



[Home](#)

Using OS-9[®] Threads

Version 2.12



RadiSys.
THE POWER OF WE

www.radisys.com
Revision A • January 2008

Copyright and publication information

This manual reflects version 2.12 of Microware OS-9 Threads. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

January 2008
Copyright ©2008 by RadiSys Corporation
All rights reserved.

EPC and RadiSys are registered trademarks of RadiSys Corporation. ASM, Brahma, DAI, DAQ, MultiPro, SAIB, Spirit, and ValuePro are trademarks of RadiSys Corporation.

DAVID, MAUI, OS-9, OS-9000, and SoftStax are registered trademarks of RadiSys Corporation. FasTrak, Hawk, and UpLink are trademarks of RadiSys Corporation.

† All other trademarks, registered trademarks, service marks, and trade names are the property of their respective owners.

Contents

Thread Definition	6
Thread Architecture	6
Using Threads.....	7
Benefits	7
Limitations.....	8
Ideal Applications	8
Example Using Threads	8
The POSIX Threads Standard.....	13
Additional Resources.....	13
Overview of OS-9 Threads	16
The OS-9 Implementation of POSIX Threads.....	16
The OS-9 Kernel	16
Managing Processes and Threads.....	16
Mutexes in OS-9	17
Thread Interruption	17
Signals.....	18
POSIX Signals.....	18
Thread Suspension	18
Support.....	18
Application Considerations.....	19
OS-9 Threads Guidelines and Issues	20
Shared Global Data Structures	20
New Process Structure.....	21
Functions to Access the Process Descriptor	21
System State Code.....	22
Static Return Values.....	22
Deadlock.....	23
Thread-safe Coding Techniques	23
Threads and Subroutine Modules.....	24
Shared Data Access Functions.....	25
Example Thread-safe Conversion of a Library	26
Miscellaneous Issues.....	33
POSIX Pthreads Library Functions	36
POSIX Pthreads Library Definitions	38
Pthreads Library Extension Functions.....	39
Pthreads Library Extension Definitions.....	39

1

Threads Overview

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

- [Thread Definition](#)
- [Using Threads](#)
- [Example Using Threads](#)
- [The POSIX Threads Standard](#)
- [Additional Resources](#)



Threads are not implemented in the OS-9® for 68K operating system.

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 these 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 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 proceeding.

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 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

Example:

```
pthread_t      a_thread;
pthread_attr_t a_thread_attribute;
extern 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 predetermined 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.

Example:

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

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,
                  &attr,
                  print_message_function,
                  (void*)message1);
```

```

pthread_create(&thread2,
              &attr,
              print_message_function,
              (void*)message2);
exit(0);
}

void *print_message_function( void *ptr )
{
    char *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\n 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:

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

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,
                  &attr,
                  print_message_function,
                  (void *) message1);
    sleep(10);

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

    sleep(10);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *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.

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

void *print_message_function( void *ptr );

main()
{
    pthread_t thread;
    pthread_attr_t attr;
    char *message1 = "Hello";
    char *message2 = "World";
    void *status;

    pthread_attr_init(&attr);
    pthread_attr_setstacksize(&attr, 4096);

    pthread_create(&thread,
                  &attr,
                  print_message_function,
                  (void *) message1);

    pthread_join(thread, &status);

    pthread_create(&thread,
                  &attr,
                  print_message_function,
                  (void *) message2);

    pthread_join(thread, &status);

    exit(0);
}
```

```

void *print_message_function( void *ptr )
{
    char *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

Additional Resources

The following suggested readings do not constitute a Microware endorsement:

- IEEE Standard POSIX 1003.1c. Institute of Electrical and Electronics Engineers.
- *Pthreads Programming*; Bradford Nichols, Dick Buttler & 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 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 sections below provide information for implementing 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 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 discuss 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 unuspendable 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 unuspendable 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. Mutexes 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_gprdesc` 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_gprdesc` 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_PROCDDESC` 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 runtime 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) $(FILES) -o=$@

```

Below 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](#)
- [POSIX Pthreads Library Definitions](#)
- [Pthreads Library Extension Functions](#)
- [Pthreads Library Extension Definitions](#)

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.1`. 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. For a longer description of each function, refer to the *Ultra C Library Reference* manual.



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
<code>pthread_attr_destroy()</code>	<code>pthread_attr_destroy()</code>
<code>pthread_attr_getdetachstate()</code>	Get Detach State Attribute
<code>pthread_attr_getstackaddr()</code>	Get Stack Address Attribute
<code>pthread_attr_getstacksize()</code>	Get Stack Size Attribute
<code>pthread_attr_init()</code>	Allocate Thread Creation Attribute Object
<code>pthread_attr_setdetachstate()</code>	Set Detached State Attribute
<code>pthread_attr_setstackaddr()</code>	Set Stack Address Attribute
<code>pthread_attr_setstacksize()</code>	Set Stack Size Attribute
<code>pthread_cancel()</code>	Cancel Target Thread
<code>pthread_cleanup_pop()</code>	Pop Cleanup Routine
<code>pthread_cleanup_push()</code>	Push Cleanup Routine
<code>pthread_cond_broadcast()</code>	Release Threads Waiting for Condition Variable
<code>pthread_cond_destroy()</code>	Free Condition Variable Object
<code>pthread_cond_init()</code>	Allocate Condition Variable Object
<code>pthread_cond_signal()</code>	Release Thread Waiting for Condition Variable
<code>pthread_cond_timedwait()</code>	Wait on Condition Variable for Specified Interval
<code>pthread_cond_wait()</code>	Wait on Condition Variable
<code>pthread_condattr_destroy()</code>	Free Condition Variable Attributes Object
<code>_pthread_condattr_getanysignal()</code>	Get Condition Variable Any Signal Attribute
<code>pthread_condattr_getpshared()</code>	Get Condition Variable Process-Shared Attribute
<code>pthread_condattr_init()</code>	Allocate Condition Variable Attributes Object
<code>_pthread_condattr_setanysignal()</code>	Set Condition Variable Any Signal Attribute
<code>pthread_condattr_setpshared()</code>	Set Condition Variable Process-Shared Attribute

Table 3-1. POSIX Library Functions (Continued)

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



The following list provides additional reading resources. These resources are not endorsed by RadiSys Corporation.

- *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

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. For a longer description of each definition, refer to the *Ultra C Library Reference* manual.



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
<code>_POSIX_THREAD_ATTR_STACKADDR</code>	Stackaddr Implementation Macro
<code>_POSIX_THREAD_ATTR_STACKSIZE</code>	Stacksize Implementation Macro
<code>_POSIX_THREAD_Prio_INHERIT</code>	Priority Inheritance Implementation Macro
<code>_POSIX_THREAD_Prio_PROTECT</code>	Priority Ceiling Implementation Macro
<code>_POSIX_THREAD_SAFE_FUNCTIONS</code>	Thread-safe Function Implementation Macro
<code>_POSIX_THREADS</code>	Posix Threads Implementation Macro
<code>PTHREAD_CANCEL_ASYNCHRONOUS</code>	Asynchronous Cancel Type
<code>PTHREAD_CANCEL_DEFERRED</code>	Deferred Cancel Type
<code>PTHREAD_CANCEL_DISABLE</code>	Disabled Cancel State
<code>PTHREAD_CANCEL_ENABLE</code>	Enabled Cancel State
<code>PTHREAD_CANCELED</code>	Cancelled Thread Exit Status
<code>PTHREAD_COND_INITIALIZER</code>	Condition Variable Initializer
<code>PTHREAD_CREATE_DETACHED</code>	Detached Thread Attribute
<code>PTHREAD_CREATE_JOINABLE</code>	Joinable Thread Attribute
<code>PTHREAD_DESTRUCTOR_ITERATIONS</code>	Number of Destruction Attempts
<code>PTHREAD_KEYS_MAX</code>	Maximum Number of Data Keys
<code>PTHREAD_MUTEX_INITIALIZER</code>	Mutex Initializer
<code>PTHREAD_ONCE_INIT</code>	Once Control Initializer
<code>PTHREAD_Prio_INHERIT</code>	Priority Inheritance Protocol
<code>PTHREAD_Prio_NONE</code>	No Mutex Protocol
<code>PTHREAD_Prio_PROTECT</code>	Priority Ceiling Protocol
<code>PTHREAD_PROCESS_PRIVATE</code>	Process Private Attribute
<code>PTHREAD_PROCESS_SHARED</code>	Process Shared Attribute
<code>PTHREAD_STACK_MIN</code>	Minimum Thread Stack Size
<code>PTHREAD_THREADS_MAX</code>	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. For a longer description of each function, refer to the *Ultra C Library Reference* manual.

Table 3-3. OS-9 Specific Threads Functions

Function Name	Function Description
<code>_pthread_attr_getinitfunction()</code>	Get Initialization Function Attribute
<code>_pthread_attr_getpriority()</code>	Get Priority Attribute
<code>_pthread_attr_setinitfunction()</code>	Set Initialization Function Attribute
<code>_pthread_attr_setpriority()</code>	Set Priority Attribute
<code>_pthread_getstatus()</code>	Get Thread Status Information
<code>_pthread_interrupt()</code>	Interrupt Target Thread
<code>_pthread_interrupt_clear()</code>	Clear Interrupt Request for Target Thread
<code>pthread_mutexattr_getkind_np()</code>	Get kind attribute of the mutex attribute
<code>pthread_mutexattr_setkind_np()</code>	Set kind attribute of the mutex attribute
<code>_pthread_resume()</code>	Decrement Suspension Counter
<code>_pthread_setpr()</code>	Set Priority for Target Thread
<code>_pthread_setsignalrange()</code>	Set Range of Signal Values
<code>_pthread_setsuspendable()</code>	Decrement Suspendability Counter
<code>_pthread_setunsuspendable()</code>	Increment Suspendability Counter
<code>_pthread_suspend()</code>	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`. For a longer description of each definition, refer to the *Ultra C Library Reference* manual.

Table 3-4. OS-9 Specific Threads Definitions

Definition	Definition Description
<code>MUTEX_FAST_NP</code>	Denotes a fast mutex
<code>MUTEX_RECURSIVE_NP</code>	Denotes a recursive mutex
<code>MUTEX_NONRECURSIVE_NP</code>	Denotes the default non-recursive POSIX compliant mutex
<code>_PT_BOOSTED</code>	Priority Boosted Status Flag
<code>_PT_CPENDING</code>	Cancel Pending Status Flag
<code>_PT_CSTATE</code>	Cancel State Status Flag
<code>_PT_CTYPE</code>	Cancel Type Status Flag
<code>_PT_DETACHED</code>	Detached Thread Status Flag

Table 3-4. OS-9 Specific Threads Definitions (Continued)

Definition	Definition Description
<code>_PT_EXIT</code>	Terminated Thread Status Flag
<code>_PT_IPENDING</code>	Interruption Pending Status Flag
<code>_PT_SFLAG</code>	Suspended Status Flag
<code>_PT_SPENDING</code>	Suspension Pending Status Flag
<code>_PT_SSTATE</code>	Suspension State Status Flag