RadiSys.

Using OS-9® Threads

Version 2.6

www.radisys.com

Revision A • April 2003

Contents

1 Threads Overview

This chapter provides a brief conceptual overview of threads. It includes the following sections:

Threads are not supported for OS-9 for 68K operating systems.

Thread Definition

A thread is a single flow of control within a process that performs a program task or a series of program tasks. Generally, threads are composed of the following abstract elements:

- **State Structure**. The state structure includes items like a thread ID, priority, age, signal mask, register context, and program counter.
- **Stack**. A thread has its own stack space for function calling.
- **Private Storage Area**. The private storage area is used for thread-specific data.
- **Attributes**. Thread attributes can be defined to provide thread-specific characteristics.

Threads share a single instance of the following abstract elements:

- **Resource Structure**. The resource structure includes items like a table of open paths, allocated memory, and attached subroutine modules.
- **Global Storage Area**. Global variables are shared among all threads within a process.

In addition, where a process contains multiple threads, the threads execute their instructions independently while sharing a common global data area.

The private storage area resides in user state and is accessed via the thread library calls. The thread registers (such as the stack pointer and program counter) are part of the thread and each thread has its own stack. The code that the thread executes, however, is not part of the thread, but is global and can be executed by any thread. In many cases, two threads of the same process will execute the same function.

All threads in a multi-threaded process share the resources of that process. They share the same allocated memory, and access the same functions and the same global data. If one thread alters a global variable, all other threads will see the change when they next access it. If one thread opens a file and reads it, all other threads can also read from the file.

Thread Architecture

Threads are fundamental elements of the OS-9 operating system. The most basic process is simply a process with a single executing thread. More complicated processes have multiple concurrent threads.

Each process has a single resource descriptor. The resource descriptor contains information such as open paths, allocated memory, and attached subroutine modules. Threads that allocate memory, open paths, or attach subroutine modules all access this common resource descriptor. This allows all threads to share these common resources.

Each process has one or more state descriptors. A state descriptor has the information necessary to maintain the state of a thread of execution: machine register image, signal related information, thread ID, and scheduling information.

Each thread is independently scheduled by the operating system. A process can have low priority threads and high priority threads. All threads in the entire systems are scheduled relative to one another regardless of the process that owns them.

Using Threads

The following sections detail the benefits and limitations of using threads, and the ideal applications for which threads should be used.

Benefits

The overriding benefit of using threads occurs when a process contains multiple threads. A multi-threaded process can perform multiple tasks simultaneously (concurrently or in parallel) within the process. For example, one thread in a process can perform I/O, another thread can perform calculations, and a third thread can operate an user interface.

Some of the common benefits of using threads are indicated below:

- **Provides Increased Throughput**. Multiple threads enable a single process to overlap computation when using one or more blocking system calls. Threads provide this overlap even though each request is coded synchronously. When a thread makes a request and waits, another thread in the process is able to continue. Thus, a process can have several blocking requests outstanding, which enables asynchronous I/O, even though the code is written synchronously.

- **Increases Responsiveness**. With multiple threads in a process, when one part of the process is blocked, the whole process is not necessarily blocked. In typical single-threaded applications, it's possible for the user to encounter a "wait" during a long task. In multiple-threaded applications, the long task can be written as a single thread, enabling the application to remain active in other threads. This can also make the application appear more responsive to the user.
- **Simplifies Interprocess Communications**. A typical multipurpose application uses pipes and sockets for interprocess communications. A multi-threaded application can be written to accomplish the same tasks using the inherently shared memory of the process. The threads in the process can maintain separate interprocess communications connections while sharing data in the global memory space.
- **Uses System Resources More Efficiently**. Multi-process programs typically access common data through shared memory. However, each of these processes must maintain both a state descriptor and a resource descriptor. The cost, in both processing time and memory space, of creating and maintaining these elements makes each process more expensive than a thread. In addition, the inherent separation between processes can require additional effort by the programmer to communicate among the different processes or to synchronize their actions.
- **Simplifies Multi-Tasking Program Structure**. Threads are inherently concurrent, which often simplifies the process of coordinating multiple tasks.
- **Standardizes Source Code**. The use of threads is standardized by the POSIX threads standard. This enables a single source to be recompiled for different platforms.

Limitations

Although there are many benefits to multi-threaded programs, threads have some limitations, including the following:

- **Increased Overhead**. This includes creating, scheduling, and terminating threads within a process. You must determine if the performance gain outweighs the increased overhead.
- **Synchronization**. Threads access global data, open files, and various shared objects with a process. Generally, the access must be synchronized in order to get predictable output from the program. This also includes scheduling your threads. It is possible that one thread in a process will complete prior to the completion of a prerequisite thread, thus producing invalid program output.

Ideal Applications

Generally, applications can be improved by using threads when they have one or both of the following characteristics:

- **Multiple Independent Tasks**. In this case, the application contains more than one task. Each task can proceed to completion independently, without relying on the completion of other tasks.
- **Benefits from Concurrent Execution**. In this case the application's multiple tasks execute faster concurrently than they do serially. Generally, this is the case when a task issues many I/O requests and must wait for the device to complete each request before preceding.

A example of a threaded application is a web server. In this case, a single process must manage multiple simultaneous network connections. This can be implemented using the boss/worker model. The boss thread listens for connection attempts from the network and creates a worker thread for each accepted connection to service the connection. Using threads, the boss portion of the code would not be hindered by slow network access trying to send a file to a client.

Java is another application for threads. The language itself directly supports threads.

Example Using Threads

The following example shows a sample “Hello World” program using threads. It also demonstrates some of the advantages and pitfalls of using threads.

The `pthread_create` function creates a new thread and takes the following four arguments:

- the thread variable or holder for the thread
- a thread attribute
- the function for the thread to call when it starts execution
- an argument to the function

For example:

```
  pthread_t          a_thread;
  pthread_attr_t     a_thread_attribute;
  void               *thread_function(void *argument);
  void               *some_argument;

  pthread_create( &a_thread, &a_thread_attribute,
thread_function,
    some_argument);
```

A thread attribute specifies the minimum stack size to be used. Some applications use the default attribute by passing `NULL` in the thread attribute parameter position. Unlike processes created by the OS-9 fork function, which begin execution at a pre-determined point, threads begin execution at the function specified in `pthread_create`.

Following is an example of a multi-threaded application that prints the "Hello World” message on `stdout`. This requires two thread variables and a function for the new threads to call when they start execution. In addition, there must be a way to specify that each thread should print a different message. One approach is to partition the words into separate character strings and to give each thread a different string as its "startup" parameter.

For example:

```
  void *print_message_function( void *ptr );

  main()
  {
     pthread_t thread1, thread2;
     pthread_attr_t attr;
     char *message1 = "Hello";
     char *message2 = "World";
     pthread_attr_init(&attr);
     pthread_attr_setstacksize(&attr, 4096);

     pthread_create(&thread1, tattr, print_message,
(void*)message1);
     pthread_create(&thread2, tattr, print_message,
(void*)message2);
     exit(0);
  }

  void *print_message_function( void *ptr )
  {
     char *message;
     message = (char *) ptr;
     printf("%s ", message);
     return NULL;
  }
```

Note the function prototype for `print_message_function` and the casts preceding the message arguments in the `pthread_create` call. The program creates the first thread by calling `pthread_create` and passing "Hello" as its startup argument; the second thread is created with "World" as its argument. When the first thread begins execution it starts at the `print_message_function` with its Hello argument. It prints `Hello` and comes to the end of the function. A thread terminates with the return value of its initial function if it leaves its initial function. Therefore, the first thread terminates after printing `Hello`. When the second thread executes it prints `World` and likewise terminates.

While the above program appears reasonable, there are two major flaws. First, the threads execute concurrently; there is no guarantee that the first thread reaches the `printf` function prior to the second thread. Therefore, its possible for the program to output "World Hello" rather than “Hello World“.

Also, there is a more subtle point. Note the call to `exit` made by the parent thread in the main block. If the parent thread executes the `exit` call prior to either of the child threads executing `printf`, no output will be generated. This happens because the `exit` function exits the entire process, terminating all threads. Any thread, parent or child, who calls exit can terminate all the other threads along with the process. Threads wishing to terminate explicitly must use the `pthread_exit` function.

The result is that the Hello World program has two race conditions: the race for the exit call and the race to see which child reaches the `printf` call first.

Below is an example of how the race conditions can be remedied. Since the objective is for each child thread to finish before the parent thread, you could insert a delay in the parent to give the children time to reach `printf`. You could also insert a delay prior to the `pthread_create` call that creates the second thread, which would cause the first child thread to reach the `printf` before the second thread. The resulting code is as follows:

```
  void *print_message_function( void *ptr );

  main()
  {
     pthread_t thread1, thread2;
     char *message1 = "Hello";
     char *message2 = "World";

     pthread_create( &thread1, NULL,
                    print_message_function, (void *)
message1);
     sleep(10);
     pthread_create(&thread2, NULL,
                    print_message_function, (void *)
message2);
```

```
        sleep(10);
        exit(0);
    }

    void *print_message_function( void *ptr )




    {
        char *message;
        message = (char *) ptr;
        printf("%s", message);
        return NULL;
    }
```

There are problems with this solution. It is never safe to rely on timing delays to perform synchronization. The race condition here is identical to a situation with a distributed application and a shared resource. The resource is the standard output and the distributed computing elements are the three threads. `thread1` must use `printf/stdout` prior to `thread2` and both must complete before the parent thread calls `exit`. Another obvious problem created with this solution is that the process now takes 20 seconds to run; `printf` can take less than a second.

Below is a better version that uses `pthread_join` to wait for the threads to terminate. `pthread_join` specifies a thread for which to wait and a place to put the exit status of the target thread. The calling thread blocks until the target thread terminates. The `pthread_exit` status is then returned to the calling thread.

```
    void *print_message_function( void *ptr );

    main()
    {
        pthread_t thread1, thread2;
        char *message1 = "Hello";
        char *message2 = "World";
        void *status;
```

```
     pthread_create(&thread1, NULL,
                    print_message_function, (void *)
message1);

     pthread_join(thread1, &status);

     pthread_create(&thread2, NULL,
                    print_message_function, (void *)
message2);

     pthread_join(thread2, &status);

     exit(0);
  }
  void *print_message_function( void *ptr )
  {
     char *message;
     message = (char *) ptr;
     printf("%s", message);
     return NULL;
  }
```

The POSIX Threads Standard

The IEEE Portable Operating System Interface (POSIX) standard helps developers create source-code portable applications. POSIX 1003.1c (also known as ISO/IEC 9945-1:1990c) is the portion of the overall POSIX standard describing threads. Included are functions and APIs that support multiple threads within a process.

Generally, POSIX threads (Pthreads) are a defined set of C language programming types and calls with a set of implied semantics. Pthreads implementations are usually distributed in the form of a header file (for inclusion in a program) and a library, which is linked to a program.

Pthreads is the basis for the OS-9 implementation of threads. The POSIX specification defines an API that deals with threads management, cancellation, thread-specific data, and synchronization. It provides programmers with the following basic facilities:

- thread creation—the starting of threads
- thread cancellation—asking started threads to shut down in an organized manner
- thread joining—waiting for a particular thread to terminate
- thread-specific data—storing information in a "thread local" area
- mutexes—synchronizing threads to protect critical sections (it is a simple binary semaphore-type lock).
- condition variables—waiting upon notification of an event from another thread (these are rather like simplified OS-9 events)
- threaded initialization—running an initialization function exactly once, but not allowing threads past until it has completed

Refer to the POSIX 1003.1c document for more information about the Pthreads API.

Additional Resources

The following are suggested readings and do not constitute a Microware endorsement:

- IEEE Standard POSIX 1003.1c. Institute of Electrical and Electronics Engineers.
- *Pthreads Programming*; Bradford Nichols, Dick Buttlar & Jaqueline Proulx Farrell; O'Reilly & Associates, Inc; ISBN: 1-56592-115-1.
- *POSIX.4*; Bill O. Gallmeister; O'Reilly & Associates, Inc; ISBN: 1-56592-074-0.
- *Threadtime*; Scott J. Norton & Mark D. Dipasquale; Prentice Hall; ISBN: 0-13-190067-6.

2 Using OS-9 Threads

This chapter describes the OS-9 implementation of POSIX threads. It includes the following sections:

- Overview of OS-9 Threads
- The OS-9 Implementation of POSIX Threads
- OS-9 Threads Guidelines and Issues

Overview of OS-9 Threads

The OS-9 implementation of POSIX threads (Pthreads) defines a thread as an execution context within an OS-9 process. This design enables a process to multi-task within itself. This is beneficial when the work to be done by a single process has aspects of parallelism. This is especially true when I/O is some part of the parallelism.

OS-9 threads are implemented entirely as lightweight processes; each thread acts as a process, but has a much lower overhead in terms of system resources.

The OS-9 API contains support for the following basic facilities:

- Thread creation—the starting of threads
- Thread termination—terminating a thread and returning the status
- Thread operations—setting options for already created threads
- Thread joining—the ability to wait for a particular thread/process to terminate

Refer to Chapter 3 for more information about the API.

The OS-9 Implementation of POSIX Threads

The following sections detail information regarding implementation of POSIX Threads for OS-9.

The OS-9 Kernel

In the OS-9 implementation, POSIX threads are lightweight processes. Each thread behaves like a process, but has a much lower overhead in terms of system resources. The kernel uses one resource descriptor for each process and one state descriptor for each thread. The state descriptors have only the information necessary to maintain and schedule a thread of execution.

The kernel maintains one pointer to void field of data that is swapped at context switch time. This allows multiple threads to look at an identical place in memory and see different values there, depending on which thread is looking at it. This feature is crucial for implementing thread-specific data.

In OS-9, threads within a process are siblings, so there is no concept of parenthood. There is, however, a primordial, or main thread. This is the first thread in the process.

Managing Processes and Threads

The `exit` function (and `_os_exit()`) system call shuts down the entire process, including all of its threads. To shut down just one thread, use `pthread_exit()`.

A process terminates under the following circumstances:

- if any thread in the process makes an exit system call
- if the thread running the main routine returns
- if a fatal signal is delivered
- if a thread causes an uncaught exception

A thread is started using `pthread_create()`. It needs to be passed an attribute object (or NULL to get default attributes), a start routine pointer, and a single argument (type pointer to void.) It returns an error or a thread handle.

A thread may exit with `pthread_exit`, or be terminated with `pthread_cancel` or a signal.

- `pthread_exit` is the normal thread exiting mechanism; it signifies that a thread is shutting itself down voluntarily. Signals can be dangerous, and pthreads do nothing to protect against them. However, thread cancellation is carefully managed. Threads can open themselves for arbitrary cancellation or offer to be cancelled when they call `pthread_testcancel()`, `pthread_cond_wait()`, `pthread_cond_timedwait`, or `pthread_join()`.
- If a thread exits via `pthread_exit()` or is cancelled, it will execute its cleanup stack.
- Threads normally leave information for `pthread_join()`. This is similar to the way OS-9 leaves process descriptors around for `_os_wait()`. `pthread_detach()` tells the library that it doesn't have to leave the descriptor around after the thread terminates. The thread can also be started detached by setting that state in the thread attribute object used to fork the thread.

Do not use Pthread services from within signal intercept routines.

Mutexes in OS-9

A mutex—abbreviated from mutual exclusion—is a simple binary semaphore-type lock. OS-9's mutexes can use priority inheritance or priority ceiling emulation protocol. In OS-9, a Mutex is much like a semaphore and condition variables are a form of OS-9 events. These are supported in the libraries using pre-existing kernel functionality.

Thread Interruption

The OS-9 Pthread implementation supports the concept of interruption as it relates to condition variables. Threads can issue interruption requests to other threads. If the target thread is currently blocked in a `pthread_cond_wait()` or `pthread_cond_timedwait()`, it will be interrupted. The condition variable call will return `EINTR` to its caller. If the target thread is not blocked in a condition variable wait function, the interruption will be made pending. Furthermore, the next call to a condition variable wait function by the target thread will result in `EINTR` being returned. In any case, the mutex associated with the condition variable during the wait will be reacquired, possibly causing the thread to block.

Signals

Thread interruption, cancellation, and suspensions are all implemented using OS-9 signals. Thus, if any of these mechanisms are used, the application must ensure that event waits, sleeps, semaphore operations, process waits, and other blocking operations are aware that "unexpected" signals can arrive. That is, if suspension is being used by the application, the following code will not work correctly if the thread gets suspended during the `_os_sleep`:

```
ticks = 1000;
_os_sleep(&ticks, &sig);
printf("awake\n");
```

If a suspension occurs after the thread has slept 100 ticks and resumption occurs at 150 ticks, `awake` will print after 150 ticks. Correct code would appear as follows:

```
ticks = 1000;
while (ticks)
    _os_sleep(&ticks, &sig);
printf("awake\n");
```

In addition, since these facilities are implemented with signals, it is presumed that threads will not do their own `_os_intercept()` to catch signals and will rely on the `signal()` and `intercept()` library functions for signal handling.

See `_pthread_setsignalrange()` to specify the range of signals that the Pthread layer uses. By default the Pthreads layers use signal values between 40,000 and 49,999 inclusive.

Do not use Pthread services from within signal intercept routines.

POSIX Signals

The signal handling API supports the POSIX function `pthread_kill()`, which directs a signal to a particular thread.

Thread Suspension

The following sections discusses the concerns of thread suspension.

Support

Thread suspension in OS-9 is built around OS-9 signals. When a thread is targeted for suspension it is sent a signal. The signal handler actually contains the code to suspend the thread (an `_os_sema_p()` call) and it is where the thread will block.

The suspender checks the suspendability of the target thread prior to sending the signal. If the target thread is unsuspendable then the suspender polls waiting for the target to be suspendable. Once suspendable, the signal is sent. The suspender then waits for the target thread to indicate it is suspended. If, during this wait for the target to suspend, the target thread is found in any queue but the active one, it is considered suspended. The presumption is that the thread is blocked in I/O or some other queue that is not awakened by a signal, and that once it reenters the active queue it will immediately suspend itself by entering its signal intercept routine.

The following two counters are used to support suspension:

- Suspendability Counter. This counts the number of times a thread has made itself unsuspendable. This supports the notion of nested unsuspendability. For every call to `_pthread_setunsuspendable()` there must be a call to `_pthread_setsuspendable()` for a thread to return to the suspendable state.
- Suspension Counter. This counts the number of times a thread has been requested to suspend. This supports the notion of multiple suspension calls on the same thread. Every call to `_pthread_suspend()` with a given target thread must have a call to `_pthread_resume()` before the target thread will continue to execute.

Application Considerations

The following points discuss issues that are important for designers of applications that use the suspension API. If the application has no need for suspension, these issues do not apply.

In order for the suspension mechanism to work correctly there are a few ground rules that must be followed while a thread is unsuspendable:

- It cannot change the state of the signal mask from masked to unmasked across a "primary" `pthread_setunsuspendable()` call. That is, if signals were masked when the thread set itself unsuspendable for the first time (a non-nested call to `_pthread_setunsuspendable()`), they must remain masked for the entire unsuspendable duration.
- It cannot leave the active queue. Leaving the active queue will be interpreted by the suspender as being "as good as" suspended. The `_pthread_suspend()` call will return to its caller reporting that the target thread has been suspended.

Since thread suspension can happen asynchronously with respect to the target thread's activities, it's possible that the suspended thread may be holding a resource at the time it is suspended. For example, if a thread has claimed a semaphore, but gets suspended before it can release it, other threads that want that same semaphore may block for a very long time waiting for it to be released.

It is for this reason that setting the thread to unsuspendable precedes many lock acquisitions and releases of those same locks are followed by calls to set the thread back to suspendable.

As mentioned previously, certain activities are not permitted while in the unsuspendable state. Thus, the following C library services may not be available (so they should be considered unavailable) if any thread that may have been using them has been suspended:

- `rename()`
- `stdio` functions (all those functions that use `FILE` structures, including those that use `FILE` structures implicitly, for example `printf` and `vprintf`
- `readv()` and `writev()`

Masking signals is the same as setting a thread unsuspendable since a suspension request is implemented by sending a signal from the suspender to the target thread. The suspender will poll waiting for the target thread to receive the signal before it will consider it suspended.

The suspension mechanism implemented in the Pthread layer was designed to be general purpose. That is, design decisions were made that favored working for the maximum number of applications. The results of these decisions are the limitations listed above. More elegant or efficient means of thread suspension could easily be designed for specific applications. If a different approach is used, all the limitations and ground rules listed above need not apply.

OS-9 Threads Guidelines and Issues

This section provides developers with some background and guidelines regarding the considerations and complications when working with threads.

The information in this section was derived from the book *Pthreads Programming* from O'Reilly & Associates. Refer to this book for more information.

These guidelines do not fully address how to design thread oriented code, they merely serve as pointers for writing thread-safe library routines.

Shared Global Data Structures

If multiple threads need access to the same global data structure simultaneously there must be some form of synchronization. This synchronization is probably best accomplished with OS-9 semaphores because they offer the best performance.

The synchronization of access to global data structures can be achieved at a variety of levels (or granularities). For example, consider a linked list accessed by multiple threads simultaneously. The semaphore could simply be locked prior to any access and unlocked after the access. This might be called coarse granularity. A more complicated locking mechanism could be implemented that would provide locking based on the desired operation (e.g. insert, delete, read, write) and/or on individual elements of the linked list. This could be called fine granularity.

An alternative to synchronizing simultaneous access to global variables is to make a separate copy of the global data for each thread. Doing this allows any number of threads to be simultaneously executing the code, but with the additional overhead of numerous copies of the global data area.

At the Pthreads layer, two locking mechanisms are available: mutexes and condition variables. Mutexs are classic binary semaphores. Condition variables offer a thread a way to wait for an event to occur without polling for its occurrence.

It is the programmer's responsibility to ensure that proper locking is done. Nothing in the compiler or operating system will alert the user if the application is violating locking procedures.

Existing code that uses global variables needs to be analyzed to determine whether or not multiple threads using the code will have a problem. In most cases they will.

New Process Structure

The structure used to define a process has changed significantly from the one used in previous versions of the operating system. In order to accommodate lightweight processes (or threads), the information kept in the pre-3.0 process descriptor has been split into two structures. One structure holds information about the process' execution context including the stack, signal and debug information (this structure is `pr_desc`) while the second structure holds the process' resource information, which includes allocated memory, linked modules, and a reference to the process' I/O descriptor (this structure is `pr_rsrc`).

A process that is multi-threaded will have one `pr_desc` structure for each of its threads but will have only one `pr_rsrc` structure.

These new structures are defined in the `process.h` header file, which is located in `/mwos/OS9000/SRC/DEFS`. To maintain backward compatibility, the definitions of these structures are conditional on the definition of `_USE_V3_0_PROCDESC`. If this value is not defined, only the pre-3.0 version definitions in `process.h` will be visible.

Functions to Access the Process Descriptor

Two new functions have been added to allow user applications code to acquire copies of the process descriptor structures. These are `_os_get_prdesc` and `_os_get_prsrc`. These functions return copies of the `pr_desc` and `pr_rsrc` structures respectively for the specified process or thread.

The `_os_gprdsc` function supplied with previous versions of the operating system will continue to return the pre-3.0 version process descriptor structure. The contents of the two process descriptor structures are marshalled by the kernel into the pre-3.0 structure. For users developing code that will work on all OS-9 systems (non-68K), the `_os_gprdsc` function is the preferred way to obtain process descriptor information.

System State Code

For system state code backward compatibility, the `process.h` file contains macros that define the old process descriptor field names so that they map to the correct fields in the new structures. To make system state code compatible with OS-9 v3.0 (non-68K) the user should define `_USE_V3_0_PROCDESC` before including `process.h` in source files and then recompile the code.

Static Return Values

Functions that return values from static variables do not work correctly in a threaded environment. For example, this function may not work correctly when simultaneously called by two threads:

```
char *upper_case(char *str)
{
    static char retbuf[100];
    int i = 0;
    while (*str)
        retbuf[i++] = toupper(*str++);
    return retbuf;
}
```

If a thread gets time sliced before `return retbuf;` (or before the calling thread uses the data) another thread would be able to call this function and change the contents of the buffer.

This problem is difficult to correct. Either the prototype must change so that the caller passes in a buffer to hold the upper-case version, or the return buffer must be dynamically allocated and the caller must be aware that it has to free the buffer after using it. In both cases, the caller's code will have to change to support threading.

This function could be documented as not being thread-safe, forcing the user of the function to create a lock that spans from just prior to the call to just after the final use of the return value. For example,

```
char *uc;
upper_case_lock();
uc = upper_case("Test String");
printf("Upper case version = '%x'\n", uc);
upper_case_unlock();
```

If all code in an application used this same basic technique, `upper_case()` would no longer suffer from threading problems.

The optimal solution is to use the Pthreads key mechanisms to create buffers on a per-thread basis for this function to use. This would allow the API and usage to remain consistent for the client programmer.

Deadlock

Deadlock occurs when two different threads attempt to claim the same mutexes, but in a different order. Consider the following two pseudo-code sequences:

Thread #1

```
 mutex_lock(A);
.
.
.
mutex_lock(B);
```

Thread #2

```
 mutex_lock(B);
.
.
.
mutex_lock(A);
```

The following sequence of events will result in a deadlock:

- Thread #1 gets mutex A
- Thread #1 gets time sliced by the operating system
- Thread #2 gets mutex B
- Thread #2 blocks trying to get semaphore A
- Thread #2 runs again and blocks trying to get semaphore B

At this point, both threads are permanently locked. The only way to avoid this situation is to ensure that all threads in all cases attempt to acquire common locks in the same order.

Thread-safe Coding Techniques

The following points describe thread-safe coding techniques:

- Always lock and unlock synchronization mechanisms as appropriate. Failing to unlock a semaphore usually results in a deadlock. This deadlock may happen to the thread that failed to unlock or it may happen to another thread. Either way, it can be a long time or a long distance away from where the original problem was caused. Use the "best" locking strategy available in the time permitted. That is, a correct non-optimal implementation is always better than a more optimized implementation that pushes the schedule back in order to achieve correctness.
- Do not write functions that return information from static (or global) variables. Although it generally introduces some sort of memory allocation into the system, it is the correct way to return a buffer of information. If only the called function knows the size of the buffer, then create a function that allocates the buffer and a destroy function that frees it (or, specify that the user must free it).
- Avoid deadlock by acquiring locks in the same order all the time.

Threads and Subroutine Modules

This section describes porting an existing subroutine module for use by both threaded and non-threaded applications.

For more information about general subroutine modules, see the OS-9 Technical Manual. The Additional Resources section in Chapter 1 provides a list of background material for threading related issues.

The following procedure describes one way to port an existing subroutine module:

Step 1. Recompile the subroutine module for threading.

A non-threaded application functions much like a threaded application, with only one active thread. Thus, once multi-threaded applications are supported, non-threaded applications are also supported. The largest difference between the two is the way some global data items are handled (described below).

To recompile for threading, add the `-mt` option to the `xcc` command line.

If it is not possible to recompile the subroutine module for threading, a more complicated entry and exit mechanism can be written to "serialize" access to the subroutine module. The mechanism must limit to one, the number of threads that are allowed in the subroutine module at any given time.

Step 2. Change the protocol in the initialization function.

Change the initialization function, in a backwards compatible way, such that threaded applications pass the additional parameter `_pthread`. `_pthread` is a global variable of size pointer to `void`. It is used as a base address for accessing various thread related structures, including such items as thread-specific versions of `_procid` and `errno`.

A common way to change the protocol in a backwards compatible manner is to have threaded applications pass a distinct value for one of the old parameters and then pass an additional parameter (`_pthread`). The dispatcher can then recognize this distinct value and treat the caller as threaded.

Step 3. Change the dispatcher to handle non-threaded callers.

Change the function dispatch and return to fill in a non-threaded caller's `errno`. It must copy the caller's `errno` on entry and copy the subroutine's `errno` on exit.

For threaded users of the subroutine module, `errno` will be shared automatically since `_pthread` is shared between the application and the subroutine module.

Step 4. Change the dispatcher to handle threaded callers.

Some subroutine modules are written with the assumption they will only be called by one thread within an application. For example, if a subroutine module stores the caller's return program counter (PC) in a global variable, it will fail if two or more threads call it at the same time. This problem is normally solved by storing the return PC, for example, in a thread-specific place.

Step 5. Examine the subroutine module functions for thread safety concerns.

Examine the subroutine module functions to ensure they will still function correctly when called by multiple threads within the same process. Add the appropriate locking or thread-specific data to ensure thread safety. The following sections provide for more information.

Shared Data Access Functions

The following two C library functions can be helpful for porting an existing subroutine module. They access two different kinds of data: shared global data and thread-specific data. The shared global data is automatically shared among all modules that have the same value of `_pthread` (i.e. the application and the subroutine module). The thread-specific data is unique to each thread and is visible to all modules that have the same value of `_pthread`.

The functions described below must be used to access this data.

- `_pthread_local_slot()`
  `u_int32 *_pthread_local_slot(int32 slot)`
  This function is used when reading or writing thread-specific versions of "core" C run-time variables. `errno` is a classic example of a local slot. For threaded applications, there exists one `errno` per thread. `_pthread_local_slot()` is used to get the address of the calling thread's version of `errno`.

  The `slot` parameter is the slot number. Slot numbers are defined in `MWOS/SRC/DEFS/pthread.h`. Once a slot number has been assigned to a variable, it will not change in a subsequent release.

  `_pthread_local_slot()` returns the address of the storage for a specific slot number. This makes it equally easy to read or write the variable.

`_pthread_local_slot()` automatically saves and restores any modified registers except the return value. This makes it easier to call from assembly language.

This might be used in the dispatcher during function exit to copy the version of `errno` generated by the code within a subroutine module back to the application's version of `errno`.

A module's global data pointer and `_pthread` value must be valid prior to calling `_pthread_local_slot()`.

- `_pthread_global_slot()`
  `u_int32 *_pthread_global_slot(int32 slot)`

  This function is used when reading or writing global versions of "core" C run-time variables. `_mainid`, the process ID of a thread's host process, is the only example of such a variable.

  `_pthread_global_slot()` is used to get the address of the global version of `_mainid`.

  The `slot` parameter is the slot number. Slot numbers are defined in `MWOS/SRC/DEFS/pthread.h`. Once a slot number has been assigned to a variable, it will not change in a subsequent release.

  `_pthread_global_slot()` returns the address of the storage for a specific slot number. This makes it equally easy to read or write the variable.

  `_pthread_global_slot()` automatically saves and restores any modified registers except the return value. This makes it easier to call from assembly language.

  A module's global data pointer and `_pthread` value must be valid prior to calling `_pthread_global_slot()`.

Example Thread-safe Conversion of a Library

This section describes converting an existing library to a thread-safe library. As shown in the following examples, it is possible to convert a non-thread-safe function to a thread-safe function without changing the API. That is, existing applications do not need source code changes to use the new thread-safe version of the library.

In the following example it is assumed the library contains the following two functions:

```
#include <string.h>
#include <ctype.h>

char *upper_case(char *str)
{
   static char retbuf[100];
   int i = 0;

   if (strlen(str) > 99)
      return NULL;

   while (*str)
      retbuf[i++] = toupper(*str++);
   retbuf[i] = '\0';

   return retbuf;
}

int rand_seed;

int   random()
{
   rand_seed = rand_seed * 1103515245 + 12345;
   return (unsigned int)(rand_seed / 65536) % 32768;
}
```

These functions are not thread-safe. If two threads call `upper_case()` at the same time, their data may become mixed up in the static return buffer `retbuf`. If two threads call `random()` at the same time, the value written to `rand_seed` may not be the same as it would have been if the threads had called `random()` in sequence.

The make files for the library consist of a high-level make file that runs a low-level make file. The high-level make file, `makefile`, is as follows:

```
-b


sh4 : .
   $(MAKE) -f make.gen PROC=SH4 TARGET=-
tp=sh4,lc,ld,lcd,lb
```

The low-level makefile, `make.gen`, is as follows:

```
RDIR     = RELS.$(PROC)
ODIR     = /mwos/OS9000/$(PROC)/LIB
LIB      = randomlib.l
LGOPTS   = -c

CFLAGS   = -cw $(TARGET)

FILES    = $(RDIR)/libsource.r

$(ODIR)/$(LIB) : $(FILES)
        libgen $(LGOPTS) $(FILES) -o=$@
```

Following is a series of steps that describe creating a threading and non-threading version of the above library.

Step 1. Locate functions that are not thread-safe.

Functions that use global data are generally not thread-safe. The `rdump` utility can be used to print the data requirements for a relocatable object file (ROF). Running `rdump` on the ROF generated by the source and make files above results in the following:

```
Module name:  libsource.c
TyLa/RvAt:    0000/0000
Asm valid:    Yes
Create date:  Jan 29, 2001 15:20:32
Edition:      0
Threads:      none
CPU/ROF type: SuperH(SH-4)/15
  Section     Init     Uninit
   Code:      00000070
   Data:      00000000 00000000
 Remote:      00000000 00000068
  Debug:      00000000
  Stack:      00000000
Entry point:  00000000
Excpt entry:  ffffffff
```

Note the 0x68 (104) bytes of uninitialized remote data.

Step 2. Determine how to make functions thread-safe.

There are a variety of ways to handle non thread-safe functions, including the following:

- Document the attribute. If the non thread-safe functions will not be used by multiple threads at the same time, the functions could simply be documented as non thread-safe.
- Change the API. If backwards compatibility is not an issue this is usually the best course of action. In the example, if you passed a buffer to hold the upper-case conversion string then the function would be thread-safe.
- Change the semantics. Again, if backwards compatibility is not a concern, the semantics of a function could be changed. In the example, a buffer to hold the conversion could be dynamically allocated, but the caller would have to know to free the buffer after it was done with it.
- Correct the problem using thread-safety techniques. Fix the function to be thread-safe by adding synchronization or thread-specific data.

In the example, `upper_case()` is fixed by adding thread-specific data, and `random()` is fixed by adding locking. This has the advantage that neither function's API is changed.

Step 3. Conditionalize source code with `_OS9THREAD`.

The automatically defined `_OS9THREAD` macro is used to conditionalize the code to fix the threading issues. When threading is specified in the compiler, `_OS9THREAD` is defined during preprocessing. The code is conditionalized so that both a threaded and a non-threaded version of the library can be built.

For `upper_case()` code is added to create a thread-specific data key and initialize it with a 100 byte buffer for each calling thread. For `random()`, a semaphore is added that ensures that only one thread is using `rand_seed` at one time.

Below is the new source code:

```
#include <string.h>
#include <ctype.h>
#ifdef _OS9THREAD
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>

/* key for upper_case’s thread-specific data */
static pthread_key_t upper_case_key;

/* once block to control threads creating the key */
static pthread_once_t upper_case_once =
PTHREAD_ONCE_INIT;

/* prototype for destructor function */
static void upper_case_key_destroy(void *data);

static void upper_case_key_create(void)
{
   int err;

   err = pthread_key_create(&upper_case_key,
            upper_case_key_destroy);
   if (err != 0) {
      fprintf(stderr,
         "failed to create upper_case() key - %s\n",
            strerror(err));
      exit(err);
   }
}

static void upper_case_key_destroy(void *data)
{
   if (data)
      free(data);
}
```

```
#endif /* _OS9THREAD */

char *upper_case(char *str)
{
  int i = 0;

#ifdef _OS9THREAD
  char *retbuf;
  int err;

  /* ensure key for thread-specific data exists */
  pthread_once(&upper_case_once,
upper_case_key_create);

  /* get the value of the key for this thread */
  retbuf = pthread_getspecific(upper_case_key);
  if (retbuf == NULL) {
    /* need to allocate it */
    retbuf = (char *)malloc(100);
    if (retbuf == NULL)
      return NULL;

    /* set it on the key for next time */
    err = pthread_setspecific(upper_case_key, retbuf);
    if (err != 0)
      return NULL;
  }
#else
  static char retbuf[100];
#endif

  if (strlen(str) > 99)
    return NULL;

  while (*str)
    retbuf[i++] = toupper(*str++);
  retbuf[i] = '\0';

  return retbuf;
}
```

```
int rand_seed;

#ifdef _OS9THREAD
static semaphore sem;
#endif

int  random()
{
   int ret;

#ifdef _OS9THREAD
   /* ensure semaphore is initialized */
   (void)_os_sema_init(&sem);

   /* wait for lock */
   while (_os_sema_p(&sem))
      ;
#endif

   rand_seed = rand_seed * 1103515245 + 12345;
   ret = (unsigned int)(rand_seed / 65536) % 32768;

#ifdef _OS9THREAD
   /* release lock */
   (void)_os_sema_v(&sem);
#endif

   return ret;
}
```

Step 4. Change the make files to build the threaded version.

The make files should be changed to build two different versions of the library: one for non-threaded applications and one for threaded applications. The threaded version begins with the characters "mt_". This allows it to be automatically used if `-mt` is specified on the `xcc` command line.

`makefile` now appears as follows:

```
-b

sh4 : .
    $(MAKE) -f make.gen PROC=SH4 TARGET=-
tp=sh4,lc,ld,lcd,lb
    $(MAKE) -f make.gen PROC=SH4 "TARGET=-
tp=sh4,lc,ld,lcd,lb -mt" MT=mt_
make.gen looks like this:
MT       =
RDIR     = RELS.$(MT)$(PROC)
ODIR     = /MWOS.DELME/OS9000/$(PROC)/LIB
LIB      = $(MT)randomlib.l
LGOPTS   = -c

CFLAGS   = -cw $(TARGET)

FILES    = $(RDIR)/libsource.r

$(ODIR)/$(LIB) : $(FILES)
        libgen $(LGOPTS) $(FILES) -o=$@
```

Step 5. Rebuild the library.

Running the high-level make file now results in both versions of the library being built. Using the `mt_` prefix for the threading version will allow the command line "`xcc test.c -tp=sh4 -l=randomlib.l`" to be used to build a non-threaded application and the command line "`xcc test.c -tp=sh4 -l=randomlib.l -mt`" to be used to build a threaded application.

Because `mt_` was used as a prefix for the library name only, `-mt` had to be added to the command line to compile the threaded version.

Miscellaneous Issues

Following are some issues to consider related to thread support in your OS-9 system:

- Thread-safe libraries are slower and larger than non-thread-safe libraries. Global variable access has to be synchronized and this synchronization takes time, code space, and data space. In general, avoid using threading libraries unless the application is actually threaded.
- Calling a thread-safe library call from a signal handler will likely result in deadlock. If a thread has a lock from a thread-safe routine and gets a signal that causes the signal handler to call the same thread-safe routine then the thread will deadlock with itself.
- Asynchronous death (e.g. exception, kill signal) while holding a lock will result in deadlock if the lock is system global. In addition, the data structures being modified may be in an incorrect state. Pthreads has some code to assist in the clean-up, but it is only useful if the application is notified that it has been terminated.

3 OS-9 Threads Programming Reference

This chapter describes the functions used in the OS-9 Threads implementation. The following sections are included:

POSIX Pthreads Library Functions

The functions in this section are part of the POSIX standard—known as Pthreads. They are compliant with the POSIX standard, and are useful when porting to OS-9 from other operating systems that support the POSIX standard.

Table 3-1 lists all the supported POSIX library functions in alphabetical order. These functions are supported in the library `mt_clib.l`. The descriptions are intended as a reference to show which sub-set of the POSIX standard is supported in this product. If a function listed in the POSIX standard is not described in this document, then it is not currently supported.

The full POSIX standard is *ISO/IEC 9945-1 (POSIX 1003.1c)*. Refer to this standard for clarification of capabilities and function usage.

Table 3-1. POSIX Library Functions

Function Name	Function Description
`pthread_attr_destroy()`	pthread_attr_destroy()
`pthread_attr_getdetachstate()`	Get Detach State Attribute
`pthread_attr_getstackaddr()`	Get Stack Address Attribute
`pthread_attr_getstacksize()`	Get Stack Size Attribute
`pthread_attr_init()`	Allocate Thread Creation Attribute Object
`pthread_attr_setdetachstate()`	Set Detached State Attribute
`pthread_attr_setstackaddr()`	Set Stack Address Attribute
`pthread_attr_setstacksize()`	Set Stack Size Attribute
`pthread_cancel()`	Cancel Target Thread
`pthread_cleanup_pop()`	Pop Cleanup Routine
`pthread_cleanup_push()`	Push Cleanup Routine
`pthread_cond_broadcast()`	Release Threads Waiting for Condition Variable
`pthread_cond_destroy()`	Free Condition Variable Object
`pthread_cond_init()`	Allocate Condition Variable Object
`pthread_cond_signal()`	Release Thread Waiting for Condition Variable
`pthread_cond_timedwait()`	Wait on Condition Variable for Specified Interval
`pthread_cond_wait()`	Wait on Condition Variable
`pthread_condattr_destroy()`	Free Condition Variable Attributes Object
`pthread_condattr_getpshared()`	Get Condition Variable Process-Shared Attribute
`pthread_condattr_init()`	Allocate Condition Variable Attributes Object
`pthread_condattr_setpshared()`	Set Condition Variable Process-Shared Attribute
`pthread_create()`	Create New Thread

Table 3-1. POSIX Library Functions (Continued)

Function Name	Function Description
`pthread_detach()`	Orphan Target Thread
`pthread_equal()`	Compare Thread Identifiers
`pthread_exit()`	Terminate Thread
`pthread_getspecific()`	Get Thread-Specific Data Pointer
`pthread_join()`	Wait for Target Thread to Terminate
`pthread_key_create()`	Create Thread-Specific Data Key
`pthread_key_delete()`	Delete Thread-Specific Data Key
`pthread_kill()`	Send Signal to Target Thread
`pthread_mutex_destroy()`	Free Mutex Object
`pthread_mutex_getprioceiling()`	Get Mutex Priority Ceiling
`pthread_mutex_init()`	Allocate Mutex Object
`pthread_mutex_lock()`	Lock Mutex Object
`pthread_mutex_setprioceiling()`	Set Mutex Priority Ceiling
`pthread_mutex_trylock()`	Lock Mutex Object (Non-Blocking)
`pthread_mutex_unlock()`	Unlock Mutex Object
`pthread_mutexattr_destroy()`	Free Mutex Attributes Object
`pthread_mutexattr_getprioceiling()`	Get Priority Ceiling Attribute
`pthread_mutexattr_getprotocol()`	Get Protocol Attribute
`pthread_mutexattr_getpshared()`	Get Mutex Process-Shared Attribute
`pthread_mutexattr_init()`	Allocate Mutex Attributes Object
`pthread_mutexattr_setprioceiling()`	Set Priority Ceiling Attribute
`pthread_mutexattr_setprotocol()`	Set Protocol Attribute
`pthread_mutexattr_setpshared()`	Set Mutex Process-Shared Attribute
`pthread_once()`	Execute Routine Once per Process
`pthread_self()`	Get Thread Identifier
`pthread_setcancelstate()`	Set Cancel State
`pthread_setcanceltype()`	Set Cancel Type
`pthread_setspecific()`	Set Thread-Specific Data Pointer
`pthread_testcancel()`	Test for Pending Cancel

Further descriptions of functionality and usage are available:

- *Pthreads Programming*; Bradford Nichols, Dick Buttlar & Jaqueline Proulx Farrell; O'Reilly & Associates, Inc; ISBN: 1-56592-115-1
- *POSIX.4*; Bill O. Gallmeister; O'Reilly & Associates, Inc; ISBN: 1-56592-074-0

- *Threadtime*; Scott J. Norton & Mark D. Dipasquale; Prentice Hall; ISBN: 0-13-190067-6

The above list is not a RadiSys endorsement. The texts listed are suggested readings only.

POSIX Pthreads Library Definitions

The functions and definitions in this section are unique to OS-9 and are not part of the POSIX standard, or compatible with any other operating system's libraries. They provide extra functionality not required in the POSIX specification.

Table 3-2 lists the POSIX definitions in alphabetical order. These definitions are supported in the header file `pthread.h`. The descriptions are intended as a reference to show which sub-set of the POSIX standard is supported in this product. If a definition listed in the POSIX standard is not described in this document, then it is not currently supported.

The full POSIX standard is *ISO/IEC 9945-1 (POSIX 1003.1c)*. Please refer to this standard for clarification of capabilities and function usage.

Table 3-2. POSIX Library Definitions

Definition	Definition Description
`_POSIX_THREAD_ATTR_STACKADDR`	Stackaddr Implementation Macro
`_POSIX_THREAD_ATTR_STACKSIZE`	Stacksize Implementation Macro
`_POSIX_THREAD_PRIO_INHERIT`	Priority Inheritance Implementation Macro
`_POSIX_THREAD_PRIO_PROTECT`	Priority Ceiling Implementation Macro
`_POSIX_THREAD_SAFE_FUNCTIONS`	Thread-safe Function Implementation Macro
`_POSIX_THREADS`	Posix Threads Implementation Macro
`PTHREAD_CANCEL_ASYNCHRONOUS`	Asynchronous Cancel Type
`PTHREAD_CANCEL_DEFERRED`	Deferred Cancel Type
`PTHREAD_CANCEL_DISABLE`	Disabled Cancel State
`PTHREAD_CANCEL_ENABLE`	Enabled Cancel State
`PTHREAD_CANCELED`	Cancelled Thread Exit Status

Table 3-2. POSIX Library Definitions (Continued)

Definition	Definition Description
PTHREAD_COND_INITIALIZER	Condition Variable Initializer
PTHREAD_CREATE_DETACHED	Detached Thread Attribute
PTHREAD_CREATE_JOINABLE	Joinable Thread Attribute
PTHREAD_DESTRUCTOR_ITERATIONS	Number of Destruction Attempts
PTHREAD_KEYS_MAX	Maximum Number of Data Keys
PTHREAD_MUTEX_INITIALIZER	Mutex Initializer
PTHREAD_ONCE_INIT	Once Control Initializer
PTHREAD_PROCESS_PRIVATE	Process Private Attribute
PTHREAD_PROCESS_SHARED	Process Shared Attribute
PTHREAD_STACK_MIN	Minimum Thread Stack Size
PTHREAD_THREADS_MAX	Maximum Number of Threads per Process

Pthreads Library Extension Functions

The definitions in this section support the Pthreads library extensions.

Table 3-3 lists the OS-9 extensions to the POSIX Pthread library. These functions provide extra functionality not available under POSIX or other operating systems.

Table 3-3. OS-9 Specific Threads Functions

Function Name	Function Description
_pthread_attr_getinitfunction()	Get Initialization Function Attribute
_pthread_attr_getpriority()	Get Priority Attribute
_pthread_attr_setinitfunction()	Set Initialization Function Attribute
_pthread_attr_setpriority()	Set Priority Attribute
_pthread_getstatus()	Get Thread Status Information
_pthread_interrupt()	Interrupt Target Thread
_pthread_interrupt_clear()	Clear Interrupt Request for Target Thread
_pthread_resume()	Decrement Suspension Counter
_pthread_setpr()	Set Priority for Target Thread
_pthread_setsignalrange()	Set Range of Signal Values
_pthread_setsuspendable()	Decrement Suspendability Counter

Table 3-3. OS-9 Specific Threads Functions (Continued)

Function Name	Function Description
`_pthread_setunsuspendable()`	Increment Suspendability Counter
`_pthread_suspend()`	Increment Suspension Counter

Pthreads Library Extension Definitions

The definitions in this section support the POSIX library functions.

Table 3-4 lists the definitions for the OS-9 extensions to the POSIX Pthread library. These definitions provide extra functionality not available under POSIX or other operating systems. The definitions are supported in the header file `pthread.h`.

Table 3-4. OS-9 Specific Threads Definitions

Definition	Definition Description
`_PT_BOOSTED`	Priority Boosted Status Flag
`_PT_CPENDING`	Cancel Pending Status Flag
`_PT_CSTATE`	Cancel State Status Flag
`_PT_CTYPE`	Cancel Type Status Flag
`_PT_DETACHED`	Detached Thread Status Flag
`_PT_EXIT`	Terminated Thread Status Flag
`_PT_IPENDING`	Interruption Pending Status Flag
`_PT_SFLAG`	Suspended Status Flag
`_PT_SPENDING`	Suspension Pending Status Flag
`_PT_SSTATE`	Suspension State Status Flag

Function Descriptions

This section lists all the functions and descriptions in alphabetical order (without regard for numbers and underscores).

Table 3-5 lists all the functions and descriptions, in alphabetical order. These functions are supported in the library `mt_clib.l`.

Table 3-5. Complete List of Functions and Descriptions

Function Name	Function Description
`pthread_attr_destroy()`	pthread_attr_destroy()
`pthread_attr_getdetachstate()`	Get Detach State Attribute
`_pthread_attr_getinitfunction()`	Get Initialization Function Attribute
`_pthread_attr_getpriority()`	Get Priority Attribute
`pthread_attr_getstackaddr()`	Get Stack Address Attribute
`pthread_attr_getstacksize()`	Get Stack Size Attribute
`pthread_attr_init()`	Allocate Thread Creation Attribute Object
`pthread_attr_setdetachstate()`	Set Detached State Attribute

Table 3-5. Complete List of Functions and Descriptions (Continued)

Function Name	Function Description
_pthread_attr_setinitfunction()	Set Initialization Function Attribute
_pthread_attr_setpriority()	Set Priority Attribute
pthread_attr_setstackaddr()	Set Stack Address Attribute
pthread_attr_setstacksize()	Set Stack Size Attribute
pthread_cancel()	Cancel Target Thread
pthread_cleanup_pop()	Pop Cleanup Routine
pthread_cleanup_push()	Push Cleanup Routine
pthread_cond_broadcast()	Release Threads Waiting for Condition Variable
pthread_cond_destroy()	Free Condition Variable Object
pthread_cond_init()	Allocate Condition Variable Object
pthread_cond_signal()	Release Thread Waiting for Condition Variable
pthread_cond_timedwait()	Wait on Condition Variable for Specified Interval
pthread_cond_wait()	Wait on Condition Variable
pthread_condattr_destroy()	Free Condition Variable Attributes Object
pthread_condattr_getpshared()	Get Condition Variable Process-Shared Attribute
pthread_condattr_init()	Allocate Condition Variable Attributes Object
pthread_condattr_setpshared()	Set Condition Variable Process-Shared Attribute
pthread_create()	Create New Thread
pthread_detach()	Orphan Target Thread
pthread_equal()	Compare Thread Identifiers
pthread_exit()	Terminate Thread
pthread_getspecific()	Get Thread-Specific Data Pointer
_pthread_getstatus()	Get Thread Status Information
_pthread_interrupt()	Interrupt Target Thread
_pthread_interrupt_clear()	Clear Interrupt Request for Target Thread
pthread_join()	Wait for Target Thread to Terminate
pthread_key_create()	Create Thread-Specific Data Key
pthread_key_delete()	Delete Thread-Specific Data Key
pthread_kill()	Send Signal to Target Thread
pthread_mutex_destroy()	Free Mutex Object
pthread_mutex_getprioceiling()	Get Mutex Priority Ceiling
pthread_mutex_init()	Allocate Mutex Object
pthread_mutex_lock()	Lock Mutex Object
pthread_mutex_setprioceiling()	Set Mutex Priority Ceiling
pthread_mutex_trylock()	Lock Mutex Object (Non-Blocking)
pthread_mutex_unlock()	Unlock Mutex Object

Table 3-5. Complete List of Functions and Descriptions (Continued)

Function Name	Function Description
`pthread_mutexattr_destroy()`	Free Mutex Attributes Object
`pthread_mutexattr_getprioceiling()`	Get Priority Ceiling Attribute
`pthread_mutexattr_getprotocol()`	Get Protocol Attribute
`pthread_mutexattr_getpshared()`	Get Mutex Process-Shared Attribute
`pthread_mutexattr_init()`	Allocate Mutex Attributes Object
`pthread_mutexattr_setprioceiling()`	Set Priority Ceiling Attribute
`pthread_mutexattr_setprotocol()`	Set Protocol Attribute
`pthread_mutexattr_setpshared()`	Set Mutex Process-Shared Attribute
`pthread_once()`	Execute Routine Once per Process
`_pthread_resume()`	Decrement Suspension Counter
`pthread_self()`	Get Thread Identifier
`pthread_setcancelstate()`	Set Cancel State
`pthread_setcanceltype()`	Set Cancel Type
`_pthread_setpr()`	Set Priority for Target Thread
`_pthread_setsignalrange()`	Set Range of Signal Values
`pthread_setspecific()`	Set Thread-Specific Data Pointer
`_pthread_setsuspendable()`	Decrement Suspendability Counter
`_pthread_setunsuspendable()`	Increment Suspendability Counter
`_pthread_suspend()`	Increment Suspension Counter
`pthread_testcancel()`	Test for Pending Cancel

pthread_attr_destroy()

Free Thread Attribute Object

Syntax

```
#include <pthread.h>
int pthread_attr_destroy(pthread_attr_t *attr);
```

Description

`pthread_attr_destroy()` tells the library that a pthread attribute object will no longer be used. The attribute, in effect, becomes uninitialized.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	an invalid `pthread_attr_t` pointer was passed

See Also

`pthread_attr_init()`
`pthread_create()`

Example

```
err = pthread_attr_destroy(&attr);
if (err != 0)
   fprintf(stderr, "error destroying attribute - %s\n",
strerror(err));
```

pthread_attr_getdetachstate()

Get Detach State Attribute

Syntax

```
#include <pthread.h>
int pthread_attr_getdetachstate(
     const pthread_attr_t  *attr,
     int                   *detachstate);
```

Description

`pthread_attr_getdetachstate()` gets the detach state attribute in the attribute object. The integer pointed to by `detachstate` will be written with `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`attr` or `detachstate` is invalid or the object pointed to by `attr` is not properly initialized.

See Also

`pthread_attr_init()`
`pthread_attr_setdetachstate()`
`pthread_create()`
`pthread_detach()`
`pthread_join()`

Example

```
err = pthread_attr_getdetachstate(&attr, &state);
if (err != 0)
   fprintf(stderr, "error getting detach state - %s\n",
strerror(err));
```

_pthread_attr_getinitfunction()

Get Initialization Function Attribute

Syntax

```
#include <pthread.h>
int _pthread_attr_getinitfunction(
    const pthread_attr_t *attr,
    int (**initfunc)(void *),
    void **initfunc_arg,
    void **initfunc_gp,
    void **initfunc_cp);
```

Description

`_pthread_attr_getinitfunction()` returns the initialization function pointer and initialization function argument fields from an attribute objects. `attr` is a pointer to an initialized pthread attribute object. `initfunc` points to a place to store the initialization function pointer. `initfunc_arg` points to a place to store the initialization function argument. `initfunc_gp` points to a place to store the initialization function global pointer. `initfunc_cp` points to a place to store the initialization function constant pointer.

Refer to `_pthread_attr_setinitfunction()` for more information about these fields.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`attr` does not refer to an initialized attributes object. `initfunc` or `initfunc_arg` is invalid.

See Also

`pthread_attr_init()`
`_pthread_attr_setinitfunction()`
`pthread_create()`

Example

```
err = _pthread_attr_getinitfunction(&attr, &initfunc,
&initfunc_arg, &gp, &cp);
if (err != 0)
   fprintf(stderr, "error getting initialization
function - %s\n",
   strerror(err));
```

_pthread_attr_getpriority()

Get Priority Attribute

Syntax

```
#include <pthread.h>
int _pthread_attr_getpriority(
     const pthread_attr_t  *attr,
     u_int32               *priority);
```

Description

`_pthread_attr_getpriority()` sets the `u_int32` pointed to by `priority` with the current priority setting from the specified pthread attribute object pointed to by `attr`. A value of 0 indicates that threads created with the specified attribute object will adopt the priority of the creating thread. A non-zero value indicates the desired priority for the created thread.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`attr` or `priority` is invalid or the object pointed to by `attr` is not properly initialized.

See Also

`pthread_attr_init()`
`_pthread_attr_setpriority()`
`pthread_create()`

Example

```
err = _pthread_attr_getpriority(&attr, &pr);
if (err != 0)
  fprintf(stderr, "error getting priority - %s\n",
strerror(err));
```

pthread_attr_getstackaddr()

Get Stack Address Attribute

Syntax

```
#include <pthread.h>
int pthread_attr_getstackaddr(
     const pthread_attr_t  *attr,
     void                  **stackaddr);
```

Description

`pthread_attr_getstackaddr()` gets the thread stack address attribute in the attribute object.

`pthread_attr_getstackaddr()` stores the thread stack address attribute value in `stackaddr` if successful.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`attr` or `stackaddr` is invalid or the object pointed to by `attr` is not properly initialized.

See Also

`pthread_attr_init()`
`pthread_attr_setstackaddr()`
`pthread_create()`

Example

```
err = pthread_attr_getstackaddr(&attr, &stack);
if (err != 0)
   fprintf(stderr, “error getting stack address - %s\n”,
strerror(err));
printf(“Highest stack address is 0x%x\n”, stack);
```

pthread_attr_getstacksize()

Get Stack Size Attribute

Syntax

```
#include <pthread.h>
int pthread_attr_getstacksize(
     const pthread_attr_t  *attr,
     size_t                *stacksize);
```

Description

`pthread_attr_getstacksize()` gets the thread stack size attribute in the attribute object.

`pthread_attr_getstacksize()` stores the thread stack size attribute value in `stacksize` if successful.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`attr` or `stacksize` is invalid or the object pointed to by `attr` is not properly initialized.

See Also

`pthread_attr_init()`
`pthread_attr_setstacksize()`
`pthread_create()`

Example

```
err = pthread_attr_getstacksize(&attr, &size);
if (err != 0)
   fprintf(stderr, “error getting stack size - %s\n”,
strerror(err));
printf(“Stack size will be %u\n”, size);
```

pthread_attr_init()

Allocate Thread Creation Attribute Object

Syntax

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

Description

`pthread_attr_init()` sets default values into the pthread creation attribute object. The default values for a thread creation attribute object are shown in Table 3-6:

Table 3-6. Default values for thread creation attribute

Attribute	Default Value
Stack Size	`PTHREAD_STACK_MIN`
Stack Address	`NULL (system allocated stack)`
Detach State	`PTHREAD_CREATE_JOINABLE`
Priority	`0 (priority of creator)`
Initialization Function	`NULL (none)`

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`ENOMEM`	Insufficient memory exists to initialize the attribute.
`EINVAL`	`attr` is invalid.

See Also

`pthread_create()`

Example

```
err = pthread_attr_init(&attr);
if (err != 0)
   fprintf(stderr, “error initializing attribute -
%s\n”, strerror(err));
```

pthread_attr_setdetachstate()

Set Detached State Attribute

Syntax

```
#include <pthread.h>
int pthread_attr_setdetachstate(
     pthread_attr_t  *attr,
     int             detachstate);
```

Description

`pthread_attr_setdetachstate()` sets the detach state attribute of the specified attribute object. Valid values for detachstate are `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`.

Threads created as joinable retain information upon exit so that status can be returned when `pthread_join()` is used, unless `pthread_detach()` is used to detach the thread.

Threads created as detached automatically free all resources upon exit and cannot be used with `pthread_join()`. These type of threads are forked as "orphan" OS-9 threads.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`attr` or `detachstate` is not valid or `attr` is not properly initialized.

See Also

pthread_attr_init()
pthread_attr_getdetachstate()
pthread_create()
pthread_detach()
pthread_join()

Example

```
err = pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_DETACHED);
if (err != 0)
   fprintf(stderr, “error setting to detached state -
%s\n”, strerror(err));
```

_pthread_attr_setinitfunction()

Set Initialization Function Attribute

Syntax

```
#include <pthread.h>
int _pthread_attr_setinitfunction(
    const pthread_attr_t *attr,
    int (*initfunc)(void *),
    void *initfunc_arg,
    void *gp,
    void *cp);
```

Description

`_pthread_attr_setinitfunction()` sets the initialization function address, argument, global data and constant pointer fields of an attribute object. `attr` is a pointer to an initialized pthread attribute object. `initfunc` points to the initialization function. `initfunc_arg` is the argument to pass as the initialization functions sole argument. `gp` specifies the global data pointer that should be in place when calling `initfunc`. `cp` specifies the constant pointer that should be in place when calling `inifunc`.

If a constant pointer is not applicable for a particular processor or the code is compiled in such a way that a constant pointer is not needed, the value of `cp` may be `NULL`. Passing `NULL` as the `initfunc` parameter disables the calling of an initialization function. Passing `NULL` as the `initfunc_arg` parameter simply specifies that the value of the initialization function parameter should be `NULL`.

The initialization function has the following prototype:

```
int initfunc(void *initfunc_arg);
```

The value of the argument is the value of the `initfunc_arg` parameter in the thread's creation attributes object. If the initialization function returns a non-zero value, that value is converted to a pointer to void and passed to `pthread_exit()`, thus terminating the created thread without ever calling the intended start function.

The initialization function is called in the context of the created thread before the call to `pthread_create()` returns to its caller. The function may perform application specific thread initialization as necessary. The function could be useful in eliminating any race conditions that may exist between the creating thread and created thread since it is known that the initialization code will run in the created thread prior to the return from `pthread_create()`.

Although `pthread_self()` will function correctly, the value it returns should not be communicated to any other threads. The created thread has, technically, not finished its initialization, thus it is not ready to handle all thread operations. The initialization function should not interact with any other threads.

The initialization function runs at the priority of the creating thread, instead of the priority specified for the created thread. That is, if a high priority thread is creating a low priority thread with an initialization function, the initialization function will execute at high priority in the context of the low priority thread.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`attr` does not refer to an initialized attributes object.

See Also

`pthread_attr_init()`
`_pthread_attr_getinitfunction()`
`pthread_create()`
`pthread_exit()`
`get_static()`
`get_const()`

Example

```
err = _pthread_attr_setinitfunction(&attr,
thread_startup, &sema);
if (err != 0)
     fprintf(stderr, "error setting initialization
function - %s\n",
     strerror(err));
```

_pthread_attr_setpriority()

Set Priority Attribute

Syntax

```
#include <pthread.h>
int _pthread_attr_setpriority(
     pthread_attr_t  *attr,
     u_int32         priority);
```

Description

`_pthread_attr_setpriority()` sets the priority attribute of the pthread attribute object pointed to by `attr` to `priority`. A priority of 0 indicates that created threads should adopt the priority of the creating thread. A non-zero value specifies the desired priority for the created thread.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	If `attr` is invalid or the object pointed to by `attr` is not properly initialized or the value of priority is out of range for a thread priority (0 to 65535).

See Also

`pthread_attr_init()`
`_pthread_attr_getpriority()`
`pthread_create()`
`_pthread_setpr()`

Example

```
err = _pthread_attr_setpriority(&attr, 255);
if (err != 0)
   fprintf(stderr, "error setting priority - %s\n",
strerror(err));
```

pthread_attr_setstackaddr()

Set Stack Address Attribute

Syntax

```
#include <pthread.h>
int pthread_attr_setstackaddr(
     pthread_attr_t  *attr,
     void            *stackaddr);
```

Description

`pthread_attr_setstackaddr()` allows a thread to specify a particular pre-allocated thread stack. The address specified is the desired stack pointer for the created thread. The specified stack must be at least `PTHREAD_STACK_MIN` in size.

The `stackaddr` parameter is rounded down to an eight-byte boundary. To get the actual stack address used for created threads with a given attribute object. Use `pthread_attr_getstackaddr()` to get the actual address passed to the next created thread.

There is a matrix of possibilities for the two functions `pthread_attr_setstacksize()` and `pthread_attr_setstackaddr()`. Either one can be called independent of the other one being called. The behavior depends upon the following matrix, shown in Table 3-7.

Table 3-7. Function Behavior

Setstackaddr()	Setstacksize()	Resultant behavior
Not called	Not called	A system-allocated stack, of size PTHREAD_STACK_MIN, will be given to created threads.
Called	Not called	The specified stack address will be passed to created threads. The size will be assumed to be PTHREAD_STACK_MIN.

Table 3-7. Function Behavior (Continued)

Setstackaddr()	Setstacksize()	Resultant behavior
Not called	Called	A system-allocated stack of the size specified will be passed to created threads.
Called	Called	The specified stack address will be passed to created threads. The size will be assumed to be the size set by pthread_attr_setstacksize().

Be aware of the following requirements when setting the stack address explicitly:

- The address passed to this function will be passed directly to threads created with this attribute object. Make sure that the top of the stack is passed (the highest RAM address of the stack).
- Do not create more than one thread with a given stack address.
- The stack should be "pre-loaded" with a NULL link pointer to ensure proper stack back-tracing.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

EINVAL	`attr` is invalid or `attr` is not properly initialized.

See Also

`pthread_attr_init()`
`pthread_attr_getstackaddr()`
`pthread_create()`

Example

```
stack = malloc(PTHREAD_STACK_MIN);
if (stack == NULL)
   fprintf(stderr, “error allocating stack - %s\n”,
strerror(errno));
memset(stack, 0, PTHREAD_STACK_MIN);
err = pthread_attr_setstackaddr(&attr, stack +
stacksize);
if (err != 0)
   fprintf(stderr, “error setting stack address - %s\n”,
strerror(err));
```

pthread_attr_setstacksize()

Set Stack Size Attribute

Syntax

```
#include <pthread.h>
int pthread_attr_setstacksize(
     pthread_attr_t  *attr,
     size_t          stacksize);
```

Description

`pthread_attr_setstacksize()` sets the stack size that will be allocated for threads that are created with the specified attribute object.

The `stacksize` parameter is rounded down to an eight-byte boundary. To get the actual stack size used for created threads with a given attribute object. Use `pthread_attr_getstacksize()` to get the actual stack size attribute used to create threads.

There is a matrix of possibilities for the two functions `pthread_attr_setstacksize()` and `pthread_attr_setstackaddr()`. Either one can be called independent of the other one being called. The behavior depends upon the following matrix, shown in Table 3-8.

Table 3-8. Function Behavior

Setstackaddr()	Setstacksize()	Resultant behavior
Not called	Not called	A system-allocated stack, of size PTHREAD_STACK_MIN, will be given to created threads.
Called	Not called	The specified stack address will be passed to created threads. The size will be assumed to be PTHREAD_STACK_MIN.

Table 3-8. Function Behavior (Continued)

Setstackaddr()	Setstacksize()	Resultant behavior
Not called	Called	A system-allocated stack of the size specified will be passed to created threads.
Called	Called	The specified stack address will be passed to created threads. The size will be assumed to be the size set by pthread_attr_setstacksize().

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`attr` is invalid, `attr` is not properly initialized or `stacksize` is less than `PTHREAD_STACK_MIN`.

See Also

`pthread_attr_init()`
`pthread_attr_getstacksize()`
`pthread_create()`
`PTHREAD_STACK_MIN`

Example

```
err = pthread_attr_setstacksize(&attr, 4096);
if (err != 0)
   fprintf(stderr, "error setting stack size - %s\n",
strerror(err));
```

pthread_cancel()

Cancel Target Thread

Syntax

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

Description

`pthread_cancel()` cancels the target thread unless it is not currently cancelable. If the thread is not cancelable, the request is held pending until it reaches a cancellation point. The call to `pthread_cancel()` returns immediately regardless of the cancelability of the target thread.

If the specified thread has asynchronous cancels enabled it will terminate immediately without doing any sort of cleanup.

When a thread processes a deferred cancel the cleanup routines are called, thread specific data destructors are called, and the thread is terminated with the exit status `PTHREAD_CANCELED`.

Cancelling an asynchronous cancel type thread is guaranteed to cause a loss of resources. For example, the memory allocated to implement thread safety for C library functions will be lost. Use deferred cancellation whenever possible.
In addition, cancelling an asynchronous cancel type thread that is in a queue waiting for a resource will most likely cause the process to exit with an exception.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`ESRCH`	No thread could be found corresponding to that specified by the given thread ID.

See Also

`pthread_cond_timedwait()`
`pthread_cond_wait()`
`pthread_exit()`
`pthread_join()`
`pthread_setcancelstate()`
`pthread_setcanceltype()`

Example

```
err = pthread_cancel(worker);
if (err != 0)
   fprintf(stderr, "error cancelling worker - %s\n",
strerror(err));
```

pthread_cleanup_pop()
Pop Cleanup Routine

Syntax

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
```

Description

`pthread_cleanup_pop()` removes the routine at the top of the cancellation cleanup stack of the calling thread and invokes the popped thread if `execute` is nonzero.

`pthread_cleanup_pop()` and `pthread_cleanup_push()` have to be in the same lexical scope.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

None

See Also

`pthread_cancel()`
`pthread_cleanup_push()`
`pthread_setcancelstate()`
`pthread_setcanceltype()`

Example

```
pthread_cleanup_pop(1);     /* pop and call top cleanup
function */
```

pthread_cleanup_push()

Push Cleanup Routine

Syntax

```
#include <pthread.h>
void pthread_cleanup_push(
     void (*routine)(void *),
     void  *arg);
```

Description

`pthread_cleanup_push()` is similar to the ANSI `atexit()` function. It allows a thread to push a series of routines that should be called if the thread is terminated by `pthread_testcancel()` or `pthread_exit()`. The routines are called in the reverse order that they were pushed onto the cleanup stack. That is, the most recently pushed routine is called first, followed by the next most recent, and so on.

Each `pthread_cleanup_push()` invocation must have an associated `pthread_cleanup_pop()` invocation in the same lexical scope. This is strictly enforced by having the `pthread_cleanup_push()` macro begin with an open brace ({) and the `pthread_cleanup_pop()` macro end with a close brace (}).

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

None

See Also

pthread_cancel()
pthread_cleanup_pop()
pthread_setcancelstate()
pthread_setcanceltype()

Example

```
err = pthread_mutex_lock(mutx);
if (err != 0)
   fprintf(stderr, "error locking mutex - %s\n",
strerror(err));
pthread_cleanup_push(pthread_mutex_unlock, mutx);
err = pthread_cond_wait(condvar, mutx);    /*
cancellation point */
if (err != 0)
   fprintf(stderr, "error during cond_wait - %s\n",
strerror(err));
pthread_cleanup_pop(1);    /* unlock mutx */
```

pthread_cond_broadcast()

Release Threads Waiting for Condition Variable

Syntax

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Description

`pthread_cond_broadcast()` releases every thread waiting on the specified condition variable.

If more than one thread is blocked on a condition variable, the OS-9 scheduler determines the order in which threads are activated. When each thread is unblocked it returns from its call to `pthread_cond_wait()` or `pthread_cond_timedwait()`. The thread owns the mutex with which it called `pthread_cond_wait()` or `pthread_cond_timedwait()`. The thread(s) that are unblocked contend for the mutex in the normal fashion, as if each had called `pthread_mutex_lock()`.

`pthread_cond_broadcast()` may be called by a thread whether or not that thread currently owns the mutex that threads calling `pthread_cond_wait()` or `pthread_cond_timedwait()` have associated with the condition variable during their waits. However, if predictable scheduling behavior is required, then that mutex should be locked by the thread calling `pthread_cond_broadcast()`.

`pthread_cond_broadcast()` has no effect if there are no threads currently blocked on `cond`.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value `cond` does not refer to an initialized condition variable.

See Also

`pthread_cond_init()`
`pthread_cond_timedwait()`
`pthread_cond_wait()`
`pthread_cond_signal()`

Example

```
err = pthread_cond_broadcast(cond);
if (err != 0)
   fprintf(stderr, "failed to signal readers - %s\n",
strerror(err));
err = pthread_mutex_unlock(data_lock);
if (err != 0)
   fprintf(stderr, "failed to unlock data lock - %s\n",
strerror(err));
```

pthread_cond_destroy()

Free Condition Variable Object

Syntax

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

Description

The function `pthread_cond_destroy()` destroys the given condition variable specified by `cond`; the object becomes, in effect, uninitialized.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EBUSY`	An attempt to destroy the object referenced by `cond` while it is in use by another thread. For example, while being used in a `pthread_cond_wait()` or a `pthread_cond_timedwait()`.
`EINVAL`	The value specified by `cond` is invalid.

See Also

`pthread_cond_broadcast()`
`pthread_cond_init()`
`pthread_cond_signal()`
`pthread_cond_timedwait()`
`pthread_cond_wait()`

Example

```
err = pthread_cond_destroy(&cond);
if (err != 0)
   fprintf(stderr, "failed to destroy condvar - %s\n",
strerror(err));
```

pthread_cond_init()

Allocate Condition Variable Object

Syntax

```
#include <pthread.h>
int pthread_cond_init(
     pthread_cond_t            *cond,
     const pthread_condattr_t  *attr);
```

Description

The function `pthread_cond_init()` initializes the condition variable referenced by `cond` with attributes referenced by `attr`. If `attr` is `NULL`, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable becomes initialized.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EAGAIN`	The system lacked the necessary resources (other than memory) to initialize another condition variable.
`ENOMEM`	Insufficient memory exists to initialize the condition variable.
`EBUSY`	An attempt to reinitialize the object referenced by `cond` (a previously initialized, but not yet destroyed, condition variable) has been detected.
`EINVAL`	The value specified by `cond` or `attr` is invalid.

See Also

`pthread_cond_broadcast()`
`pthread_cond_destroy()`
`pthread_cond_signal()`
`pthread_cond_timedwait()`
`pthread_cond_wait()`
`pthread_condattr_init()`
`PTHREAD_COND_INITIALIZER`

Example

```
err = pthread_cond_init(&cond, NULL);
if (err != 0)
   fprintf(stderr, “failed to initialize condvar -
%s\n”, strerror(err));
```

pthread_cond_signal()

Release Thread Waiting for Condition Variable

Syntax

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

Description

`pthread_cond_signal()` releases one thread waiting on the specified condition variable.

When the thread is unblocked it returns from its call to `pthread_cond_wait()` or `pthread_cond_timedwait()`. The thread owns the mutex with which it called `pthread_cond_wait()` or `pthread_cond_timedwait()`. The thread that is unblocked contends for the mutex in the normal fashion, as if it had called `pthread_mutex_lock()`.

`pthread_cond_signal()` may be called by a thread whether or not that thread currently owns the mutex that threads calling `pthread_cond_wait()` or `pthread_cond_timedwait()` have associated with the condition variable during their waits. However, if predictable scheduling behavior is required, then that mutex should be locked by the thread calling `pthread_cond_signal()`.

`pthread_cond_signal()` has no effect if there are no threads currently blocked on `cond`.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value `cond` does not refer to an initialized condition variable.

See Also

`pthread_cond_init()`
`pthread_cond_timedwait()`
`pthread_cond_wait()`
`pthread_cond_broadcast()`

Example

```
err = pthread_cond_signal(cond);
if (err != 0)
   fprintf(stderr, "failed to signal worker - %s\n",
strerror(err));
err = pthread_mutex_unlock(work_que_lock);
if (err != 0)
   fprintf(stderr, "failed to unlock work queue - %s\n",
strerror(err));
```

pthread_cond_timedwait()

Wait on Condition Variable for Specified Interval

Syntax

```
#include <pthread.h>
int pthread_cond_timedwait(
     pthread_cond_t          *cond,
     pthread_mutex_t         *mutex,
     const struct timespec   *abstime);
```

Description

`pthread_cond_timedwait()` is used to block on a condition variable until an absolute time is reached. It must be called with `mutex` locked by the calling thread or `EINVAL` will be returned.

This function releases the mutex and causes the calling thread to block on the condition variable `cond`. If another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_signal()` or `pthread_cond_broadcast()` in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon return, the mutex is locked and is owned by the calling thread. When using condition variables, there is always a boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_timedwait()` may occur. Since the return from `pthread_cond_timedwait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon each return.

The effect of using more than one mutex for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable will result in `EINVAL` errors being returned. That is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this dynamic binding ends when the last concurrent wait returns.

A condition wait is a cancellation point. When the cancelability enable state of a thread is set to `PTHREAD_CANCEL_DEFERRED`, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is re-acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to `pthread_cond_timedwait()`, but at that point notices the cancellation request and instead of returning to the caller of `pthread_cond_timedwait()`, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to `pthread_cond_timedwait()` does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The `timespec` pointed to by `abstime` specifies an absolute time in GMT that the call should return if the thread is not awakened by a `pthread_cond_signal()` or `pthread_cond_broadcast()`.

The Microware Pthread implementation supports the concept of interruption as it relates to condition variable waits. If a thread has a pending interruption or is interrupted while blocked, `pthread_cond_timedwait()` will return `EINTR`. The mutex will be re-acquired prior to return.

If successful, returns a value of 0; otherwise, returns an error.

This function contains a cancel point.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`ETIMEDOUT`	The time specified by `abstime` to

	`pthread_cond_timedwait()` has passed.
`EINVAL`	The value specified by `cond`, `mutex`, or `abstime` is invalid. Different mutexes are supplied for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable. The mutex is not owned by the current thread at the time of the call.

Additional Error

`EINTR`	`_pthread_interrupt()` was called with this thread as the target prior to or during this call.

See Also

`pthread_cond_broadcast()`
`pthread_cond_signal()`
`pthread_cond_wait()`

Example

```
err = _os_getime(&tspec->tv_sec, &ticks);
if (err != SUCCESS)
   fprintf(stderr, "error getting GMT - %s\n", strerror(err));
tspec->tv_sec += 5; /* give up after 5 seconds */
tspec->tv_nsec = 0;
err = pthread_cond_timedwait(cond, mutx, tspec);
switch (err) {
case EINTR:
   fputs("timed wait interrupted\n", stderr);
   break;
case ETIMEDOUT:
   fputs("timed wait timed out\n", stderr);
   break;
default:
   fprintf(stderr, "error on timed wait - %s\n", strerror(err));
   break;
}
```

pthread_cond_wait()
Wait on Condition Variable

Syntax

```
#include <pthread.h>
int pthread_cond_wait(
     pthread_cond_t    *cond,
     pthread_mutex_t   *mutex);
```

Description

`pthread_cond_wait()` is used to block on a condition variable. It must be called with `mutex` locked by the calling thread or `EINVAL` will be returned.

This function releases the mutex and causes the calling thread to block on the condition variable `cond`. If another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_signal()` or `pthread_cond_broadcast()` in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon return, the mutex is locked and is owned by the calling thread. When using condition variables, there is always a boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_wait()` may occur. Since the return from `pthread_cond_wait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon each return.

The effect of using more than one mutex for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable will result in `EINVAL` errors being returned. That is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this dynamic binding ends when the last concurrent wait returns.

A condition wait is a cancellation point. When the cancelability enable state of a thread is set to `PTHREAD_CANCEL_DEFERRED`, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is re-acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to `pthread_cond_wait()`, but at that point notices the cancellation request and instead of returning to the caller of `pthread_cond_wait()`, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to `pthread_cond_wait()` does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The Microware Pthread implementation supports the concept of interruption as it relates to condition variable waits. If a thread has a pending interruption or is interrupted while blocked, `pthread_cond_wait()` will return `EINTR`. The mutex will be re-acquired prior to return.

If successful, returns a value of 0; otherwise, returns an error.

This function contains a cancel point.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `cond` or `mutex` is invalid. Different mutexes are supplied for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable. The mutex is not owned by the current thread at the time of the call.

Additional Error

`EINTR`	`_pthread_interrupt()` was called with this thread as the target prior to or during this call.

See Also

`pthread_cond_broadcast()`
`pthread_cond_signal()`
`pthread_cond_timedwait()`

Example

```
err = pthread_cond_wait(cond, mutx);
if (err != 0)
   fprintf(stderr, "failed to wait on condvar\n",
strerror(err));
```

pthread_condattr_destroy()

Free Condition Variable Attributes Object

Syntax

```
#include <pthread.h>
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Description

`pthread_condattr_destroy()` destroys a condition variable attributes object; the object becomes, in effect, uninitialized.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid.

See Also

`pthread_cond_init()`
`pthread_condattr_init()`

Example

```
err = pthread_condattr_destroy(attr);
if (err != 0)
   fprintf(stderr, “failed to destory condattr - %s\n”,
strerror(err));
```

pthread_condattr_getpshared()

Get Condition Variable Process-Shared Attribute

Syntax

```
#include <pthread.h>
int pthread_condattr_getpshared(
     const pthread_condattr_t  *attr,
     int                       *pshared);
```

Description

`pthread_condattr_getpshared()` obtains the value of the process-shared attribute from the attributes object referenced by `attr`.

If successful, returns 0 and stores the value of the process-shared attribute of `attr` into the object referenced by the `pshared` parameter. Otherwise, an error number is returned.

This facility is not currently supported in Microware's Pthreads implementation. The process-shared attribute can be changed, but both values behave like `PTHREAD_PROCESS_PRIVATE`. `_POSIX_THREAD_PROCESS_SHARED` is not currently defined.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by attr is invalid.

See Also

`pthread_condattr_init()`
`pthread_condattr_setpshared()`

Example

```
err = pthread_condattr_getpshared(attr, &pshare);
if (err != 0)
   fprintf(stderr, “failed to get pshared attribute -
%s\n”, strerror(err));
```

pthread_condattr_init()

Allocate Condition Variable Attributes Object

Syntax

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
```

Description

`pthread_condattr_init()` initializes a condition variable attributes object `attr` with the default value for all of the attributes. These are shown in Table 3-9.

Table 3-9. Default Attribute Values

Attribute	Default Value
Process-shared	`PTHREAD_PROCESS_PRIVATE`

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`ENOMEM`	Insufficient memory exists to initialize the condition variable attributes object.
`EINVAL`	`attr` is an invalid value.

See Also

`pthread_cond_init()`
`pthread_condattr_destroy()`

Example

```
err = pthread_condattr_init(&condattr);
if (err != 0)
   fprintf(stderr, “failed to init condattr - %s\n”,
strerror(err));
```

pthread_condattr_setpshared()

Set Condition Variable Process-Shared Attribute

Syntax

```
#include <pthread.h>
int pthread_condattr_setpshared(
     pthread_condattr_t  *attr,
     int                 pshared);
```

Description

`pthread_condattr_setpshared()` sets the process-shared attribute in an initialized attributes object referenced by `attr`. Valid values for pshared are `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

If successful, returns a value of 0; otherwise, returns an error.

This facility is not currently supported in the Microware Pthreads implementation. The process-shared attribute can be changed, but `PTHREAD_PROCESS_SHARED` behaves exactly like `PTHREAD_PROCESS_PRIVATE`. `_POSIX_THREAD_PROCESS_SHARED` is not currently defined.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid. The new value specified for the attribute is outside the range of legal values for that attribute.

See Also

`pthread_condattr_init()`
`pthread_condattr_getpshared()`

Example

```
err = pthread_condattr_setpshared(&condattr,
PTHREAD_PROCESS_PRIVATE);
if (err != 0)
   fprintf(stderr, "failed to set to private - %s\n",
strerror(err));
```

pthread_create()

Create New Thread

Syntax

```
#include <pthread.h>
int pthread_create(
     pthread_t                *thread,
     const pthread_attr_t     *attr,
     void                     *(*start_routine) (void *),
     void                     *arg);
```

Description

`pthread_create()` is used to create a new thread, with attributes specified by `attr`, within a process. If `attr` is `NULL`, the default attributes are used. If the attributes specified by `attr` are modified later, the attributes of the thread are not affected. Upon successful completion, `pthread_create()` stores the ID of the created thread in the location referenced by thread.

The thread starts by executing `start_routine` with `arg` as its sole argument. If the `start_routine` returns, the effect is as if there was an implicit call to `pthread_exit()` using the return value of `start_routine` as the exit status.

The thread in which `main()` was originally invoked differs from this. When this thread returns from `main()`, the effect is as if there was an implicit call to `exit()` using the return value of `main()` as the exit status.

If `pthread_create()` fails, no new thread is created, and the contents of the location referenced by `thread` are undefined.

The pthread `attr` structure is used when threads are created.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User

Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EAGAIN`	The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process `PTHREAD_THREADS_MAX` would be exceeded.
`EINVAL`	The value specified by `attr` is invalid.

See Also

`_os_thfork()`
`pthread_exit()`
`pthread_join()`
`pthread_detach()`

Example

```
err = pthread_create(&tid, &worker_attr, worker_loop,
NULL);
if (err != 0)
   fprintf(stderr, "error creating worker - %s\n",
strerror(err));
```

pthread_detach()

Orphan Target Thread

Syntax

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

Description

`pthread_detach()` orphans the designated thread. Any thread within the caller's process can be detached unless it is already in detached state.

The `pthread_detach()` function is used to indicate that storage for the thread can be reclaimed when that thread terminates. If thread has not terminated, `pthread_detach()` does not cause it to terminate. Multiple `pthread_detach()` calls on the same target thread result in `EINVAL` being returned.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `thread` does not refer to a thread that can be joined.
`ESRCH`	No thread could be found corresponding to that specified by the given thread ID.

See Also

`pthread_join()`

Example

```
err = pthread_detach(io_thread);
if (err != 0)
   fprintf(stderr, “error detaching I/O thread - %s\n”,
strerror(err));
```

pthread_equal()

Compare Thread Identifiers

Syntax

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

Description

`pthread_equal()` tests whether two thread IDs are the same.

`pthread_equal()` returns a nonzero value if the two thread IDs are equal; otherwise 0 is returned.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

None

See Also

`pthread_self()`

Example

```
if (pthread_equal(worker[0], dead))
   fputs("worker #0 died”, stderr);
```

pthread_exit()

Terminate Thread

Syntax

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

Description

`pthread_exit()` terminates the calling thread. If any thread is waiting on a join on this thread, they are released and passed `value_ptr` as the exit status.

`pthread_exit()` terminates the calling thread and makes the value `value_ptr` available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped, shall be popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions are called in an unspecified order. Thread termination does not release any application visible process resources (e.g. allocated memory, open paths, etc.). Nor does it perform any process level cleanup actions like calling any `atexit()` routines that may exist.

An implicit call to `pthread_exit()` is made when a thread other than the thread in which `main()` was first invoked returns from the start routine that was used to create it. The return value of the function serves as the exit status of the thread.

`pthread_exit()` returns immediately without doing anything if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `pthread_exit()`.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the `pthread_exit()` `value_ptr` parameter value.

The process exits with an exit status of 0 after the last thread has been terminated. The behavior is as if the implementation called `exit()` with a zero argument at the time of thread termination.

Calling `pthread_exit` from the thread in which main was first invoked does not necessarily cause the process to exit. The process will continue to run until all threads have terminated or an exit call is made.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

None

See Also

`exit()`
`pthread_create()`
`pthread_join()`
`pthread_detach()`
`PTHREAD_DESTRUCTOR_ITERATIONS`

Example

```
pthread_exit((void *)SUCCESS);
```

pthread_getspecific()

Get Thread-Specific Data Pointer

Syntax

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
```

Description

`pthread_getspecific()` returns the value currently bound to the specified key on behalf of the calling thread.

`pthread_getspecific()` returns the thread-specific data value associated with the given key. If no thread-specific data value is currently associated with key, then the value `NULL` is returned.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

None

See Also

`pthread_key_create()`
`pthread_setspecific()`

Example

```
thread_data = pthread_getspecific(thread_data_key);
if (thread_data == NULL) {
   thread_data = malloc(sizeof(thread_data_t));
   if (thread_data == NULL)
      fprintf(stderr, “memory allocation error - %s\n”,
strerror(errno));
   pthread_setspecific(thread_data_key, thread_data);
}
```

_pthread_getstatus()

Get Thread Status Information

Syntax

```
#include <pthread.h>
int _pthread_getstatus(
     pthread_t           thread,
     _pthread_status_t   *status);
```

Description

`_pthread_getstatus()` returns various pieces of information related to the target thread in the structure pointed to by `status`. The following table describes the fields of the `_pthread_status_t` structure:

Table 3-10. _pthread_status_t Structure Fields

Type	Name	Description
`u_int32` (bit masks follow)	status	Bits for various boolean information:
`_PT_DETACHED` 0 = joinable thread 1 = detached thread		
`_PT_EXIT` 0 = thread has terminated 1 = thread has not yet terminated		
`_PT_CSTATE` 0 = cancels enabled 1 = cancels disabled		
`_PT_CTYPE` 0 = deferred cancels 1 = asynchronous cancels		
`_PT_CPENDING` 0 = no cancel request pending 1 = cancel request pending		
`_PT_SSTATE`		

Table 3-10. _pthread_status_t Structure Fields (Continued)

Type	Name	Description
0 = suspendable 1 = unsuspendable		

`_PT_SPENDING`
0 = no suspend request pending
1 = suspend request pending
`_PT_SFLAG`
0 = not suspended
1 = suspended
`_PT_BOOSTED`
0 = not priority boosted
1 = priority boosted
`_PT_IPENDING`
0 = no interruption pending
1 = interruption pending		

`thread_t`	tid	OS-9 thread identifier of thread
`thread_t`	creator	OS-9 thread identifier of thread's creator
`void *`	stack	stack base (highest address)
`size_t`	stack_size	Stack size in bytes
`u_int16`	priority	thread's priority
`u_int16`	bpriority	thread's boosted priority
`u_int32 [2]`	resv	reserved space for future additional status information

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The passed thread or status pointer is NULL.
`ESRCH`	The specified target thread is not valid.

Example

```
err = _pthread_getstatus(child, &stats);
if (err != 0)
   fprintf(stderr, "failed to get status for child -
%s", strerror(err));
printf("child's OS-9 thread ID is %u\n", stats.tid);
```

_pthread_interrupt()
Interrupt Target Thread

Syntax

```
#include <pthread.h>
int _pthread_interrupt(pthread_t thread);
```

Description

`_pthread_interrupt()` interrupts any `pthread_cond_wait()` or `pthread_cond_timedwait()` being done by the specified thread. If the thread is not currently blocked in `pthread_cond_wait()` or `pthread_cond_timedwait()`, `_pthread_interrupt()` makes the interruption pending.

`_pthread_interrupt()` is implemented as if the target thread can atomically check for a pending interrupt and then block in `pthread_cond_timedwait()` or `pthread_cond_wait()` if none is pending. That is, there is no window between when a thread checks for a pending interrupt and when the thread actually blocks where an interruption request could be missed.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`thread` is invalid.
`ESRCH`	`thread` is not a valid thread.

See Also

`pthread_cond_timedwait()`
`pthread_cond_wait()`
`_pthread_getstatus()`
`_pthread_interrupt_clear()`

Example

```
err = _pthread_interrupt(wait_thread);
if (err != 0)
   fprintf(stderr, “failed to interrupt waiter - %s\n”,
strerror(err));
```

_pthread_interrupt_clear()

Clear Interrupt Request for Target Thread

Syntax

```
#include <pthread.h>
int _pthread_interrupt_clear(
   pthread_t thread,
   int *old_status);
```

Description

`_pthread_interrupt_clear()` clears any pending interrupt for the specified thread. This function might be useful if other interruptible operations are defined for a particular application. Refer to `_pthread_getstatus()` for more information on determining if a particular thread has an interruption pending.

The value of the interruption status for the target thread is returned at the integer pointed to by `old_status`. If the old status is not required, `NULL` may be passed for `old_status`.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`thread` is invalid.
`ESRCH`	`thread` is not a valid thread.

See Also

`pthread_cond_timedwait()`
`pthread_cond_wait()`
`_pthread_interrupt()`
`_pthread_getstatus()`

Example

```
err = _pthread_interrupt_clear(pthread_self());
if (err != 0)
   fprintf(stderr, "failed to clear interruption -
%s\n", strerror(err));
```

pthread_join()

Wait for Target Thread to Terminate

Syntax

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

Description

The `pthread_join()` function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated. On return from a successful `pthread_join()` call with a non-NULL `value_ptr` argument, the value passed to `pthread_exit()` by the terminating thread is stored in the location referenced by `value_ptr`.

When a `pthread_join()` returns successfully, the target thread has been terminated. Multiple simultaneous calls to `pthread_join()` specifying the same target thread results in one thread successfully getting the exit status and the remainder getting `EOS_NOCHLD` as the result of `pthread_join()`.

Exited but remaining unjoined threads count against the maximum number of threads a process may have, `PTHREAD_THREADS_MAX`.

If successful, returns a value of 0; otherwise, returns an error.

This function contains a cancel point.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `thread` does not refer to a thread that can be joined.
`ESRCH`	No thread could be found corresponding to that specified by the given thread ID.
`EDEADLK`	A deadlock was detected, or the value of `thread` specifies the calling thread.

See Also

`pthread_create()`
`pthread_detach()`
`pthread_exit()`

Example

```
err = pthread_join(child, &status);
if (err != 0)
   fprintf(stderr, "error waiting for child - %s\n",
strerror(err));
printf("Child's exit status was %u\n", status);
```

pthread_key_create()

Create Thread-Specific Data Key

Syntax

```
#include <pthread.h>
int pthread_key_create(
     pthread_key_t  *key,
     void           (*destructor) (void *));
```

Description

`pthread_key_create()` creates a thread-specific data key visible to all threads in the process. Key values provided by `pthread_key_create()` are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by `pthread_setspecific()` are maintained on a per-thread basis and persist for the life of the calling thread.

If successful, `pthread_key_create()` stores the newly created key value at `*key` and returns 0. Otherwise, an error number is returned.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EAGAIN`	The system lacked the necessary resources to create another thread-specific data key, or the limit on the total number of keys per process, `PTHREAD_KEYS_MAX`, has been exceeded.
`EINVAL`	The key value is invalid.
`ENOMEM`	Insufficient memory exists to create the key.

See Also

`pthread_getspecific()`
`pthread_key_delete()`
`pthread_setspecific()`
`PTHREAD_KEYS_MAX`

Example

```
err = pthread_key_create(&thread_data_key,
free_thread_data);
if (err != 0)
   fprintf(stderr, “failed to create key - %s\n”,
strerror(err));
```

pthread_key_delete()

Delete Thread-Specific Data Key

Syntax

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
```

Description

`pthread_key_delete()` deletes a thread-specific data key previously returned by `pthread_key_create()`. The thread-specific data values associated with key need not be `NULL` at the time `pthread_key_delete()` is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after `pthread_key_delete()` is called.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The key value is invalid.

See Also

`pthread_key_create()`
`pthread_getspecific()`
`pthread_setspecific()`

Example

```
err = pthread_key_delete(thread_data_key);
if (err != 0)
   fprintf(stderr, “error deleting key - %s\n”,
strerror(err));
```

pthread_kill()

Send Signal to Target Thread

Syntax

```
#include <signal.h>
int pthread_kill(pthread_t thread, int sig);
```

Description

`pthread_kill()` sends the specified signal to the designated thread.

`pthread_kill()` works much like `kill()` or `_os_send()` except `pthread_kill()` takes a `pthread_t` instead of a `process_id`. Unlike `kill()` and `_os_send()`, `pthread_kill()` can not be used to send signals to other processors.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	thread is an invalid `pthread_t` value.
`ESRCH`	thread is not a valid thread ID.

See Also

`signal()`
`_os_sigmask()`

Example

```
err = pthread_kill(worker, SYNC_SIG);
if (err != 0)
   fprintf(stderr, “error signaling worker - %s\n”,
strerror(err));
```

pthread_mutex_destroy()

Free Mutex Object

Syntax

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Description

`pthread_mutex_destroy()` destroys the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EBUSY`	Attempt to destroy the object referenced by `mutex` while it is locked or referenced. For example, while being used in a `pthread_cond_wait()` or `pthread_cond_timedwait()` by another thread.
`EINVAL`	The value specified by `mutex` is invalid.

See Also

`pthread_mutex_init()`

Example

```
err = pthread_mutex_destroy(mutx);
if (err != 0)
   fprintf(stderr, "error destroying mutex - %s\n",
strerror(err));
```

pthread_mutex_getprioceiling()

Get Mutex Priority Ceiling

Syntax

```
#include <pthread.h>
int pthread_mutex_getprioceiling(
     const pthread_mutex_t      *mutex,
     int                        *prioceiling);
```

Description

`pthread_mutex_getprioceiling()` obtains the value of the priority ceiling value from the mutex object referenced by `mutex`.

The value stored at `prioceiling` will be the current value of the priority ceiling for the mutex. Valid priority ceilings are in the range 0 to 65535 (0xffff).

If successful, returns 0 and stores the value of the priority ceiling of `mutex` into the integer referenced by the `prioceiling` parameter. Otherwise, returns an error number.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `mutex` is invalid.

See Also

`pthread_mutexattr_init()`
`pthread_mutex_setprioceiling()`

Example

```
err = pthread_mutex_getprioceiling(mutex, &pc);
if (err != 0)
   fprintf(stderr, “error getting priority ceiling -
%s\n”, strerror(err));
```

pthread_mutex_init()

Allocate Mutex Object

Syntax

```
#include <pthread.h>
int pthread_mutex_init(
     pthread_mutex_t            *mutex,
     const pthread_mutexattr_t  *attr);
```

Description

The `pthread_mutex_init()` function initializes the mutex referenced by `mutex` with attributes specified by `attr`.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EAGAIN`	The system lacked the necessary resources (other than memory) to initialize another mutex.
`EBUSY`	An attempt to reinitialize the object referenced by `mutex` (a previously initialized, but not yet destroyed, mutex).
`EINVAL`	The value specified by `attr` or `mutex` is invalid.

See Also

pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
pthread_mutex_destroy()
PTHREAD_MUTEX_INITIALIZER

Example

```
err = pthread_mutex_init(&glob_mutex, NULL);
if (err != 0)
   fprintf(stderr, “error initializing mutex - %s\n”,
strerror(err));
```

pthread_mutex_lock()

Lock Mutex Object

Syntax

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Description

The mutex object referenced by `mutex` is locked by calling `pthread_mutex_lock()`. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by `mutex` in the locked state with the calling thread as its owner. An attempt by the current owner of a mutex to relock the mutex results in an `EDEADLK` error.

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

If priority inheritance is enabled for the specified mutex and a thread with a lower priority already owns the mutex then the owning thread's priority will be raised to the level of calling thread.

After the lock is acquired, if priority protection is enabled for the specified mutex and the specified ceiling priority is greater than the thread's current priority the thread's priority will be raised.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `mutex` does not refer to an initialized mutex object.
`EDEADLK`	The current thread already owns the mutex.

See Also

`pthread_mutex_trylock()`
`pthread_mutex_unlock()`

Example

```
err = pthread_mutex_lock(&glob_mutex);
if (err != 0)
   fprintf(stderr, "error locking mutex - %s\n",
strerror(err));
```

pthread_mutex_setprioceiling()

Set Mutex Priority Ceiling

Syntax

```
#include <pthread.h>
int pthread_mutex_setprioceiling(
     pthread_mutex_t       *attr,
     int                   ceiling);
```

Description

`pthread_mutex_setprioceiling()` is used to set the priority ceiling value in an initialized mutex object referenced by `mutex`.

`ceiling` must be a valid OS-9 priority value; it must be in the range 0 to 65535 (0xffff).

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `mutex` is invalid. The new value specified for the attribute is outside the range of legal values for that attribute.

See Also

`pthread_mutexattr_init()`
`pthread_mutex_getprioceiling()`

Example

```
err = pthread_mutex_setprioceiling(mutex, 255);
if (err != 0)
   fprintf(stderr, “error setting priority ceiling -
%s\n”, strerror(err));
```

pthread_mutex_trylock()

Lock Mutex Object (Non-Blocking)

Syntax

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Description

`pthread_mutex_trylock()` is a non-blocking mutex lock operation. If `mutex` is currently unowned, the calling thread is made the owner. If `mutex` is currently owned (by any thread, including the calling thread), `EBUSY` is returned.

Returns 0 if a lock on the mutex object referenced by `mutex` is acquired; otherwise, returns an error number.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EBUSY`	The mutex could not be acquired because it was already locked.
`EINVAL`	The value specified by `mutex` does not refer to an initialized mutex object.

See Also

`pthread_mutex_lock()`
`pthread_mutex_unlock()`

Example

```
err = pthread_mutex_trylock(&glob_mutex);
if (err != 0 && err != EBUSY)
   fprintf(stderr, "error trying to lock glob_mutex -
%s\n", strerror(err));
```

pthread_mutex_unlock()

Unlock Mutex Object

Syntax

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Description

`pthread_mutex_unlock()` is called by the owner of the mutex object referenced by `mutex` to release it. A `pthread_mutex_unlock()` call by a thread that is not the owner of the mutex results in an `EPERM` error. Calling `pthread_mutex_unlock()` when the mutex object is unlocked also results in an `EPERM` error.

If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock()` is called, the mutex becomes available, and is given to the next waiting thread.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `mutex` does not refer to an initialized mutex object.
`EPERM`	The current thread does not own the mutex.

See Also

`pthread_mutex_lock()`

`pthread_mutex_trylock()`

Example

```
err = pthread_mutex_unlock(&glob_mutex);
if (err != 0)
   fprintf(stderr, "error unlocking glob_mutex - %s\n",
strerror(err));
```

pthread_mutexattr_destroy()

Free Mutex Attributes Object

Syntax

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t
*attr);
```

Description

`pthread_mutexattr_destroy()` destroys a mutex attributes object; the object becomes, in effect, uninitialized.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid.

See Also

`pthread_mutex_init()`
`pthread_mutexattr_init()`

Example

```
err = pthread_mutexattr_destroy(mutex_attr);
if (err != 0)
   fprintf(stderr, "error destroying attr - %s\n",
strerror(err));
```

pthread_mutexattr_getprioceiling()

Get Priority Ceiling Attribute

Syntax

```
#include <pthread.h>
int pthread_mutexattr_getprioceiling(
     const pthread_mutexattr_t  *attr,
     int                        *prioceiling);
```

Description

`pthread_mutexattr_getprioceiling()` obtains the value of the priority ceiling attribute from the mutex attributes object referenced by `attr`.

The value stored at `prioceiling` will be the current value of the priority ceiling attribute. Valid priority ceilings are in the range 0 to 65535 (0xffff).

If successful, returns 0 and stores the value of the priority ceiling attribute of `attr` into the integer referenced by the `prioceiling` parameter. Otherwise, returns an error number.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid.

See Also

`pthread_mutexattr_init()`
`pthread_mutexattr_setprioceiling()`

Example

```
err = pthread_mutexattr_getprioceiling(mutex_attr,
&pc);
if (err != 0)
   fprintf(stderr, “error getting priority ceiling -
%s\n”, strerror(err));
```

pthread_mutexattr_getprotocol()

Get Protocol Attribute

Syntax

```
#include <pthread.h>
int pthread_mutexattr_getprotocol(
     const pthread_mutexattr_t  *attr,
     int                        *protocol);
```

Description

`pthread_mutexattr_getprotocol()` obtains the value of the protocol attribute from the mutex attributes object referenced by `attr`.

The value stored at `protocol` will be one of `PTHREAD_PRIO_NONE`, `PTHREAD_PRIO_INHERIT`, or `PTHREAD_PRIO_PROTECT`.

If successful, returns 0 and stores the value of the protocol attribute of `attr` into the integer referenced by the `protocol` parameter. Otherwise, returns an error number.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid.

See Also

`pthread_mutexattr_init()`
`pthread_mutexattr_setprotocol()`

Example

```
err = pthread_mutexattr_getprotocol(mutex_attr, &prot);
if (err != 0)
   fprintf(stderr, “error getting protocol - %s\n”,
strerror(err));
```

pthread_mutexattr_getpshared()

Get Mutex Process-Shared Attribute

Syntax

```
#include <pthread.h>
int pthread_mutexattr_getpshared(
     const pthread_mutexattr_t  *attr,
     int                        *pshared);
```

Description

`pthread_mutexattr_getpshared()` obtains the value of the process-shared attribute from the attributes object referenced by `attr`.

The value stored at `pshared` will be either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

If successful, returns 0 and stores the value of the process-shared attribute of `attr` into the object referenced by the `pshared` parameter. Otherwise, returns an error number.

This facility is not currently supported in Microware's Pthreads implementation. The process-shared attribute can be changed, but both values behave like `PTHREAD_PROCESS_PRIVATE`. `_POSIX_THREAD_PROCESS_SHARED` is not currently defined.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid.

See Also

`pthread_mutexattr_init()`
`pthread_mutexattr_setpshared()`

Example

```
err = pthread_mutexattr_getpshared(mutex_attr,
&pshared);
if (err != 0)
   fprintf(stderr, "error getting pshared - %s\n",
strerror(err));
```

pthread_mutexattr_init()

Allocate Mutex Attributes Object

Syntax

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

Description

`pthread_mutexattr_init()` initializes a mutex attributes object `attr` with a default value for all of the attributes.

The default values for the attributes are shown in <Bold><links>Table 3-11.

Table 3-11. Default attribute values for mutex attribute object

Attribute	Default Value
Process-shared	`PTHREAD_PROCESS_PRIVATE`
Protocol	`PTHREAD_PRIO_NONE`
Priority Ceiling	`<none>`

Returns 0 if successful or an error code if unsuccessful.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The key value is invalid.
`ENOMEM`	Insufficient memory exists to initialize the mutex attributes object.

See Also

pthread_mutex_init()
pthread_mutexattr_destroy()

Example

```
err = pthread_mutexattr_init(mutex_attr);
if (err != 0)
   fprintf(stderr, "error initializing attr - %s\n",
strerror(err));
```

pthread_mutexattr_setprioceiling()

Set Priority Ceiling Attribute

Syntax

```
#include <pthread.h>
int pthread_mutexattr_setprioceiling(
     pthread_mutexattr_t  *attr,
     int                  ceiling);
```

Description

`pthread_mutexattr_setprioceiling()` is used to set the priority ceiling attribute in an initialized attributes object referenced by `attr`.

`ceiling` must be a valid OS-9 priority value; it must be in the range 0 to 65535 (0xffff).

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid. The new value specified for the attribute is outside the range of legal values for that attribute.

See Also

`pthread_mutexattr_init()`
`pthread_mutexattr_getprioceiling()`

Example

```
err = pthread_mutexattr_setprioceiling(mutex_attr,
255);
if (err != 0)
   fprintf(stderr, “error setting priority ceiling -
%s\n”, strerror(err));
```

pthread_mutexattr_setprotocol()

Set Protocol Attribute

Syntax

```
#include <pthread.h>
int pthread_mutexattr_setprotocol(
     pthread_mutexattr_t  *attr,
     int                  protocol);
```

Description

`pthread_mutexattr_setprotocol()` is used to set the protocol attribute in an initialized attributes object referenced by `attr`.

`protocol` must be either `PTHREAD_PRIO_NONE`, `PTHREAD_PRIO_INHERIT`, or `PTHREAD_PRIO_PROTECT`,.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid. The new value specified for the attribute is outside the range of legal values for that attribute.

See Also

`pthread_mutexattr_init()`
`pthread_mutexattr_getprotocol()`

Example

```
err = pthread_mutexattr_setprotocol(mutex_attr, param);
if (err != 0)
   fprintf(stderr, “error setting protocol attr - %s\n”,
strerror(err));
```

pthread_mutexattr_setpshared()

Set Mutex Process-Shared Attribute

Syntax

```
#include <pthread.h>
int pthread_mutexattr_setpshared(
     pthread_mutexattr_t  *attr,
     int                  pshared);
```

Description

`pthread_mutexattr_setpshared()` is used to set the process-shared attribute in an initialized attributes object referenced by `attr`.

`pshared` must be either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

If successful, returns a value of 0; otherwise, returns an error.

This facility is not currently supported in Microware's Pthreads implementation. The process-shared attribute can be changed, but both values behave like `PTHREAD_PROCESS_PRIVATE`. `_POSIX_THREAD_PROCESS_SHARED` is not currently defined.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The value specified by `attr` is invalid. The new value specified for the attribute is outside the range of legal values for that attribute.

See Also

pthread_mutexattr_init()
pthread_mutexattr_getpshared()

Example

```
err = pthread_mutexattr_setpshared(mutex_attr,
PTHREAD_PROCESS_PRIVATE);
if (err != 0)
   fprintf(stderr, “error setting pshared attr - %s\n”,
strerror(err));
```

pthread_once()

Execute Routine Once per Process

Syntax

```
#include <pthread.h>
int pthread_once(
     pthread_once_t  *once_control,
     void            (*init_routine) (void));
```

Description

The first call to `pthread_once()` by any thread in a process with a given `once_control` calls the `init_routine()` with no arguments. Subsequent calls of `pthread_once()` with the same `once_control` will not call the `init_routine()`. On return from `pthread_once()` by any thread, it is guaranteed that `init_routine()` has completed. The `once_control` parameter is used to determine whether the associated initialization routine has been called.

`pthread_once()` is not a cancellation point. However, if `init_routine()` is a cancellation point and is canceled, the effect on `once_control` is as if `pthread_once()` was never called.

The behavior of `pthread_once()` is undefined if `once_control` has automatic storage duration or is not initialized by `PTHREAD_ONCE_INIT`.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`once` is an invalid pointer to a `pthread_once_t` object. `once` does not point to an initialized object. `init_routine` is an invalid address.

See Also

`PTHREAD_ONCE_INIT`

Example

```
err = pthread_once(get_key_once, create_data_key);
if (err != 0)
   fprintf(stderr, "error creating data key - %s\n",
strerror(err));
```

_pthread_resume()

Decrement Suspension Counter

Syntax

```
#include <LIB/pthread.h>
int _pthread_resume(pthread_t thread, int *status);
```

Description

`_pthread_resume()` decrements the suspension counter for the specified target thread. The suspension status of the target thread is returned at the `int` pointed to by status. The `int` is as follows:

- 0 if the target thread was not suspended
- 1 if the target thread went from suspended to not suspended
- > 1 if the target thread remained suspended

A suspension counter is used to support multiple suspension requests with the same target thread. An equal number of resume requests must be made for the thread to continue execution.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`thread` or `status` is NULL.
`ESRCH`	`thread` is not a valid thread.

See Also

`_pthread_setsuspendable()`
`_pthread_setunsuspendable()`
`_pthread_suspend()`

Example

```
err = _pthread_resume(worker, &level);
if (err != 0)
   fprintf(stderr, “error resuming worker - %s\n”,
strerror(err));
```

pthread_self()
Get Thread Identifier

Syntax

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Description

`pthread_self()` returns the calling thread's thread ID.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

None

See Also

`pthread_equal()`

Example

```
if (pthread_self() == worker[0])
   fputs("thread is worker #0\n", stdout);
```

pthread_setcancelstate()

Set Cancel State

Syntax

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
```

Description

`pthread_setcancelstate()` sets a thread's cancel state. `state` can be either `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`. The previous value of the thread's cancel state is returned at `oldstate`.

Any cancel requests made against a thread while its state is `PTHREAD_CANCEL_DISABLE` will be held pending until the state is changed to `PTHREAD_CANCEL_ENABLE`.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`state` is neither `PTHREAD_CANCEL_ENABLE` nor `PTHREAD_CANCEL_DISABLE`. `oldstate` is an invalid address.

See Also

`pthread_setcanceltype()`
`pthread_cancel()`
`pthread_cleanup_push()`
`pthread_cleanup_pop()`
`pthread_testcancel()`

Example

```
err = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,
&oldstate);
if (err != 0)
   fprintf(stderr, “error setting cancel state - %s\n”,
strerror(err));
```

pthread_setcanceltype()

Set Cancel Type

Syntax

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
```

Description

`pthread_setcanceltype()` sets a thread's cancel type. `type` can be either `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`. The previous value of the thread's cancel type is returned at `oldtype`.

When a thread's cancel type is `PTHREAD_CANCEL_DEFERRED` cancel requests against it wait to take effect until the next call to `pthread_testcancel()`.

When a thread's cancel type is `PTHREAD_CANCEL_ASYNCHRONOUS` cancel requests are acted upon when they are made. That is, when a thread calls `pthread_cancel()` with a target thread that has cancellation enabled and asynchronous, the target thread will immediately cancel.

If successful, returns a value of 0; otherwise, returns an error.

Cancelling an asynchronous cancel type thread causes a loss of resources. For example, the memory allocated to implement thread safety for C library functions will be lost. Use deferred cancellation whenever possible.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`type` is neither `PTHREAD_CANCEL_DEFERRED` nor `PTHREAD_CANCEL_ASYNCHRONOUS`.

See Also

`pthread_setcancelstate()`
`pthread_cancel()`
`pthread_cleanup_push()`
`pthread_cleanup_pop()`
`pthread_testcancel()`

Example

```
err = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,
&oldtype);
if (err != 0)
   fprintf(stderr, "error setting cancel type - %s\n",
strerror(err));
```

_pthread_setpr()

Set Priority for Target Thread

Syntax

```
#include <pthread.h>
int _pthread_setpr(pthread_t thread, u_int32 priority);
```

Description

`_pthread_setpr()` sets the OS-9 priority of `thread` to `priority`. This call must be used by threaded applications instead of `_os_setpr()` to ensure that priority inversion avoidance is properly supported for mutexes. Calling `_os_setpr()` directly results in undefined behavior as it relates to priority inversion.

Use `_pthread_getstatus()` to determine the priority of a thread.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	`thread` is invalid. `priority` is out of range. Valid range of `priority` is 0-65538.
`ESRCH`	`thread` is an invalid thread ID.

See Also

`_pthread_getstatus()`
`pthread_mutex_destroy()`
`pthread_mutex_destroy()`
`_pthread_attr_setpriority()`
`_pthread_attr_getpriority()`

Example

```
err = _pthread_setpr(reactor_shutdown, HIGH_PRIORITY);
if (err != 0)
   fprintf(stderr, "failed to set priority - %s",
strerror(err));
```

_pthread_setsignalrange()

Set Range of Signal Values

Syntax

```
#include <pthread.h>
int _pthread_setsignalrange(
     signal_code  low,
     signal_code  high);
```

Description

`_pthread_setsignalrange()` is used to specify the set of signal values that the Pthread library uses internally. Using this function will cause the Pthread library to use signals in the range `low` to `(high - 1)`.

Use this function if your application uses the same set of signal values as the Pthread library. By default, the Pthread library will use signals in the range 40,000 to 49,999 inclusive.

A minimum of 1000 signal values must be specified. The Pthreads library uses about 5 signals per thread as well as 1 per timed condition variable wait.

The new set of signals may not overlap the current set of signal values. This is to ensure integrity of any already allocated signal numbers.

`_pthread_setsignalrange()` returns `0` if successful or an error code if not.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	If less than 1000 signal values are in the range or `high` is less than `low` or the specified range overlaps with the signal range currently in use.

Example

```
err = _pthread_setsignalrange(2000, 3500);
if (err != SUCCESS)
   fprintf(stderr, "error setting signal range - %s\n”,
strerror(err));
```

pthread_setspecific()

Set Thread-Specific Data Pointer

Syntax

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void
*value);
```

Description

`pthread_setspecific()` function associates a thread-specific value with a key obtained via a previous call to `pthread_key_create()`. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

If successful, returns a value of 0; otherwise, returns an error.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

Possible Errors

`ENOMEM`	Insufficient memory exists to associate the value with the key.
`EINVAL`	The key value is invalid.

See Also

`pthread_key_create()`
`pthread_getspecific()`

Example

```
err = pthread_setspecific(thread_data_key,
thread_data);
if (err != 0)
   fprintf(stderr, “error setting thread data - %s\n”,
strerror(err));
```

_pthread_setsuspendable()

Decrement Suspendability Counter

Syntax

```
#include <pthread.h>
int _pthread_setsuspendable(void);
```

Description

`_pthread_setsuspendable()` decrements the suspendability counter for the calling thread. When this counter is at 0, the thread is suspendable.

This call would be used by applications that contain thread suspension and resource locking. Before taking a common lock a thread would set itself unsuspendable. This prevents the thread from holding a common lock while it is in the suspended state. Holding a common lock while suspended could cause deadlock for the remaining unsuspended threads. After unlocking the common lock the thread would call this function to return itself to the suspendable state.

Calling this function from a suspendable thread yields no change in state.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

None

See Also

`_pthread_resume()`
`_pthread_setunsuspendable()`

`_pthread_suspend()`

Example

The following example illustrates how to execute a semaphore protected critical section. Using the mechanisms shown below, a thread calling `_pthread_suspend()` can be assured that `glob_lock` will not be claimed by the suspended thread.

```
_pthread_setunsuspendable();

ec = _os_sema_p(&glob_lock);
if (ec != SUCCESS) {
   fprintf(stderr, "failed to get semaphore\n");
   pthread_exit((void *)ec);
}

/* critical section code */

ec = _os_sema_v(&glob_lock);
if (ec != SUCCESS) {
   fprintf(stderr, "failed to release semaphore\n");
   pthread_exit((void *)ec);
}

_pthread_setsuspendable();
```

_pthread_setunsuspendable()

Increment Suspendability Counter

Syntax

```
#include <pthread.h>
int _pthread_setunsuspendable(void);
```

Description

`_pthread_setunsuspendable()` increments the suspendability counter for the calling thread. When this counter is greater than 0, the thread is unsuspendable. This call does not return until the unsuspendable state is achieved.

This call is used by applications that contain thread suspension and resource locking. Before taking a common lock a thread would use this call to set itself unsuspendable. This prevents the thread from holding a common lock while it is in the suspended state. After unlocking the common lock the thread would call `_pthread_setsuspendable()` to return itself to the normal suspendable state.

Calling this function from a unsuspendable thread simply increases the suspendability counter. It is expected that each `_pthread_setunsuspendable()` call has a matching `_pthread_setsuspendable()` call.

Calling this function more than `0xffffffff` times without any intervening `_pthread_setsuspendable()` calls results in undefined behavior. Fewer `_pthread_setsuspendable()` calls than `_pthread_setunsuspendable()` calls will be required to return to the normal suspendable state.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

Errors from memory allocation and getting a process descriptor if called when signals are masked.

See Also

`_pthread_resume()`
`_pthread_setsuspendable()`
`_pthread_suspend()`

Example

Refer to the example provided for `_pthread_setunsuspendable()`.

_pthread_suspend()

Increment Suspension Counter

Syntax

```
#include <pthread.h>
int _pthread_suspend(pthread_t thread, unsigned int
*count);
```

Description

`_pthread_suspend()` increments the suspension counter for the target thread specified by `thread`. The target thread's suspension counter prior to the suspension request is returned at the unsigned integer pointed to by `count`. A counter is used to support multiple suspension requests on the same target thread. An equal number of resume requests must be made before the target thread will resume execution.

This call does not return until the target thread has been successfully suspended. That is, if the target thread has set itself unsuspendable then this call will poll until the target sets itself back to suspendable.

Refer to the section on Thread Suspension for more information on what services are guaranteed while threads are suspended.

Returns 0 if the thread's suspension counter was successfully incremented or an error number if not.

Attributes

Operating System:	OS-9
State:	User

Library

`mt_clib.l`

Possible Errors

`EINVAL`	The specified thread or count pointer is `NULL`.
`ESRCH`	The specified thread is invalid or has terminated.
`EDEADLK`	The specified thread is the calling thread and there is only one thread in the process.

See Also

`_pthread_resume()`
`_pthread_setsuspendable()`
`_pthread_setunsuspendable()`

Example

```
err = _pthread_suspend(child, &count);
if (err != 0) {
   fprintf(stderr, "failed to suspend child\n");
   pthread_exit((void *)err);
}

/* do some activity with child suspended */

err = _pthread_resume(child, &status);
if (err != 0) {
   fprintf(stderr, "failed to resume child\n");
   pthread_exit((void *)err);
}
```

pthread_testcancel()

Test for Pending Cancel

Syntax

```
#include <pthread.h>
void pthread_testcancel(void);
```

Description

`pthread_testcancel()` checks for a pending, deferred cancel request. If there is one, cancellation cleanup handlers are called in the reverse order in which they were pushed, thread specific data destructors are called in an unspecified order, and the thread is terminated with `PTHREAD_CANCELED` as its status.

If the cancel state of the thread is `PTHREAD_CANCEL_DISABLE`, this call has no effect.

`pthread_testcancel()` does not return if a cancel is pending.

Attributes

Operating System:	OS-9
State:	User
Compatibility:	POSIX

Library

`mt_clib.l`

See Also

`pthread_cancel()`
`pthread_setcancelstate()`
`pthread_setcanceltype()`
`PTHREAD_CANCELED`

Example

```
pthread_testcancel();
```

Definition Descriptions

This section lists all the definitions and descriptions in alphabetical order (without regard for numbers and underscores).

Table 3-12 lists all the definitions and descriptions, in alphabetical order. These definitions appear in the header file `pthread.h`.

Table 3-12. Definition Descriptions

Function Name	Function Description
`_POSIX_THREAD_ATTR_STACKADDR`	Stackaddr Implementation Macro
`_POSIX_THREAD_ATTR_STACKSIZE`	Stacksize Implementation Macro
`_POSIX_THREAD_PRIO_INHERIT`	Priority Inheritance Implementation Macro
`_POSIX_THREAD_PRIO_PROTECT`	Priority Ceiling Implementation Macro
`_POSIX_THREAD_SAFE_FUNCTIONS`	Thread-safe Function Implementation Macro
`_POSIX_THREADS`	Posix Threads Implementation Macro
`_PT_BOOSTED`	Priority Boosted Status Flag
`_PT_CPENDING`	Cancel Pending Status Flag
`_PT_CSTATE`	Cancel State Status Flag
`_PT_CTYPE`	Cancel Type Status Flag
`_PT_DETACHED`	Detached Thread Status Flag
`_PT_EXIT`	Terminated Thread Status Flag
`_PT_IPENDING`	Interruption Pending Status Flag
`_PT_SFLAG`	Suspended Status Flag
`_PT_SPENDING`	Suspension Pending Status Flag
`_PT_SSTATE`	Suspension State Status Flag
`PTHREAD_CANCEL_ASYNCHRONOUS`	Asynchronous Cancel Type
`PTHREAD_CANCEL_DEFERRED`	Deferred Cancel Type
`PTHREAD_CANCEL_DISABLE`	Disabled Cancel State
`PTHREAD_CANCEL_ENABLE`	Enabled Cancel State
`PTHREAD_CANCELED`	Cancelled Thread Exit Status
`PTHREAD_COND_INITIALIZER`	Condition Variable Initializer

Table 3-12. Definition Descriptions (Continued)

Function Name	Function Description
`PTHREAD_CREATE_DETACHED`	Detached Thread Attribute
`PTHREAD_CREATE_JOINABLE`	Joinable Thread Attribute
`PTHREAD_DESTRUCTOR_ITERATIONS`	Number of Destruction Attempts
`PTHREAD_KEYS_MAX`	Maximum Number of Data Keys
`PTHREAD_MUTEX_INITIALIZER`	Mutex Initializer
`PTHREAD_ONCE_INIT`	Once Control Initializer
`PTHREAD_PROCESS_PRIVATE`	Process Private Attribute
`PTHREAD_PROCESS_SHARED`	Process Shared Attribute
`PTHREAD_STACK_MIN`	Minimum Thread Stack Size
`PTHREAD_THREADS_MAX`	Maximum Number of Threads per Process

_POSIX_THREAD_ATTR_STACKADDR

Stackaddr Implementation Macro

Syntax

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

Description

The presence of the macro `_POSIX_THREAD_ATTR_STACKADDR` indicates that the OS-9 implementation of Pthreads supports `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_attr_getstackaddr()`
`pthread_attr_setstackaddr()`

_POSIX_THREAD_ATTR_STACKSIZE

Stacksize Implementation Macro

Syntax

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

Description

The presence of the macro `_POSIX_THREAD_ATTR_STACKSIZE` indicates that the OS-9 implementation of Pthreads supports `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_attr_getstacksize()`
`pthread_attr_setstacksize()`

_POSIX_THREAD_PRIO_INHERIT

Priority Inheritance Implementation Macro

Syntax

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

Description

The presence of the macro `_POSIX_THREAD_PRIO_INHERIT` indicates that the OS-9 implementation of Pthreads has the priority inheritance mechanism to avoid priority inversion.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

_POSIX_THREAD_PRIO_PROTECT

Priority Ceiling Implementation Macro

Syntax

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

Description

The presence of the macro `_POSIX_THREAD_PRIO_PROTECT` indicates that the OS-9 implementation of Pthreads has the priority ceiling emulation protocol mechanism to avoid priority inversion.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

_POSIX_THREAD_SAFE_FUNCTIONS
Thread-safe Function Implementation Macro

Syntax

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

Description

The presence of the macro `_POSIX_THREAD_SAFE_FUNCTIONS` indicates that the OS-9 implementation of Pthreads implements thread-safe functions.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

_POSIX_THREADS

Posix Threads Implementation Macro

Syntax

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

Description

The presence of the macro `_POSIX_THREADS` indicates that the OS-9 implementation of Pthreads supports the POSIX threads API.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

_PT_BOOSTED

Priority Boosted Status Flag

Syntax

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

Description

`_PT_BOOSTED` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread is running at its default priority. If set, the thread is running at a higher priority due to priority inheritance or priority ceiling emulation protocol.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`pthread_mutexattr_setprotocol()`
`pthread_mutexattr_setprioceiling()`

_PT_CPENDING
Cancel Pending Status Flag

Syntax

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

Description

`_PT_CPENDING` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread has no cancel pending. If set, the thread has a cancel pending.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`pthread_cancel()`

_PT_CSTATE
Cancel State Status Flag

Syntax

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

Description

`_PT_CSTATE` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread has cancelling enabled. If set, the thread has a cancelling disabled.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`pthread_setcancelstate()`

_PT_CTYPE

Cancel Type Status Flag

Syntax

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

Description

`_PT_CTYPE` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread has cancels marked as deferred. If set, the thread has cancels marked as asynchronous.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`pthread_setcanceltype()`

_PT_DETACHED

Detached Thread Status Flag

Syntax

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

Description

`_PT_DETACHED` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread is joinable. If set, the thread is detached.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`pthread_create()`
`pthread_detach()`
`pthread_attr_setdetachstate()`
`pthread_join()`

_PT_EXIT

Terminated Thread Status Flag

Syntax

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

Description

`_PT_EXIT` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread has not yet terminated. If set, the thread has terminated and is available for joining, if not detached.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`pthread_exit()`
`pthread_cancel()`

_PT_IPENDING
Interruption Pending Status Flag

Syntax

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

Description

`_PT_IPENDING` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread has no interrupt pending. If set, the thread has an interrupt pending.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`_pthread_interrupt()`
`_pthread_interrupt_clear()`
`pthread_cond_wait()`
`pthread_cond_timedwait()`

_PT_SFLAG
Suspended Status Flag

Syntax

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

Description

`_PT_SFLAG` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread is not suspended. If set, the thread is suspended.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`_pthread_suspend()`
`_pthread_resume()`
`_pthread_setunsuspendable()`
`_pthread_setsuspendable()`

_PT_SPENDING

Suspension Pending Status Flag

Syntax

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

Description

`_PT_SPENDING` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread has no suspend pending. If set, the thread has a suspend pending.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`_pthread_suspend()`
`_pthread_resume()`
`_pthread_setunsuspendable()`
`_pthread_setsuspendable()`

_PT_SSTATE

Suspension State Status Flag

Syntax

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

Description

`_PT_SSTATE` is a bit mask for the `status` field of the `_pthread_status_t` structure. If clear, the thread has suspension enabled. If set, the thread has suspension disabled.

Attributes

Operating System:	OS-9

See Also

`_pthread_getstatus()`
`_pthread_setunsuspendable()`
`_pthread_setsuspendable()`

PTHREAD_CANCEL_ASYNCHRONOUS

Asynchronous Cancel Type

Syntax

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

Description

`PTHREAD_CANCEL_ASYNCHRONOUS` is used to specify the asynchronous cancel type to `pthread_setcanceltype()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_setcanceltype()`
`pthread_setcancelstate()`
`pthread_cancel()`

PTHREAD_CANCEL_DEFERRED

Deferred Cancel Type

Syntax

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

Description

`PTHREAD_CANCEL_DEFERRED` is used to specify the deferred cancel type to `pthread_setcanceltype()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_setcanceltype()`
`pthread_setcancelstate()`
`pthread_cancel()`

PTHREAD_CANCEL_DISABLE
Disabled Cancel State

Syntax

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

Description

`PTHREAD_CANCEL_DISABLE` is used to specify that cancels are disabled to `pthread_setcancelstate()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_setcanceltype()`
`pthread_setcancelstate()`
`pthread_cancel()`

PTHREAD_CANCEL_ENABLE
Enabled Cancel State

Syntax

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

Description

`PTHREAD_CANCEL_ENABLE` is used to specify that cancels are enabled to `pthread_setcancelstate()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_setcanceltype()`
`pthread_setcancelstate()`
`pthread_cancel()`

PTHREAD_CANCELED
Cancelled Thread Exit Status

Syntax

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

Description

`PTHREAD_CANCELED` is the exit status of a thread that has been canceled and recognized the cancellation.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_cancel()`
`pthread_exit()`

PTHREAD_COND_INITIALIZER
Condition Variable Initializer

Syntax

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

Description

`PTHREAD_COND_INITIALIZER` is used to initialize a variable of type `pthread_cond_t`. Using this macro is an alternative to calling `pthread_cond_init()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_cond_init()`

PTHREAD_CREATE_DETACHED

Detached Thread Attribute

Syntax

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

Description

`PTHREAD_CREATE_DETACHED` specifies that threads created with the attribute be detached. It is passed to `pthread_attr_setdetachstate()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_create()`
`pthread_attr_setdetachstate()`

PTHREAD_CREATE_JOINABLE
Joinable Thread Attribute

Syntax

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

Description

`PTHREAD_CREATE_JOINABLE` specifies that threads created with the attribute be joinable. It is passed to `pthread_attr_setdetachstate()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_create()`
`pthread_attr_setdetachstate()`

PTHREAD_DESTRUCTOR_ITERATIONS

Number of Destruction Attempts

Syntax

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

Description

`PTHREAD_DESTRUCTOR_ITERATIONS` is the number of times the Pthread library will call the set of destructors for non-NULL thread-specific data keys when a thread exits. After this many iterations, non-NULL thread-specific data key values will be ignored.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_exit()`
`pthread_key_create()`
`pthread_setspecific()`

PTHREAD_KEYS_MAX
Maximum Number of Data Keys

Syntax

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

Description

`PTHREAD_KEYS_MAX` is the maximum number of thread-specific data keys a process may have. Attempts to create more keys will result in `EAGAIN` being returned from `pthread_key_create()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_key_create()`
`pthread_key_delete()`

PTHREAD_MUTEX_INITIALIZER

Mutex Initializer

Syntax

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

Description

`PTHREAD_MUTEX_INITIALIZER` is used to initialize a variable of type `pthread_mutex_t`. Using this macro is an alternative to calling `pthread_mutex_init()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_mutex_init()`

PTHREAD_ONCE_INIT
Once Control Initializer

Syntax

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

Description

`PTHREAD_ONCE_INIT` must be used to initialize a global or file static `pthread_once_t` variable. Failure to do so will result in `EINVAL` being returned from `pthread_once()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_once()`

PTHREAD_PROCESS_PRIVATE
Process Private Attribute

Syntax

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

Description

`PTHREAD_PROCESS_PRIVATE` is used to specify that mutexes or condition variables created with the attribute be private to the creating process. It can be passed to `pthread_mutexattr_setpshared()` and `pthread_condattr_setpshared()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_condattr_init()`
`pthread_condattr_setpshared()`
`pthread_mutexattr_init()`

PTHREAD_PROCESS_SHARED

Process Shared Attribute

Syntax

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

Description

`PTHREAD_PROCESS_SHARED` is used to specify that mutexes or condition variables created with the attribute be shared among processes. It can be passed to `pthread_mutexattr_setpshared()` and `pthread_condattr_setpshared()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_condattr_init()`
`pthread_condattr_setpshared()`
`pthread_mutexattr_init()`

PTHREAD_STACK_MIN

Minimum Thread Stack Size

Syntax

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

Description

`PTHREAD_STACK_MIN` is the minimum amount of stack a thread is allowed to be created with. The value is minimal. If a threads is to have many nested function calls or a large amount of automatic storage, additional stack should be allocated with `pthread_attr_setstacksize()`.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_attr_init()`
`pthread_attr_setstacksize()`
`pthread_create()`

PTHREAD_THREADS_MAX
Maximum Number of Threads per Process

Syntax

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

Description

`PTHREAD_THREADS_MAX` is the maximum number of threads a process can have. OS-9 places no artificial limit on this number. System resources will run out before this maximum is reached.

Attributes

Operating System:	OS-9
Compatibility:	POSIX

See Also

`pthread_create()`
`pthread_exit()`