the **theKey** variable, thus overriding *K1*. As a result, thread *A* will use *K2*, assuming it is *K1*. This situation should be avoided for the following reasons:

- Key K1 is lost, thus its storage will never be reclaimed until the process terminates. Because the number of keys is limited, you may not have enough keys.
- If thread *A* stores a thread-specific data using the **theKey** variable before call 2, the data will be bound to key K1. After call 2, the **theKey** variable contains K2; if thread *A* then tries to fetch its thread-specific data, it would always get **NULL**.

Ensuring that keys are created uniquely can be done in the following ways:

- Using the one-time initialization facility.
- Creating the key before the threads that will use it. This is often possible, for example, when using a pool of threads with thread-specific data to perform similar operations. This pool of threads is usually created by one thread, the initial (or another "driver") thread.

It is the programmer's responsibility to ensure the uniqueness of key creation. The threads library provides no way to check if a key has been created more than once.

Destructor routine

A destructor routine may be associated with each thread-specific data key. Whenever a thread is terminated, if there is non-**NULL,** thread-specific data for this thread bound to any key, the destructor routine associated with that key is called. This allows dynamically allocated thread-specific data to be automatically freed when the thread is terminated. The destructor routine has one parameter, the value of the thread-specific data.

For example, a thread-specific data key may be used for dynamically allocated buffers. A destructor routine should be provided to ensure that when the thread terminates the buffer is freed, the **free** subroutine can be used as follows:

```
pthread_key_create(&key, free);
```

More complex destructors may be used. If a multithreaded **grep** command, using a thread per file to scan, has thread-specific data to store a structure containing a work buffer and the thread's file descriptor, the destructor routine may be as follows:

```
 typedef struct {
       FILE *stream;
       char *buffer;
} data_t;
...

void destructor(void *data)
{
```

```
        fclose(((data_t *)data)->stream);
        free(((data_t *)data)->buffer);
        free(data);
        *data = NULL;
}
```

Destructor calls can be repeated up to four times.

Key destruction

A thread-specific data key can be destroyed by calling the **pthread_key_delete** subroutine. The **pthread_key_delete** subroutine does not actually call the destructor routine for each thread having data. After a data key is destroyed, it can be reused by another call to the **pthread_key_create** subroutine. Thus, the **pthread_key_delete** subroutine is useful especially when using many data keys. For example, in the following code fragment, the loop would never end: `/* bad example - do not write such code! */`

```
pthread_key_t key;


while (pthread_key_create(&key, NULL))
        pthread_key_delete(key);
```

Using thread-specific data

Thread-specific data is accessed using the **pthread_getspecific** and **pthread_setspecific** subroutines. The **pthread_getspecific** subroutine reads the value bound to the specified key and is specific to the calling thread; the **pthread_setspecific** subroutine sets the value.

Setting successive values

The value bound to a specific key should be a pointer, which can point to any kind of data. Thread-specific data is typically used for dynamically allocated storage, as in the following code fragment: `private_data = malloc(...);`

```
pthread_setspecific(key, private_data);
```

When setting a value, the previous value is lost. For example, in the following code fragment, the value of the **old** pointer is lost, and the storage it pointed to may not be recoverable: `pthread_setspecific(key, old);`

```
...
pthread_setspecific(key, new);
```

It is the programmer's responsibility to retrieve the old thread-specific data value to reclaim storage before setting the new value. For example, it is possible to implement a **swap_specific** routine in the following manner: `int`

```
swap_specific(pthread_key_t key, void **old_pt, void *new)
{
        *old_pt = pthread_getspecific(key);
        if (*old_pt == NULL)
                return -1;
        else
                return pthread_setspecific(key, new);
}
```

Such a routine does not exist in the threads library because it is not always necessary to retrieve the previous value of thread-specific data. Such a case occurs, for example, when thread-specific data are pointers to specific locations in a memory pool allocated by the initial thread.

Using destructor routines

When using dynamically allocated thread-specific data, the programmer must provide a destructor routine when calling the **pthread_key_create** subroutine. The programmer must also ensure that, when releasing the storage allocated for thread-specific data, the pointer is set to **NULL**. Otherwise, the destructor routine might be called with an illegal parameter. For example:

```
pthread_key_create(&key, free);
...


...
private_data = malloc(...);
pthread_setspecific(key, private_data);
...


/* bad example! */
...
pthread_getspecific(key, &data);
free(data);
...
```

When the thread terminates, the destructor routine is called for its thread-specific data. Because the value is a pointer to already released memory, an error can occur. To correct this, the following code fragment should be substituted:

```
 /* better example! */
...
pthread_getspecific(key, &data);
free(data);
pthread_setspecific(key, NULL);
...
```

When the thread terminates, the destructor routine is not called, because there is no thread-specific data.

Using non-pointer values

Although it is possible to store values that are not pointers, it is not recommended for the following reasons:

- Casting a pointer into a scalar type may not be portable.
- The **NULL** pointer value is implementation-dependent; several systems assign the **NULL** pointer a non-zero value.

If you are sure that your program will never be ported to another system, you may use integer values for thread-specific data.

Parent topic: Multithreaded programming

Related concepts:

Creating complex synchronization objects

The subroutines provided in the threads library can be used as primitives to build more complex synchronization objects.

Long Locks

The mutexes provided by the threads library are low-contention objects and should not be held for a very long time. Long locks are implemented with mutexes and condition variables, so that a long lock can be held for a long time without affecting the performance of the program. Long locks should not be used if cancelability is enabled.

A long lock has the **long_lock_t** data type. It must be initialized by the **long_lock_init** routine. The **long_lock**, **long_trylock**, and **long_unlock** subroutine performs similar operations to the **pthread_mutex_lock**, **pthread_mutex_trylock**, and **pthread_mutex_unlock** subroutine.

The following example shows a typical use of condition variables. In this example, the lock owner is not checked. As a result, any thread can unlock any lock. Error handling and cancelation handling are not performed.

```
typedef struct {
	pthread_mutex_t lock;
	pthread_cond_t cond;
	int free;
	int wanted;
} long_lock_t;

void long_lock_init(long_lock_t *ll)
{
	pthread_mutex_init(&ll->lock, NULL);
	pthread_cond_init(&ll->cond);
	ll->free = 1;
	ll->wanted = 0;
}

void long_lock_destroy(long_lock_t *ll)
{
	pthread_mutex_destroy(&ll->lock);
	pthread_cond_destroy(&ll->cond);
}

void long_lock(long_lock_t *ll)
{
	pthread_mutex_lock(&ll->lock);
	ll->wanted++;
	while(!ll->free)
		pthread_cond_wait(&ll->cond);
	ll->wanted--;
	ll->free = 0;
	pthread_mutex_unlock(&ll->lock);
}
```

```
int long_trylock(long_lock_t *ll)
{
	int got_the_lock;

	pthread_mutex_lock(&ll->lock);
	got_the_lock = ll->free;
	if (got_the_lock)
		ll->free = 0;
	pthread_mutex_unlock(&ll->lock);
	return got_the_lock;
}

void long_unlock(long_lock_t *ll)
{
	pthread_mutex_lock(&ll->lock);
	ll->free = 1;
	if (ll->wanted)
		pthread_cond_signal(&ll->cond);
	pthread_mutex_unlock(&ll->lock);
}
```

Semaphores

Traditional semaphores in UNIX systems are interprocess-synchronization facilities. For specific usage, you can implement interthread semaphores.

A semaphore has the **sema_t** data type. It must be initialized by the **sema_init** routine and destroyed with the **sema_destroy** routine. The semaphore wait and semaphore post operations are respectively performed by the **sema_p** and **sema_v** routines.

In the following basic implementation, error handling is not performed, but cancelations are properly handled with cleanup handlers whenever required: `typedef`

```
struct {
	pthread_mutex_t lock;
	pthread_cond_t cond;
	int count;
} sema_t;

void sema_init(sema_t *sem)
{
	pthread_mutex_init(&sem->lock, NULL);
	pthread_cond_init(&sem->cond, NULL);
	sem->count = 1;
}

void sema_destroy(sema_t *sem)
{
	pthread_mutex_destroy(&sem->lock);
	pthread_cond_destroy(&sem->cond);
}
```

```
void p_operation_cleanup(void *arg)
{
        sema_t *sem;

        sem = (sema_t *)arg;
        pthread_mutex_unlock(&sem->lock);
}

void sema_p(sema_t *sem)
{
        pthread_mutex_lock(&sem->lock);
        pthread_cleanup_push(p_operation_cleanup, sem);
        while (sem->count <= 0)
                pthread_cond_wait(&sem->cond, &sem->lock);
        sem->count--;
        /*
         *  Note that the pthread_cleanup_pop subroutine will
         *  execute the p_operation_cleanup routine
         */
        pthread_cleanup_pop(1);
}

void sema_v(sema_t *sem)
{
    pthread_mutex_lock(&sem->lock);
    if (sem->count <=0)
        pthread_cond_signal(&sem->cond);
    sem->count++;
    pthread_mutex_unlock(&sem->lock);
}
```

The counter specifies the number of users that are allowed to use the semaphore. It is never strictly negative; thus, it does not specify the number of waiting users, as for traditional semaphores. This implementation provides a typical solution to the multiple wakeup problem on the **pthread_cond_wait** subroutine. The semaphore wait operation is cancelable, because the **pthread_cond_wait** subroutine provides a cancelation point.

Write-Priority Read/Write Locks

A write-priority read/write lock provides multiple threads with simultaneous read-only access to a protected resource, and a single thread with write access to the resource while excluding reads. When a writer releases a lock, other waiting writers will get the lock before any waiting reader. Write-priority read/write locks are usually used to protect resources that are more often read than written.

A write-priority read/write lock has the **rwlock_t** data type. It must be initialized by the **rwlock_init** routine. The **rwlock_lock_read** routine locks the lock for a reader (multiple readers are allowed), the **rwlock_unlock_read** routine unlocks it. The

rwlock_lock_write routine locks the lock for a writer, the **rwlock_unlock_write** routine unlocks it. The proper unlocking routine (for the reader or for the writer) must be called.
In the following example, the lock owner is not checked. As a result, any thread can unlock any lock. Routines, such as the **pthread_mutex_trylock** subroutine, are missing and error handling is not performed, but cancelations are properly handled with cleanup handlers whenever required.

```
typedef struct {
	pthread_mutex_t lock;
	pthread_cond_t rcond;
	pthread_cond_t wcond;
	int lock_count;  /* < 0 .. held by writer             */
	                 /* > 0 .. held by lock_count readers */
	                 /* = 0 .. held by nobody             */
	int waiting_writers;  /* count of wating writers      */
} rwlock_t;

void rwlock_init(rwlock_t *rwl)
{
	pthread_mutex_init(&rwl->lock, NULL);
	pthread_cond_init(&rwl->wcond, NULL);
	pthread_cond_init(&rwl->rcond, NULL);
	rwl->lock_count = 0;
	rwl->waiting_writers = 0;
}

void waiting_reader_cleanup(void *arg)
{
	rwlock_t *rwl;

	rwl = (rwlock_t *)arg;
	pthread_mutex_unlock(&rwl->lock);
}

void rwlock_lock_read(rwlock_t *rwl)
{
	pthread_mutex_lock(&rwl->lock);
	pthread_cleanup_push(waiting_reader_cleanup, rwl);
	while ((rwl->lock_count < 0) && (rwl->waiting_writers))
		pthread_cond_wait(&rwl->rcond, &rwl->lock);
	rwl->lock_count++;
	/*
	 *  Note that the pthread_cleanup_pop subroutine will
	 *  execute the waiting_reader_cleanup routine
	 */
	pthread_cleanup_pop(1);
}

void rwlock_unlock_read(rwlock_t *rwl)
```

```
{
        pthread_mutex_lock(&rwl->lock);
        rwl->lock_count--;
        if (!rwl->lock_count)
                pthread_cond_signal(&rwl->wcond);
        pthread_mutex_unlock(&rwl->lock);
}

void waiting_writer_cleanup(void *arg)
{
        rwlock_t *rwl;

        rwl = (rwlock_t *)arg;
        rwl->waiting_writers--;
        if ((!rwl->waiting_writers) && (rwl->lock_count >= 0))
                /*
                         * This only happens if we have been canceled
                         */
                        pthread_cond_broadcast(&rwl->wcond);
                        pthread_mutex_unlock(&rwl->lock);
}

void rwlock_lock_write(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        rwl->waiting_writers++;
        pthread_cleanup_push(waiting_writer_cleanup, rwl);
        while (rwl->lock_count)
                pthread_cond_wait(&rwl->wcond, &rwl->lock);
        rwl->lock_count = -1;
        /*
         *  Note that the pthread_cleanup_pop subroutine will
         *  execute the waiting_writer_cleanup routine
         */
        pthread_cleanup_pop(1);
}

void rwlock_unlock_write(rwlock_t *rwl)
{
        pthread_mutex_lock(&rwl->lock);
        l->lock_count = 0;
        if (!rwl->wating_writers)
                pthread_cond_broadcast(&rwl->rcond);
        else
                pthread_cond_signal(&rwl->wcond);
        pthread_mutex_unlock(&rwl->lock);
}
```

Readers are counted only. When the count reaches zero, a waiting writer may take the lock. Only one writer can hold the lock. When the lock is released by a writer, another writer is awakened, if there is one. Otherwise, all waiting readers are awakened.
The locking routines are cancelable, because they call the **pthread_cond_wait** subroutine. Cleanup handlers are therefore registered before calling the subroutine.

Parent topic: Multithreaded programming

Related concepts:
Thread-specific data
Synchronization overview
List of threads-processes interactions subroutines

Signal management

Signals in multithreaded processes are an extension of signals in traditional single-threaded programs.
Signal management in multithreaded processes is shared by the process and thread levels, and consists of the following:
- Per-process signal handlers
- Per-thread signal masks
- Single delivery of each signal

Signal handlers and signal masks

Signal handlers are maintained at process level. It is strongly recommended to use the **sigwait** subroutine when waiting for signals. The **sigaction** subroutine is not recommended because the list of signal handlers is maintained at process level and any thread within the process might change it. If two threads set a signal handler on the same signal, the last thread that called the **sigaction** subroutine overrides the setting of the previous thread call; and in most cases, the order in which threads are scheduled cannot be predicted.
Signal masks are maintained at thread level. Each thread can have its own set of signals that will be blocked from delivery. The **sigthreadmask** subroutine must be used to get and set the calling thread's signal mask. The **sigprocmask** subroutine must not be used in multithreaded programs; because unexpected behavior might result.
The **pthread_sigmask** subroutine is very similar to the **sigprocmask** subroutine. The parameters and usage of both subroutines are identical. When porting existing code to support the threads library, you can replace the **sigprocmask** subroutine with the **pthread_sigmask** subroutine.

Signal generation

Signals generated by some action attributable to a particular thread, such as a hardware fault, are sent to the thread that caused the signal to be generated. Signals generated in association with a process ID, a process group ID, or an asynchronous event (such as terminal activity) are sent to the process.
- The **pthread_kill** subroutine sends a signal to a thread. Because thread IDs identify threads within a process, this subroutine can only send signals to threads within the same process.
- The **kill** subroutine (and thus the **kill** command) sends a signal to a process. A thread can send a **Signal** signal to its process by executing the following call:
  `kill(getpid(), Signal);`
- The **raise** subroutine cannot be used to send a signal to the calling thread's process. The **raise** subroutine sends a signal to the calling thread, as in the following call: `pthread_kill(pthread_self(), Signal);`
  This ensures that the signal is sent to the caller of the **raise** subroutine. Thus, library routines written for single-threaded programs can easily be ported to a multithreaded system, because the **raise** subroutine is usually intended to send the signal to the caller.
- The **alarm** subroutine requests that a signal be sent later to the process, and alarm states are maintained at process level. Thus, the last thread that called the **alarm** subroutine overrides the settings of other threads in the process. In a multithreaded program, the **SIGALRM** signal is not necessarily delivered to the

thread that called the **alarm** subroutine. The calling thread might even be terminated; and therefore, it cannot receive the signal.

Handling signals

Signal handlers are called within the thread to which the signal is delivered. The following limitations to signal handlers are introduced by the threads library:

- Signal handlers might call the **longjmp** or **siglongjmp** subroutine only if the corresponding call to the **setjmp** or **sigsetjmp** subroutine is performed in the same thread. Usually, a program that wants to wait for a signal installs a signal handler that calls the **longjmp** subroutine to continue execution at the point where the corresponding **setjmp** subroutine is called. This cannot be done in a multithreaded program, because the signal might be delivered to a thread other than the one that called the **setjmp** subroutine, thus causing the handler to be executed by the wrong thread.
  Note: Using longjmp from a signal handler can result in undefined behavior.
- No **pthread** routines can be called from a signal handler. Calling a **pthread** routine from a signal handler can lead to an application deadlock.

To allow a thread to wait for asynchronously generated signals, the threads library provides the **sigwait** subroutine. The **sigwait** subroutine blocks the calling thread until one of the awaited signals is sent to the process or to the thread. There must not be a signal handler installed on the awaited signal using the **sigwait** subroutine. Typically, programs might create a dedicated thread to wait for asynchronously generated signals. Such a thread loops on a **sigwait** subroutine call and handles the signals. It is recommended that such a thread block all the signals. The following code fragment gives an example of such a signal-waiter thread:

```
#include <pthread.h>
#include <signal.h>

static pthread_mutex_t mutex;
sigset_t set;
static int sig_cond = 0;

void *run_me(void *id)
{
	int sig;
	int err;
	sigset_t sigs;
	sigset_t oldSigSet;
	sigfillset(&sigs);
	sigthreadmask(SIG_BLOCK, &sigs, &oldSigSet);

	err = sigwait(&set, &sig);

	if(err)
	{
		/* do error code */
	}
	else
	{
```

```
                printf("SIGINT caught\n");
                pthread_mutex_lock(&mutex);
                sig_cond = 1;
                pthread_mutex_unlock(&mutex);
        }

        return;
}

main()
{
        pthread_t tid;

        sigemptyset(&set);
        sigaddset(&set, SIGINT);
        pthread_sigmask(SIG_BLOCK, &set, 0);

        pthread_mutex_init(&mutex, NULL);

        pthread_create(&tid, NULL, run_me, (void *)1);

        while(1)
        {
                sleep(1);
                /* or so something here */

                pthread_mutex_lock(&mutex);
                if(sig_cond)
                {
                        /* do exit stuff */
                        return;
                }
                pthread_mutex_unlock(&mutex);

        }

}
```

If more than one thread called the **sigwait** subroutine, exactly one call returns when a matching signal is sent. Which thread is awakened cannot be predicted. If a thread is going to do **sigwait** as well as handling of some other signals for which it is not doing **sigwait**, the user-defined signal handlers need to block the sigwaiter signals for the proper handling. Note that the **sigwait** subroutine provides a cancellation point.

Because a dedicated thread is not a real signal handler, it might signal a condition to any other thread. It is possible to implement a **sigwait_multiple** routine that would awaken all threads waiting for a specific signal. Each caller of the **sigwait_multiple** routine registers a set of signals. The caller then waits on a condition variable. A

single thread calls the **sigwait** subroutine on the union of all registered signals. When the call to the **sigwait** subroutine returns, the appropriate state is set and condition variables are broadcasted. New callers to the **sigwait_multiple** subroutine cause the pending **sigwait** subroutine call to be canceled and reissued to update the set of signals being waited for.

Signal delivery

A signal is delivered to a thread, unless its action is set to ignore. The following rules govern signal delivery in a multithreaded process:

- A signal whose action is set to terminate, stop, or continue the target thread or process respectively terminates, stops, or continues the entire process (and thus all of its threads). Single-threaded programs can thus be rewritten as multithreaded programs without changing their externally visible signal behavior. For example, consider a multithreaded user command, such as the **grep** command. A user can start the command in his favorite shell and then decide to stop it by sending a signal with the **kill** command. The signal should stop the entire process running the **grep** command.
- Signals generated for a specific thread, using the **pthread_kill** or the **raise** subroutines, are delivered to that thread. If the thread has blocked the signal from delivery, the signal is set pending on the thread until the signal is unblocked from delivery. If the thread is terminated before the signal delivery, the signal will be ignored.
- Signals generated for a process, using the **kill** subroutine for example, are delivered to exactly one thread in the process. If one or more threads called the **sigwait** subroutine, the signal is delivered to exactly one of these threads. Otherwise, the signal is delivered to exactly one thread that did not block the signal from delivery. If no thread matches these conditions, the signal is set pending on the process until a thread calls the **sigwait** subroutine specifying this signal or a thread unblocks the signal from delivery.

If the action associated with a pending signal (on a thread or on a process) is set to ignore, the signal is ignored.

- List of threads-processes interactions subroutines
  This section lists threads-processes interactions subroutines.

Parent topic: Multithreaded programming

Related concepts:

Process duplication and termination

List of threads-processes interactions subroutines

This section lists threads-processes interactions subroutines.

Subroutine	Description
alarm	Causes a signal to be sent to the calling process after a specified timeout.
kill or **killpg**	Sends a signal to a process or a group of processes.
pthread_atfork	Registers fork cleanup handlers.
pthread_kill	Sends a signal to the specified thread.
pthread_sigmask	Sets the signal mask of a thread.
raise	Sends a signal to the executing thread.
sigaction, **sigvec**, or **signal**	Specifies the action to take upon delivery of a signal.
sigsuspend or **sigpause**	Atomically changes the set of blocked signals and waits for a signal.
sigthreadmask	Sets the signal mask of a thread.
sigwait	Blocks the calling thread until a specified signal is received.

Parent topic: Signal management

Related concepts:
Thread-specific data
Creating complex synchronization objects
Process duplication and termination

Process duplication and termination

Because all processes have at least one thread, creating (that is, duplicating) and terminating a process implies the creation and the termination of threads.
This section describes the interactions between threads and processes when duplicating and terminating a process.
Read the following to learn more about process duplication and termination:

Forking

Programmers call the **fork** subroutine in the following cases:

- To create a new flow of control within the same program. AIX® creates a new process.
- To create a new process running a different program. In this case, the call to the **fork** subroutine is soon followed by a call to one of the **exec** subroutines.

In a multithreaded program, the first use of the **fork** subroutine, creating new flows of control, is provided by the **pthread_create** subroutine. The **fork** subroutine should thus be used only to run new programs.
The **fork** subroutine duplicates the parent process, but duplicates only the calling thread; the child process is a single-threaded process. The calling thread of the parent process becomes the initial thread of the child process; it may not be the initial thread of the parent process. Thus, if the initial thread of the child process returns from its entry-point routine, the child process terminates.
When duplicating the parent process, the **fork** subroutine also duplicates all the synchronization variables, including their state. Thus, for example, mutexes may be held by threads that no longer exist in the child process and any associated resource may be inconsistent.
It is strongly recommended that the **fork** subroutine be used only to run new programs, and to call one of the **exec** subroutines as soon as possible after the call to the **fork** subroutine in the child process.

Fork handlers

The preceding forking rule does not address the needs of multithreaded libraries. Application programs may not be aware that a multithreaded library is in use and will call any number of library routines between the **fork** and the **exec** subroutines, just as they always have. Indeed, they may be old single-threaded programs and cannot, therefore, be expected to obey new restrictions imposed by the threads library.
On the other hand, multithreaded libraries need a way to protect their internal state during a fork in case a routine is called later in the child process. The problem arises especially in multithreaded input/output libraries, which are almost sure to be invoked between the **fork** and the **exec** subroutines to affect input/output redirection.
The **pthread_atfork** subroutine provides a way for multithreaded libraries to protect themselves from innocent application programs that call the **fork** subroutine. It also provides multithreaded application programs with a standard mechanism for protecting themselves from calls to the **fork** subroutine in a library routine or the application itself.
The **pthread_atfork** subroutine registers fork handlers to be called before and after the call to the **fork** subroutine. The fork handlers are executed in the thread that called the **fork** subroutine. The following fork handlers exist:

Subroutine	Description
Prepare	The prepare fork handler is called just before the processing of the **fork** subroutine begins.
Parent	The parent fork handler is called just after the processing of the **fork** subroutine is completed in the parent process.
Child	The child fork handler is called just after the processing of the **fork** subroutine is completed in the child process.

Process termination
The prepare fork handlers are called in last-in first-out (LIFO) order, whereas the parent and child fork handlers are called in first-in first-out (FIFO) order. This allows programs to preserve any desired locking order.
When a process terminates, by calling the **exit**, **atexit**, or **_exit** subroutine either explicitly or implicitly, all threads within the process are terminated. Neither the cleanup handlers nor the thread-specific data destructors are called. Note: The unatexit subroutine unregisters functions that were previously registered by the atexit subroutine. If the referenced function is found, it is removed from the list of functions that are called at normal program termination.

The reason for this behavior is that there is no state to leave clean and no thread-specific storage to reclaim, because the whole process terminates, including all the threads, and all the process storage is reclaimed, including all thread-specific storage.

Parent topic: Multithreaded programming

Related concepts:
Understanding threads and processes
Signal management
List of threads-processes interactions subroutines

Threads library options

This section describes special attributes of threads, mutexes, and condition variables.

The POSIX standard for the threads library specifies the implementation of some parts as optional. All subroutines defined by the threads library API are always available. Depending on the available options, some subroutines may not be implemented. Unimplemented subroutines can be called by applications, but they always return the **ENOSYS** error code.

Stack attributes

A stack is allocated for each thread. Stack management is implementation-dependent. Thus, the following information applies only to AIX®, although similar features may exist on other systems.

The stack is dynamically allocated when the thread is created. Using advanced thread attributes, it is possible for the user to control the stack size and address of the stack. The following information does not apply to the initial thread, which is created by the system.

Stack size

The stack size option enables the control of the **stacksize** attribute of a thread attributes object. This attribute specifies the minimum stack size to be used for the created thread.

The **stacksize** attribute is defined in AIX. The following attribute and subroutines are available when the option is implemented:

- The **stacksize** attribute of the thread attributes object
- The **pthread_attr_getstacksize** returns the value of the attribute
- and **pthread_attr_setstacksize** subroutines sets the value

The default value of the **stacksize** attribute is 96 KB. The minimum value of the **stacksize** attribute is 16 KB. If the assigned value is less than the minimum value, the minimum value is allocated.

In the AIX implementation of the threads library, a chunk of data, called *user thread area*, is allocated for each created thread. The area is divided into the following sections:

- A *red zone*, which is both read-protected and write-protected for stack overflow detection. There is no red zone in programs that use large pages.
- A default stack.
- A pthread structure.
- A thread structure.
- A thread attribute structure.

Note: The user thread area described here has no relationship to the **uthread** structure used in the AIX kernel. The user thread area is accessed only in user mode and is exclusively handled by the threads library, whereas the **uthread** structure only exists within the kernel environment.

Stack address POSIX option

The stack address option enables the control of the **stackaddr** attribute of a thread attributes object. This attribute specifies the location of storage to be used for the created thread's stack.

The following attribute and subroutines are available when the option is

implemented:
- The **stackaddr** attribute of the thread attributes object specifies the address of the stack that will be allocated for a thread.
- The **pthread_attr_getstackaddr** subroutine returns the value of the attribute.
- and **pthread_attr_setstackaddr** subroutine sets the value.

If no stack address is specified, the stack is allocated by the system at an arbitrary address. If you must have the stack at a known location, you can use the **stackaddr** attribute. For example, if you need a very large stack, you can set its address to an unused segment, guaranteeing that the allocation will succeed.
If a stack address is specified when calling the **pthread_create** subroutine, the system attempts to allocate the stack at the given address. If it fails, the **pthread_create** subroutine returns **EINVAL**. Because the **pthread_attr_setstackaddr** subroutine does not actually allocate the stack, it only returns an error if the specified stack address exceeds the addressing space.

Priority scheduling POSIX option

The priority scheduling option enables the control of execution scheduling at thread level. When this option is disabled, all threads within a process share the scheduling properties of the process. When this option is enabled, each thread has its own scheduling properties. For local contention scope threads, the scheduling properties are handled at process level by a library scheduler, while for global contention scope threads, the scheduling properties are handled at system level by the kernel scheduler.
The folowing attributes and subroutines are available when the option is implemented:
- The **inheritsched** attribute of the thread attributes object
- The **schedparam** attribute of the thread attributes object and the thread
- The **schedpolicy** attribute of the thread attributes objects and the thread
- The **contention-scope** attribute of the thread attributes objects and the thread
- The **pthread_attr_getschedparam** and **pthread_attr_setschedparam** subroutines
- The **pthread_getschedparam** subroutine

Checking the availability of an option

Options can be checked at compile time or at run time. Portable programs should check the availability of options before using them, so that they need not be rewritten when ported to other systems.

Compile-time checking

When an option is not available, you can stop the compilation, as in the following example:

```
#ifndef _POSIX_THREAD_ATTR_STACKSIZE
#error "The stack size POSIX option is required"
#endif
```

The **pthread.h** header file also defines the following symbols that can be used by other header files or by programs:

- _POSIX_REENTRANT_FUNCTIONS
  - Denotes that reentrant functions are required

- _POSIX_THREADS

- Denotes the implementation of the threads library

Run-time checking

The **sysconf** subroutine can be used to get the availability of options on the system where the program is executed. This is useful when porting programs between systems that have a binary compatibility, such as two versions of AIX.

The following list indicates the symbols that are associated with each option and that must be used for the *Name* parameter of the **sysconf** subroutine. The symbolic constants are defined in the **unistd.h** header file.

- Stack address
  - **_SC_THREAD_ATTR_STACKADDR**

- Stack size
  - **_SC_THREAD_ATTR_STACKSIZE**

- Priority scheduling
  - **_SC_THREAD_PRIORITY_SCHEDULING**

- Priority inheritance
  - **_SC_THREAD_PRIO_INHERIT**

- Priority protection
  - **_SC_THREAD_PRIO_PROTECT**

- Process sharing
  - **_SC_THREAD_PROCESS_SHARED**

To check the general options, use the **sysconf** subroutine with the following *Name* parameter values:

- _SC_REENTRANT_FUNCTIONS
  - Denotes that reentrant functions are required.

- _SC_THREADS
  - Denotes the implementation of the threads library.

Process sharing

AIX and most UNIX systems allow several processes to share a common data space, known as *shared memory*.The process-sharing attributes for condition variables and mutexes are meant to allow these objects to be allocated in shared memory to support synchronization among threads belonging to different processes. However, because there is no industry-standard interface for shared memory management, the process-sharing POSIX option is not implemented in the AIX threads library.

Threads data types

The following data types are defined for the threads library. The definition of these data types can vary between systems:

- pthread_t
  - Identifies a thread

- pthread_attr_t
  - Identifies a thread attributes object

- pthread_cond_t
  - Identifies a condition variable

- pthread_condattr_t
  - Identifies a condition attributes object

- **pthread_key_t**
  - Identifies a thread-specific data key
- **pthread_mutex_t**
  - Identifies a mutex
- **pthread_mutexattr_t**
  - Identifies a mutex attributes object
- **pthread_once_t**
  - Identifies a one-time initialization object

Limits and Default Values

The threads library has some implementation-dependent limits and default values. These limits and default values can be retrieved by symbolic constants to enhance the portability of programs:

- The maximum number of threads per process is 512. The maximum number of threads can be retrieved at compilation time using the **PTHREAD_THREADS_MAX** symbolic constant defined in the **pthread.h** header file. If an application is compiled with the **-D_LARGE_THREADS** flag, the maximum number of threads per process is 32767.
- The minimum stack size for a thread is 8 K. The default stack size is 96 KB. This number can be retrieved at compilation time using the **PTHREAD_STACK_MIN** symbolic constant defined in the **pthread.h** header file. Note: The maximum stack size is 256 MB, the size of a segment. This limit is indicated by the **PTHREAD_STACK_MAX** symbolic constant in the **pthread.h** header file.
- The maximum number of thread-specific data keys is limited to 508. This number can be retrieved at compilation time using the **PTHREAD_KEYS_MAX** symbolic constant defined in the **pthread.h** header file.

Default attribute values

The default values for the thread attributes object are defined in the **pthread.h** header file by the following symbolic constants:

- The default value for the **DEFAULT_DETACHSTATE** symbolic constant is **PTHREAD_CREATE_DETACHED**, which specifies the default value for the **detachstate** attribute.
- The default value for the **DEFAULT_JOINABLE** symbolic constant is **PTHREAD_CREATE_JOINABLE**, which specifies the default value for the joinable state.
- The default value for the **DEFAULT_INHERIT** symbolic constant is **PTHREAD_INHERIT_SCHED**, which specifies the default value for the **inheritsched** attribute.
- The default value for the **DEFAULT_PRIO** symbolic constant is 1, which specifies the default value for the sched_prio field of the **schedparam** attribute.
- The default value for the **DEFAULT_SCHED** symbolic constant is **SCHED_OTHER**, which specifies the default value for the **schedpolicy** attribute of a thread attributes object.
- The default value for the **DEFAULT_SCOPE** symbolic constant is **PTHREAD_SCOPE_LOCAL**, which specifies the default value for the **contention-scope** attribute.

- List of threads advanced-feature subroutines
  This section lists threads advanced-features subroutines.
- Supported interfaces
  On AIX systems, the **_POSIX_THREADS**,
  _POSIX_THREAD_ATTR_STACKADDR,
  _POSIX_THREAD_ATTR_STACKSIZE and
  _POSIX_THREAD_PROCESS_SHARED symbols are always defined.

Parent topic: Multithreaded programming

Related concepts:
Threadsafe and threaded libraries in AIX

List of threads advanced-feature subroutines

This section lists threads advanced-features subroutines.

Subroutine	Description
pthread_attr_getstackaddr	Returns the value of the stackaddr attribute of a thread attributes object.
pthread_attr_getstacksize	Returns the value of the stacksize attribute of a thread attributes object.
pthread_attr_setstackaddr	Sets the value of the stackaddr attribute of a thread attributes object.
pthread_attr_setstacksize	Sets the value of the stacksize attribute of a thread attributes object.
pthread_condattr_getpshared	Returns the value of the process-shared attribute of a condition attributes object.
pthread_condattr_setpshared	Sets the value of the process-shared attribute of a condition attributes object.
pthread_getspecific	Returns the thread-specific data associated with the specified key.
pthread_key_create	Creates a thread-specific data key.
pthread_key_delete	Deletes a thread-specific data key.
pthread_mutexattr_getpshared	Returns the value of the process-shared attribute of a mutex attributes object.
pthread_mutexattr_setpshared	Sets the value of the process-shared attribute of a mutex attributes object.
pthread_once	Executes a routine exactly once in a process.
PTHREAD_ONCE_INIT	Initializes a one-time synchronization control structure.
pthread_setspecific	Sets the thread-specific data associated with the specified key.

Parent topic: Threads library options

Supported interfaces

On AIX® systems, the **_POSIX_THREADS**, **_POSIX_THREAD_ATTR_STACKADDR**, **_POSIX_THREAD_ATTR_STACKSIZE** and **_POSIX_THREAD_PROCESS_SHARED** symbols are always defined. Therefore, the following threads interfaces are supported.

POSIX interfaces

The following is a list of POSIX interfaces:

- pthread_atfork
- pthread_attr_destroy
- pthread_attr_getdetachstate
- pthread_attr_getschedparam
- pthread_attr_getstacksize
- pthread_attr_getstackaddr
- pthread_attr_init
- pthread_attr_setdetachstate
- pthread_attr_setschedparam
- pthread_attr_setstackaddr
- pthread_attr_setstacksize
- pthread_cancel
- pthread_cleanup_pop
- pthread_cleanup_push
- pthread_detach
- pthread_equal
- pthread_exit
- pthread_getspecific
- pthread_join
- pthread_key_create
- pthread_key_delete
- pthread_kill
- pthread_mutex_destroy
- pthread_mutex_init
- pthread_mutex_lock
- pthread_mutex_trylock
- pthread_mutex_unlock
- pthread_mutexattr_destroy
- pthread_mutexattr_getpshared
- pthread_mutexattr_init
- pthread_mutexattr_setpshared
- pthread_once
- pthread_self
- pthread_setcancelstate
- pthread_setcanceltype
- pthread_setspecific
- pthread_sigmask
- pthread_testcancel
- pthread_cond_broadcast
- pthread_cond_destroy

- pthread_cond_init
- pthread_cond_signal
- pthread_cond_timedwait
- pthread_cond_wait
- pthread_condattr_destroy
- pthread_condattr_getpshared
- pthread_condattr_init
- pthread_condattr_setpshared
- pthread_create
- sigwait

Single UNIX Specification, Version 2 Interfaces

The following is a list of Single UNIX Specification, Version 2 interfaces:
- pthread_attr_getguardsize
- pthread_attr_setguardsize
- pthread_getconcurrency
- pthread_mutexattr_gettype
- pthread_mutexattr_settype
- pthread_rwlock_destroy
- pthread_rwlock_init
- pthread_rwlock_rdlock
- pthread_rwlock_tryrdlock
- pthread_rwlock_trywrlock
- pthread_rwlock_unlock
- pthread_rwlock_wrlock
- pthread_rwlockattr_destroy
- pthread_rwlockattr_getpshared
- pthread_rwlockattr_init
- pthread_rwlockattr_setpshared
- pthread_setconcurrency

On AIX systems, **_POSIX_THREAD_SAFE_FUNCTIONS** symbol is always defined. Therefore, the following interfaces are always supported:
- asctime_r
- ctime_r
- flockfile
- ftrylockfile
- funlockfile
- getc_unlocked
- getchar_unlocked
- getgrgid_r
- getgrnam_r
- getpwnam_r
- getpwuid_r
- gmtime_r
- localtime_r
- putc_unlocked

- putchar_unlocked
- rand_r
- readdir_r
- strtok_r

AIX does not support the following interfaces; the symbols are provided but they always return an error and set the errno to ENOSYS:
- pthread_mutex_getprioceiling
- pthread_mutex_setprioceiling
- pthread_mutexattr_getprioceiling
- pthread_mutexattr_getprotocol
- pthread_mutexattr_setprioceiling
- pthread_mutexattr_setprotocol

Non-threadsafe interfaces
libc.a library (standard functions):
- advance
- asctime
- brk
- catgets
- chroot
- compile
- ctime
- cuserid
- dbm_clearerr
- dbm_close
- dbm_delete
- dbm_error
- dbm_fetch
- dbm_firstkey
- dbm_nextkey
- dbm_open
- dbm_store
- dirname
- drand48
- ecvt
- encrypt
- endgrent
- endpwent
- endutxent
- fcvt
- gamma
- gcvt
- getc_unlocked
- getchar_unlocked
- getdate
- getdtablesize

- getgrent
- getgrgid
- getgrnam
- getlogin
- getopt
- getpagesize
- getpass
- getpwent
- getpwnam
- getpwuid
- getutxent
- getutxid
- getutxline
- getw
- getw
- gmtime
- l64a
- lgamma
- localtime
- lrand48
- mrand48
- nl_langinfo
- ptsname
- putc_unlocked
- putchar_unlocked
- pututxline
- putw
- rand
- random
- readdir
- re_comp
- re_exec
- regcmp
- regex
- sbrk
- setgrent
- setkey
- setpwent
- setutxent
- sigstack
- srand48
- srandom
- step
- strerror
- strtok
- ttyname
- ttyslot

- wait3

The following AIX interfaces are not threadsafe.

libc.a Library (AIX-specific functions):

- endfsent
- endttyent
- endutent
- getfsent
- getfsfile
- getfsspec
- getfstype
- getttyent
- getttynam
- getutent
- getutid
- getutline
- pututline
- setfsent
- setttyent
- setutent
- utmpname

libbsd.a library:

- timezone

libm.a and **libmsaa.a** libraries:

- gamma
- lgamma

None of the functions in the following libraries are threadsafe:

- libPW.a
- libblas.a
- libcur.a
- libcurses.a
- libplot.a
- libprint.a

The **ctermid** and **tmpnam** interfaces are not threadsafe if they are passed a NULL argument.

In a multi-threaded program it is not recommended to execute **setlocale()** subroutine simultaneously from multiple threads if one of the threads calls **setlocale()** subroutine from within a module-initialization routine.

Note: Certain subroutines may be implemented as macros on some systems. Avoid using the address of threads subroutines.

Parent topic: Threads library options

Writing reentrant and threadsafe code

In single-threaded processes, only one flow of control exists. The code executed by these processes thus need not be reentrant or threadsafe. In multithreaded programs, the same functions and the same resources may be accessed concurrently by several flows of control.

To protect resource integrity, code written for multithreaded programs must be reentrant and threadsafe.

Reentrance and thread safety are both related to the way that functions handle resources. Reentrance and thread safety are separate concepts: a function can be either reentrant, threadsafe, both, or neither.

This section provides information about writing reentrant and threadsafe programs. It does not cover the topic of writing thread-efficient programs. Thread-efficient programs are efficiently parallelized programs. You must consider thread effiency during the design of the program. Existing single-threaded programs can be made thread-efficient, but this requires that they be completely redesigned and rewritten.

Reentrance

A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions.

A non-reentrant function can often, but not always, be identified by its external interface and its usage. For example, the **strtok** subroutine is not reentrant, because it holds the string to be broken into tokens. The **ctime** subroutine is also not reentrant; it returns a pointer to static data that is overwritten by each call.

Thread safety

A threadsafe function protects shared resources from concurrent access by locks. Thread safety concerns only the implementation of a function and does not affect its external interface.

In C language, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data or other shared resources is trivially threadsafe, as in the following example:

```
/* threadsafe function */
int diff(int x, int y)
{
        int delta;

        delta = y - x;
        if (delta < 0)
                delta = -delta;

        return delta;
}
```

The use of global data is thread-unsafe. Global data should be maintained per thread or encapsulated, so that its access can be serialized. A thread may read an error code corresponding to an error caused by another thread. In AIX®, each thread has its own **errno** value.

Making a function reentrant

In most cases, non-reentrant functions must be replaced by functions with a modified interface to be reentrant. Non-reentrant functions cannot be used by

multiple threads. Furthermore, it may be impossible to make a non-reentrant function threadsafe.

Returning data

Many non-reentrant functions return a pointer to static data. This can be avoided in the following ways:

- Returning dynamically allocated data. In this case, it will be the caller's responsibility to free the storage. The benefit is that the interface does not need to be modified. However, backward compatibility is not ensured; existing single-threaded programs using the modified functions without changes would not free the storage, leading to memory leaks.
- Using caller-provided storage. This method is recommended, although the interface must be modified.

For example, a **strtoupper** function, converting a string to uppercase, could be implemented as in the following code fragment:

```
/* non-reentrant function */
char *strtoupper(char *string)
{
	static char buffer[MAX_STRING_SIZE];
	int index;

	for (index = 0; string[index]; index++)
		buffer[index] = toupper(string[index]);
	buffer[index] = 0

	return buffer;
}
```

This function is not reentrant (nor threadsafe). To make the function reentrant by returning dynamically allocated data, the function would be similar to the following code fragment:

```
/* reentrant function (a poor solution) */
char *strtoupper(char *string)
{
	char *buffer;
	int index;

	/* error-checking should be performed! */
	buffer = malloc(MAX_STRING_SIZE);

	for (index = 0; string[index]; index++)
		buffer[index] = toupper(string[index]);
	buffer[index] = 0

	return buffer;
}
```

A better solution consists of modifying the interface. The caller must provide the storage for both input and output strings, as in the following code fragment:

```
/* reentrant function (a better solution) */
```

```
char *strtoupper_r(char *in_str, char *out_str)
{
	int index;

	for (index = 0; in_str[index]; index++)
	out_str[index] = toupper(in_str[index]);
	out_str[index] = 0

	return out_str;
}
```

Te non-reentrant standard C library subroutines were made reentrant using caller-provided storage.

Keeping data over successive calls

No data should be kept over successive calls, because different threads may successively call the function. If a function must maintain some data over successive calls, such as a working buffer or a pointer, the caller should provide this data.

Consider the following example. A function returns the successive lowercase characters of a string. The string is provided only on the first call, as with the **strtok** subroutine. The function returns 0 when it reaches the end of the string. The function could be implemented as in the following code fragment:

```
/* non-reentrant function */
char lowercase_c(char *string)
{
	static char *buffer;
	static int index;
	char c = 0;

	/* stores the string on first call */
	if (string != NULL) {
		buffer = string;
		index = 0;
	}

	/* searches a lowercase character */
	for (; c = buffer[index]; index++) {
		if (islower(c)) {
			index++;
			break;
		}
	}
	return c;
}
```

h

This function is not reentrant. To make it reentrant, the static data, the **index** variable, must be maintained by the caller. The reentrant version of the function

could be implemented as in the following code fragment: /* reentrant function */

```
char reentrant_lowercase_c(char *string, int *p_index)
{
	char c = 0;

	/* no initialization - the caller should have done it */

	/* searches a lowercase character */
	for (; c = string[*p_index]; (*p_index)++) {
		if (islower(c)) {
			(*p_index)++;
			break;
		  }
	}
	return c;
}
```

The interface of the function changed and so did its usage. The caller must provide the string on each call and must initialize the index to 0 before the first call, as in the following code fragment: char *my_string;

```
char my_char;
int my_index;
...
my_index = 0;
while (my_char = reentrant_lowercase_c(my_string, &my_index)) {
	...
}
```

Making a function threadsafe

In multithreaded programs, all functions called by multiple threads must be threadsafe. However, a workaround exists for using thread-unsafe subroutines in multithreaded programs. Non-reentrant functions usually are thread-unsafe, but making them reentrant often makes them threadsafe, too.

Locking shared resources

Functions that use static data or any other shared resources, such as files or terminals, must serialize the access to these resources by locks in order to be threadsafe. For example, the following function is thread-unsafe: /* thread-unsafe function */

```
int increment_counter()
{
	static int counter = 0;

	counter++;
	return counter;
}
```

To be threadsafe, the static variable **counter** must be protected by a static lock, as in the following example: /* pseudo-code threadsafe function */

```
int increment_counter();
{
        static int counter = 0;
        static lock_type counter_lock = LOCK_INITIALIZER;

        pthread_mutex_lock(counter_lock);
        counter++;
        pthread_mutex_unlock(counter_lock);
        return counter;
}
```

In a multithreaded application program using the threads library, mutexes should be used for serializing shared resources. Independent libraries may need to work outside the context of threads and, thus, use other kinds of locks.

Workarounds for thread-unsafe functions

It is possible to use a workaround to use thread-unsafe functions called by multiple threads. This can be useful, especially when using a thread-unsafe library in a multithreaded program, for testing or while waiting for a threadsafe version of the library to be available. The workaround leads to some overhead, because it consists of serializing the entire function or even a group of functions. The following are possible workarounds:

- Use a global lock for the library, and lock it each time you use the library (calling a library routine or using a library global variable). This solution can create performance bottlenecks because only one thread can access any part of the library at any given time. The solution in the following pseudocode is acceptable only if the library is seldom accessed, or as an initial, quickly implemented workaround. `/* this is pseudo code! */`

```
lock(library_lock);
library_call();
unlock(library_lock);

lock(library_lock);
x = library_var;
unlock(library_lock);
```

- Use a lock for each library component (routine or global variable) or group of components. This solution is somewhat more complicated to implement than the previous example, but it can improve performance. Because this workaround should only be used in application programs and not in libraries, mutexes can be used for locking the library. `/* this is pseudo-code! */`

```
lock(library_moduleA_lock);
library_moduleA_call();
unlock(library_moduleA_lock);

lock(library_moduleB_lock);
x = library_moduleB_var;
unlock(library_moduleB_lock);
```

Reentrant and threadsafe libraries

Reentrant and threadsafe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within threads. It is a good programming practice to always use and write reentrant and threadsafe functions.

Using libraries

Several libraries shipped with the AIX Base Operating System are threadsafe. In the current version of AIX, the following libraries are threadsafe:

- Standard C library (**libc.a**)
- Berkeley compatibility library (**libbsd.a**)

Some of the standard C subroutines are non-reentrant, such as the **ctime** and **strtok** subroutines. The reentrant version of the subroutines have the name of the original subroutine with a suffix **_r** (underscore followed by the letter *r*).
When writing multithreaded programs, use the reentrant versions of subroutines instead of the original version. For example, the following code fragment:

```
token[0] = strtok(string, separators);
i = 0;
do {
       i++;
       token[i] = strtok(NULL, separators);
} while (token[i] != NULL);
```

should be replaced in a multithreaded program by the following code fragment:

```
char *pointer;
...
token[0] = strtok_r(string, separators, &pointer);
i = 0;
do {
       i++;
       token[i] = strtok_r(NULL, separators, &pointer);
} while (token[i] != NULL);
```

Thread-unsafe libraries may be used by only one thread in a program. Ensure the uniqueness of the thread using the library; otherwise, the program will have unexpected behavior, or may even stop.

Converting libraries

Consider the following when converting an existing library to a reentrant and threadsafe library. This information applies only to C language libraries.

- Identify exported global variables. Those variables are usually defined in a header file with the **export** keyword. Exported global variables should be encapsulated. The variable should be made private (defined with the **static** keyword in the library source code), and access (read and write) subroutines should be created.
- Identify static variables and other shared resources. Static variables are usually defined with the **static** keyword. Locks should be associated with any shared resource. The granularity of the locking, thus choosing the number of locks, impacts the performance of the library. To initialize the locks, the one-time initialization facility may be used.

- Identify non-reentrant functions and make them reentrant. For more information, see Making a Function Reentrant.
- Identify thread-unsafe functions and make them threadsafe. For more information, see Making a Function threadsafe.

Parent topic: Multithreaded programming

Parent topic: General programming concepts

Related concepts:
One-time initializations

Related information:
admin
cdc
delta
get
prs
sccsdiff
sccsfile

Developing multithreaded programs

Developing multithreaded programs is similar to developing programs with multiple processes. Developing programs also consists of compiling and debugging the code.

Compiling a multithreaded program

This section explains how to generate a multithreaded program. It describes the following:

- The required header file
- Invoking the compiler, which is used to generate multithreaded programs.

Header file

All subroutine prototypes, macros, and other definitions for using the threads library are in the **pthread.h** header file, which is located in the **/usr/include** directory. The **pthread.h** header file must be included in each source file using the threads library. The **pthread.h** header includes the **unistd.h** header, which provides the following global definitions:

- **_POSIX_REENTRANT_FUNCTIONS**
  - Specifies that all functions should be reentrant. Several header files use this symbol to define supplementary reentrant subroutines, such as the **localtime_r** subroutine.
- **_POSIX_THREADS**
  - Denotes the POSIX threads API. This symbol is used to check if the POSIX threads API is available. Macros or subroutines may be defined in different ways, depending on whether the POSIX or some other threads API is used.

The **pthread.h** file also includes **errno.h**, in which the **errno** global variable is redefined to be thread-specific. The **errno** identifier is, therefore, no longer an **l-value** in a multithreaded program.

Invoking the compiler

When compiling a multithreaded program, invoke the C compiler using one of the following commands:

- **xlc_r**
  - Invokes the compiler with default language level of **ansi**
- **cc_r**
  - Invokes the compiler with default language level of **extended**

These commands ensure that the adequate options and libraries are used to be compliant with the Single UNIX Specification, Version 2. The POSIX Threads Specification 1003.1c is a subset of the Single UNIX Specification, Version 2.
The following libraries are automatically linked with your program when using the **xlc_r** and **cc_r** commands:

- **libpthreads.a**
  - Threads library
- **libc.a**
  - Standard C library

For example, the following command compiles the **foo.c** multithreaded C source file and produces the **foo** executable file: `cc_r -o foo foo.c`

Invoking the compiler for draft 7 of POSIX 1003.1c
AIX® provides source code compatibility for Draft 7 applications. It is recommended that developers port their threaded application to the latest standard.
When compiling a multithreaded program for Draft 7 support of threads, invoke the C compiler using one of the following commands:
- xlc_r7
- Invokes the compiler with default language level of **ansi**
- cc_r7
- Invokes the compiler with default language level of **extended**

The following libraries are automatically linked with your program when using the **xlc_r7** and **cc_r7** commands:
- libpthreads_compat.a
- Draft 7 Compatibility Threads library
- libpthreads.a
- Threads library
- libc.a
- Standard C library

To achieve source code compatibility, use the compiler directive **_AIX_PTHREADS_D7**. It is also necessary to link the libraries in the following order: **libpthreads_compat.a**, **libpthreads.a**, and **libc.a**. Most users do not need to know this information, because the commands provide the necessary options. These options are provided for those who do not have the latest AIX compiler.
Porting draft 7 applications to the &Symbol.unixspec;
Differences exist between Draft 7 and the final standard include:
- Minor **errno** differences. The most prevalent is the use of **ESRCH** to denote the specified pthread could not be found. Draft 7 frequently returned **EINVAL** for this failure.
- The default state when a pthread is created is *joinable*. This is a significant change because it can result in a memory leak if ignored.
- The default pthread scheduling parameter is *scope*.
- The **pthread_yield** subroutine has been replaced by the **sched_yield** subroutine.
- The various scheduling policies associated with the mutex locks are slightly different.

Memory requirements of a multithreaded program
AIX supports up to 32768 threads in a single process. Each individual pthread requires some amount of process address space, so the actual maximum number of pthreads that a process can have depends on the memory model and the use of process address space, for other purposes. The amount of memory that a pthread needs includes the stack size and the guard region size, plus some amount for internal use. The user can control the size of the stack with the **pthread_attr_setstacksize** subroutine and the size of the guard region with the **pthread_attr_setguardsize** subroutine. Note: The soft limit on stack size imposed by the command ulimit–s applies only to the stack of the main thread of the application.

The following table indicates the maximum number of pthreads that could be created in a 32-bit process using a simple program which does nothing other than create pthreads in a loop using the NULL pthread attribute. In a real program, the actual numbers depend on other memory usage in the program. For a 64-bit process, the **ulimit** subroutine controls how many threads can be created. Therefore, the big data model is not necessary and in fact, can decrease the maximum number of threads.

Data Model	-bmaxdata	Maximum Pthreads
Small Data	n/a	1084
Big Data	0x10000000	2169
Big Data	0x20000000	4340
Big Data	0x30000000	6510
Big Data	0x40000000	8681
Big Data	0x50000000	10852
Big Data	0x60000000	13022
Big Data	0x70000000	15193
Big Data	0x80000000	17364

The **NUM_SPAREVP** environment variable can be set to control the number of spare virtual processors that are maintained by the library. It is not necessary to modify this variable. In some circumstances, applications that use only a few megabytes of memory can reduce memory overhead by setting the **NUM_SPAREVP** environment variable to a lower value. Typical settings include the number of CPUs on the system or the peak number of process threads. Setting this variable does not affect process performance. The default setting is 256. Note: The **NUM_SPAREVP** environment variable is available only in AIX 5.1.

Example of a multithreaded program

The following short multithreaded program displays "Hello!" in both English and French for five seconds. Compile with **cc_r** or **xlc_r**. F

```
#include <pthread.h>    /* include file for pthreads - the 1st */
#include <stdio.h>      /* include file for printf()           */
#include <unistd.h>     /* include file for sleep()            */

void *Thread(void *string)
{
       while (1)
               printf("%s\n", (char *)string);
       pthread_exit(NULL);
}

int main()
{
       char *e_str = "Hello!";
```

```
        char *f_str = "Bonjour !";

        pthread_t e_th;
        pthread_t f_th;

        int rc;

        rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
        if (rc)
                exit(-1);
        rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
        if (rc)
                exit(-1);
        sleep(5);

        /* usually the exit subroutine should not be used
           see below to get more information */
        exit(0);
}
```

The initial thread (executing the **main** routine) creates two threads. Both threads have the same entry-point routine (the **Thread** routine), but a different parameter. The parameter is a pointer to the string that will be displayed.

Debugging a multithreaded program

The following tools are available to debug multithreaded programs:

- Application programmers can use the **dbx** command to perform debugging. Several subcommands are available for displaying thread-related objects, including **attribute**, **condition**, **mutex**, and **thread**.
- Kernel programmers can use the kernel debug program to perform debugging on kernel extensions and device drivers. The kernel debug program provides limited access to user threads, and primarily handles kernel threads. Several subcommands support multiple kernel threads and processors, including:
  - The **cpu** subcommand, which changes the current processor
  - The **ppd** subcommand, which displays per-processor data structures
  - The **thread** subcommand, which displays thread table entries
  - The **uthread** subcommand, which displays the **uthread** structure of a thread

  For more information on the kernel debug program, see the *Kernel Extensions and Device Support Programming Concepts*.

Core File requirements of a multithreaded program

By default, processes do not generate a full core file. If an application must debug data in shared memory regions, particularly thread stacks, it is necessary to generate a full core dump. To generate full core file information, run the following command as root user: `chdev -l sys0 -a fullcore=true`

Each individual pthread adds to the size of the generated core file. The amount of core file space that a pthread needs includes the stack size, which the user can control with the **pthread_attr_setstacksize** subroutine. For pthreads created with

the NULL pthread attribute, each pthread in a 32-bit process adds 128 KB to the size of the core file, and each pthread in a 64-bit process adds 256 KB to the size of the core file.

Parent topic: Multithreaded programming

Related concepts:
Threadsafe and threaded libraries in AIX
Creating threads
Scheduling threads
Developing multithreaded programs

Developing multithreaded programs to examine and modify pthread library objects

The pthread debug library (**libpthdebug.a**) provides a set of functions that enable application developers to examine and modify pthread library objects.
This library can be used for both 32-bit applications and 64-bit applications. This library is threadsafe. The pthread debug library contains a 32-bit shared object and a 64-bit shared object.
The pthread debug library provides applications with access to the pthread library information. This includes information on pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, and information about the state of the pthread library.
Note: All data (addresses, registers) returned by this library is in 64-bit format both for 64-bit and 32-bit applications. It is the application's responsibility to convert these values into 32-bit format for 32-bit applications. When debugging a 32-bit application, the top half of addresses and registers is ignored.
The pthread debug library does not report information on mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, and read/write lock attibutes that have the pshared value of **PTHREAD_PROCESS_SHARED**.

Initialization

The application must initialize a pthread debug library session for each pthreaded process. The **pthdb_sessison_init** function must be called from each pthreaded process after the process has been loaded. The pthread debug library supports one session for a single process. The application must assign a unique user identifier and pass it to the **pthdb_session_init** function, which in turn assigns a unique session identifier that must be passed as the first parameter to all other **pthread debug library** functions, except **pthdb_session_pthreaded** function, in return. Whenever the pthread debug library invokes a call back function, it will pass the unique application assigned user identifier back to the application. The **pthdb_session_init** function checks the list of call back functions provided by the application, and initializes the session's data structures. Also, this function sets the session flags. An appplication must pass the **PTHDB_FLAG_SUSPEND** flag to the **pthdb_session_init** Function. See the **pthdb_session_setflags** function for a full list of flags.

Call back functions

The pthread debug library uses the call back functions to obtain and write data, as well as to give storage management to the application. Required call back functions for an application are as follows:

- **read_data**
  - Retrieves **pthread library** object information
- **alloc**
  - Allocates memory in the pthread debug library
- **realloc**
  - Reallocates memory in the pthread debug library
- **dealloc**
  - Frees allocated memory in the pthread debug library

Optional call back functions for an application are as follows:

- **read_regs**

- Necessary only for the **pthdb_pthread_context** and **pthdb_pthread_setcontext** subroutines

- write_data

- Necessary only for the **pthdb_pthread_setcontext** subroutine

- write_regs

- Necessary only for **pthdb_pthread_setcontext** subroutine

Update function

Each time the application is stopped, after the session has been initialized, it is necessary to call the **pthdb_session_update** function. This function sets or resets the lists of pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, pthread specific keys, and active keys. It uses call back functions to manage memory for the lists.

Context functions

The **pthdb_pthread_context** function obtains the context information, and the **pthdb_pthread_setcontext** function sets the context. The **pthdb_pthread_context** function obtains the context information of a pthread from either the kernel or the pthread data structure in the application's address space. If the pthread is not associated with a kernel thread, the context information saved by the pthread library is obtained. If a pthread is associated with a kernel thread, the information is obtained from the application using the call back functions. The application must determine if the kernel thread is in kernel mode or user mode and then to provide the correct information for that mode.

When a pthread with kernel thread is in kernel mode, you cannot get the full user mode context because the kernel does not save it in one place. The **getthrds** function can be used to obtain part of this information, because it always saves the user mode stack. The application can discover this by checking the **thrdsinfo64.ti_scount** structure. If this is non-zero, the user mode stack is available in the **thrdsinfo64.ti_ustk** structure. From the user mode stack, it is possible to determine the instruction address register (IAR) and the call back frames, but not the other register values. The **thrdsinfo64** structure is defined in **procinfo.h** file.

List functions

The pthread debug library maintains lists for pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variables attributes, read/write locks, read/write lock attributes, pthread specific keys and active keys, each represented by a type-specific handle. The **pthdb_*object*** functions return the next handle in the appropriate list, where *object* is one of the following: **pthread, attr, mutex, mutexattr, cond, condattr, rwlock, rwlockattr** or **key**. If the list is empty or the end of the list is reached, **PTHDB_INVALID_*OBJECT*** is reported, where *OBJECT* is one of the following: **PTHREAD, ATTR, MUTEX, MUTEXATTR, COND, CONDATTR, RWLOCK, RWLOCKATTR** or **KEY**.

Field functions

Detailed information about an object can be obtained by using the appropriate object member function, **pthdb_*object_field***, where *object* is one of the following: **pthread, attr, mutex, mutexattr, cond, condattr, rwlock, rwlockattr** or **key** and where *field* is the name of a field of the detailed information for the object.

Customizing the session

The **pthdb_session_setflags** function allows the application to change the flags that customize the session. These flags control the number of registers that are read or written during context operations.

The **pthdb_session_flags** function obtains the current flags for the session.

Terminating the session

At the end of the session, the session data structures must be deallocated, and the session data must be deleted. This is accomplished by calling the **pthdb_session_destroy** function, which uses a call back function to deallocate the memory. All of the memory allocated by the **pthdb_session_init**, and **pthdb_session_update** functions will be deallocated.

Example of connecting to the pthread debug library

The following example shows how an application can connect to the pthread debug library: `/* includes */`

```
#include &lt;thread.h>
#include &lt;ys/pthdebug.h>

...

int my_read_data(pthdb_user_t user, pthdb_symbol_t symbols[],int count)
{
 int rc;

 rc=memcpy(buf,(void *)addr,len);
 if (rc==NULL) {
   fprintf(stderr,&odq;Error message\n&cdq;);
   return(1);
 }
 return(0);
}
int my_alloc(pthdb_user_t user, size_t len, void **bufp)
{
 *bufp=malloc(len);
 if(!*bufp) {
   fprintf(stderr,&odq;Error message\n&cdq;);
   return(1);
 }
 return(0);
}
int my_realloc(pthdb_user_t user, void *buf, size_t len, void **bufp)
{
 *bufp=realloc(buf,len);
 if(!*bufp) {
   fprintf(stderr,“Error message\n”);
   return(1);
 }
 return(0);
```

```
}
int my_dealloc(pthdb_user_t user,void *buf)
{
 free(buf);
 return(0);
}

status()
{
 pthdb_callbacks_t callbacks =
                   {  NULL,
                      my_read_data,
                      NULL,
                      NULL,
                      NULL,
                      my_alloc,
                      my_realloc,
                      my_dealloc,
                      NULL
                   };

 ...

 rc=pthread_suspend_others_np();
 if (rc!=0)
   deal with error

 if (not initialized)
   rc=pthdb_session_init(user,exec_mode,PTHDB_SUSPEND|PTHDB_REGS,callbacks,
                         &session);
   if (rc!=PTHDB_SUCCESS)
      deal with error

 rc=pthdb_session_update(session);
 if (rc!=PTHDB_SUCCESS)
       deal with error

  retrieve pthread object information using the object list functions and
  the object field functions

 ...

 rc=pthread_continue_others_np();
 if (rc!=0)
   deal with error
}

...
```

```
main()
{
 ...
}
```

Parent topic: Multithreaded programming

Developing multithreaded program debuggers

The pthread debug library (**libpthdebug.a**) provides a set of functions that allows developers to provide debug capabilities for applications that use the pthread library. The pthread debug library is used to debug both 32-bit and 64-bit pthreaded applications. This library is used to debug targeted debug processes only. It can also be used to examine pthread information of its own application. This library can be used by a multithreaded debugger to debug a multithreaded application. Multithreaded debuggers are supported in the **libpthreads.a** library, which is threadsafe. The pthread debug library contains a 32-bit shared object and a 64-bit shared object.

Debuggers using the ptrace facility must link to the 32-bit version of the library, because the ptrace facility is not supported in 64-bit mode. Debuggers using the /proc facility can link to either the 32-bit version or the 64-bit version of this library. The pthread debug library provides debuggers with access to pthread library information. This includes information on pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, and information about the state of the pthread library. This library also provides help with controlling the execution of pthreads.

Note: All data (addresses, registers) returned by this library is in 64-bit format both for 64-bit and 32-bit applications. It is the debugger's responsibility to convert these values into 32-bit format for 32-bit applications. When debugging a 32-bit application, the top half of addresses and registers is ignored.

The pthread debug library does not report mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, and read/write lock attributes that have the pshared value of PTHREAD_PROCESS_SHARED.

Initialization

The debugger must initialize a pthread debug library session for each debug process. This cannot be done until the pthread library has been initialized in the debug process. The **pthdb_session_pthreaded** function has been provided to tell the debugger when the pthread library has been initialized in the debug process. Each time the **pthdb_session_pthreaded** function is called, it checks to see if the pthread library has been initialized. If initialized, it returns PTHDB_SUCCESS. Otherwise, it returns PTHDB_NOT_PTHREADED. In both cases, it returns a function name that can be used to set a breakpoint for immediate notification that the pthread library has been initialized. Therefore, the **pthdb_session_pthreaded** function provides the following methods for determining when the pthread library has been initialized:

- The debugger calls the function each time the debug process stops, to see if the program that is being debugged is pthreaded.
- The debugger calls the function once and if the program that is being debugged is not pthreaded, sets a breakpoint to notify the debugger when the debug process is pthreaded.

After the debug process is pthreaded, the debugger must call the **pthdb_session_init** function, to initialize a session for the debug process. The pthread debug library supports one session for a single debug process. The debugger must assign a unique user identifier and pass it to **pthdb_session_init** which in turn will assign a unique session identifier which must be passed as the

first parameter to all other pthread debug library functions, except **pthdb_session_pthreaded**, in return. Whenever the pthread debug library invokes a call back function, it will pass the unique debugger assigned user identifier back to the debugger. The **pthdb_session_init** function checks the list of call back functions provided by the debugger, and initializes the session's data structures. Also, this function sets the session flags. See the **pthdb_session_setflags** function in *Technical Reference: Base Operating System and Extensions, Volume 1*.

Call back functions

The pthread debug library uses call back functions to do the following:
- Obtain addresses and data
- Write data
- Give storage management to the debugger
- Aid debugging of the pthread debug library

Update function

Each time the debugger is stopped, after the session has been initialized, it is necessary to call the **pthdb_session_update** function. This function sets or resets the lists of pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, pthread specific keys, and active keys. It uses call back functions to manage memory for the lists.

Hold and unhold functions

Debuggers must support hold and unhold of threads for the following reasons:
- To allow a user to single step a single thread, it must be possible to hold one or more of the other threads.
- For users to continue through a subset of available threads, it must be possible to hold threads not in the set.

The following list of functions perform hold and unhold tasks:
- The **pthdb_pthread_hold** function sets the *hold state* of a pthread to hold.
- The **pthdb_pthread_unhold** function sets the *hold state* of a pthread to unhold. Note: The **pthdb_pthread_hold** and **pthdb_pthread_unhold** functions must always be used, whether or not a pthread has a kernel thread.
- The **pthdb_pthread_holdstate** function returns the *hold state* of the pthread.
- The **pthdb_session_committed** function reports the function name of the function that is called after all of the hold and unhold changes are committed. A breakpoint can be placed at this function to notify the debugger when the hold and unhold changes have been committed.
- The **pthdb_session_stop_tid** function informs the **pthread debug library**, which informs the **pthread library** the thread ID (TID) of the thread that stopped the debugger.
- The **pthdb_session_commit_tid** function returns the list of kernel threads, one kernel thread at a time, that must be continued to commit the hold and unhold changes. This function must be called repeatedly, until PTHDB_INVALID_TID is reported. If the list of kernel threads is empty, it is not necessary to continue any threads for the commit operation.

The debugger can determine when all of the hold and unhold changes have been committed in the following ways:
- Before the commit operation (continuing all of the tids returned by the **pthdb_session_commit_tid** function) is started, the debugger can call the **pthdb_session_committed** function to get the function name and set a breakpoint. (This method can be done once for the life of the process.)
- Before the commit operation is started, the debugger calls the **pthdb_session_stop_tid** function with the TID of the thread that stopped the debugger. When the commit operation is complete, the pthread library ensures that the same TID is stopped as before the commit operation.

To hold or unhold pthreads, use the follow the following procedure, before continuing a group of pthreads or single-stepping a single pthread:
1. Use the **pthdb_pthread_hold** and **pthdb_pthread_unhold** functions to set up which pthreads will be held and which will be unheld.
2. Select the method that will determine when all of the hold and unhold changes have been committed.
3. Use the **pthdb_session_commit_tid** function to determine the list of TIDs that must be continued to commit the hold and unhold changes.
4. Continue the TIDs in the previous step and the thread that stopped the debugger.

The **pthdb_session_continue_tid** function allows the debugger to obtain the list of kernel threads that must be continued before it proceeds with single-stepping a single pthread or continuing a group of pthreads. This function must be called repeatedly, until PTHDB_INVALID_TID is reported. If the list of kernel threads is not empty, the debugger must continue these kernel threads along with the others that it is explicitly interested in. The debugger is responsible for parking the stop thread and continuing the stop thread. The stop thread is the thread that caused the debugger to be entered.

Context functions

The **pthdb_pthread_context** function obtains the context information and the **pthdb_pthread_setcontext** function sets the context. The **pthdb_pthread_context** function obtains the context information of a pthread from either the kernel or the pthread data structure in the debug process's address space. If the pthread is not associated with a kernel thread, the context information saved by the pthread library is obtained. If a pthread is associated with a kernel thread, the information is obtained from the debugger using call backs. It is the debugger's responsibility to determine if the kernel thread is in kernel mode or user mode and then to provide the correct information for that mode.

When a pthread with kernel thread is in kernel mode, you cannot get the full user mode context because the kernel does not save it in one place. The **getthrds** function can be used to obtain part of this information, because it always saves the user mode stack. The debugger can discover this by checking the **thrdsinfo64.ti_scount** structure. If this is non-zero, the user mode stack is available in the **thrdsinfo64.ti_ustk** structure. From user mode stack, it is possible to determine the instruction address register (IAR) and the call back frames, but not the other register values. The **thrdsinfo64** structure is defined in **procinfo.h** file.

List functions

The pthread debug library maintains lists for pthreads, pthread attributes, mutexes, mutex attributes, condition variables, condition variables attributes, read/write locks, read/write lock attributes, pthread specific keys and active keys, each represented by a type-specific handle. The **pthdb_*object*** functions return the next handle in the appropriate list, where *object* is one of the following: **pthread**, **attr**, **mutex**, **mutexattr**, **cond**, **condattr**, **rwlock**, **rwlockattr** or **key**. If the list is empty or the end of the list is reached, **PTHDB_INVALID_*object*** is reported, where *object* is one of the following: **PTHREAD**, **ATTR**, **MUTEX**, **MUTEXATTR**, **COND**, **CONDATTR**, **RWLOCK**, **RWLOCKATTR** or **KEY**.

Field Functions

Detailed information about an object can be obtained by using the appropriate object member function, **pthdb_*object*_*field***, where ***object*** is one of the following: **pthread**, **attr**, **mutex**, **mutexattr**, **cond**, **condattr**, **rwlock**, **rwlockattr** or **key** and where ***field*** is the name of a field of the detailed information for the object.

Customizing the session

The **pthdb_session_setflags** function allows the debugger to change the flags that customize the session. These flags control the number of registers that are read or written to during context operations, and to control the printing of debug information. The **pthdb_session_flags** function obtains the current flags for the session.

Terminating the session

At the end of the debug session, the session data structures must be deallocated, and the session data must be deleted. This is accomplished by calling the **pthdb_session_destroy** function, which uses a call back function to deallocate the memory. All of the memory allocated by the **pthdb_session_init** and **pthdb_session_update** functions will be deallocated.

Example of hold/unhold functions

The following pseudocode example shows how the debugger uses the hold/unhold code:

```
/* includes */


#include <sys/pthdebug.h>


main()
{
   tid_t stop_tid; /* thread which stopped the process */
   pthdb_user_t user = <unique debugger value>;
   pthdb_session_t session; /* <unique library value> */
   pthdb_callbacks_t callbacks = <callback functions>;
   char *pthreaded_symbol=NULL;
   char *committed_symbol;
   int pthreaded = 0;
   int pthdb_init = 0;
   char *committed_symbol;


   /* fork/exec or attach to the program that is being debugged */


   /* the program that is being debugged uses ptrace()/ptracex() with PT_TRACE_ME */
```

```
  while (/* waiting on an event */)
  {
    /* debugger waits on the program that is being debugged */

    if (pthreaded_symbol==NULL) {
      rc = pthdb_session_pthreaded(user, &callbacks, pthreaded_symbol);
      if (rc == PTHDB_NOT_PTHREADED)
      {
          /* set breakpoint at pthreaded_symbol */
      }
      else
        pthreaded=1;
    }
    if (pthreaded == 1 && pthdb_init == 0) {
        rc = pthdb_session_init(user, &session, PEM_32BIT, flags, &callbacks);
        if (rc)
            /* handle error and exit */
        pthdb_init=1;
    }

    rc = pthdb_session_update(session)
    if ( rc != PTHDB_SUCCESS)
/* handle error and exit */

    while (/* accepting debugger commands */)
    {
        switch (/* debugger command */)
        {
            ...
            case DB_HOLD:
                /* regardless of pthread with or without kernel thread */
                rc = pthdb_pthread_hold(session, pthread);
                if (rc)
                    /* handle error and exit */
            case DB_UNHOLD:
                /* regardless of pthread with or without kernel thread */
                rc = pthdb_pthread_unhold(session, pthread);
                if (rc)
                    /* handle error and exit */
            case DB_CONTINUE:
                /* unless we have never held threads for the life */
                /* of the process */
                if (pthreaded)
                {
                    /* debugger must handle list of any size */
                    struct pthread commit_tids;
                    int commit_count = 0;
```

```
                /* debugger must handle list of any size */
                struct pthread continue_tids;
                int continue_count = 0;

   rc = pthdb_session_committed(session, committed_symbol);
   if (rc != PTHDB_SUCCESS)
/* handle error */
          /* set break point  at committed_symbol */

                /* gather any tids necessary to commit hold/unhold */
                /* operations */
                do
                {
                    rc = pthdb_session_commit_tid(session,
                                        &commit_tids.th[commit_count++]);
                    if (rc != PTHDB_SUCCESS)
                        /* handle error and exit */
                } while (commit_tids.th[commit_count - 1] != PTHDB_INVALID_TID);

                /* set up thread which stopped the process to be */
                /* parked using the stop_park function*/

   if (commit_count > 0) {
                  rc = ptrace(PTT_CONTINUE, stop_tid, stop_park, 0,
                                                        &commit_tids);
                  if (rc)
                      /* handle error and exit */

                  /* wait on process to stop */
   }

                /* gather any tids necessary to continue */
                /* interesting threads */
                do
                {
                    rc = pthdb_session_continue_tid(session,
                                    &continue_tids.th[continue_count++]);
                     if (rc != PTHDB_SUCCESS)
                        /* handle error and exit */
                } while (continue_tids.th[continue_count - 1] != PTHDB_INVALID_TID);

                /* add interesting threads to continue_tids */

                /* set up thread which stopped the process to be parked */
                /* unless it is an interesting thread */

                rc = ptrace(PTT_CONTINUE, stop_tid, stop_park, 0,
                                                        &continue_tids);
```

```
                    if (rc)
                        /* handle error and exit */
                }
            case DB_EXIT:
  rc = pthdb_session_destroy(session);
  /* other clean up code */
  exit(0);
            ...
        }
     }

   }
   exit(0);
}
```

Parent topic: Multithreaded programming

Benefits of threads

Multithreaded programs can improve performance compared to traditional parallel programs that use multiple processes. Furthermore, improved performance can be obtained on multiprocessor systems using threads.

Managing threads

Managing threads; that is, creating threads and controlling their execution, requires fewer system resources than managing processes. Creating a thread, for example, only requires the allocation of the thread's private data area, usually 64 KB, and two system calls. Creating a process is far more expensive, because the entire parent process addressing space is duplicated.

The threads library API is also easier to use than the library for managing processes. Thread creation requires only the **pthread_create** subroutine.

Inter-thread communications

Inter-thread communication is far more efficient and easier to use than inter-process communication. Because all threads within a process share the same address space, they need not use shared memory. Protect shared data from concurrent access by using mutexes or other synchronization tools.

Synchronization facilities provided by the threads library ease implementation of flexible and powerful synchronization tools. These tools can replace traditional inter-process communication facilities, such as message queues. Pipes can be used as an inter-thread communication path.

Multiprocessor systems

On a multiprocessor system, multiple threads can concurrently run on multiple CPUs. Therefore, multithreaded programs can run much faster than on a uniprocessor system. They can also be faster than a program using multiple processes, because threads require fewer resources and generate less overhead. For example, switching threads in the same process can be faster, especially in the M:N library model where context switches can often be avoided. Finally, a major advantage of using threads is that a single multithreaded program will work on a uniprocessor system, but can naturally take advantage of a multiprocessor system, without recompiling.

Limitations

Multithreaded programming is useful for implementing parallelized algorithms using several independent entities. However, there are some cases where multiple processes should be used instead of multiple threads.

Many operating system identifiers, resources, states, or limitations are defined at the process level and, thus, are shared by all threads in a process. For example, user and group IDs and their associated permissions are handled at process level. Programs that need to assign different user IDs to their programming entities need to use multiple processes, instead of a single multithreaded process. Other examples include file-system attributes, such as the current working directory, and the state and maximum number of open files. Multithreaded programs may not be appropriate if these attributes are better handled independently. For example, a multi-processed program can let each process open a large number of files without interference from other processes.

Parent topic: Multithreaded programming

Related concepts:
Threadsafe and threaded libraries in AIX