Multithreaded Programming Guide

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043-1100
U.S.A.

Part No: 802-5938
August 1997

Adobe PostScript

Please Recycle

Contents

Preface

The *Multithreaded Programming Guide* describes the multithreaded programming interfaces for POSIX and Solaris threads in the Solaris™ 2.5 system. This guide shows application programmers how to create new multithreaded programs and how to add multithreading to existing programs.

Although this guide covers both the POSIX and Solaris threads implementations, most topics assume a POSIX threads interest. Information applying to only Solaris threads is covered in a special chapter.

To understand this guide, a reader must be familiar with

- A UNIX® SVR4 system—preferably the Solaris 2.5 system
- The C programming language—multithreading is implemented through the libthread library
- The principles of concurrent programming (as opposed to sequential programming)—multithreading requires a different way of thinking about function interactions. Some books you might want to read are:
  - *Algorithms for Mutual Exclusion* by Michel Raynal (MIT Press, 1986)
  - *Concurrent Programming* by Alan Burns & Geoff Davies (Addison-Wesley, 1993)
  - *Distributed Algorithms and Protocols* by Michel Raynal (Wiley, 1988)
  - *Operating System Concepts* by Silberschatz, Peterson, & Galvin (Addison-Wesley, 1991)
  - *Principles of Concurrent Programming* by M. Ben-Ari (Prentice-Hall, 1982)

Ordering Sun Documents

The SunDocs^SM program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of SunExpress™ On The Internet at `http://www.sun.com/sunexpress`.

How This Guide Is Organized

Chapter 1," gives a structural overview of threads implementation in this release.

Chapter 2," discusses the general POSIX threads library routines, emphasizing creating a thread with default attributes.

Chapter 3," covers creating a thread with nondefault attributes.

Chapter 4," covers the threads library synchronization routines.

Chapter 5," discusses changes to the operating system to support multithreading.

Chapter 6," covers multithreading safety issues.

Chapter 7," covers the basics of compiling and debugging multithreaded applications.

Chapter 8," describes some of the tools available for gathering performance and debugging information about your multithreaded programs.

Chapter 9," covers the Solaris threads (as opposed to POSIX threads) interfaces.

Chapter 10," discusses issues that affect programmers writing multithreaded applications.

Appendix A," shows how code can be designed for POSIX threads.

Appendix B," shows an example of building a barrier in Solaris threads.

Appendix C," lists the safety levels of library routines.

You can find additional useful information about multithreaded programming by browsing the following World Wide Web (WWW) site:

`http://www.sun.com/sunsoft/Products/Developer-products/sig/threads`

What Typographic Changes and Symbols Mean

Table P–1 describes the type changes and symbols used in this guide.

TABLE P–1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123()`	Commands, files, directories, and C functions; code examples	The `fork1()()` function is new. Use `ls -a` to list all files.
AaBbCc123	Variables, titles, and emphasized words	The *stack_size* value is set by... You *must* specify a zero value.
`AaBbCc123`	What you type, contrasted with on-screen computer output	system% **`cc prog.c`**
page(#)	The man page name and section in the *Solaris Reference Manual*	See thr_create(3T).

Sections of program code in the main text are enclosed in boxes:

```
nt test (100);

 main()
 {
  register int a, b, c, d, e, f;

  test(a) = b & test(c & 0x1) & test(d & 0x1);
```

CHAPTER **1**

Covering Multithreading Basics

The word *multithreading* can be translated as *multiple threads of control* or *multiple flows of control*. While a traditional UNIX process always has contained and still does contain a single thread of control, multithreading (MT) separates a process into many execution threads, each of which runs independently.

Multithreading your code can

- Improve application responsiveness
- Use multiprocessors more efficiently
- Improve program structure
- Use fewer system resources

This chapter explains some multithreading terms, benefits, and concepts. If you are ready to start using multithreading, skip to Chapter 2.

- "Defining Multithreading Terms" on page 1
- "Meeting Multithreading Standards" on page 3
- "Benefiting From Multithreading" on page 3
- "Understanding Basic Multithreading Concepts" on page 4

Defining Multithreading Terms

Table 1–1 introduces some of the terms used in this book.

TABLE 1–1 Multithreading Terms

Term	Definition
Process	The UNIX environment (such as file descriptors, user ID, and so on) created with the `fork(2)` system call, which is set up to run a program.
Thread	A sequence of instructions executed within the context of a process.
pthreads (POSIX threads)	A POSIX 1003.1c compliant threads interface.
Solaris threads	A SunSoft™ threads interface that is not POSIX compliant. A predecessor of pthreads.
Single-threaded	Restricting access to a single thread.
Multithreaded	Allowing access to two or more threads.
User- or Application-level threads	Threads managed by the threads library routines in user (as opposed to kernel) space.
Lightweight processes	Threads in the kernel that execute kernel code and system calls (also called LWPs).
Bound threads	Threads that are permanently bound to LWPs.
Unbound threads	A default Solaris thread that context switches very quickly without kernel support.
Attribute object	Contains opaque data types and related manipulation functions used to standardize some of the configurable aspects of POSIX threads, mutual exclusion locks (mutexes), and condition variables.
Mutual exclusion locks	Functions that lock and unlock access to shared data.
Condition variables	Functions that block threads until a change of state.
Counting semaphore	A memory-based synchronization mechanism.
Parallelism	A condition that arises when at least two threads are *executing* simultaneously.
Concurrency	A condition that exists when at least two threads are *making progress*. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.

Meeting Multithreading Standards

The concept of multithreaded programming goes back to at least the 1960s. Its development on UNIX systems began in the mid-1980s. While there is agreement about what multithreading is and the features necessary to support it, the interfaces used to implement multithreading have varied greatly.

For several years a group called POSIX (Portable Operating System Interface) 1003.4a has been working on standards for multithreaded programming. The standard has now been ratified. This Multithreaded Programming Guide is based on the POSIX standards: P1003.1b final draft 14 (realtime), and P1003.1c final draft 10 (multithreading).

This guide covers both POSIX threads (also called *pthreads*) and Solaris threads. Solaris threads were available in the Solaris 2.4 release, and are not functionally different from POSIX threads. However, because POSIX threads are more portable than Solaris threads, this guide covers multithreading from the POSIX perspective. Subjects specific to Solaris threads only are covered in the Chapter 9.;

Benefiting From Multithreading

Improving Application Responsiveness

Any program in which many activities are not dependent upon each other can be redesigned so that each activity is defined as a thread. For example, the user of a multithreaded GUI does not have to wait for one activity to complete before starting another.

Using Multiprocessors Efficiently

Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors.

Numerical algorithms and applications with a high degree of parallelism, such as matrix multiplications, can run much faster when implemented with threads on a multiprocessor.

Improving Program Structure

Many programs are more efficiently structured as multiple independent or semi-independent units of execution instead of as a single, monolithic thread. Multithreaded programs can be more adaptive to variations in user demands than single threaded programs.

Using Fewer System Resources

Programs that use two or more processes that access common data through shared memory are applying more than one thread of control.

However, each process has a full address space and operating systems state. The cost of creating and maintaining this large amount of state information makes each process much more expensive than a thread in both time and space.

In addition, the inherent separation between processes can require a major effort by the programmer to communicate between the threads in different processes, or to synchronize their actions.

Combining Threads and RPC

By combining threads and a remote procedure call (RPC) package, you can exploit nonshared-memory multiprocessors (such as a collection of workstations). This combination distributes your application relatively easily and treats the collection of workstations as a multiprocessor.

For example, one thread might create child threads. Each of these children could then place a remote procedure call, invoking a procedure on another workstation. Although the original thread has merely created threads that are now running in parallel, this parallelism involves other computers.

Understanding Basic Multithreading Concepts

Concurrency and Parallelism

In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution.

In the same multithreaded process in a shared-memory multiprocessor environment, each thread in the process can run on a separate processor at the same time, resulting in parallel execution.

When the process has fewer or as many threads as there are processors, the threads support system in conjunction with the operating system ensure that each thread runs on a different processor.

For example, in a matrix multiplication that has the same number of threads and processors, each thread (and each processor) computes a row of the result.

Looking at Multithreading Structure

Traditional UNIX already supports the concept of threads—each process contains a single thread, so programming with multiple processes is programming with multiple threads. But a process is also an address space, and creating a process involves creating a new address space.

Creating a thread is much less expensive when compared to creating a new process, because the newly created thread uses the current process address space. The time it takes to switch between threads is much less than the time it takes to switch between processes, partly because switching between threads does not involve switching between address spaces.

Communicating between the threads of one process is simple because the threads share everything—address space, in particular. So, data produced by one thread is immediately available to all the other threads.

The interface to multithreading support is through a subroutine library, `libpthread` for POSIX threads, and `libthread` for Solaris threads. Multithreading provides flexibility by decoupling kernel-level and user-level resources.

User-Level Threads

Threads are the primary programming interface in multithreaded programming. User-level threads[1] are handled in user space and avoid kernel context switching penalties. An application can have hundreds of threads and still not consume many kernel resources. How many kernel resources the application uses is largely determined by the application.

Threads are visible only from within the process, where they share all process resources like address space, open files, and so on. The following state is unique to each thread.

- Thread ID

1. User-level threads are named to distinguish them from kernel-level threads, which are the concern of systems programmers, only. Because this book is for application programmers, kernel-level threads are not discussed.

- Register state (including PC and stack pointer)
- Stack
- Signal mask
- Priority
- Thread-private storage

Because threads share the process instructions and most of the process data, a change in shared data by one thread can be seen by the other threads in the process. When a thread needs to interact with other threads in the same process, it can do so without involving the operating system.

By default, threads are very lightweight. But, to get more control over a thread (for instance, to control scheduling policy more), the application can bind the thread. When an application binds threads to execution resources, the threads become kernel resources (see “System Scope (Bound Threads)” on page 8for more information).

To summarize, user-level threads are:

- Inexpensive to create because they do not need to create their own address space. They are bits of virtual memory that are allocated from your address space at run time.
- Fast to synchronize because synchronization is done at the application level, not at the kernel level.
- Easily managed by the threads library; either `libpthread` or `libthread`.

Lightweight Processes

The threads library uses underlying threads of control called *lightweight processes* that are supported by the kernel. You can think of an LWP as a virtual CPU that executes code or system calls.

You usually do not need to concern yourself with LWPs to program with threads. The information here about LWPs is provided as background, so you can understand the differences in scheduling scope, described on “Process Scope (Unbound Threads)” on page 7.

Note - The LWPs in the Solaris 2.x system are *not* the same as the LWPs in the SunOS™ 4.0 LWP library, which are not supported in the Solaris 2.x system.

Much as the `stdio` library routines such as `fopen()` and `fread()` use the `open()` and `read()` functions, the threads interface uses the LWP interface, and for many of the same reasons.

Lightweight processes (LWPs) bridge the user level and the kernel level. Each process contains one or more LWP, each of which runs one or more user threads. The creation of a thread usually involves just the creation of some user context, but not the creation of an LWP.

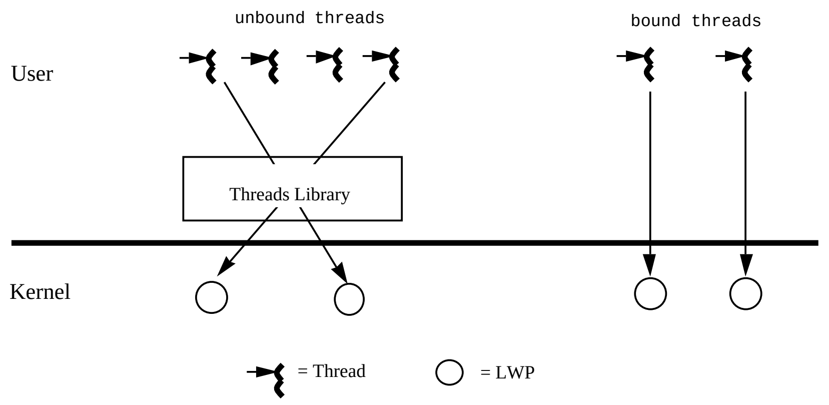


Figure 1–1 User-level Threads and Lightweight Processes

Each LWP is a kernel resource in a kernel pool, and is allocated (attached) and de-allocated (detached) to a thread on a per thread basis. This happens as threads are scheduled or created and destroyed.

Scheduling

POSIX specifies three scheduling policies: first-in-first-out (`SCHED_FIFO`), round-robin (`SCHED_RR`), and custom (`SCHED_OTHER`). `SCHED_FIFO` is a queue-based scheduler with different queues for each priority level. `SCHED_RR` is like FIFO except that each thread has an execution time quota.

Both `SCHED_FIFO` and `SCHED_RR` are POSIX Realtime extensions. `SCHED_OTHER` is the default scheduling policy.

See "LWPs and Scheduling Classes" on page 104for information about the `SCHED_OTHER` policy, and about emulating some properties of the POSIX `SCHED_FIFO` and `SCHED_RR` policies.

Two scheduling scopes are available: process scope for unbound threads and system scope for bound threads. Threads with differing scope states can coexist on the same system and even in the same process. In general, the scope sets the range in which the threads scheduling policy is in effect.

Process Scope (Unbound Threads)

Unbound threads are created `PTHREAD_SCOPE_PROCESS`. These threads are scheduled in user space to attach and detach from available LWPs in the LWP pool. LWPs are available to threads in this process only; that is threads are scheduled on these LWPs.

In most cases, threads should be `PTHREAD_SCOPE_PROCESS`. This allows the threads to float among the LWPs, and this improves threads performance (and is equivalent to creating a Solaris thread in the THR_UNBOUND state). The threads library decides, with regard to other threads, which threads get serviced by the kernel.

System Scope (Bound Threads)

Bound threads are created PTHREAD_SCOPE_SYSTEM. A boundthread is permanently attached to an LWP.

Each bound thread is bound to an LWP for the lifetime of the thread. This is equivalent to creating a Solaris thread in the THR_BOUND state. You can bind a thread to give it an alternate signal stack or to use special scheduling attributes with Realtime scheduling. All scheduling is done by the Operating System.

Note - In neither case, bound or unbound, can a thread be directly accessed by or moved to another process.

Cancellation

Thread cancellation allows a thread to terminate the execution of any other thread in the process. The target thread (the one being cancelled) can keep cancellation requests pending and can perform application-specific cleanup when it acts upon the cancellation notice.

The pthreads cancellation feature permits either asynchronous or deferred termination of a thread. Asynchronous cancellation can occur at any time; deferred cancellation can occur only at defined points. Deferred cancellation is the default type.

Synchronization

Synchronization allows you to control program flow and access to shared data for concurrently executing threads.

The three synchronization models are mutex locks, condition variables, and semaphores.

- Mutex locks allow only one thread at a time to execute a specific section of code, or to access specific data.
- Condition variables block threads until a particular condition is true.
- Counting semaphores typically coordinate access to resources. The count is the limit on how many threads can have access to a semaphore. When the count is reached, the semaphore blocks.

CHAPTER **2**

Basic Threads Programming

The Threads Library

This chapter introduces the basic threads programming routines from the POSIX threads library, `libpthread(3T)`. This chapter covers *default threads*, or threads with default attribute values, which are the kind of threads that are most often used in multithreaded programming.

Chapter 3,; explains how to create and use threads with nondefault attributes.

Note - Attributes are specified only at thread creation time; they cannot be altered when the thread is being used.

The POSIX (libpthread) routines introduced here have programming interfaces that are similar to the original (`libthread`) Solaris multithreading library.

The following brief roadmap directs you to the discussion of a particular task and its associated man page.

- "Create a Default Thread" on page 10
- "Wait for Thread Termination" on page 11
- "Detaching a Thread " on page 14
- "Create a Key for Thread-Specific Data" on page 15
- "Delete the Thread-Specific Data Key" on page 16
- "Set the Thread-Specific Data Key" on page 17
- "Get the Thread-Specific Data Key" on page 17
- "Get the Thread Identifier" on page 20

- “Compare Thread IDs” on page 21
- “Initializing Threads” on page 21
- “Yield Thread Execution” on page 22
- “Set the Thread Priority” on page 22
- “Get the Thread Priority” on page 23
- “Send a Signal to a Thread” on page 24
- “Access the Signal Mask of the Calling Thread” on page 25
- “Forking Safely” on page 26
- “Terminate a Thread” on page 26
- “Cancel a Thread” on page 29
- “Enable or Disable Cancellation” on page 30
- “Set Cancellation Type” on page 30
- “Create a Cancellation Point” on page 31
- “Push a Handler Onto the Stack” on page 32
- “Pull a Handler Off the Stack” on page 32

Create a Default Thread

When an attribute object is not specified, it is `NULL`, and the default thread is created with the following attributes:

- Unbound
- Nondetached
- With a default stack and stack size
- With the parent’s priority

You can also create a default attribute object with `pthread_attr_init()`, and then use this attribute object to create a default thread. See the section “Initialize Attributes” on page 37for details.

pthread_create(3T)

Use `pthread_create()` to add a new thread of control to the current process.

```
Prototype:
int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
    void*(*start_routine)(void *), void *arg);


#include <pthread.h>
```

```
pthread_attr_t ()tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;

/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The `pthread_create()` function is called with *attr* having the necessary state behavior. *start_routine* is the function with which the new thread begins execution. When *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine* (see "pthread_create(3T) " on page 10).

When `pthread_create()` is successful, the ID of the thread created is stored in the location referred to as *tid*.

Creating a thread using a `NULL` attribute argument has the same effect as using a default attribute; both create a default thread. When *tattr* is initialized, it acquires the default behavior.

Return Values

`pthread_create()` returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `pthread_create()` fails and returns the corresponding value.

`EAGAIN`

A system limit is exceeded, such as when too many LWPs have been created.

`EINVAL`

The value of *tattr* is invalid.

Wait for Thread Termination

pthread_join(3T)

Use the `pthread_join()` function to wait for a thread to terminate.

```
Prototype:
int pthread_join(thread_t tid, void **status);
```

```
#include <pthread.h>

pthread_t tid;
int ret;
int status;

/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);

/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

The `pthread_join()` function blocks the calling thread until the specified thread terminates.

The specified thread must be in the current process and must not be detached. For information on thread detachment, see "Set Detach State" on page 39.

When *status* is not `NULL`, it points to a location that is set to the exit status of the terminated thread when `pthread_join()` returns successfully.

Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of ESRCH.

After `pthread_join()` returns, any stack storage associated with the thread can be reclaimed by the application.

Return Values

Returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `pthread_join()` fails and returns the corresponding value.

`ESRCH`

tid is not a valid, undetached thread in the current process.

`EDEADLK`

tid specifies the calling thread.

`EINVAL`

The value of *tid* is invalid.

The `pthread_join()` routine takes two arguments, giving you some flexibility in its use. When you want the caller to wait until a specific thread terminates, supply that thread's ID as the first argument.

If you are interested in the exit code of the defunct thread, supply the address of an area to receive it.

Remember that `pthread_join()` works only for target threads that are nondetached. When there is no reason to synchronize with the termination of a particular thread, then that thread should be detached.

Think of a detached thread as being the thread you use in most instances and reserve nondetached threads for only those situations that require them.

A Simple Threads Example

In Code Example 2–1, one thread executes the procedure at the top, creating a helper thread that executes the procedure `fetch()`, which involves a complicated database lookup and might take some time.

The main thread wants the results of the lookup but has other work to do in the meantime. So it does those other things and then waits for its helper to complete its job by executing `pthread_join()`.

An argument, *pbe*, to the new thread is passed as a stack parameter. This can be done here because the main thread waits for the spun-off thread to terminate. In general, though, it is better to `malloc(3C)` storage from the heap instead of passing an address to thread stack storage, which can disappear or be reassigned if the thread terminated.

CODE EXAMPLE 2–1 A Simple Threads Program

```
void mainline (...)
{
        struct phonebookentry *pbe;
        pthread_attr_t tattr;
        pthread_t helper;
        int status;

        pthread_create(&helper, NULL, fetch, &pbe);

            /* do something else for a while */

        pthread_join(helper, &status);
        /* it’s now safe to use result */
}

void fetch(struct phonebookentry *arg)
{
        struct phonebookentry *npbe;
        /* fetch value from a database */

        npbe = search (prog_name)
            if (npbe != NULL)
                *arg = *npbe;
        pthread_exit(0);
}

struct phonebookentry {
        char name[64];
        char phonenumber[32];
        char flags[16];
}
```

Detaching a Thread

pthread_detach(3T)

`pthread_detach(3T)` is an alternative to `pthread_join(3T)` to reclaim storage for a thread that is created with a *detachstate* attribute set to `PTHREAD_CREATE_JOINABLE`.

```
Prototype:
int pthread_detach(thread_t tid);
```

```
#include <pthread.h>

pthread_t tid;
int ret;

/* detach thread tid */
ret = pthread_detach(tid);
```

The `pthread_detach( )` function is used to indicate to the implementation that storage for the thread *tid* can be reclaimed when the thread terminates. If *tid* has not terminated, `pthread_detach( )` does not cause it to terminate. The effect of multiple `pthread_detach( )` calls on the same target thread is unspecified.

Return Values

`pthread_detach( )` returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `pthread_detach( )` fails and returns the corresponding value.

`EINVAL`

tid is not a valid thread.

`ESRCH`

tid is not a valid, undetached thread in the current process.

Create a Key for Thread-Specific Data

Single-threaded C programs have two basic classes of data—local data and global data. For multithreaded C programs a third class is added—*thread-specific data* (TSD). This is very much like global data, except that it is private to a thread.

Thread-specific data is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a *key* that is global to all threads in the process. Using the *key*, a thread can access a pointer (*void* *) that is maintained per-thread.

pthread_keycreate(3T)

Use `pthread_keycreate()` to allocate a *key* that is used to identify thread-specific data in a process. The *key* is global to all threads in the process, and all threads initially have the value `NULL` associated with the key when it is created.

`pthread_keycreate()` is called once for each *key* before the *key* is used. There is no implicit synchronization.

Once a *key* has been created, each thread can bind a value to the *key*. The values are specific to the thread and are maintained for each thread independently. The per-thread binding is deallocated when a thread terminates if the *key* was created with a `destructor` function.

```
Prototype:
int pthread_key_create(pthread_key_t *key,
    void (*destructor) (void *));
```

```
#include <pthread.h>

pthread_key_t key;
int ret;

/* key create without destructor */
ret = pthread_key_create(&key, NULL);

/* key create with destructor */
ret = pthread_key_create(&key, destructor);
```

When `pthread_keycreate()` returns successfully, the allocated key is stored in the location pointed to by *key*. The caller must ensure that the storage and access to this key are properly synchronized.

An optional destructor function, `destructor`, can be used to free stale storage. When a key has a non-`NULL` `destructor` function and the thread has a non-`NULL` value associated with that key, the `destructor` function is called with the current associated value when the thread exits. The order in which the `destructor` functions are called is unspecified.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `pthread_keycreate()` fails and returns the corresponding value.

`EAGAIN`

The *key* name space is exhausted.

`ENOMEM`

Not enough virtual memory is available in this process to create a new key.

Delete the Thread-Specific Data Key

pthread_keydelete(3T)

Use `pthread_keydelete()` to destroy an existing thread-specific data key. Any memory associated with the key can be freed because the key has been invalidated and will return an error if ever referenced. There is no comparable function in Solaris threads.

```
Prototype:
int pthread_key_delete(pthread_key_t key);


#include <pthread.h>

pthread_key_t key;
int ret;

/* key previously created */
ret = pthread_key_delete(key);
```

Once a *key* has been deleted, any reference to it with the `pthread_setspecific()` or `pthread_getspecific()` call results in the `EINVAL` error.

It is the responsibility of the programmer to free any thread-specific resources before calling the delete function. This function does not invoke any of the destructors.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_keycreate()` fails and returns the corresponding value.

`EINVAL`

The *key* value is invalid.

Set the Thread-Specific Data Key

pthread_setspecific(3T)

Use `pthread_setspecific()` to set the thread-specific binding to the specified thread-specific data key.

```
Prototype:
int pthread_setspecific(pthread_key_t key, const void *value);
```

```
#include <pthread.h>

pthread_key_t key;
void *value;
int ret;

/* key previously created */
ret = pthread_setspecific(key, value);
```

Return Values

`pthread_setspecific()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `pthread_setspecific()` fails and returns the corresponding value.

`ENOMEM`

Not enough virtual memory is available.

`EINVAL`

key is invalid.

Note - `pthread_setspecific()` does not free its storage. If a new binding is set, the existing binding must be freed; otherwise, a memory leak can occur..

Get the Thread-Specific Data Key

pthread_getspecific(3T)

Use `pthread_getspecific()` to get the calling thread's binding for *key*, and store it in the location pointed to by *value*.

```
Prototype:
int pthread_getspecific(pthread_key_t key);
```

```
#include <pthread.h>

pthread_key_t key;
void *value;

/* key previously created */
value = pthread_getspecific(key);
```

Return Values

No errors are returned.

Global and Private Thread-Specific Data Example

Code Example 2–2 shows an excerpt from a multithreaded program. This code is executed by any number of threads, but it has references to two global variables, *errno* and *mywindow*, that really should be references to items private to each thread.

CODE EXAMPLE 2–2 Thread-Specific Data—Global but Private

```
body() {
    ...

    while (write(fd, buffer, size) == -1) {
        if (errno != EINTR) {
            fprintf(mywindow, "%s\n", strerror(errno));
            exit(1);
        }
    }

    ...

}
```

References to `errno` should get the system error code from the routine called by this thread, not by some other thread. So, references to `errno` by one thread refer to a different storage location than references to `errno` by other threads.

The *mywindow* variable is intended to refer to a `stdio` stream connected to a window that is private to the referring thread. So, as with `errno`, references to *mywindow* by one thread should refer to a different storage location (and, ultimately, a different window) than references to *mywindow* by other threads. The only difference here is that the threads library takes care of `errno`, but the programmer must somehow make this work for *mywindow*.

The next example shows how the references to *mywindow* work. The preprocessor converts references to *mywindow* into invocations of the `_mywindow()` procedure.

This routine in turn invokes `pthread_getspecific()`, passing it the *mywindow_key* global variable (it really is a global variable) and an output parameter, *win*, that receives the identity of this thread's window.

CODE EXAMPLE 2–3 Turning Global References Into Private References

```
thread_key_t mywin_key;

FILE *_mywindow(void) {
    FILE *win;

    pthread_getspecific(mywin_key, &win);
    return(win);
}

#define mywindow _mywindow()

void routine_uses_win( FILE *win) {
    ...
}

void thread_start(...) {
    ...
    make_mywin();
    ...
    routine_uses_win( mywindow )
    ...
}
```

The *mywin_key* variable identifies a class of variables for which each thread has its own private copy; that is, these variables are thread-specific data. Each thread calls `make_mywin()` to initialize its window and to arrange for its instance of *mywindow* to refer to it.

Once this routine is called, the thread can safely refer to *mywindow* and, after `_mywindow()`, the thread gets the reference to its private window. So, references to *mywindow* behave as if they were direct references to data private to the thread.

Code Example 2–4 shows how to set this up.

CODE EXAMPLE 2–4 Initializing the Thread-Specific Data

```
void make_mywindow(void) {
    FILE **win;
    static pthread_once_t mykeycreated = PTHREAD_ONCE_INIT;

    pthread_once(&mykeycreated, mykeycreate);

    win = malloc(sizeof(*win));
    create_window(win, ...);

    pthread_setspecific(mywindow_key, win);

 }

void mykeycreate(void) {
    pthread_keycreate(&mywindow_key, free_key);
 }

void free_key(void *win) {
    free(win);
}
```

First, get a unique value for the key, *mywin_key*. This key is used to identify the thread-specific class of data. So, the first thread to call `make_mywin()` eventually calls `pthread_keycreate()`, which assigns to its first argument a unique *key*. The second argument is a `destructor` function that is used to deallocate a thread's instance of this thread-specific data item once the thread terminates.

The next step is to allocate the storage for the caller's instance of this thread-specific data item. Having allocated the storage, a call is made to the `create_window()` routine, which sets up a window for the thread and sets the storage pointed to by *win* to refer to it. Finally, a call is made to `pthread_setspecific()`, which associates the value contained in *win* (that is, the location of the storage containing the reference to the window) with the key.

After this, whenever this thread calls `pthread_getspecific()`, passing the global *key*, it gets the value that was associated with this key by this thread when it called `pthread_setspecific()`.

When a thread terminates, calls are made to the `destructor` functions that were set up in `pthread_key_create()`. Each `destructor` function is called only if the terminating thread established a value for the *key* by calling `pthread_setspecific()`.

Get the Thread Identifier

pthread_self(3T)

Use `pthread_self()` to get the `ID` of the calling thread.

```
Prototype:
pthread_t  pthread_self(void);


#include <pthread.h>

pthread_t tid;

tid = pthread_self();
```

Return Values

Returns the `ID` of the calling thread.

Compare Thread IDs

pthread_equal(3T)

Use `pthread_equal()` to compare the thread identification numbers of two threads.

```
Prototype:
int  pthread_equal(pthread_t tid1, pthread_t tid2);
```

```
#include <pthread.h>

pthread_t tid1, tid2;
int ret;

ret = pthread_equal(tid1, tid2);
```

Return Values

Returns a non-zero value when *tid1* and *tid2* are equal; otherwise, `zero` is returned. When either *tid1* or *tid2* is an invalid thread identification number, the result is unpredictable.

Initializing Threads

pthread_once(3T)

Use `pthread_once()` to call an initialization routine the first time `pthread_once(3T)` is called. Subsequent calls to `pthread_once()` have no effect.

```
Prototype:
int  pthread_once(pthread_once_t *once_control,
    void (*init_routine)(void));
```

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
int ret;

ret = pthread_once(&once_control, init_routine);
```

The *once_control* parameter determines whether the associated initialization routine has been called.

Return Values

`pthread_once()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_once()` fails and returns the corresponding value.

`EINVAL`

once_control or *init_routine* is `NULL`.

Yield Thread Execution

sched_yield(3R)

Use `sched_yield()` to cause the current thread to yield its execution in favor of another thread with the same or greater priority.

```
Prototype:
int  sched_yield(void);
```

```
#include <sched.h>

int ret;

ret = sched_yield();
```

Return Values

Returns zero after completing successfully. Otherwise -1 is returned and *errno* is set to indicate the error condition.

`ENOSYS`

`sched_yield(3R)` is not supported in this implementation.

Set the Thread Priority

pthread_setschedparam(3T)

Use `pthread_setschedparam()` to modify the priority of an existing thread. This function has no effect on scheduling policy.

```
Prototype:
int  pthread_setschedparam(pthread_t tid, int policy,
    const struct sched_param *param);
```

```
#include <pthread.h>

pthread_t tid;
int ret;
struct sched_param param;
int priority;

/* sched_priority will be the priority of the thread */
sched_param.sched_priority = priority;

/* only supported policy, others will result in ENOTSUP */
policy = SCHED_OTHER;

/* scheduling parameters of target thread */
ret = pthread_setschedparam(tid, policy, &param);
```

Return Values

`pthread_setschedparam()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the `pthread_setschedparam()` function fails and returns the corresponding value.

`EINVAL`

The value of the attribute being set is not valid.

`ENOTSUP`

An attempt was made to set the attribute to an unsupported value.

Get the Thread Priority

pthread_getschedparam(3T)

`pthread_getschedparam()` gets the priority of the existing thread.

```
Prototype:
int  pthread_getschedparam(pthread_t tid, int policy,
    struct schedparam *param);
```

```
#include <pthread.h>

pthread_t tid;
sched_param param;
int priority;
int policy;
int ret;

/* scheduling parameters of target thread */
ret = pthread_getschedparam (tid, &policy, &param);
```

```
/* sched_priority contains the priority of the thread */
priority = param.sched_priority;
```

Return Values

`pthread_getschedparam()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`ESRCH`

The value specified by *tid* does not refer to an existing thread.

Send a Signal to a Thread

pthread_kill(3T)

Use `pthread_kill()` to send a signal to a thread.

```
Prototype:
int  pthread_kill(thread_t tid, int sig);


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


int sig;
pthread_t tid;
int ret;

ret = pthread_kill(tid, sig);
```

`pthread_kill()` sends the signal *sig* to the thread specified by *tid*. *tid* must be a thread within the same process as the calling thread. The *sig* argument must be from the list given in signal(5).

When *sig* is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of *tid*.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, `pthread_kill()` fails and returns the corresponding value.

`EINVAL`

sig is not a valid signal number.

`ESRCH`

tid cannot be found in the current process.

Access the Signal Mask of the Calling Thread

pthread_sigmask(3T)

Use `pthread_sigmask()` to change or examine the signal mask of the calling thread.

```
Prototype:
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);
```

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


int ret;
sigset_t old, new;

ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* set new mask */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

how determines how the signal set is changed. It can have one of the following values:

- `SIG_BLOCK`—Add *new* to the current signal mask, where *new* indicates the set of signals to block.
- `SIG_UNBLOCK`—Delete *new* from the current signal mask, where new indicates the set of signals to unblock.
- `SIG_SETMASK`—Replace the current signal mask with *new*, where *new* indicates the new signal mask.

When the value of *new* is `NULL`, the value of *how* is not significant and the signal mask of the thread is unchanged. So, to inquire about currently blocked signals, assign a `NULL` value to the *new* argument.

The *old* variable points to the space where the previous signal mask is stored, unless it is `NULL`.

Return Values

Returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_sigmask()` fails and returns the corresponding value.

`EINVAL`

The value of *how* is not defined.

Forking Safely

pthread_atfork(3T)

See the discussion about `pthread_atfork()` on "The Solution—pthread_atfork(3T)" on page 100.

```
Prototype:

int pthread_atfork(void (*prepare) (void), void (*parent) (void),
    void (*child) (void) );
```

Terminate a Thread

pthread_exit(3T)

Use `pthread_exit()` to terminate a thread.

```
Prototype:
void  pthread_exit(void *status);
```

```
#include <pthread.h>

int status;

pthread_exit(&status); /* exit with status */
```

The `pthread_exit()` function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by *status* are retained until the thread is waited for (blocked). Otherwise, *status* is ignored and the thread's ID can be reclaimed immediately. For information on thread detachment, see "Set Detach State" on page 39.

Return Values

The calling thread terminates with its exit status set to the contents of *status* if *status* is not `NULL`.

Finishing Up

A thread can terminate its execution in the following ways:

- By returning from its first (outermost) procedure, the threads start routine; see `pthread_create(3T)`
- By calling `pthread_exit()`, supplying an exit status
- By termination with POSIX cancel functions; see `pthread_cancel()`

The default behavior of a thread is to linger until some other thread has acknowledged its demise by "joining" with it. This is the same as the default `pthread_create()` attribute being non-detached; see `pthread_detach(3T)`. The result of the join is that the joining thread picks up the exit status of the dying thread and the dying thread vanishes.

An important special case arises when the initial thread — the one calling `main()`,— returns from calling `main()` or calls `exit(3C)`. This action causes the entire process to be terminated, along with all its threads. So take care to ensure that the initial thread does not return from `main()` prematurely.

Note that when the main thread merely calls `pthread_exit(3T)`, it terminates only itself—the other threads in the process, as well as the process, continue to exist. (The process terminates when all threads terminate.)

Cancellation

The POSIX threads library introduces the ability to cancel threads to threads programming. Cancellation allows a thread to terminate the execution of any other thread, or all threads, in the process. Cancellation is an option when all further operations of a related set of threads are undesirable or unnecessary. A good method is to cancel all threads, restore the process to a consistent state, and then return to the point of origin.

One example of thread cancellation is an asynchronously generated cancel condition, such as, when a user requesting to close or exit some running application. Another example is the completion of a task undertaken by a number of threads. One of the threads might ultimately complete the task while the others continue to operate. Since they are serving no purpose at that point, they all should be cancelled.

There are dangers in performing cancellations. Most deal with properly restoring invariants and freeing shared resources. A thread that is cancelled without care might leave a mutex in a locked state, leading to a deadlock. Or it might leave a region of memory allocated with no way to identify it and therefore no way to free it.

The `pthreads` library specifies a cancellation interface that permits or forbids cancellation programmatically. The library defines the set of points at which cancellation can occur *(cancellation points)*. It also allows the scope of *cancellation handlers*, which provide clean up services, to be defined so that they are sure to operate when and where intended.

Placement of cancellation points and the effects of cancellation handlers must be based on an understanding of the application. A mutex is explicitly not a cancellation point and should be held only the minimal essential time.

Limit regions of asynchronous cancellation to sequences with no external dependencies that could result in dangling resources or unresolved state conditions. Take care to restore cancellation state when returning from some alternate, nested cancellation state. The interface provides features to facilitate restoration: `pthread_setcancelstate(3T)` preserves the current cancel state in a referenced variable; `pthread_setcanceltype(3T)` preserves the current cancel type in the same way.

Cancellations can occur under three different circumstances:

- Asynchronously
- At various points in the execution sequence as defined by the standard
- At discrete points specified by the application

By default, cancellation can occur only at well-defined points as defined by the POSIX standard.

In all cases, take care that resources and state are restored to a condition consistent with the point of origin.

Cancellation Points

Be careful to cancel a thread only when cancellation is safe. The `pthreads` standard specifies several cancellation points, including:

- Programmatically establish a thread cancellation point through a `pthread_testcancel(3T)` call.
- Threads waiting for the occurrence of a particular condition in `pthread_cond_wait(3T)` or `pthread_cond_timedwait(3T)`.
- Threads waiting for termination of another thread in `pthread_join(3T)`.
- Threads blocked on `sigwait(2)`.
- Some standard library calls. In general, these are functions in which threads can block; see the man page `cancellation(3T)` for a list.

Cancellation is enabled by default. At times you might want an application to disable cancellation. This has the result of deferring all cancellation requests until they are enabled again.

See "pthread_setcancelstate(3T) " on page 30for information about disabling cancellation.

Cancel a Thread

pthread_cancel(3T)

Use `pthread_cancel()` to cancel a thread.

Prototype:

```
int pthread_cancel(pthread_t thread);
```

```
#include <pthread.h>

pthread_t thread;
int ret;

ret = pthread_cancel(thread);
```

How the cancellation request is treated depends on the state of the target thread. Two functions, `pthread_setcancelstate(3T)` and `pthread_setcanceltype(3T)`, determine that state.

Return Values

`pthread_cancel()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`ESRCH`

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

Enable or Disable Cancellation

pthread_setcancelstate(3T)

Use `pthread_setcancelstate()` to enable or disable thread cancellation. When a thread is created, thread cancellation is enabled by default.

Prototype:

```
int pthread_setcancelstate(int state, int *oldstate);
```

```
#include <pthread.h>

int oldstate;
int ret;

/* enabled */
ret = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);

/* disabled */
ret = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
```

Return Values

`pthread_setcancelstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the `pthread_setcancelstate()` function fails and returns the corresponding value.

`EINVAL`

The state is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

Set Cancellation Type

pthread_setcanceltype(3T)

Use `pthread_setcanceltype()` to set the cancellation type to either deferred or asynchronous mode. When a thread is created, the cancellation type is set to deferred mode by default. In deferred mode, the thread can be cancelled only at cancellation points. In asynchronous mode, a thread can be cancelled at any point during its execution. Using asynchronous mode is discouraged.

Prototype:

```
int pthread_setcanceltype(int type, int *oldtype);
```

```
#include <pthread.h>

int oldtype;
int ret;

/* deferred mode */
ret = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);

/* async mode*/
ret = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);
```

Return Values

`pthread_setcanceltype()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The type is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

Create a Cancellation Point

pthread_testcancel(3T)

Use `pthread_testcancel()` to establish a cancellation point for a thread.

Prototype:

```
void pthread_testcancel(void);
```

```
#include <pthread.h>

pthread_testcancel();
```

The `pthread_testcancel()` function is effective when thread cancellation is enabled and in deferred mode. Calling this function while cancellation is disabled has no effect.

Be careful to insert `pthread_testcancel()` only in sequences where it is safe to cancel a thread. In addition to programmatically establishing cancellation points through the `pthread_testcancel()` call, the pthreads standard specifies several cancellation points. See “Cancellation Points” on page 28for more details.

There is no return value.

Push a Handler Onto the Stack

Use cleanup handlers to restore conditions to a state consistent with that at the point of origin, such as cleaning up allocated resources and restoring invariants. Use the `pthread_cleanup_push(3T)` and `pthread_cleanup_pop(3T)` functions to manage the handlers.

Cleanup handlers are pushed and popped in the same lexical scope of a program. They should always match; otherwise compiler errors will be generated.

pthread_cleanup_push(3T)

Use the `pthread_cleanup_push()` function to push a cleanup handler onto a cleanup stack (LIFO).

Prototype:

```
void pthread_cleanup_push(void(*routine)(void *), void *args);
```

```
#include <pthread.h>

/* push the handler "routine" on cleanup stack */
pthread_cleanup_push (routine, arg);
```

Pull a Handler Off the Stack

pthread_cleanup_pop(3T)

Use the `pthread_cleanup_pop()` function to pull the cleanup handler off the cleanup stack.

A nonzero argument in the pop function removes the handler from the stack and executes it. An argument of zero pops the handler without executing it.

`pthread_cleanup_pop()` is effectively called with a nonzero argument if a thread either explicitly or implicitly calls pthread_exit(3T) or if the thread accepts a cancel request.

Prototype:

```
void pthread_cleanup_pop(int execute);
```

```
#include <pthread.h>

/* pop the "func" out of cleanup stack and execute "func" */
pthread_cleanup_pop (1);

/* pop the "func" and DONT execute "func" */
pthread_cleanup_pop (0);
```

There are no return values.

CHAPTER **3**

Thread Create Attributes

The previous chapter covered the basics of threads creation using default attributes. This chapter discusses setting attributes at thread creation time.

Note that only pthreads uses attributes and cancellation, so the API covered in this chapter is for POSIX threads only. Otherwise, the *functionality* for Solaris threads and `pthreads` is largely the same. (See Chapter 9;for more information about similarities and differences.)

Attributes

Attributes are a way to specify behavior that is different from the default. When a thread is created with `pthread_create(3T)` or when a synchronization variable is initialized, an attribute object can be specified. The defaults are usually sufficient.

Note - Attributes are specified only at thread creation time; they cannot be altered while the thread is being used.

An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type.

Once an attribute is initialized and configured, it has process-wide scope. The suggested method for using attributes is to configure all required state specifications at one time in the early stages of program execution. The appropriate attribute object can then be referred to as needed.

Using attribute objects has two primary advantages.

- First, it adds to code portability.

  Even though supported attributes might vary between implementations, you need not modify function calls that create thread entities because the attribute object is hidden from the interface.

  If the target port supports attributes that are not found in the current port, provision must be made to manage the new attributes. This is an easy porting task though, because attribute objects need only be initialized once in a well-defined location.

- Second, state specification in an application is simplified.

  As an example, consider that several sets of threads might exist within a process, each providing a separate service, and each with its own state requirements.

  At some point in the early stages of the application, a thread attribute object can be initialized for each set. All future thread creations will then refer to the attribute object initialized for that type of thread. The initialization phase is simple and localized, and any future modifications can be made quickly and reliably.

Attribute objects require attention at process exit time. When the object is initialized, memory is allocated for it. This memory must be returned to the system. The `pthreads` standard provides function calls to destroy attribute objects.

Initialize Attributes

pthread_attr_init(3T)

Use `pthread_attr_init()` to initialize object attributes to their default values. The storage is allocated by the thread system during execution.

Prototype:

```
int pthread_attr_init(pthread_attr_t *tattr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* initialize an attribute to the default value */
ret = pthread_attr_init(&tattr);
```

The default values for attributes (*tattr*) are:

TABLE 3–1 Default Attribute Values for *tattr*

Attribute	Value	Result
scope	PTHREAD_SCOPE_PROCESS	New thread is unbound – not permanently attached to LWP.
detachstate	PTHREAD_CREATE_JOINABLE	Exit status and thread are preserved after the thread terminates.
stackaddr	NULL	New thread has system-allocated stack address.
stacksize	1 megabyte	New thread has system-defined stack size.
priority		New thread inherits parent thread priority.

TABLE 3–1 Default Attribute Values for *tattr* *(continued)*

Attribute	Value	Result
inheritsched	`PTHREAD_INHERIT_SCHED`	New thread inherits parent thread scheduling priority.
schedpolicy	`SCHED_OTHER`	New thread uses Solaris-defined fixed priority scheduling; threads run until preempted by a higher-priority thread or until they block or yield.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`ENOMEM`

Returned when there is not enough memory to initialize the thread attributes object.

Destroy Attributes

pthread_attr_destroy(3T)

Use `pthread_attr_destroy()` to remove the storage allocated during initialization. The attribute object becomes invalid.

```
Prototype:
int pthread_attr_destroy(pthread_attr_t *tattr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

Return Values

`pthread_attr_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Indicates that the value of *tattr* was not valid.

Set Detach State

pthread_attr_setdetachstate(3T)

When a thread is created detached (`PTHREAD_CREATE_DETACHED`), its thread *ID* and other resources can be reused as soon as the thread terminates. Use `pthread_attr_setdetachstate()` when the calling thread does not want to wait for the thread to terminate.

When a thread is created nondetached (`PTHREAD_CREATE_JOINABLE`), it is assumed that you will be waiting for it. That is, it is assumed that you will be executing a `pthread_join(3T)()` on the thread.

Whether a thread is created detached or nondetached, the process does not exit until all threads have exited. See "Finishing Up" on page 27for a discussion of process termination caused by premature exit from `main()`.

Prototype:

```
int pthread_attr_setdetachstate(pthread_attr_t *tattr,int detachstate);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* set the thread detach state */
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

Note - When there is no explicit synchronization to prevent it, a newly created, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `pthread_create()`.

For nondetached (`PTHREAD_CREATE_JOINABLE`) threads, it is very important that some thread join with it after it terminates—otherwise the resources of that thread are not released for use by new threads. This commonly results in a memory leak. So when you do not want a thread to be joined, create it as a detached thread.

CODE EXAMPLE 3–1 Creating a Detached Thread

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init()(&tattr);
ret = pthread_attr_setdetachstate()(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create()(&tid, &tattr, start_routine, arg);
```

Return Values

`pthread_attr_setdetachstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Indicates that the value of *detachstate* or *tattr* was not valid.

Get Detach State

pthread_attr_getdetachstate(3T)

Use `pthread_attr_getdetachstate()` to retrieve the thread create state, which can be either detached or joined.

Prototype:

```
int pthread_attr_getdetachstate(const pthread_attr_t *tattr,
    int *detachstate;


#include <pthread.h>

pthread_attr_t tattr;
int detachstate;
int ret;

/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

Return Values

`pthread_attr_getdetachstate( )` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Indicates that the value of *detachstate* is `NULL` or *tattr* is invalid.

Set Scope

pthread_attr_setscope(3T)

Use `pthread_attr_setscope( )` to create a bound thread (`PTHREAD_SCOPE_SYSTEM`) or an unbound thread (`PTHREAD_SCOPE_PROCESS`).

Note - Both thread types are accessible only within a given process.

`Prototype`:

```
int pthread_attr_setscope(pthread_attr_t *tattr,int scope);


#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* bound thread */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);

/* unbound thread */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_PROCESS);
```

Notice that there are three function calls in this example: one to initialize the attributes, one to set any variations from the default attributes, and one to create the pthreads.

```
#include <pthread.h>

pthread_attr_t attr;
pthread_t tid;
void start_routine;
void arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);
```

```
/* BOUND behavior */
ret =  pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

Return Values

`pthread_attr_setscope()` returns zero after completing *successfully*. Any other returned value indicates that an error occurred. If the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

An attempt was made to set *tattr* to a value that is not valid.

Get Scope

pthread_attr_getscope(3T)

Use `pthread_attr_getscope()` to retrieve the thread scope, which indicates whether the thread is bound or unbound.

Prototype:

```
int pthread_attr_getscope(pthread_attr_t *tattr, int *scope);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

Return Values

`pthread_attr_getscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value of *scope* is `NULL` or *tattr* is invalid.

Set Scheduling Policy

pthread_attr_setschedpolicy(3T)

Use `pthread_attr_setschedpolicy()` to set the scheduling policy. The POSIX draft standard specifies scheduling policy attributes of `SCHED_FIFO` (first-in-first-out), `SCHED_RR` (round-robin), or `SCHED_OTHER` (an implementation-defined method).

`SCHED_FIFO` and `SCHED_RR` are optional in POSIX, and are supported for real time bound threads, only.

Currently, only the Solaris `SCHED_OTHER` default value is supported in pthreads. For a discussion of scheduling, see the section “Scheduling” on page 7.

`Prototype`:

```
int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int policy;
int ret;

/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

An attempt was made to set *tattr* to a value that is not valid.

`ENOTSUP`

An attempt was made to set the attribute to an unsupported value.

Get Scheduling Policy

pthread_attr_getschedpolicy(3T)

Use `pthread_attr_getschedpolicy()` to retrieve the scheduling policy. Currently, only the Solaris-based `SCHED_OTHER` default value is supported in pthreads.

Prototype:

```
int pthread_attr_getschedpolicy(pthread_attr_t *tattr, int *policy);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int policy;
int ret;

/* get scheduling policy of thread */
ret = pthread_attr_getschedpolicy (&tattr, &policy);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The parameter *policy* is `NULL` or *tattr* is invalid.

Set Inherited Scheduling Policy

pthread_attr_setinheritsched(3T)

An *inherit* value of `PTHREAD_INHERIT_SCHED` (the default) means that the scheduling policies defined in the creating thread are to be used, and any scheduling attributes defined in the `pthread_create()` call are to be ignored. If `PTHREAD_EXPLICIT_SCHED` is used, the attributes from the `pthread_create()` call are to be used.

Prototype:

```
int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int inherit;
int ret;

/* use the current scheduling policy */
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

An attempt was made to set *tattr* to a value that is not valid.

`ENOTSUP`

An attempt was made to set the attribute to an unsupported value.

Get Inherited Scheduling Policy

pthread_attr_getinheritsched(3T)

Prototype:

```
int pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int inherit;
int ret;

/* get scheduling policy and priority of the creating thread */
ret = pthread_attr_getinheritsched (&tattr, &inherit);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The parameter *inherit* is `NULL` or *tattr* is invalid.

Set Scheduling Parameters

pthread_attr_setschedparam(3T)

Scheduling parameters are defined in the param structure; only priority is supported. Newly created threads run with this priority.

Prototype:

```
int pthread_attr_setschedparam(pthread_attr_t *tattr,
    const struct sched_param *param);


#include <pthread.h>

pthread_attr_t tattr;
int newprio;
sched_param param;
newprio = 30;

/* set the priority; others are unchanged */
param.sched_priority = newprio;

/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following conditions occur, the function fails and returns the corresponding value.

EINVAL

The value of *param* is NULL or *tattr* is invalid.

You can manage pthreads priority two ways. You can set the priority attribute before creating a child thread, or you can change the priority of the parent thread and then change it back.

Get Scheduling Parameters

pthread_attr_getschedparam(3T)

Prototype:

```
int pthread_attr_getschedparam(pthread_attr_t *tattr,
    const struct sched_param *param);
```

```
#include <pthread.h>

pthread_attr_t attr;
struct sched_param param;
int ret;

/* get the existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL

The value of *param* is NULL or *tattr* is invalid.

Creating a Thread With a Specified Priority

You can set the priority attribute before creating the thread. The child thread is created with the new priority that is specified in the sched_param structure (this structure also contains other scheduling information).

It is always a good idea to get the existing parameters, change the priority, xxx the thread, and then reset the priority.

Code Example 3–2 shows an example of this.

CODE EXAMPLE 3–2 Creating a Prioritized Thread

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

pthread_attr_t tattr;
pthread_t tid;
int ret;
int newprio = 20;
sched_param param;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);

/* safe to get existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);

/* set the priority; others are unchanged */
param.sched_priority = newprio;

/* setting the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);

/* with new priority specified */
ret = pthread_create (&tid, &tattr, func, arg);
```

Set Stack Size

pthread_attr_setstacksize(3T)

The *stacksize* attribute defines the size of the stack (in bytes) that the system will allocate. The size should not be less than the system-defined minimum stack size. See "About Stacks" on page 49for more information.

Prototype:

```
int pthread_attr_setstacksize(pthread_attr_t *tattr, int size);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int size;
int ret;

size = (PTHREAD_STACK_MIN + 0x4000);

/* setting a new size */
ret = pthread_attr_setstacksize(&tattr, size);
```

In the example above, *size* contains the size, in number of bytes, for the stack that the new thread uses. If *size* is zero, a default size is used. In most cases, a zero value works best.

`PTHREAD_STACK_MIN` is the amount of stack space required to start a thread. This does not take into consideration the threads routine requirements that are needed to execute application code.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value returned is less than the value of `PTHREAD_STACK_MIN`, or exceeds a system-imposed limit, or *tattr* is not valid.

Get Stack Size

pthread_attr_getstacksize(3T)

Prototype:

```
int pthread_attr_getstacksize(pthread_attr_t *tattr, size_t *size);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int size;
int ret;

/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &size);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value returned is less than the value of `PTHREAD_STACK_MIN`, or exceeds a system-imposed limit.

About Stacks

Typically, thread stacks begin on page boundaries and any specified size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows result in sending a `SIGSEGV` signal to the offending thread. Thread stacks allocated by the caller are used as is.

When a stack is specified, the thread should also be created `PTHREAD_CREATE_JOINABLE`. That stack cannot be freed until the `pthread_join(3T)` call for that thread has returned, because the thread's stack cannot be freed until the thread has terminated. The only reliable way to know if such a thread has terminated is through `pthread_join(3T)`.

Generally, you do not need to allocate stack space for threads. The `threads` library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of `mmap()` to make the allocations.)

Each thread stack created by the threads library has a red zone. The library creates the red zone by appending a page to the top of a stack to catch stack overflows. This page is invalid and causes a memory fault if it is accessed. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

Note - Because runtime stack requirements vary, you should be absolutely certain that the specified stack will satisfy the runtime requirements needed for library calls and dynamic linking.

There are very few occasions when it is appropriate to specify a stack, its size, or both. It is difficult even for an expert to know if the right size was specified. This is because even a program compliant with ABI standards cannot determine its stack size statically. Its size is dependent on the needs of the particular runtime environment in which it executes.

Building Your Own Stack

When you specify the size of a thread stack, be sure to account for the allocations needed by the invoked function and by each function called. The accounting should include calling sequence needs, local variables, and information structures.

Occasionally you want a stack that is a bit different from the default stack. An obvious situation is when the thread needs more than one megabyte of stack space. A less obvious situation is when the default stack is too large. You might be creating thousands of threads and not have enough virtual memory to handle the gigabytes of stack space that this many default stacks require.

The limits on the maximum size of a stack are often obvious, but what about the limits on its minimum size? There must be enough stack space to handle all of the stack frames that are pushed onto the stack, along with their local variables, and so on.

You can get the absolute minimum limit on stack size by calling the macro PTHREAD_STACK_MIN, which returns the amount of stack space required for a thread that executes a `NULL` procedure. Useful threads need more than this, so be very careful when reducing the stack size.

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;

int size = PTHREAD_STACK_MIN + 0x4000;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* only size specified in tattr*/
```

```
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

When you allocate your own stack, be sure to append a red zone to its end by calling `mprotect(2)`.

Set Stack Address

pthread_attr_setstackaddr(3T)

The *stackaddr* attribute defines the base of the thread's stack. If this is set to non-null (`NULL` is the default) the system initializes the stack at that address.

Prototype:

```
int pthread_attr_setstackaddr(pthread_attr_t *tattr,void *stackaddr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
void *base;
int ret;

base = (void *) malloc(PTHREAD_STACK_MIN + 0x4000);

/* setting a new address */
ret = pthread_attr_setstackaddr(&tattr, base);
```

In the example above, *base* contains the address for the stack that the new thread uses. If *base* is `NULL`, then `pthread_create(3T)` allocates a stack for the new thread with at least `PTHREAD_STACK_MIN` bytes.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value of *base* or *tattr* is incorrect.

This example shows how to create a thread with a custom stack address.

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
```

```
void *stackbase;

stackbase = (void *) malloc(size);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the base address in the attribute */
ret = pthread_attr_setstackaddr(&tattr, stackbase);

/* only address specified in attribute tattr */
ret = pthread_create(&tid, &tattr, func, arg);
```

This example shows how to create a thread with both a custom stack address and a custom stack size.

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;

int size = PTHREAD_STACK_MIN + 0x4000;
stackbase = (void *) malloc(size);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* setting the base address in the attribute */
ret = pthread_attr_setstackaddr(&tattr, stackbase);

/* address and size specified */
ret = pthread_create(&tid, &tattr, func, arg);
```

Get Stack Address

pthread_attr_getstackaddr(3T)

Prototype:

```
int pthread_attr_getstackaddr(pthread_attr_t *tattr,void * *stackaddr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
void *base;
int ret;

/* getting a new address */
ret = pthread_attr_getstackaddr (&tattr, *base);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value or *base* or *tattr* is incorrect.

CHAPTER 4

Programming with Synchronization Objects

This chapter describes the synchronization types available with threads and discusses when and how to use synchronization.

Synchronization objects are variables in memory that you access just like data. Threads in different processes can communicate with each other through synchronization objects placed in threads-controlled shared memory, even though the threads in different processes are generally invisible to each other.

Synchronization objects can also be placed in files and can have lifetimes beyond that of the creating process.

The available types of synchronization objects are:

- Mutex Locks
- Condition Variables
- Semaphores

Here are situations that can profit from the use of synchronization:

- When synchronization is the only way to ensure consistency of shared data.
- When threads in two or more processes can use a single synchronization object jointly. Note that the synchronization object should be initialized by only one of

the cooperating processes, because reinitializing a synchronization object sets it to the *unlocked* state.

- When synchronization can ensure the safety of mutable data.
- When a process can map a file and have a thread in this process get a record's lock. Once the lock is acquired, any other thread in any process mapping the file that tries to acquire the lock is blocked until the lock is released.
- Even when accessing a single primitive variable, such as an integer. On machines where the integer is not aligned to the bus data width or is larger than the data width, a single memory load can use more than one memory cycle. While this cannot happen on the SPARC® architecture, portable programs cannot rely on this.

Note - On 32-bit architectures a `long long` is not atomic[1] and is read and written as two 32-bit quantities. The types `int`, `char`, `float`, and pointers are atomic on SPARC and x86 machines.

Mutual Exclusion Lock Attributes

Use mutual exclusion locks (mutexes) to serialize thread execution. Mutual exclusion locks synchronize threads, usually by ensuring that only one thread at a time executes a critical section of code. Mutex locks can also preserve single-threaded code.

To change the default mutex attributes, you can declare and initialize an attribute object. Often, the mutex attributes are set in one place at the beginning of the application so they can be located quickly and modified easily. The following table lists the functions discussed in this section that manipulate mutex attributes.

1. An *atomic* operation cannot be divided into smaller operations.

TABLE 4–1 Mutex Attributes Routines

"Initialize a Mutex Attribute Object" on page 57	"pthread_mutexattr_init(3T)" on page 57
"Destroy a Mutex Attribute Object" on page 58	"pthread_mutexattr_destroy(3T)" on page 58
"Set the Scope of a Mutex " on page 59	"pthread_mutexattr_setpshared(3T)" on page 59
"Get the Scope of a Mutex " on page 60	"pthread_mutexattr_getpshared(3T)" on page 60

The differences between Solaris and POSIX, when defining the scope of a mutex, are shown in Table 4–2.

TABLE 4–2 Mutex Scope Comparison

Solaris	**POSIX**	**Definition**
`USYNC_PROCESS`	`PTHREAD_PROCESS_SHARED`	Use to synchronize threads in this and other processes
`USYNC_THREAD`	`PTHREAD_PROCESS_PRIVATE`	Use to synchronize threads in this process only

Initialize a Mutex Attribute Object

pthread_mutexattr_init(3T)

Use `pthread_mutexattr_init()` to initialize attributes associated with this object to their default values. Storage for each attribute object is allocated by the threads system during execution.

The default value of the *pshared* attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized mutex can be used within a process.

```
Prototype:
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

```
#include <pthread.h

pthread_mutexattr_t mattr;
int ret;

/* initialize an attribute to default value */
ret = pthread_mutexattr_init(&mattr);
```

mattr is an `opaque` type that contains a system-allocated attribute object. The possible values of *mattr*'s scope are `PTHREAD_PROCESS_PRIVATE` (the default) and `PTHREAD_PROCESS_SHARED`.

Before a mutex attribute object can be reinitialized, it must first be destroyed by `pthread_mutexattr_destroy(3T)`. The `pthread_mutexattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If either of the following conditions occurs, the function fails and returns the corresponding value.

`ENOMEM`

There is not enough memory to initialize the thread attributes object.

`EINVAL`

The value specified by *mattr* is invalid.

Destroy a Mutex Attribute Object

pthread_mutexattr_destroy(3T)

`pthread_mutexattr_destroy()` deallocates the storage space used to maintain the attribute object created by `pthread_mutexattr_init()`.

```
Prototype:
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr)
```

```
#include <pthread.h

pthread_mutexattr_t mattr;
int ret;

/* destroy an attribute */
ret = pthread_mutexattr_destroy(&mattr);
```

Return Values

`pthread_mutexattr_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *mattr* is invalid.

Set the Scope of a Mutex

pthread_mutexattr_setpshared(3T)

The scope of a mutex variable can be either process private (intraprocess) or system wide (interprocess). If the mutex is created with the pshared attribute set to the `PTHREAD_PROCESS_SHARED` state, and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the `USYNC_PROCESS` flag in `mutex_init()` in the original Solaris threads.

```
Prototype:
int pthread_mutexattr_setpshared(pthread_mutexattr_t *mattr,
    int pshared);
```

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

ret = pthread_mutexattr_init(&mattr);
/*
 * resetting to its default value: private
 */
ret = pthread_mutexattr_setpshared(&mattr,
     PTHREAD_PROCESS_PRIVATE);
```

If the mutex pshared attribute is set to `PTHREAD_PROCESS_PRIVATE`, only those threads created by the same process can operate on the mutex.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *mattr* is invalid.

Get the Scope of a Mutex

pthread_mutexattr_getpshared(3T)

```
Prototype:
int pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr,
    int *pshared);


#include <pthread.h>

pthread_mutexattr_t mattr;
int pshared, ret;

/* get pshared of mutex */
ret = pthread_mutexattr_getpshared(&mattr, &pshared);
```

Get the current value of *pshared* for the attribute object *mattr*. It is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *mattr* is invalid.

Using Mutual Exclusion Locks

After the attributes for a mutex are configured, you initialize the mutex itself. The following functions are used to initialize or destroy, lock or unlock, or try to lock a mutex. Table 4–3 lists the functions discussed in this chapter that manipulate mutex locks.

TABLE 4–3 Routines for Mutual Exclusion Locks

"Initialize a Mutex" on page 61	"pthread_mutex_init(3T)" on page 61
"Lock a Mutex" on page 62	"pthread_mutex_lock(3T)" on page 62
"Unlock a Mutex" on page 63	"pthread_mutex_unlock(3T)" on page 63
"Lock with a Nonblocking Mutex" on page 64	"pthread_mutex_trylock(3T)" on page 64
"Destroy a Mutex" on page 65	"pthread_mutex_destroy(3T)" on page 65

The default scheduling policy, `SCHED_OTHER`, does not specify the order in which threads can acquire a lock. When multiple threads are waiting for a mutex, the order of acquisition is undefined. When there is contention, the default behavior is to unblock threads in priority order.

Initialize a Mutex

pthread_mutex_init(3T)

Use `pthread_mutex_init()` to initialize the mutex pointed at by *mp* to its default value (*mattr* is `NULL`), or to specify mutex attributes that have already been set with `pthread_mutexattr_init()`.

```
Prototype:
int pthread_mutex_init(pthread_mutex_t *mp,
    const pthread_mutexattr_t *mattr);
```

```
#include <pthread.h>

pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;

/* initialize a mutex to its default value */
```

```
ret = pthread_mutex_init(&mp, NULL);

/* initialize a mutex */
ret = pthread_mutex_init(&mp, &mattr);
```

When the mutex is initialized, it is in an unlocked state.

The effect of *mattr* being NULL is the same as passing the address of a default mutex attribute object, but without the memory overhead.

Statically defined mutexes can be initialized directly to have default attributes with the macro PTHREAD_MUTEX_INITIALIZER.

A mutex lock must not be reinitialized or destroyed while other threads might be using it. Program failure will result if either action is not done correctly. If a mutex is reinitialized or destroyed, the application must be sure the mutex is not currently in use.

Return Values

pthread_mutex_init() returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

EBUSY

The mutex cannot be reinitialized or modified because it still exists.

EINVAL

The attribute value is invalid. The mutex has not been modified.

EFAULT

The address for the mutex pointed at by *mp* is invalid.

Lock a Mutex

pthread_mutex_lock(3T)

```
Prototype:
int pthread_mutex_lock(pthread_mutex_t *mp);


#include <pthread.h>

pthread_mutex_t mp;
int ret;
```

```
ret = pthread_ mutex_lock(&mp); /* acquire the mutex */
```

Use `pthread_mutex_lock()` to lock the mutex pointed to by *mp*. When the mutex is already locked, the calling thread blocks and the mutex waits on a prioritized queue. When `pthread_mutex_lock()` returns, the mutex is locked and the calling thread is the owner.

Return Values

`pthread_mutex_lock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *mp* does not refer to an initialized mutex object.

`EDEADLK`

The current thread already owns the mutex.

Unlock a Mutex

pthread_mutex_unlock(3T)

Use `pthread_mutex_unlock()` to unlock the mutex pointed to by *mp*.

```
Prototype:
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_ mutex_unlock(&mp); /* release the mutex */
```

The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner). When any other threads are waiting for the mutex to become available, the thread at the head of the queue is unblocked.

Return Values

`pthread_mutex_unlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *mp* does not refer to an initialized mutex object.

Lock with a Nonblocking Mutex

pthread_mutex_trylock(3T)

Use `pthread_mutex_trylock()` to attempt to lock the mutex pointed to by *mp*.

```
Prototype:
int pthread_mutex_trylock(pthread_mutex_t *mp);


#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_ mutex_trylock(&mp); /* try to lock the mutex */
```

`pthread_mutex_trylock()` is a nonblocking version of `pthread_mutex_lock()`. When the mutex is already locked, this call returns with an error. Otherwise, the mutex is locked and the calling thread is the owner.

Return Values

`pthread_mutex_trylock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EBUSY`

The mutex pointed to by *mp* was already locked.

`EINVAL`

The value specified by *mp* does not refer to an initialized mutex object.

Destroy a Mutex

pthread_mutex_destroy(3T)

Use `pthread_mutex_destroy()` to destroy any state associated with the mutex pointed to by *mp*.

```
Prototype:
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_mutex_destroy(&mp); /* mutex is destroyed */
```

Note that the space for storing the mutex is not freed.

Return Values

`pthread_mutex_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *mp* does not refer to an initialized mutex object.

Mutex Lock Code Examples

Here are some code fragments showing mutex locking.

CODE EXAMPLE 4–1 Mutex Lock Example

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;

void
increment_count()
{
     pthread_mutex_lock(&count_mutex);
    count = count + 1;
     pthread_mutex_unlock(&count_mutex);
}

long long
```

```
get_count()
{
    long long c;

    pthread_mutex_lock(&count_mutex);
     c = count;
    pthread_mutex_unlock(&count_mutex);
     return (c);
}
```

The two functions in Code Example 4–1 use the mutex lock for different purposes. The `increment_count()` function uses the mutex lock simply to ensure an atomic update of the shared variable. The `get_count()` function uses the mutex lock to guarantee that the 64-bit quantity *count* is read atomically. On a 32-bit architecture, a `long long` is really two 32-bit quantities.

Reading an integer value is an atomic operation because integer is the common word size on most machines.

Using Locking Hierarchies

You will occasionally want to access two resources at once. Perhaps you are using one of the resources, and then discover that the other resource is needed as well. As shown in , there could be a problem if two threads attempt to claim both resources but lock the associated mutexes in different orders. In this example, if the two threads lock mutexes 1 and 2 respectively, then a deadlock occurs when each attempts to lock the other mutex.

CODE EXAMPLE 4–2 Deadlock

Thread 1	**Thread 2**
pthread_mutex_lock(&m1);	pthread_mutex_lock(&m2);
/* use resource 1 */	/* use resource 2 */
pthread_mutex_lock(&m2);	pthread_mutex_lock(&m1);
/* use resources1 and 2 */	/* use resources 1 and 2 */
pthread_mutex_unlock(&m2);	pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m1);	pthread_mutex_unlock(&m2);

The best way to avoid this problem is to make sure that whenever threads lock multiple mutexes, they do so in the same order. This technique is known as lock hierarchies: order the mutexes by logically assigning numbers to them.

Also, honor the restriction that you cannot take a mutex that is assigned n when you are holding any mutex assigned a number greater than n.

Note - The `lock_lint` tool can detect the sort of deadlock problem shown in this example. The best way to avoid such deadlock problems is to use lock hierarchies. When locks are always taken in a prescribed order, deadlock should not occur.

However, this technique cannot always be used—sometimes you must take the mutexes in an order other than prescribed. To prevent deadlock in such a situation, use `pthread_mutex_trylock()`. One thread must release its mutexes when it discovers that deadlock would otherwise be inevitable.

shows how this is done.

CODE EXAMPLE 4–3 Conditional Locking

Thread 1	Thread 2

pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);
	for (; ;)
	{ pthread_mutex_lock(&m2);

	if(pthread_mutex_trylock(&m1)==0)
	/* got it! */
	break;

/* no processing */	/* didn't get it */

pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
	pthread_mutex_unlock(&m2);
	}

	/* get locks; no processing */
	pthread_mutex_unlock(&m1);
	pthread_mutex_unlock(&m2);

In this example, thread 1 locks mutexes in the prescribed order, but thread 2 takes them out of order. To make certain that there is no deadlock, thread 2 has to take mutex 1 very carefully; if it were to block waiting for the mutex to be released, it is likely to have just entered into a deadlock with thread 1.

To ensure this does not happen, thread 2 calls `pthread_mutex_trylock()`, which takes the mutex if it is available. If it is not, thread 2 returns immediately, reporting failure. At this point, thread 2 must release mutex 2, so that thread 1 can lock it, and then release both mutex 1 and mutex 2.

Nested Locking with a Singly Linked List

Code Example 4–4 and Code Example 4–5 show how to take three locks at once, but prevent deadlock by taking the locks in a prescribed order.

CODE EXAMPLE 4–4 Singly Linked List Structure

```
typedef struct node1 {
    int value;
    struct node1 *link;
    pthread_mutex_t lock;
} node1_t;

node1_t ListHead;
```

This example uses a singly-linked list structure with each node containing a mutex. To remove a node from the list, first search the list starting at *ListHead* (which itself is never removed) until the desired node is found.

To protect this search from the effects of concurrent deletions, lock each node before any of its contents are accessed. Because all searches start at *ListHead*, there is never a deadlock because the locks are always taken in list order.

When the desired node is found, lock both the node and its predecessor since the change involves both nodes. Because the predecessor's lock is always taken first, you are again protected from deadlock. Code Example 4–5 shows the C code to remove an item from a singly linked list.

CODE EXAMPLE 4–5 Singly-Linked List with Nested Locking

```
node1_t *delete(int value)
{
    node1_t *prev, *current;

    prev = &ListHead;
    pthread_mutex_lock(&prev->lock);

     while ((current = prev->link) != NULL) {
        pthread_mutex_lock(&current->lock);
        if (current->value == value) {
            prev->link = current->link;
            pthread_mutex_unlock(&current->lock);

             pthread_mutex_unlock(&prev->lock);
            current->link = NULL;
            return(current);
        }
        pthread_mutex_unlock(&prev->lock);
        prev = current;
    }
    pthread_mutex_unlock(&prev->lock);
    return(NULL);
}
```

Nested Locking with a Circular Linked List

Code Example 4–6 modifies the previous list structure by converting it into a circular list. There is no longer a distinguished head node; now a thread might be associated with a particular node and might perform operations on that node and its neighbor. Note that lock hierarchies do not work easily here because the obvious hierarchy (following the links) is circular.

CODE EXAMPLE 4–6 Circular Linked List Structure

```
typedef struct node2 {
    int value;
    struct node2 *link;
    pthread_mutex_t lock;
} node2_t;
```

Here is the C code that acquires the locks on two nodes and performs an operation involving both of them.

CODE EXAMPLE 4–7 Circular Linked List with Nested Locking

```
void Hit Neighbor(node2_t *me) {
   while (1) {
      pthread_mutex_lock(&me->lock);
      if (pthread_mutex_lock(&me->link->lock)!= 0) {
         /* failed to get lock */
         pthread_mutex_unlock(&me->lock);
         continue;
      }
      break;
   }
   me->link->value += me->value;
   me->value /=2;
   pthread_mutex_unlock(&me->link->lock);
   pthread_mutex_unlock(&me->lock);
}
```

Condition Variable Attributes

Use condition variables to atomically block threads until a particular condition is true. Always use condition variables together with a mutex lock.

With a condition variable, a thread can atomically block until a condition is satisfied. The condition is tested under the protection of a mutual exclusion lock (mutex).

When the condition is false, a thread usually blocks on a condition variable and atomically releases the mutex waiting for the condition to change. When another thread changes the condition, it can signal the associated condition variable to cause one or more waiting threads to wake up, acquire the mutex again, and reevaluate the condition.

Condition variables can be used to synchronize threads among processes when they are allocated in memory that can be written to and is shared by the cooperating processes.

The scheduling policy determines how blocking threads are awakened. For the default `SCHED_OTHER`, threads are awakened in priority order.

The attributes for condition variables must be set and initialized before the condition variables can be used. The functions that manipulate condition variable attributes are listed in Table 4–4.

TABLE 4–4 Condition Variable Attributes

"Initialize a Condition Variable Attribute" on page 71	"pthread_condattr_init(3T)" on page 71
"Remove a Condition Variable Attribute" on page 72	"pthread_condattr_destroy(3T)" on page 72
"Set the Scope of a Condition Variable" on page 73	"pthread_condattr_setpshared(3T)" on page 73
"Get the Scope of a Condition Variable" on page 74	"pthread_condattr_getpshared(3T)" on page 74

The differences between Solaris and POSIX threads, when defining the scope of a condition variable, are shown in Table 4–5.

TABLE 4–5 Condition Variable Scope Comparison

Solaris	POSIX	Definition
`USYNC_PROCESS`	`PTHREAD_PROCESS_SHARED`	Use to synchronize threads in this and other processes
`USYNC_THREAD`	`PTHREAD_PROCESS_PRIVATE`	Use to synchronize threads in this process only

Initialize a Condition Variable Attribute

pthread_condattr_init(3T)

Use `pthread_condattr_init()` to initialize attributes associated with this object to their default values. Storage for each attribute object is allocated by the threads system during execution. The default value of the *pshared* attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized condition variable can be used within a process.

```
Prototype:
int pthread_condattr_init(pthread_condattr_t *cattr);
```

```
#include pthread.h
pthread_condattr_t  cattr;
int ret;

/* initialize an attribute to default value */
ret = pthread_condattr_init(&cattr);
```

cattr is an opaque data type that contains a system-allocated attribute object. The possible values of *cattr*'s scope are `PTHREAD_PROCESS_PRIVATE` (the default) and `PTHREAD_PROCESS_SHARED`.

Before a condition variable attribute can be reused, it must first be reinitialized by pthread_condattr_destroy(3T). The `pthread_condattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result.

Return Values

`pthread_condattr_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`ENOMEM`

There is not enough memory to initialize the thread attributes object.

`EINVAL`

The value specified by *cattr* is invalid.

Remove a Condition Variable Attribute

pthread_condattr_destroy(3T)

Use `pthread_condattr_destroy()` to remove storage and render the attribute object invalid.

```
Prototype:
int pthread_condattr_destroy(pthread_condattr_t *cattr);


#include <pthread.h
pthread_condattr_t cattr;
int ret;

/* destroy an attribute */
ret
 = pthread_condattr_destroy(&cattr);
```

Return Values

`pthread_condattr_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *cattr* is invalid.

Set the Scope of a Condition Variable

pthread_condattr_setpshared(3T)

`pthread_condattr_setpshared()` sets the scope of a condition variable to either process private (intraprocess) or system wide (interprocess). If the condition variable is created with the pshared attribute set to the `PTHREAD_PROCESS_SHARED` state, and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the `USYNC_PROCESS` flag in `mutex_init()` in the original Solaris threads.

If the mutex pshared attribute is set to `PTHREAD_PROCESS_PRIVATE` (default value), only those threads created by the same process can operate on the mutex. Using `PTHREAD_PROCESS_PRIVATE` results in the same behavior as with the `USYNC_THREAD` flag in the original Solaris threads `cond_init()` call, which is that of a local condition variable. `PTHREAD_PROCESS_SHARED` is equivalent to a global condition variable.

```
Prototype:
int pthread_condattr_setpshared(pthread_condattr_t *cattr,
    int pshared);


#include <pthread.h>

pthread_condattr_t cattr;
int ret;

/* all processes */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

/* within a process */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

Return Values

`pthread_condattr_setpshared()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value of *cattr* is invalid, or the *pshared* value is invalid.

Get the Scope of a Condition Variable

pthread_condattr_getpshared(3T)

`pthread_condattr_getpshared()` gets the current value of *pshared* for the attribute object *cattr*. The value is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

```
Prototype:
int pthread_condattr_getpshared(const pthread_condattr_t *cattr,
    int *pshared);


#include <pthread.h>

pthread_condattr_t cattr;
int pshared;
int ret;

/* get pshared value of condition variable */
ret = pthread_condattr_getpshared(&cattr, &pshared);
```

Return Values

`pthread_condattr_getpshared()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value of *cattr* is invalid.

Using Condition Variables

This section explains using condition variables. Table 4–6 lists the functions that are available.

TABLE 4–6 Condition Variables Functions

"Initialize a Condition Variable" on page 75	"pthread_cond_init(3T)" on page 75
"Block on a Condition Variable" on page 76	"pthread_cond_wait(3T)" on page 76
"Unblock a Specific Thread" on page 78	"pthread_cond_signal(3T)" on page 78
"Block Until a Specified Event" on page 79	"pthread_cond_timedwait(3T)" on page 79
"Unblock All Threads" on page 80	"pthread_cond_broadcast(3T)" on page 80
"Destroy Condition Variable State" on page 82	"pthread_cond_destroy(3T)" on page 82

Initialize a Condition Variable

pthread_cond_init(3T)

Use `pthread_cond_init()` to initialize the condition variable pointed at by *cv* to its default value (*cattr* is NULL), or to specify condition variable attributes that are already set with `pthread_condattr_init()`. The effect of *cattr* being NULL is the same as passing the address of a default condition variable attribute object, but without the memory overhead.

```
Prototype:
int pthread_cond_init(pthread_cond_t *cv,
    const pthread_condattr_t *cattr);
```

```
#include <pthread.h>

pthread_cond_t cv;
pthread_condattr_t cattr;
int ret;

/* initialize a condition variable to its default value */
```

```
ret = pthread_cond_init(&cv, NULL);

/* initialize a condition variable */
ret = pthread_cond_init(&cv, &cattr);
```

Statically-defined condition variables can be initialized directly to have default attributes with the macro `PTHREAD_COND_INITIALIZER`. This has the same effect as dynamically allocating `pthread_cond_init()` with null attributes. No error checking is done.

Multiple threads must not simultaneously initialize or reinitialize the same condition variable. If a condition variable is reinitialized or destroyed, the application must be sure the condition variable is not in use.

Return Values

`pthread_cond_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *cattr* is invalid.

`EBUSY`

The condition variable is being used.

`EAGAIN`

The necessary resources are not available.

`ENOMEM`

There is not enough memory to initialize the condition variable.

Block on a Condition Variable

pthread_cond_wait(3T)

Use `pthread_cond_wait()` to atomically release the mutex pointed to by *mp* and to cause the calling thread to block on the condition variable pointed to by *cv*.

```
Prototype:
int pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);
```

```
#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mp;
int ret;

/* wait on condition variable */
ret = pthread_cond_wait(&cv, &mp);
```

The blocked thread can be awakened by a `pthread_cond_signal()`, a `pthread_cond_broadcast()`, or when interrupted by delivery of a signal.

Any change in the value of a condition associated with the condition variable cannot be inferred by the return of `pthread_cond_wait()`, and any such condition must be reevaluated.

The `pthread_cond_wait()` routine always returns with the mutex locked and owned by the calling thread, even when returning an error.

This function blocks until the condition is signaled. It atomically releases the associated mutex lock before blocking, and atomically acquires it again before returning.

In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when it changes the condition value. This causes one or all of the threads waiting on the condition to unblock and to try to acquire the mutex lock again.

Because the condition can change before an awakened thread returns from `pthread_cond_wait()`, the condition that caused the wait must be retested before the mutex lock is acquired. The recommended test method is to write the condition check as a `while()` loop that calls `pthread_cond_wait()`.

```
    pthread_mutex_lock();
        while(condition_is_false)
            pthread_cond_wait();
    pthread_mutex_unlock();
```

No specific order of acquisition is guaranteed when more than one thread blocks on the condition variable.

Note - `pthread_cond_wait()` is a cancellation point. If a cancel is pending and the calling thread has cancellation enabled, the thread terminates and begins executing its cleanup handlers while continuing to hold the lock.

Return Values

`pthread_cond_wait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *cv* or *mp* is invalid.

Unblock a Specific Thread

pthread_cond_signal(3T)

Use `pthread_cond_signal()` to unblock one thread that is blocked on the condition variable pointed to by *cv*.

```
Prototype:
int pthread_cond_signal(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* one condition variable is signaled */
ret = pthread_cond_signal(&cv);
```

Call `pthread_cond_signal()` under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

The scheduling policy determines the order in which blocked threads are awakened. For `SCHED_OTHER`, threads are awakened in priority order.

When no threads are blocked on the condition variable, then calling `pthread_cond_signal()` has no effect.

Return Values

`pthread_cond_signal()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

cv points to an illegal address.

CODE EXAMPLE 4–8 Using `pthread_cond_wait()` and `pthread_cond_signal()`

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
```

```
unsigned count;

decrement_count()
{
    pthread_mutex_lock(&count_lock);
    while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}

increment_count()
{
    pthread_mutex_lock(&count_lock);
    if (count == 0)
        pthread_cond_signal(&count_nonzero);
    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}
```

Block Until a Specified Event

pthread_cond_timedwait(3T)

```
Prototype:
int pthread_cond_timedwait(pthread_cond_t *cv,
    pthread_mutex_t *mp, const struct timespec *abstime);


#include <pthread.h>
#include <time.h>

pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;

/* wait on condition variable */
ret = pthread_cond_timedwait(&cv, &mp, &abstime);
```

Use `pthread_cond_timedwait()` as you would use `pthread_cond_wait()`, except that `pthread_cond_timedwait()` does not block past the time of day specified by *abstime*. `pthread_cond_timedwait()` always returns with the mutex locked and owned by the calling thread, even when it is returning an error.

The `pthread_cond_timedwait()` function blocks until the condition is signaled or until the time of day, specified by the last argument, has passed.

Note - `pthread_cond_timedwait()` is also a cancellation point.

Return Values

`pthread_cond_timedwait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

cv or *abstime* points to an illegal address.

`ETIMEDOUT`

The time specified by *abstime* has passed.

The time-out is specified as a time of day so that the condition can be retested efficiently without recomputing the value, as shown in Code Example 4–9.

CODE EXAMPLE 4–9 Timed Condition Wait

```
pthread_timestruc_t to;
pthread_mutex_t m;
pthread_cond_t c;
...
pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
    err = pthread_cond_timedwait(&c, &m, &to);
    if (err == ETIMEDOUT) {
        /* timeout, do something */
        break;
    }
}
pthread_mutex_unlock(&m);
```

Unblock All Threads

pthread_cond_broadcast(3T)

```
Prototype:
int pthread_cond_broadcast(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* all condition variables are signaled */
ret = pthread_cond_broadcast(&cv);
```

Use `pthread_cond_broadcast()` to unblock all threads that are blocked on the condition variable pointed to by *cv*, specified by `pthread_cond_wait()`. When no threads are blocked on the condition variable, `pthread_cond_broadcast()` has no effect.

Return Values

`pthread_cond_broadcast()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

cv points to an illegal address.

Condition Variable Broadcast Example

Since `pthread_cond_broadcast()` causes all threads blocked on the condition to contend again for the mutex lock, use it with care. For example, use `pthread_cond_broadcast()` to allow threads to contend for varying resource amounts when resources are freed, as shown in Code Example 4–10.

CODE EXAMPLE 4–10 Condition Variable Broadcast

```
pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    while (resources < amount) {
        pthread_cond_wait(&rsrc_add, &rsrc_lock);
    }
    resources -= amount;
    pthread_mutex_unlock(&rsrc_lock);
}

add_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    resources += amount;
    pthread_cond_broadcast(&rsrc_add);
    pthread_mutex_unlock(&rsrc_lock);
}
```

Note that in `add_resources()` it does not matter whether *resources* is updated first or if `pthread_cond_broadcast()` is called first inside the mutex lock.

Call `pthread_cond_broadcast()` under the protection of the same mutex that is used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

Destroy Condition Variable State

pthread_cond_destroy(3T)

Use `pthread_cond_destroy()` to destroy any state associated with the condition variable pointed to by *cv*.

```
Prototype:
int pthread_cond_destroy(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* Condition variable is destroyed */
ret = pthread_cond_destroy(&cv);
```

Note that the space for storing the condition variable is not freed.

Return Values

`pthread_cond_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

The value specified by *cv* is invalid.

The Lost Wake-Up Problem

Calling `pthread_cond_signal()` or `pthread_cond_broadcast()` when the thread does not hold the mutex lock associated with the condition can lead to *lost wake-up* bugs.

A lost wake-up occurs when

- A thread calls `pthread_cond_signal()` or `pthread_cond_broadcast()`

- *And* another thread is between the test of the condition and the call to `pthread_cond_wait()`
- *And* no threads are waiting.

  The signal has no effect, and therefore is lost.

The Producer/Consumer Problem

This problem is one of the small collection of standard, well-known problems in concurrent programming: a finite-size buffer and two classes of threads, producers and consumers, put items into the buffer (producers) and take items out of the buffer (consumers).

A producer must wait until the buffer has space before it can put something in, and a consumer must wait until something is in the buffer before it can take something out.

A condition variable represents a queue of threads waiting for some condition to be signaled.

Code Example 4–11 has two such queues, one (*less*) for producers waiting for a slot in the buffer, and the other (*more*) for consumers waiting for a buffer slot containing information. The example also has a mutex, as the data structure describing the buffer must be accessed by only one thread at a time.

CODE EXAMPLE 4–11 The Producer/Consumer Problem and Condition Variables

```
typedef struct {
    char buf[BSIZE];
    int occupied;
    int nextin;
    int nextout;
    pthread_mutex_t mutex;
    pthread_cond_t more;
    pthread_cond_t less;
} buffer_t;

buffer_t buffer;
```

As Code Example 4–12 shows, the producer thread acquires the mutex protecting the `buffer` data structure and then makes certain that space is available for the item being produced. If not, it calls `pthread_cond_wait()`, which causes it to join the queue of threads waiting for the condition *less*, representing *there is room in the buffer*, to be signaled.

At the same time, as part of the call to `pthread_cond_wait()`, the thread releases its lock on the mutex. The waiting producer threads depend on consumer threads to signal when the condition is true (as shown in Code Example 4–12). When the condition is signaled, the first thread waiting on *less* is awakened. However, before the thread can return from `pthread_cond_wait()`, it must acquire the lock on the mutex again.

This ensures that it again has mutually exclusive access to the buffer data structure. The thread then must check that there really is room available in the buffer; if so, it puts its item into the next available slot.

At the same time, consumer threads might be waiting for items to appear in the buffer. These threads are waiting on the condition variable *more*. A producer thread, having just deposited something in the buffer, calls `pthread_cond_signal()` to wake up the next waiting consumer. (If there are no waiting consumers, this call has no effect.)

Finally, the producer thread unlocks the mutex, allowing other threads to operate on the buffer data structure.

CODE EXAMPLE 4–12 The Producer/Consumer Problem – the Producer

```
void producer(buffer_t *b, char item)
{
    pthread_mutex_lock(&b->mutex);

    while (b->occupied >= BSIZE)
        pthread_cond_wait(&b->less, &b->mutex);

    assert(b->occupied < BSIZE);

    b->buf[b->nextin++] = item;

    b->nextin %= BSIZE;
    b->occupied++;

    /* now: either b->occupied < BSIZE and b->nextin is the index
       of the next empty slot in the buffer, or
       b->occupied == BSIZE and b->nextin is the index of the
       next (occupied) slot that will be emptied by a consumer
       (such as b->nextin == b->nextout) */

    pthread_cond_signal(&b->more);

    pthread_mutex_unlock(&b->mutex);
}
```

Note the use of the `assert()` statement; unless the code is compiled with `NDEBUG` defined, `assert()` does nothing when its argument evaluates to true (that is, nonzero), but causes the program to abort if the argument evaluates to false (zero). Such assertions are especially useful in multithreaded programs— they immediately point out runtime problems if they fail, and they have the additional effect of being useful comments.

The comment that begins `/* now: either b->occupied ...` could better be expressed as an assertion, but it is too complicated as a Boolean-valued expression and so is given in English.

Both the assertion and the comments are examples of invariants. These are logical statements that should not be falsified by the execution of the program, except during brief moments when a thread is modifying some of the program variables mentioned in the invariant. (An assertion, of course, should be true whenever any thread executes it.)

Using invariants is an extremely useful technique. Even if they are not stated in the program text, think in terms of invariants when you analyze a program.

The invariant in the producer code that is expressed as a comment is always true whenever a thread is in the part of the code where the comment appears. If you move this comment to just after the `mutex_unlock()`, this does not necessarily remain true. If you move this comment to just after the `assert()`, this is still true.

The point is that this invariant expresses a property that is true at all times, except when either a producer or a consumer is changing the state of the buffer. While a thread is operating on the buffer (under the protection of a mutex), it might temporarily falsify the invariant. However, once the thread is finished, the invariant should be true again.

Code Example 4–13 shows the code for the consumer. Its flow is symmetric with that of the producer.

CODE EXAMPLE 4–13 The Producer/Consumer Problem – the Consumer

```
char consumer(buffer_t *b)
{
    char item;
    pthread_mutex_lock(&b->mutex);
    while(b->occupied <= 0)
        pthread_cond_wait(&b->more, &b->mutex);

    assert(b->occupied > 0);

    item = b->buf[b->nextout++];
    b->nextout %= BSIZE;
    b->occupied--;

    /* now: either b->occupied > 0 and b->nextout is the index
       of the next occupied slot in the buffer, or
       b->occupied == 0 and b->nextout is the index of the next
       (empty) slot that will be filled by a producer (such as
       b->nextout == b->nextin) */

    pthread_cond_signal(&b->less);
    pthread_mutex_unlock(&b->mutex);

    return(item);
}
```

Semaphores

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed.

Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter.

In the computer version, a semaphore appears to be a simple integer. A thread waits for permission to proceed and then signals that it has proceeded by performing a P operation on the semaphore.

The semantics of the operation are such that the thread must wait until the semaphore's value is positive, then change the semaphore's value by subtracting one from it. When it is finished, the thread performs a V operation, which changes the semaphore's value by adding one to it. It is crucial that these operations take place atomically—they cannot be subdivided into pieces between which other actions on the semaphore can take place. In the P operation, the semaphore's value must be positive just before it is decremented (resulting in a value that is guaranteed to be nonnegative and one less than what it was before it was decremented).

In both P and V operations, the arithmetic must take place without interference. If two V operations are performed simultaneously on the same semaphore, the net effect should be that the semaphore's new value is two greater than it was.

The mnemonic significance of P and V is lost on most of the world, as Dijkstra is Dutch. However, in the interest of true scholarship: P stands for prolagen, a made-up word derived from proberen te verlagen, which means *try to decrease*. V stands for verhogen, which means *increase*. This is discussed in one of Dijkstra's technical notes, EWD 74.

sem_wait(3R) and sem_post(3R) correspond to Dijkstra's P and V operations. sem_trywait(3R) is a conditional form of the P operation: if the calling thread cannot decrement the value of the semaphore without waiting, the call returns immediately with a nonzero value.

There are two basic sorts of semaphores: binary semaphores, which never take on values other than zero or one, and counting semaphores, which can take on arbitrary nonnegative values. A binary semaphore is logically just like a mutex.

However, although it is not enforced, mutexes should be unlocked only by the thread holding the lock. There is no notion of "the thread holding the semaphore," so any thread can perform a V (or sem_post(3R)) operation.

Counting semaphores are about as powerful as conditional variables (used in conjunction with mutexes). In many cases, the code might be simpler when it is implemented with counting semaphores rather than with condition variables (as shown in the next few examples).

However, when a mutex is used with condition variables, there is an implied bracketing—it is clear which part of the program is being protected. This is not necessarily the case for a semaphore, which might be called the *go to* of concurrent programming—it is powerful but too easy to use in an unstructured, indeterminate way.

Counting Semaphores

Conceptually, a semaphore is a nonnegative integer count. Semaphores are typically used to coordinate access to resources, with the semaphore count initialized to the number of free resources. Threads then atomically increment the count when resources are added and atomically decrement the count when resources are removed.

When the semaphore count becomes zero, indicating that no more resources are present, threads trying to decrement the semaphore block wait until the count becomes greater than zero.

TABLE 4–7 Routines for Semaphores

"Initialize a Semaphore" on page 88	"sem_init(3R)" on page 88
"Increment a Semaphore" on page 90	"sem_post(3R)" on page 90
"Block on a Semaphore Count" on page 90	"sem_wait(3R)" on page 90
"Decrement a Semaphore Count" on page 91	"sem_trywait(3R)" on page 91
"Destroy the Semaphore State" on page 92	"sem_destroy(3R)" on page 92

Because semaphores need not be acquired and released by the same thread, they can be used for asynchronous event notification (such as in signal handlers). And, because semaphores contain state, they can be used asynchronously without acquiring a mutex lock as is required by condition variables. However, semaphores are not as efficient as mutex locks.

By default, there is no defined order of unblocking if multiple threads are waiting for a semaphore.

Semaphores must be initialized before use, but they do not have attributes.

Initialize a Semaphore

sem_init(3R)

```
Prototype:
int sem_init(sem_t *sem, int pshared, unsigned int value);


#include <semaphore.h>

sem_t sem;
int pshared;
int ret;
int value;

/* initialize a private semaphore */
pshared = 0;
value = 1;
ret = sem_init(&sem, pshared, value);
```

Use `sem_init()` to initialize the semaphore variable pointed to by *sem* to *value* amount. If the value of pshared is zero, then the semaphore cannot be shared between processes. If the value of pshared is nonzero, then the semaphore can be shared between processes.

Multiple threads must not initialize the same semaphore.

A semaphore must not be reinitialized while other threads might be using it.

Return Values

`sem_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

The value argument exceeds `SEM_VALUE_MAX`.

`ENOSPC`

A resource required to initialize the semaphore has been exhausted. The limit on semaphores `SEM_NSEMS_MAX` has been reached.

`EPERM`

The process lacks the appropriate privileges to initialize the semaphore.

Initializing Semaphores with Intraprocess Scope

When *pshared* is 0, the semaphore can be used by all the threads in this process only.

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be used within this process only */
ret = sem_init(&sem, 0, count);
```

Initializing Semaphores with Interprocess Scope

When *pshared* is nonzero, the semaphore can be shared by other processes.

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be shared among processes */
ret = sem_init(&sem, 1, count);
```

Named Semaphores

The functions `sem_open(3R)`, `sem_getvalue(3R)`, `sem_close(3R)`, and `sem_unlink(3R)` are available to `open`, `retrieve`, `close`, and `remove` named semaphores. Using `sem_open()`, you can create a semaphore that has a name defined in the file system name space.

Named semaphores are like process shared semaphores, except that they are referenced with a pathname rather than a *pshared* value.

For more information about named semaphores, see `sem_open(3R)`, `sem_getvalue(3R)`, `sem_close(3R)`, and `sem_unlink(3R)`.

Increment a Semaphore

sem_post(3R)

```
Prototype:
int sem_post(sem_t *sem);


#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_post(&sem); /* semaphore is posted */
```

Use `sem_post()` to atomically increment the semaphore pointed to by *sem*. When any threads are blocked on the semaphore, one of them is unblocked.

Return Values

`sem_post()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

sem points to an illegal address.

Block on a Semaphore Count

sem_wait(3R)

```
Prototype:
int sem_wait(sem_t *sem);


#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_wait(&sem); /* wait for semaphore */
```

Use `sem_wait()` to block the calling thread until the count in the semaphore pointed to by *sem* becomes greater than zero, then atomically decrement it.

Return Values

`sem_wait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

sem points to an illegal address.

`EINTR`

A signal interrupted this function.

Decrement a Semaphore Count

sem_trywait(3R)

```
Prototype:
int sem_trywait(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_trywait(&sem); /* try to wait for semaphore*/
```

Use `sem_trywait()` to try to atomically decrement the count in the semaphore pointed to by *sem* when the count is greater than zero. This function is a nonblocking version of `sem_wait()`; that is it returns immediately if unsuccessful.

Return Values

`sem_trywait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

sem points to an illegal address.

`EINTR`

A signal interrupted this function.

`EAGAIN`

The semaphore was already locked, so it cannot be immediately locked by the `sem_trywait()` operation.

Destroy the Semaphore State

sem_destroy(3R)

```
Prototype:
int sem_destroy(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_destroy(&sem); /* the semaphore is destroyed */
```

Use `sem_destroy()` to destroy any state associated with the semaphore pointed to by *sem*. The space for storing the semaphore is not freed.

Return Values

`sem_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

sem points to an illegal address.

The Producer/Consumer Problem, Using Semaphores

The data structure in Code Example 4–14 is similar to that used for the condition variables example (see Code Example 4–11). Two semaphores represent the number of full and empty buffers and ensure that producers wait until there are empty buffers and that consumers wait until there are full buffers.

CODE EXAMPLE 4–14 The Producer/Consumer Problem with Semaphores

```
typedef struct {
    char buf[BSIZE];
    sem_t occupied;
    sem_t empty;
    int nextin;
    int nextout;
    sem_t pmut;
    sem_t cmut;
} buffer_t;

buffer_t buffer;

sem_init(&buffer.occupied, 0, 0);

 sem_init(&buffer.empty,0, BSIZE);
sem_init(&buffer.pmut, 0, 1);
sem_init(&buffer.cmut, 0, 1);
buffer.nextin = buffer.nextout = 0;
```

Another pair of (binary) semaphores plays the same role as mutexes, controlling access to the buffer when there are multiple producers and multiple empty buffer slots, and when there are multiple consumers and multiple full buffer slots. Mutexes would work better here, but would not provide as good an example of semaphore use.

CODE EXAMPLE 4–15 The Producer/Consumer Problem – the Producer

```
void producer(buffer_t *b, char item) {
    sem_wait(&b->empty);

     sem_wait(&b->pmut);

    b->buf[b->nextin] = item;
    b->nextin++;
    b->nextin %= BSIZE;

    sem_post(&b->pmut);

     sem_post(&b->occupied);
}
```

CODE EXAMPLE 4–16 The Producer/Consumer Problem – the Consumer

```
char consumer(buffer_t *b) {
    char item;

    sem_wait(&b->occupied);

    sem_wait(&b->cmut);

    item = b->buf[b->nextout];
    b->nextout++;
    b->nextout %= BSIZE;

    sem_post(&b->cmut);

    sem_post(&b->empty);

    return(item);
}
```

Synchronization Across Process Boundaries

Each of the synchronization primitives can be set up to be used across process boundaries. This is done quite simply by ensuring that the synchronization variable is located in a shared memory segment and by calling the appropriate `init()` routine, after the primitive has been initialized with its shared attribute set as interprocess.

Producer/Consumer Problem Example

Code Example 4–17 shows the producer/consumer problem with the producer and consumer in separate processes. The main routine maps zero-filled memory (that it shares with its child process) into its address space.

A child process is created that runs the consumer. The parent runs the producer.

This example also shows the drivers for the producer and consumer. The `producer_driver()` simply reads characters from `stdin` and calls `producer()`. The `consumer_driver()` gets characters by calling `consumer()` and writes them to `stdout`.

The data structure for Code Example 4–17 is the same as that used for the condition variables example (see Code Example 4–4). Two semaphores represent the number of full and empty buffers and ensure that producers wait until there are empty buffers and that consumers wait until there are full buffers.

CODE EXAMPLE 4–17 Synchronization Across Process Boundaries

```
main() {
    int zfd;
    buffer_t *buffer;
    pthread_mutexattr_t mattr;
    pthread_condattr_t cvattr_less, cvattr_more;

    zfd = open("/dev/zero", O_RDWR);
    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
    buffer->occupied = buffer->nextin = buffer->nextout = 0;

    pthread_mutex_attr_init(&mattr);
    pthread_mutexattr_setpshared(&mattr,
        PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&buffer->lock, &mattr);
    pthread_condattr_init(&cvattr_less);
    pthread_condattr_setpshared(&cvattr_less, PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&buffer->less, &cvattr_less);
    pthread_condattr_init(&cvattr_more);
    pthread_condattr_setpshared(&cvattr_more,
        PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&buffer->more, &cvattr_more);

    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}

void producer_driver(buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, ‘\0’);
            break;
        } else
            producer(b, (char)item);
    }
}

void consumer_driver(buffer_t *b) {
    char item;

    while (1) {
        if ((item = consumer(b)) == ’\0’)
            break;
        putchar(item);
    }
}
```

Interprocess Locking without the Threads Library

Although not generally recommended, it is possible in Solaris threads to do interprocess locking without using the threads library. If this is something you want to do, see the instructions in “Using LWPs Between Processes” on page 186.

Comparing Primitives

The most basic synchronization primitive in threads is the mutual exclusion lock. So, it is the most efficient mechanism in both memory use and execution time. The basic use of a mutual exclusion lock is to serialize access to a resource.

The next most efficient primitive in threads is the condition variable. The basic use of a condition variable is to block on a change of state; that is it provides a thread wait facility. Remember that a mutex lock must be acquired before blocking on a condition variable and must be unlocked after returning from `pthread_cond_wait()`. The mutex lock must also be held across the change of state that occurs before the corresponding call to `pthread_cond_signal()`.

The semaphore uses more memory than the condition variable. It is easier to use in some circumstances because a semaphore variable functions on state rather than on control. Unlike a lock, a semaphore does not have an owner. Any thread can increment a semaphore that has blocked.

CHAPTER **5**

Programming with the Operating System

This chapter describes how multithreading interacts with the Solaris operating system and how the operating system has changed to support multithreading.

Process Creation–Forking Issues

The default handling of the `fork()` function in the Solaris operating system is somewhat different from the way `fork()` is handled in POSIX threads, although the Solaris operating system does support both mechanisms.

Table 5–1 compares the differences and similarities of Solaris and pthreads `fork()` handling. When the comparable interface is not available either in POSIX threads or in Solaris threads, the '-' character appears in the table column.

TABLE 5–1 Comparing POSIX and Solaris `fork()` Handling

	Solaris Operating System Interface	**POSIX Threads Interface**
Fork One Model	`fork1(2)`	`fork(2)`
Fork All Model	`fork(2)`	—
Fork-Safety	—	`pthread_atfork(3T)`

The Fork One Model

As shown in Table 5–1, the behavior of the pthreads `fork(2)` function is the same as that of the Solaris `fork1(2)` function. Both the pthreads `fork(2)` function and the Solaris `fork1(2)` create a new process, duplicating the complete address space in the child, but duplicating only the calling thread in the child process.

This is useful when the child process immediately calls `exec()`, which is what happens after most calls to `fork()`. In this case, the child process does not need a duplicate of any thread other than the one that called `fork()`.

In the child, do not call any library functions after calling fork() and before calling `exec()` because one of the library functions might use a lock that was held in the parent at the time of the `fork()`. The child process may execute only Async-Signal-Safe operations until one of the `exec()` handlers is called.

The Fork One Safety Problem and Solution

In addition to all of the usual concerns such as locking shared data, a library should be well-behaved with respect to forking a child process when only one thread is running (the one that called `fork()`). The problem is that the sole thread in the child process might try to grab a lock that is held by a thread that wasn't duplicated in the child.

This is not a problem most programs are likely to run into. Most programs call `exec()` in the child right after the return from `fork()`. However, if the program wishes to carry out some actions in the child before the call to `exec()`, or never calls `exec()`, then the child *could* encounter deadlock scenarios.

Each library writer should provide a safe solution, although not providing a fork-safe library is not a large concern because this condition is rare.

For example, assume that T1 is in the middle of printing something (and so is holding a lock for `printf()`), when T2 forks a new process. In the child process, if the sole thread (T2) calls `printf()`, it promptly deadlocks.

The POSIX `fork()` or Solaris `fork1()` duplicates only the thread that calls it. (Calling the Solaris `fork()` duplicates all threads, so this issue does not come up.)

To prevent deadlock, ensure that no such locks are being held at the time of forking. The most obvious way to do this is to have the forking thread acquire all the locks that could possibly be used by the child. Because you cannot do this for locks like those in `printf()` (because `printf()` is owned by libc), you must ensure that `printf()` is not being used at `fork()` time.

To manage the locks in your library:

- Identify all the locks used by the library.
- Identify the locking order for the locks used by the library. (If a strict locking order is not used, then lock acquisition must be managed carefully.)
- Arrange to acquire those locks at fork time. In Solaris threads this must be done manually, obtaining the locks just before calling `fork1()`, and releasing them right after.

In the following example, the list of locks used by the library is {`L1,...Ln`}, and the locking order for these locks is also `L1...Ln`.

```
mutex_lock(L1);
mutex_lock(L2);
fork1(...);
mutex_unlock(L1);
mutex_unlock(L2);
```

In pthreads, you can add a call to `pthread_atfork(f1, f2, f3)` in your library's `.init()` section, where `f1()`, `f2()`, `f3()` are defined as follows:

```
f1() /* This is executed just before the process forks. */
{
 mutex_lock(L1); |
 mutex_lock(...); | -- ordered in lock order
 mutex_lock(Ln); |
 } V

f2() /* This is executed in the child after the process forks. */
 {
 mutex_unlock(L1);
 mutex_unlock(...);
 mutex_unlock(Ln);
 }

f3() /* This is executed in the parent after the process forks. */
 {
 mutex_unlock(L1);
 mutex_unlock(...);
 mutex_unlock(Ln);
 }
```

Another example of deadlock would be a thread in the parent process—other than the one that called Solaris `fork1(2)`—that has locked a mutex. This mutex is copied into the child process in its locked state, but no thread is copied over to unlock the mutex. So, any thread in the child that tries to lock the mutex waits forever.

Virtual Forks–vfork(2)

The standard `vfork(2)` function is unsafe in multithreaded programs. `vfork(2)` is like `fork1(2)` in that only the calling thread is copied in the child process. As in nonthreaded implementations, `vfork()` does not copy the address space for the child process.

Be careful that the thread in the child process does not change memory before it calls `exec(2)`. Remember that `vfork()` gives the parent address space to the child. The parent gets its address space back after the child calls `exec()` or exits. It is important that the child not change the state of the parent.

For example, it is dangerous to create new threads between the call to `vfork()` and the call to `exec()`. This is an issue only if the fork one model is used, and only if the child does more than just call `exec()`. Most libraries are not fork-safe, so use `pthread_atfork()` to implement fork safety.

The Solution—pthread_atfork(3T)

Use `pthread_atfork()` to prevent deadlocks whenever you use the fork one model.

```
#include <pthread.h>

int pthread_atfork(void (*prepare) (void), void (*parent) (void),
    void (*child) (void) );
```

The `pthread_atfork()` function declares `fork()` handlers that are called before and after `fork()` in the context of the thread that called `fork()`.

- The *prepare* handler is called before `fork()` starts.
- The *parent* handler is called after `fork()` returns in the parent.
- The *child* handler is called after `fork()` returns in the child.

Any one of these can be set to `NULL`. The order in which successive calls to `pthread_atfork()` are made is significant.

For example, a *prepare* handler could acquire all the mutexes needed, and then the *parent* and *child* handlers could release them. This ensures that all the relevant locks are held by the thread that calls the fork function *before* the process is forked, preventing the deadlock in the child.

Using the fork all model avoids the deadlock problem described in "The Fork One Safety Problem and Solution" on page 98.

Return Values

`pthread_atfork()` returns a zero when it completes successfully. Any other returned value indicates that an error occurred. If the following condition is detected, pthread_atfork(3T) fails and returns the corresponding value.

`ENOMEM`

Insufficient table space exists to record the fork handler addresses.

The Fork All Model

The Solaris fork(2) function duplicates the address space and all the threads (and LWPs) in the child. This is useful, for example, when the child process never calls `exec(2)` but does use its copy of the parent address space. The fork all functionality is not available in POSIX threads.

Note that when one thread in a process calls Solaris `fork(2)`, threads that are blocked in an interruptible system call return `EINTR`.

Also, be careful not to create locks that are held by both the parent and child processes. This can happen when locks are allocated in memory that is sharable (that is mmap'ed with the `MAP_SHARED` flag). Note that this is not a problem if the fork one model is used.

Choosing the Right Fork

You determine whether `fork()` has a "fork all" or a "fork one" semantic in your application by linking with the appropriate library. Linking with `–lthread` gives you the "fork all" semantic for `fork()`, and linking with `–lpthread` gives the "fork one" semantic for `fork()` (see Figure 7–1for an explanation of compiling options).

Cautions for Any Fork

Be careful when using global state after a call to any `fork()` function.

For example, when one thread reads a file serially and another thread in the process successfully calls one of the forks, each process then contains a thread that is reading the file. Because the seek pointer for a file descriptor is shared after a `fork()`, the thread in the parent gets some data while the thread in the child gets the other. This introduces gaps in the sequential read accesses.

Process Creation–exec(2)and exit(2) Issues

Both the `exec(2)` and `exit(2)` system calls work as they do in single-threaded processes except that they destroy all the threads in the address space. Both calls block until all the execution resources (and so all active threads) are destroyed.

When `exec()` rebuilds the process, it creates a single lightweight process (LWP) . The process startup code builds the initial thread. As usual, if the initial thread returns, it calls `exit()` and the process is destroyed.

When all the threads in a process exit, the process exits. A call to any `exec()` function from a process with more than one thread terminates all threads, and loads and executes the new executable image. No destructor functions are called.

Timers, Alarms, and Profiling

The "End of Life" announcements for per-LWP timers (see `timer_create(3R)`) and per-thread alarms (see `alarm(2)` or `setitimer(2)`) were made in the Solaris 2.5 release. Both features are now supplemented with the per-process variants described in this section.

Originally, each LWP had a unique Realtime interval timer and alarm that a thread bound to the LWP could use. The timer or alarm delivered one signal to the thread when the timer or alarm expired.

Each LWP also had a virtual time or profile interval timer that a thread bound to the LWP could use. When the interval timer expired, either `SIGVTALRM` or `SIGPROF`, as appropriate, was sent to the LWP that owned the interval timer.

Per-LWP POSIX Timers

In the Solaris 2.3 and 2.4 releases, the `timer_create(3R)` function returned a timer object whose timer ID was meaningful only within the calling LWP and whose expiration signals were delivered to that LWP. Because of this, the only threads that could use the POSIX timer facility were bound threads.

Even with this restricted use, POSIX timers in Solaris 2.3 and 2.4 multithreaded applications were unreliable about masking the resulting signals and delivering the associated value from the sigvent structure.

With the Solaris 2.5 release, an application that is compiled defining the macro `_POSIX_PER_PROCESS_TIMERS`, or with a value greater that `199506L` for the symbol `_POSIX_C_SOURCE`, can create per-process timers.

Applications compiled with a release before the Solaris 2.5 release, or without the feature test macros, will continue to create per-LWP POSIX timers. In some future release, calls to create per-LWP timers will return per-process timers.

The timer IDs of per-process timers are usable from any LWP, and the expiration signals are generated for the process rather than directed to a specific LWP.

The per-process timers are deleted only by `timer_delete(3R)` or when the process terminates.

Per-Thread Alarms

In the Solaris 2.3 and 2.4 releases, a call to `alarm(2)` or `setitimer(2)` was meaningful only within the calling LWP. Such timers were deleted automatically when the creating LWP terminated. Because of this, the only threads that could use these were bound threads.

Even with this restricted use, `alarm()` and `setitimer()` timers in Solaris 2.3 and 2.4 multithreaded applications were unreliable about masking the signals from the bound thread that issued these calls. When such masking was not required, then these two system calls worked reliably from bound threads.

With the Solaris 2.5 release, an application linking with `–lpthread` (POSIX) threads will get per-process delivery of `SIGALRM` when calling alarm(). The `SIGALRM` generated by alarm() is generated for the process rather than directed to a specific LWP. Also, the alarm is reset when the process terminates.

Applications compiled with a release before the Solaris 2.5 release, or not linked with `–lpthread`, will continue to see a per-LWP delivery of signals generated by `alarm()` and `setitimer()`.

In some future release, calls to `alarm()` or to `setitimer()` with the `ITIMER_REAL` flag will cause the resulting `SIGALRM` to be sent to the process. For other flags, `setitmer()` will continue to be per-LWP. Flags other than the `ITIMER_REAL` flag, used by `setitimer()`, will continue to result in the generated signal being delivered to the LWP that issued the call, and so are usable only from bound threads.

Profiling

You can profile each LWP with `profil(2)`, giving each LWP its own buffer, or sharing buffers between LWPs. Profiling data is updated at each clock tick in LWP user time. The profile state is inherited from the creating LWP.

Nonlocal Goto—setjmp(3C) and longjmp(3C)

The scope of `setjmp()` and `longjmp()` is limited to one thread, which is fine most of the time. However, this does mean that a thread that handles a signal can `longjmp()` only when `setjmp()` is performed in the same thread.

Resource Limits

Resource limits are set on the entire process and are determined by adding the resource use of all threads in the process. When a soft resource limit is exceeded, the offending thread is sent the appropriate signal. The sum of the resources used in the process is available through getrusage(3B).

LWPs and Scheduling Classes

As mentioned in the "Scheduling" section of the "Covering Multithreading Basics", the Solaris pthreads implementation supports only the `SCHED_OTHER` scheduling policy. The others are optional under POSIX.

The POSIX `SCHED_FIFO` and `SCHED_RR` policies can be duplicated or emulated using the standard Solaris mechanisms. These scheduling mechanisms are described in this section.

The Solaris kernel has three classes of scheduling. The highest priority scheduling class is Realtime (`RT`). The middle priority scheduling class is `system`. The `system` class cannot be applied to a user process. The lowest priority scheduling class is timeshare (`TS`), which is also the default class.

Scheduling class is maintained for each LWP. When a process is created, the initial LWP inherits the scheduling class and priority of the creating LWP in the parent process. As more LWPs are created to run unbound threads, they also inherit this scheduling class and priority.

All unbound threads in a process have the same scheduling class and priority. Each scheduling class maps the priority of the LWP it is scheduling to an overall dispatching priority according to the configurable priority of the scheduling class.

Bound threads have the scheduling class and priority of their underlying LWPs. Each bound thread in a process can have a unique scheduling class and priority that is visible to the kernel. Bound threads are scheduled with respect to all other LWPs in the system.

Thread priorities regulate access to LWP resources. By default LWPs are in the timesharing class. For compute-bound multithreading, thread priorities are not very useful. For multithreaded applications that do a lot of synchronization using the MT libraries, thread priorities become more meaningful.

The scheduling class is set by priocntl(2). How you specify the first two arguments determines whether just the calling LWP or all the LWPs of one or more processes are affected. The third argument of `priocntl()` is the command, which can be one of the following.

- `PC_GETCID`. Get the class ID and class attributes for a specific class.
- `PC_GETCLINFO`. Get the class name and class attributes for a specific class.
- `PC_GETPARMS`. Get the class identifier and the class-specific scheduling parameters of a process, an LWP with a process, or a group of processes.
- `PC_SETPARMS`. Set the class identifier and the class-specific scheduling parameters of a process, an LWP with a process, or a group of processes.

Use `priocntl()` only on bound threads. To affect the priority of unbound threads, use pthread_setprio(3T).

Timeshare Scheduling

Timeshare scheduling distributes the processing resource fairly among the LWPs in this scheduling class. Other parts of the kernel can monopolize the processor for short intervals without degrading response time as seen by the user.

The `priocntl(2)` call sets the nice(2) level of one or more processes. The `priocntl()` call also affects the `nice()` level of all the timesharing class LWPs in the process. The `nice()` level ranges from 0 to +20 normally and from -20 to +20 for processes with superuser privilege. The lower the value, the higher the priority.

The dispatch priority of time-shared LWPs is calculated from the instantaneous CPU use rate of the LWP and from its `nice()` level. The `nice()` level indicates the relative priority of the LWPs to the timeshare scheduler.

LWPs with a greater `nice()` value get a smaller, but nonzero, share of the total processing. An LWP that has received a larger amount of processing is given lower priority than one that has received little or no processing.

Realtime Scheduling

The Realtime class (RT) can be applied to a whole process or to one or more LWPs in a process. This requires superuser privilege.

Unlike the `nice(2)` level of the timeshare class, LWPs that are classified Realtime can be assigned priorities either individually or jointly. A `priocntl(2)` call affects the attributes of all the Realtime LWPs in the process.

The scheduler always dispatches the highest-priority Realtime LWP. It preempts a lower-priority LWP when a higher-priority LWP becomes runnable. A preempted LWP is placed at the head of its level queue.

A Realtime LWP retains control of a processor until it is preempted, it suspends, or its Realtime priority is changed. LWPs in the RT class have absolute priority over processes in the TS class.

A new LWP inherits the scheduling class of the parent process or LWP. An RT class LWP inherits the parent's time slice, whether finite or infinite.

An LWP with a finite time slice runs until it terminates, blocks (for example, to wait for an I/O event), is preempted by a higher-priority runnable Realtime process, or the time slice expires.

An LWP with an infinite time slice ceases execution only when it terminates, blocks, or is preempted.

LWP Scheduling and Thread Binding

The threads library automatically adjusts the number of LWPs in the pool used to run unbound threads. Its objectives are:

- To prevent the program from being blocked by a lack of unblocked LWPs.

  For example, if there are more runnable unbound threads than LWPs and all the active threads block in the kernel in indefinite waits (such as while reading a tty), the process cannot progress until a waiting thread returns.

- To make efficient use of LWPs.

  For example, if the library creates one LWP for each thread, many LWPs will usually be idle and the operating system is overloaded by the resource requirements of the unused LWPs.

Keep in mind that LWPs are time-sliced, not threads. This means that when there is only one LWP, there is no time slicing within the process—threads run on the LWP until they block (through interthread synchronization), are preempted, or terminate.

You can assign priorities to threads with `pthread_setprio(3T)`; lower-priority unbound threads are assigned to LWPs only when no higher-priority unbound threads are available. Bound threads, of course, do not compete for LWPs because they have their own. Note that the thread priority that is set with `pthread_setprio()` regulates threads access to LWPs, not to CPUs.

Bind threads to your LWPs to get precise control over whatever is being scheduled. This control is not possible when many unbound threads compete for an LWP.

In particular, a lower-priority unbound thread could be on a higher priority LWP and running on a CPU, while a higher-priority unbound thread assigned to a lower-priority LWP is not running. In this sense, thread priorities are just a hint about access to CPUs.

Realtime threads are useful for getting a quick response to external stimuli. Consider a thread used for mouse tracking that must respond instantly to mouse clicks. By binding the thread to an LWP, you guarantee that there is an LWP available when it is needed. By assigning the LWP to the Realtime scheduling class, you ensure that the LWP is scheduled quickly in response to mouse clicks.

SIGWAITING—Creating LWPs for Waiting Threads

The library usually ensures that there are enough LWPs in its pool for a program to proceed.

When all the LWPs in the process are blocked in indefinite waits (such as blocked reading from a tty or network), the operating system sends the new signal, `SIGWAITING`, to the process. This signal is handled by the threads library. When the process contains a thread that is waiting to run, a new LWP is created and the appropriate waiting thread is assigned to it for execution.

The `SIGWAITING` mechanism does not ensure that an additional LWP is created when one or more threads are compute bound and another thread becomes runnable. A compute-bound thread can prevent multiple runnable threads from being started because of a shortage of LWPs.

This can be prevented by calling `thr_setconcurrency(3T)`. While using `thr_setconcurrency()` with POSIX threads is not POSIX compliant, its use is recommended to avoid LWP shortages for unbound threads in some computationally-intensive situations. (The only way to be *completely* POSIX compliant *and* avoid LWP shortages is to create only `PTHREAD_SCOPE_SYSTEM` bound threads.)

See “Thread Concurrency (Solaris Threads Only)” on page 199for more information about using the thr_setconcurrency(3T) function.

In Solaris threads, you can also use `THR_NEW_LWP` in calls to `thr_create(3T)` to create another LWP.

Aging LWPs

When the number of active threads is reduced, some of the LWPs in the pool are no longer needed. When there are more LWPs than active threads, the threads library destroys the unneeded LWPs. The library ages LWPs—they are deleted when they are unused for a “long” time, default is five minutes.

Extending Traditional Signals

The traditional UNIX signal model is extended to threads in a fairly natural way. The key characteristics are that the signal disposition is process-wide, but the signal mask is per-thread. The process-wide disposition of signals is established using the traditional mechanisms (`signal(2)`, `sigaction(2)`, and so on).

When a signal handler is marked `SIG_DFL` or `SIG_IGN`, the action on receipt of the signal (exit, core dump, stop, continue, or ignore) is performed on the entire receiving process, affecting all threads in the process. For these signals that don’t have handlers, the issue of which thread picks the signal is unimportant, because the action on receipt of the signal is carried out on the whole process. See `signal(5)` for basic information about signals.

Each thread has its own signal mask. This lets a thread block some signals while it uses memory or another state that is also used by a signal handler. All threads in a process share the set of signal handlers set up by `sigaction(2)` and its variants.

A thread in one process cannot send a signal to a specific thread in another process. A signal sent by `kill(2)` or `sigsend(2)` to a process is handled by any one of the receptive threads in the process.

Unbound threads cannot use alternate signal stacks. A bound thread can use an alternate stack because the state is associated with the execution resource. An alternate stack must be enabled for the signal through `sigaction(2)`, and declared and enabled through `signaltstack(2)`.

An application can have per-thread signal handlers based on the per-process signal handlers. One way is for the process-wide signal handler to use the identifier of the thread handling the signal as an index into a table of per-thread handlers. Note that there is no thread zero.

Signals are divided into two categories: traps and exceptions (synchronously generated signals) and interrupts (asynchronously generated signals).

As in traditional UNIX, if a signal is pending, additional occurrences of that signal have no additional effect—a pending signal is represented by a bit, not by a counter. In other words, signal delivery is idempotent.

As is the case with single-threaded processes, when a thread receives a signal while blocked in a system call, the thread might return early, either with the `EINTR` error code, or, in the case of I/O calls, with fewer bytes transferred than requested.

Of particular importance to multithreaded programs is the effect of signals on `pthread_cond_wait(3T)`. This call usually returns in response to a `pthread_cond_signal(3T)` or a `pthread_cond_broadcast(3T)`, but, if the waiting thread receives a traditional UNIX signal, it returns with the error code `EINTR`. See "Interrupted Waits on Condition Variables (Solaris Threads Only)" on page 116for more information.

Synchronous Signals

Traps (such as `SIGILL`, `SIGFPE`, `SIGSEGV`) result from something a thread does to itself, such as dividing by zero or explicitly sending itself a signal. A trap is handled only by the thread that caused it. Several threads in a process can generate and handle the same type of trap simultaneously.

Extending the idea of signals to individual threads is easy for synchronous signals—the signal is dealt with by the thread that caused the problem.

However, if the thread has not chosen to deal with the problem, such as by establishing a signal handler with `sigaction(2)`, the handler is invoked on the thread that receives the synchronous signal.

Because such a synchronous signal usually means that something is seriously wrong with the whole process, and not just with a thread, terminating the process is often a good choice.

Asynchronous Signals

Interrupts (such as `SIGINT` and `SIGIO`) are asynchronous with any thread and result from some action outside the process. They might be signals sent explicitly by other threads, or they might represent external actions such as a user typing `Control-c`. Dealing with asynchronous signals is more complicated than dealing with synchronous signals.

An interrupt can be handled by any thread whose signal mask allows it. When more than one thread is able to receive the interrupt, only one is chosen.

When multiple occurrences of the same signal are sent to a process, then each occurrence can be handled by a separate thread, as long as threads are available that do not have it masked. When all threads have the signal masked, then the signal is marked *pending* and the first thread to unmask the signal handles it.

Continuation Semantics

Continuation semantics are the traditional way to deal with signals. The idea is that when a signal handler returns, control resumes where it was at the time of the interruption. This is well suited for asynchronous signals in single-threaded processes, as shown in Code Example 5–1.

This is also used as the exception-handling mechanism in some programming languages, such as PL/1.

CODE EXAMPLE 5–1 Continuation Semantics

```
unsigned int nestcount;

unsigned int A(int i, int j) {
    nestcount++;

    if (i==0)
        return(j+1)
    else if (j==0)
        return(A(i-1, 1));
    else
        return(A(i-1, A(i, j-1)));
}

void sig(int i) {
    printf("nestcount = %d\n", nestcount);
}

main() {
    sigset(SIGINT, sig);
    A(4,4);
}
```

Operations on Signals

pthread_sigsetmask(3T)

`pthread_sigsetmask(3T)` does for a thread what `sigprocmask(2)` does for a process—it sets the `thread`'s signal mask. When a new thread is created, its initial mask is inherited from its creator.

The call to `sigprocmask( )` in a multithreaded process is equivalent to a call to `pthread_sigsetmask( )`. See the **`sigprocmask`**(2)page for more information.

pthread_kill(3T)

`pthread_kill(3T)` is the thread analog of `kill(2)`—it sends a signal to a specific thread.This, of course, is different from sending a signal to a process. When a signal is sent to a process, the signal can be handled by any thread in the process. A signal sent by `pthread_kill( )` can be handled only by the specified thread.

Note than you can use `pthread_kill()` to send signals only to threads in the current process. This is because the thread identifier (type *thread_t*) is local in scope—it is not possible to name a thread in any process but your own.

Note also that the action taken (handler, `SIG_DFL`, `SIG_IGN`) on receipt of a signal by the target thread is global, as usual. This means, for example, that if you send SIG*XXX* to a thread, and the SIG*XXX* signal disposition for the process is to kill the process, then the whole process is killed when the target thread receives the signal.

sigwait(2)

For multithreaded programs, sigwait(2) is the preferred interface to use, because it deals so well with aysynchronously-generated signals.

`sigwait()` causes the calling thread to wait until any signal identified by its set argument is delivered to the thread. While the thread is waiting, signals identified by the set argument are unmasked, but the original mask is restored when the call returns.

All signals identified by the set argument must be blocked on all threads, including the calling thread; otherwise, `sigwait()` may not work correctly.

Use `sigwait()` to separate threads from asynchronous signals. You can create one thread that is listening for asynchronous signals while your other threads are created to block any asynchronous signals that might be set to this process.

New `sigwait()` *Implementations*

Two versions of `sigwait()` are available in the Solaris 2.5 release: the new Solaris 2.5 version, and the POSIX.1c version. New applications and libraries should use the POSIX standard interface, as the Solaris version might not be available in future releases.

Note - The new Solaris 2.5 `sigwait()` does not override the signal's ignore disposition. Applications relying on the older `sigwait(2)` behavior can break unless you install a dummy signal handler to change the disposition from `SIG_IGN` to having a handler, so calls to `sigwait()` for this signal catch it.

The syntax for the two versions of `sigwait()` is shown below.

```
#include <signal.h>

/* the Solaris 2.5 version*/
int sigwait(sigset_t *set);

/* the POSIX.1c version */
int sigwait(const sigset_t *set, int *sig);
```

When the signal is delivered, the POSIX.1c `sigwait()` clears the pending signal and places the signal number in *sig*. Many threads can call `sigwait()` at the same time, but only one thread returns for each signal that is received.

With `sigwait()` you can treat asynchronous signals synchronously—a thread that deals with such signals simply calls `sigwait()` and returns as soon as a signal arrives. By ensuring that all threads (including the caller of `sigwait()`) have such signals masked, you can be sure that signals are handled only by the intended handler and that they are handled safely.

By always masking all signals in all threads, and just calling `sigwait()` as necessary, your application will be much safer for threads that depend on signals.

Usually, you use `sigwait()` to create one or more threads that wait for signals. Because `sigwait()` can retrieve even masked signals, be sure to block the signals of interest in all other threads so they are not accidentally delivered.

When the signals arrive, a thread returns from `sigwait()`, handles the signal, and waits for more signals. The signal-handling thread is not restricted to using Async-Signal-Safe functions and can synchronize with other threads in the usual way. (The Async-Signal-Safe category is defined in “MT Interface Safety Levels” on page 124.)

Note - `sigwait()` should *never* be used with synchronous signals.

---

sigtimedwait(2)

`sigtimedwait(2)` is similar to `sigwait(2)` except that it fails and returns an error when a signal is not received in the indicated amount of time.

Thread-Directed Signals

The UNIX signal mechanism is extended with the idea of thread-directed signals. These are just like ordinary asynchronous signals, except that they are sent to a particular thread instead of to a process.

Waiting for asynchronous signals in a separate thread can be safer and easier than installing a signal handler and processing the signals there.

A better way to deal with asynchronous signals is to treat them synchronously. By calling `sigwait(2)`, discussed on “sigwait(2)” on page 111, a thread can wait until a signal occurs.

CODE EXAMPLE 5–2 Asynchronous Signals and sigwait(2)

```
main() {
    sigset_t set;
    void runA(void);
    int sig;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    pthread_sigsetmask(SIG_BLOCK, &set, NULL);
    pthread_create(NULL, 0, runA, NULL, PTHREAD_DETACHED, NULL);

    while (1) {
        sigwait(&set, &sig);
        printf("nestcount = %d\n", nestcount);
        printf("received signal %d\n", sig);
    }
}

void runA() {
    A(4,4);
    exit(0);
}
```

This example modifies the code of Code Example 5–1: the main routine masks the `SIGINT` signal, creates a child thread that calls the function A of the previous example, and then issues `sigwait()` to handle the `SIGINT` signal.

Note that the signal is masked in the compute thread because the compute thread inherits its signal mask from the main thread. The main thread is protected from `SIGINT` while, and only while, it is not blocked inside of `sigwait()`.

Also, note that there is never any danger of having system calls interrupted when you use `sigwait()`.

Completion Semantics

Another way to deal with signals is with completion semantics.

Use completion semantics when a signal indicates that something so catastrophic has happened that there is no reason to continue executing the current code block. The signal handler runs instead of the remainder of the block that had the problem. In other words, the signal handler completes the block.

In Code Example 5–3, the block in question is the body of the `then` part of the `if` statement. The call to `setjmp(3C)` saves the current register state of the program in *jbuf* and returns 0, thereby executing the block.

CODE EXAMPLE 5–3 Completion Semantics

```
sigjmp_buf jbuf;
void mult_divide(void) {
    int a, b, c, d;
    void problem();

    sigset(SIGFPE, problem);
    while (1) {
        if (sigsetjmp(&jbuf) == 0) {
            printf("Three numbers, please:\n");
            scanf("%d %d %d", &a, &b, &c);
            d = a*b/c;
            printf("%d*%d/%d = %d\n", a, b, c, d);
        }
    }
}

void problem(int sig) {
    printf("Couldn't deal with them, try again\n");
    siglongjmp(&jbuf, 1);
}
```

If a `SIGFPE` (a floating-point exception) occurs, the signal handler is invoked.

The signal handler calls `siglongjmp(3C)`, which restores the register state saved in *jbuf*, causing the program to return from `sigsetjmp()` again (among the registers saved are the program counter and the stack pointer).

This time, however, `sigsetjmp(3C)` returns the second argument of `siglongjmp()`, which is 1. Notice that the block is skipped over, only to be executed during the next iteration of the `while` loop.

Note that you can use `sigsetjmp(3C)` and `siglongjmp(3C)` in multithreaded programs, but be careful that a thread never does a `siglongjmp()` using the results of another thread's `sigsetjmp()`.

Also, `sigsetjmp()` and `siglongjmp()` save and restore the signal mask, but `setjmp(3C)` and `longjmp(3C)` do not.

It is best to use `sigsetjmp()` and `siglongjmp()` when you work with signal handlers.

Completion semantics are often used to deal with exceptions. In particular, the Ada® programming language uses this model.

Note - Remember, sigwait(2) should *never* be used with synchronous signals.

Signal Handlers and Async-Signal Safety

A concept similar to thread safety is Async-Signal safety. Async-Signal-Safe operations are guaranteed not to interfere with operations that are being interrupted.

The problem of Async-Signal safety arises when the actions of a signal handler can interfere with the operation that is being interrupted.

For example, suppose a program is in the middle of a call to `printf(3S)` and a signal occurs whose handler itself calls `printf()`. In this case, the output of the two `printf()` statements would be intertwined. To avoid this, the handler should not call `printf()` itself when `printf()` might be interrupted by a signal.

This problem cannot be solved by using synchronization primitives because any attempted synchronization between the signal handler and the operation being synchronized would produce immediate deadlock.

Suppose that `printf()` is to protect itself by using a mutex. Now suppose that a thread that is in a call to `printf()`, and so holds the lock on the mutex, is interrupted by a signal.

If the handler (being called by the thread that is still inside of `printf()`) itself calls `printf()`, the thread that holds the lock on the mutex will attempt to take it again, resulting in an instant deadlock.

To avoid interference between the handler and the operation, either ensure that the situation never arises (perhaps by masking off signals at critical moments) or invoke only Async-Signal-Safe operations from inside signal handlers.

Because setting a thread's mask is an inexpensive user-level operation, you can inexpensively make functions or sections of code fit in the Async-Signal-Safe category.

The only routines that POSIX guarantees to be Async-Signal-Safe are listed in Table 5–2. Any signal handler can safely call in to one of these functions.

TABLE 5–2 Async-Signal-Safe Functions

`_exit()`	`fstat()`	`read()`	`sysconf()`
`access()`	`getegid()`	`rename()`	`tcdrain()`
`alarm()`	`geteuid()`	`rmdir()`	`tcflow()`
`cfgetispeed()`	`getgid()`	`setgid()`	`tcflush()`
`cfgetospeed()`	`getgroups()`	`setpgid()`	`tcgetattr()`
`cfsetispeed()`	`getpgrp()`	`setsid()`	`tcgetpgrp()`
`cfsetospeed()`	`getpid()`	`setuid()`	`tcsendbreak()`
`chdir()`	`getppid()`	`sigaction()`	`tcsetattr()`
`chmod()`	`getuid()`	`sigaddset()`	`tcsetpgrp()`
`chown()`	`kill()`	`sigdelset()`	`time()`

TABLE 5–2 Async-Signal-Safe Functions *(continued)*

`close()`	`link()`	`sigemptyset()`	`times()`
`creat()`	`lseek()`	`sigfillset()`	`umask()`
`dup2()`	`mkdir()`	`sigismember()`	`uname()`
`dup()`	`mkfifo()`	`sigpending()`	`unlink()`
`execle()`	`open()`	`sigprocmask()`	`utime()`
`execve()`	`pathconf()`	`sigsuspend()`	`wait()`
`fcntl()`	`pause()`	`sleep()`	`waitpid()`
`fork()`	`pipe()`	`stat()`	`write()`

Interrupted Waits on Condition Variables (Solaris Threads Only)

When a signal is delivered to a thread while the thread is waiting on a condition variable, the old convention (assuming that the process is not terminated) is that interrupted calls return `EINTR`.

The ideal new condition would be that when `cond_wait(3T)` and `cond_timedwait(3T)` return, the lock has been retaken on the mutex.

This is what is done in Solaris threads: when a thread is blocked in `cond_wait()` or `cond_timedwait()` and an unmasked, caught signal is delivered to the thread, the handler is invoked and the call to `cond_wait()` or `cond_timedwait()` returns `EINTR` with the mutex locked.

This implies that the mutex is locked in the signal handler because the handler might have to clean up after the thread. While this is true in the Solaris 2.5 release, it might change in the future, so do not rely upon this behavior.

Note - In POSIX threads, `pthread_cond_wait(3T)` returns from signals, but this is not an error, `pthread_cond_wait()` returns zero as a spurious wakeup.

Handler cleanup is illustrated by Code Example 5–4.

CODE EXAMPLE 5–4 Condition Variables and Interrupted Waits

```
int sig_catcher() {
    sigset_t set;
    void hdlr();

    mutex_lock(&mut);

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigsetmask(SIG_UNBLOCK, &set, 0);

    if (cond_wait(&cond, &mut) == EINTR) {
        /* signal occurred and lock is held */
        cleanup();
        mutex_unlock(&mut);
        return(0);
    }
    normal_processing();
    mutex_unlock(&mut);
    return(1);
}

void hdlr() {
    /* lock is held in the handler */
    ...
}
```

Assume that the `SIGINT` signal is blocked in all threads on entry to `sig_catcher()` and that `hdlr()` has been established (with a call to `sigaction(2)`) as the handler for the `SIGINT` signal. When an unmasked and caught instance of the `SIGINT` signal is delivered to the thread while it is in `cond_wait()`, the thread first reacquires the lock on the mutex, then calls `hdlr()`, and then returns `EINTR` from `cond_wait()`.

Note that whether `SA_RESTART` has been specified as a flag to `sigaction()` has no effect here; `cond_wait(3T)` is not a system call and is not automatically restarted. When a caught signal occurs while a thread is blocked in `cond_wait()`, the call always returns `EINTR`. Again, the application should not rely on an interrupted `cond_wait()` reacquiring the mutex, because this behavior could change in the future.

I/O Issues

One of the attractions of multithreaded programming is I/O performance. The traditional UNIX API gave you little assistance in this area—you either used the facilities of the file system or bypassed the file system entirely.

This section shows how to use threads to get more flexibility through I/O concurrency and multibuffering. This section also discusses the differences and similarities between the approaches of synchronous I/O (with threads) and asynchronous I/O (with and without threads).

I/O as a Remote Procedure Call

In the traditional UNIX model, I/O appears to be synchronous, as if you were placing a remote procedure call to the I/O device. Once the call returns, then the I/O has completed (or at least it appears to have completed—a write request, for example, might merely result in the transfer of the data to a buffer in the operating system).

The advantage of this model is that it is easy to understand because, as a programmer you are very familiar with the concept of procedure calls.

An alternative approach not found in traditional UNIX systems is the asynchronous model, in which an I/O request merely starts an operation. The program must somehow discover when the operation completes.

This approach is not as simple as the synchronous model, but it has the advantage of allowing concurrent I/O and processing in traditional, single-threaded UNIX processes.

Tamed Asynchrony

You can get most of the benefits of asynchronous I/O by using synchronous I/O in a multithreaded program. Where, with asynchronous I/O, you would issue a request and check later to determine when it completes, you can instead have a separate thread perform the I/O synchronously. The main thread can then check (perhaps by calling pthread_join(3T)) for the completion of the operation at some later time.

Asynchronous I/O

In most situations there is no need for asynchronous I/O, since its effects can be achieved with the use of threads, with each thread doing synchronous I/O. However, in a few situations, threads cannot achieve what asynchronous I/O can.

The most straightforward example is writing to a tape drive to make the tape drive stream. Streaming prevents the tape drive from stopping while it is being written to and moves the tape forward at high speed while supplying a constant stream of data that is written to tape.

To do this, the tape driver in the kernel must issue a queued write request when the tape driver responds to an interrupt that indicates that the previous tape-write operation has completed.

Threads cannot guarantee that asynchronous writes will be ordered because the order in which threads execute is indeterminate. Specifying the order of a write to a tape, for example, is not possible.

Asynchronous I/O Operations

```
#include <sys/asynch.h>

int aioread(int fildes, char *bufp, int bufs, off_t offset,

     int whence, aio_result_t *resultp);

int aiowrite(int filedes, const char *bufp, int bufs,
    off_t offset, int whence, aio_result_t *resultp);

aio_result_t *aiowait(const struct timeval *timeout);


int aiocancel(aio_result_t *resultp);
```

`aioread(3)` and `aiowrite(3)` are similar in form to `pread(2)` and `pwrite(2)`, except for the addition of the last argument. Calls to `aioread()` and `aiowrite()` result in the initiation (or queueing) of an I/O operation.

The call returns without blocking, and the status of the call is returned in the structure pointed to by *resultp*. This is an item of type `aio_result_t` that contains the following:

```
int aio_return;
int aio_errno;
```

When a call fails immediately, the failure code can be found in `aio_errno`. Otherwise, this field contains `AIO_INPROGRESS`, meaning that the operation has been successfully queued.

You can wait for an outstanding asynchronous I/O operation to complete by calling `aiowait(3)`. This returns a pointer to the `aio_result_t` structure supplied with the original `aioread(3)` or `aiowrite(3)` call.

This time `aio_result_t` contains whatever `read(2)` or `write(2)` would have returned if one of them had been called instead of the asynchronous version. If the `read()` or `write()` is successful, `aio_return` contains the number of bytes that were read or written; if it was not successful, `aio_return` is -1, and `aio_errno` contains the error code.

`aiowait()` takes a *timeout* argument, which indicates how long the caller is willing to wait. As usual, a `NULL` pointer here means that the caller is willing to wait indefinitely, and a pointer to a structure containing a zero value means that the caller is unwilling to wait at all.

You might start an asynchronous I/O operation, do some work, then call `aiowait()` to wait for the request to complete. Or you can use `SIGIO` to be notified, asynchronously, when the operation completes.

Finally, a pending asynchronous I/O operation can be cancelled by calling `aiocancel()`. This routine is called with the address of the result area as an argument. This result area identifies which operation is being cancelled.

Shared I/O and New I/O System Calls

When multiple threads are performing I/O operations at the same time with the same file descriptor, you might discover that the traditional UNIX I/O interface is not thread-safe. The problem occurs with nonsequential I/O. This uses the `lseek(2)` system call to set the file offset, which is then used in the next `read(2)` or `write(2)` call to indicate where in the file the operation should start. When two or more threads are issuing `lseeks()` to the same file descriptor, a conflict results.

To avoid this conflict, use the `pread(2)` and `pwrite(2)` system calls.

```
#include <sys/types.h>
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

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

These behave just like read(2) and write(2) except that they take an additional argument, the file offset. With this argument, you specify the offset without using lseek(2), so multiple threads can use these routines safely for I/O on the same file descriptor.

Alternatives to getc(3S) and putc(3S)

An additional problem occurs with standard I/O. Programmers are accustomed to routines such as `getc(3S)` and `putc(3S)` being very quick—they are implemented as macros. Because of this, they can be used within the inner loop of a program with no concerns about efficiency.

However, when they are made thread safe they suddenly become more expensive—they now require (at least) two internal subroutine calls, to lock and unlock a mutex.

To get around this problem, alternative versions of these routines are supplied, `getc_unlocked(3S)` and `putc_unlocked(3S)`.

These do not acquire locks on a mutex and so are as quick as the original, nonthread-safe versions of `getc(3S)` and `putc(3S)`.

However, to use them in a thread-safe way, you must explicitly lock and release the mutexes that protect the standard I/O streams, using `flockfile(3S)` and `funlockfile(3S)`. The calls to these latter routines are placed outside the loop, and the calls to `getc_unlocked()` or `putc_unlocked()` are placed inside the loop.

CHAPTER 6

Safe and Unsafe Interfaces

This chapter defines MT-safety levels for functions and libraries.

Thread Safety

Thread safety is the avoidance of data races—situations in which data are set to either correct or incorrect values, depending upon the order in which multiple threads access and modify the data.

When no sharing is intended, give each thread a private copy of the data. When sharing is important, provide explicit synchronization to make certain that the program behaves in a deterministic manner.

A procedure is thread safe when it is logically correct when executed simultaneously by several threads. At a practical level, it is convenient to recognize three levels of safety.

- Unsafe
- Thread safe—Serializable
- Thread safe—MT-safe

An unsafe procedure can be made thread safe and serializable by surrounding it with statements to lock and unlock a mutex. Code Example 6–1shows three simplified implementations of `fputs()`, initially thread unsafe.

Next is a serializable version of this routine with a single mutex protecting the procedure from concurrent execution problems. Actually, this is stronger synchronization than is usually necessary. When two threads are sending output to different files using `fputs()`, one need not wait for the other—the threads need synchronization only when they are sharing an output file.

The last version is MT-safe. It uses one lock for each file, allowing two threads to print to different files at the same time. So, a routine is MT-safe when it is thread safe and its execution does not negatively affect performance.

CODE EXAMPLE 6–1 Degrees of Thread Safety

```
/* not thread-safe */
fputs(const char *s, FILE *stream) {
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
}

/* serializable */
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m);
    for (p=s; *p; p++)
        putc((int)*p, stream);

    mutex_unlock(&m);
}

/* MT-Safe */
mutex_t m[NFILE];
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m[fileno(stream)]);
    for (p=s; *p; p++)
        putc((int)*p, stream);
    mutex_unlock(&m[fileno(stream)]0;
}
```

MT Interface Safety Levels

The *man Pages(3): Library Routines* use the following categories to describe how well an interface supports threads (these categories are explained more fully in the `Intro(3)` reference manual page).

Safe	This code can be called from a multithreaded application.
Safe with exceptions	See the NOTES sections of these pages for a description of the exceptions.
Unsafe	This interface is not safe to use with multithreaded applications unless the application arranges for only one thread at a time to execute within the library.
MT-Safe	This interface is fully prepared for multithreaded access in that it is both *safe* and it supports some concurrency.
MT-Safe with exceptions	See the NOTES sections of these pages in the *man Pages(3): Library Routines* for a list of the exceptions.
Async-Signal-Safe	This routine can safely be called from a signal handler. A thread that is executing an Async-Signal-Safe routine does not deadlock with itself when it is interrupted by a signal.
Fork1-Safe	This interface releases locks it has held whenever the Solaris fork1(2) or the POSIX fork(2) is called.

See the table in Appendix C,; for the safety levels of interfaces from the *man Pages(3): Library Routines*. Check the man page to be sure of the level.

Some functions have purposely not been made safe for the following reasons.

- Making the interface MT-Safe would have negatively affected the performance of single-threaded applications.
- The library has an Unsafe interface. For example, a function might return a pointer to a buffer in the stack. You can use reentrant counterparts for some of these functions. The reentrant function name is the original function name with "`_r`" appended.

Caution - There is no way to be certain that a function whose name does not end in "`_r`" is MT-Safe other than by checking its reference manual page. Use of a function identified as not MT-Safe must be protected by a synchronizing device or by restriction to the initial thread.

Reentrant Functions for Unsafe Interfaces

For most functions with Unsafe interfaces, an MT-Safe version of the routine exists. The name of the new MT-Safe routine is always the name of the old Unsafe routine with "`_r`" appended. The Table 6–1 "`_r`" routines are supplied in the Solaris system.

TABLE 6–1 Reentrant Functions

asctime_r(3c)	gethostbyname_r(3n)	getservbyname_r(3n)
ctermid_r(3s)	gethostent_r(3n)	getservbyport_r(3n)
ctime_r(3c)	getlogin_r(3c)	getservent_r(3n)
fgetgrent_r(3c)	getnetbyaddr_r(3n)	getspent_r(3c)
fgetpwent_r(3c)	getnetbyname_r(3n)	getspnam_r(3c)
fgetspent_r(3c)	getnetent_r(3n)	gmtime_r(3c)
gamma_r(3m)	getnetgrent_r(3n)	lgamma_r(3m)
getauclassent_r(3)	getprotobyname_r(3n)	localtime_r(3c)
getauclassnam_r(3)	getprotobynumber_r(3n)	nis_sperror_r(3n)
getauevent_r(3)	getprotoent_r(3n)	rand_r(3c)
getauevnam_r(3)	getpwent_r(3c)	readdir_r(3c)
getauevnum_r(3)	getpwnam_r(3c)	strtok_r(3c)
getgrent_r(3c)	getpwuid_r(3c)	tmpnam_r(3s)
getgrgid_r(3c)	getrpcbyname_r(3n)	ttyname_r(3c)
getgrnam_r(3c)	getrpcbynumber_r(3n)	
gethostbyaddr_r(3n)	getrpcent_r(3n)	

Async-Signal-Safe Functions

Functions that can safely be called from signal handlers are *Async-Signal-Safe*. The POSIX standard defines and lists Async-Signal-Safe functions (IEEE Std 1003.1-1990, 3.3.1.3 (3)(f), page 55). In addition to the POSIX Async-Signal-Safe functions, these three functions from the Solaris threads library are also Async- Signal-Safe.

- sema_post(3T)
- thr_sigsetmask(3T), similar to pthread_sigmask(3T)

- thr_kill(3T), similar to pthread_kill(3T)

MT Safety Levels for Libraries

All routines that can potentially be called by a thread from a multithreaded program should be MT-Safe.

This means that two or more activations of a routine must be able to *correctly* execute concurrently. So, every library interface that a multithreaded program uses must be MT-Safe.

Not all libraries are now MT-Safe. The commonly used libraries that are MT-Safe are listed in Table 6–2. Additional libraries will eventually be modified to be MT-Safe.

TABLE 6–2 Some MT-Safe Libraries

Library	**Comments**
`lib/libc`	Interfaces that are not safe have thread-safe interfaces of the form `*_r` (often with different semantics)
`lib/libdl_stubs`	To support static switch compiling
`lib/libintl`	Internationalization library
`lib/libm`	Math library compliant with System V Interface Definition, Edition 3, X/Open and ANSI C
`lib/libmalloc`	Space-efficient memory allocation library; see malloc(3X)
`lib/libmapmalloc`	Alternative `mmap(2)`-based memory allocation library; see `mapmalloc(3X)`
`lib/libnsl`	The TLI interface, XDR, RPC clients and servers, `netdir`, `netselect` and `getXXbyYY` interfaces are not safe, but have thread-safe interfaces of the form `getXXbyYY_r`
`lib/libresolv`	Thread-specific errno support
`lib/libsocket`	Socket library for making network connections
`lib/libw`	Wide character and wide string functions for supporting multibyte locales

TABLE 6–2 Some MT-Safe Libraries *(continued)*

Library	Comments
`lib/straddr`	Network name-to-address translation library
`lib/libX11`	X11 Windows library routines
`lib/libC`	C++ runtime shared objects

Unsafe Libraries

Routines in libraries that are not guaranteed to be MT-Safe can safely be called by multithreaded programs only when such calls are single-threaded.

CHAPTER **7**

Compiling and Debugging

This chapter describes how to compile and debug multithreaded programs.

- "Compiling a Multithreaded Application " on page 129
- "Debugging a Multithreaded Program" on page 133

Compiling a Multithreaded Application

There are many options to consider for header files, define flags, and linking.

Preparing for Compilation

The following items are required to compile and link a multithreaded program. Except for the C compiler, all should come with your Solaris 2.x system.

- A standard C compiler
- Include files:
  - `<thread.h>` and `<pthread.h>`
  - `<errno.h>`, `<limits.h>`, `<signal.h>`, `<unistd.h>`
- The regular Solaris linker, ln(1)
- The Solaris threads library (`libthread`), the POSIX threads library (`libpthread`), and possibly the POSIX realtime library (`libposix4`) for semaphores
- MT-safe libraries (`libc`, `libm`, `libw`, `libintl`, `libnsl`, `libsocket`, `libmalloc`, `libmapmalloc`, and so on)

Choosing Solaris or POSIX Semantics

Certain functions, including the ones listed below, have different semantics in the POSIX 1003.1c standard than in the Solaris 2.4 release, which was based on an earlier POSIX draft. Function definitions are chosen at compile time. See the *man Pages(3): Library Routines* for a description of the differences in expected parameters and return values.

TABLE 7–1 Functions with POSIX/Solaris Semantic Differences

`sigwait(2)`	
`ctime_r(3C)`	`asctime_r(3C)`
`ftrylockfile(3S)` - new	`getlogin_r(3C)`
`getgrnam_r(3C)`	`getgrgid_r(3C)`
`getpwnam_r(3C)`	`getpwuid_r(3C)`
`readdir_r(3C)`	`ttyname_r(3C)`

The Solaris fork(2) function duplicates all threads (*fork-all* behavior), while the POSIX fork(2) function duplicates only the calling thread (*fork-one* behavior), as does the Solaris `fork1( )` function.

The handling of an alarm(2) is also different: a Solaris alarm goes to the thread's LWP, while a POSIX alarm goes to the whole process (see "Per-Thread Alarms" on page 103;).

Including `<thread.h>` or `<pthread.h>`

The include file `<thread.h>`, used with the `-lthread` library, compiles code that is upward compatible with earlier releases of the Solaris system. This library contains both interfaces—those with Solaris semantics and those with POSIX semantics. To call `thr_setconcurrency(3T)` with POSIX threads, your program needs to include `<thread.h>`.

The include file `<pthread.h>`, used with the `-lpthread` library, compiles code that is conformant with the multithreading interfaces defined by the POSIX 1003.1c standard. For complete POSIX compliance, the define flag `_POSIX_C_SOURCE` should be set to a (`long`) value ≥ `199506`:

```
cc [flags] file... -D_POSIX_C_SOURCE=N  (where N  199506L)
```

You can mix Solaris threads and POSIX threads in the same application, by including both `<thread.h>` and `<pthread.h>`, and linking with either the `-lthread` or `-lpthread` library.

In mixed use, Solaris semantics prevail when compiling with `–D_REENTRANT` and linking with `-lthread`, whereas POSIX semantics prevail when compiling with `–D_POSIX_C_SOURCE` and linking with `-lpthread`.

Defining _REENTRANT or _POSIX_C_SOURCE

For POSIX behavior, compile applications with the `–D_POSIX_C_SOURCE` flag set ≥ `199506L`. For Solaris behavior, compile multithreaded programs with the `–D_REENTRANT` flag. This applies to every module of an application.

For mixed applications (for example, Solaris threads with POSIX semantics), compile with the `–D_REENTRANT` and `–D_POSIX_PTHREAD_SEMANTICS` flags.

To compile a single-threaded application, define neither the `–D_REENTRANT` nor the `–D_POSIX_C_SOURCE` flag. When these flags are not present, all the old definitions for `errno`, `stdio`, and so on, remain in effect.

To summarize, POSIX applications that define `-D_POSIX_C_SOURCE` get the POSIX 1003.1c semantics for the routines listed in Table 7–1. Applications that define only `<–D_REENTRANT` get the Solaris semantics for these routines. Solaris applications that define `–D_POSIX_PTHREAD_SEMANTICS` get the POSIX semantics for these routines, but can still use the Solaris threads interface.

Linking With libthread or libpthread

For POSIX threads behavior, load the `libpthread` library. For Solaris threads behavior, load the `libthread` library. Some POSIX programmers might want to link with `–lthread` to preserve the Solaris distinction between `fork()` and `fork1()`. All that `-lpthread` really does is to make `fork()` behave the same way as the Solaris `fork1()` call, and change the behavior of `alarm(2)`.

To use `libthread`, specify `–lthread` before `–lc` on the `ld` command line, or last on the `cc` command line.

To use `libpthread`, specify `–lpthread` before `–lc` on the `ld` command line, or last on the `cc` command line.

Do not link a nonthreaded program with `–lthread` or `–lpthread`. Doing so establishes multithreading mechanisms at link time that are initiated at run time. These slow down a single-threaded application, waste system resources, and produce misleading results when you debug your code.

This diagram summarizes the compile options:

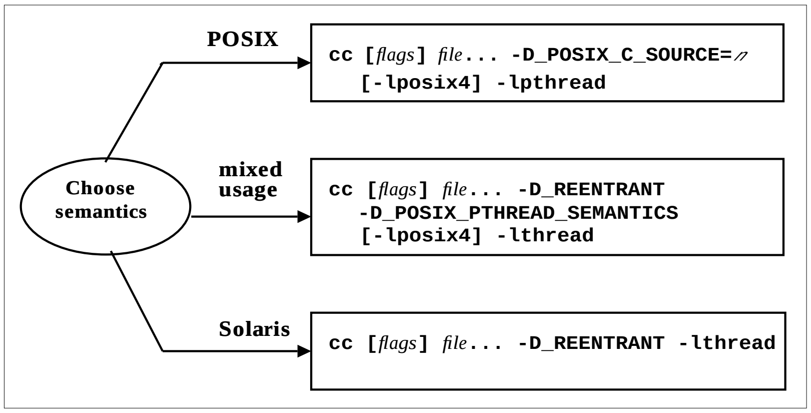

POSIX
cc [*flags*] *file*... -D_POSIX_C_SOURCE=199506L
[-lposix4] -lpthread
Choose semantics
mixed usage
cc [*flags*] *file*... -D_REENTRANT
-D_POSIX_PTHREAD_SEMANTICS
[-lposix4] -lthread
Solaris
cc [*flags*] *file*... -D_REENTRANT -lthread

Figure 7–1 Compilation Flowchart

In mixed usage, you need to include both `thread.h` and `pthread.h`.

All calls to `libthread` and `libpthread` are no-ops if the application does not link `–lthread` or `–lpthread`. The runtime library `libc` has many predefined `libthread` and `libpthread` stubs that are null procedures. True procedures are interposed by `libthread` or `libpthread` when the application links both `libc` and the thread library.

The behavior of the C library is undefined if a program is constructed with an `ld` command line that includes the following *incorrect* fragment:

```
.o’s ... -lc -lthread ... (this is incorrect)
or
 .o’s ... -lc -lpthread ... (this is incorrect)
```

Note - For C++ programs that use threads, use the `–mt` option, rather than `–lthread`, to compile and link your application. The `–mt` option links with `libthread` and ensures proper library linking order. Using `–lthread` might cause your program to core dump.

Linking with -lposix4 for POSIX Semaphores

The Solaris semaphore routines, `sema_*(3T)`, are contained in the `libthread` library. By contrast, you link with the `-lposix4` library to get the standard `sem_*(3R)` POSIX 1003.1c semaphore routines described in “ Semaphores ” on page 86.;

Link Old With New

Table 7–2 shows that multithreaded object modules should be linked with old object modules only with great caution.

TABLE 7–2 Compiling With and Without the `_REENTRANT` Flag

The File Type	Compiled	Reference	And Return
Old object files (non-threaded) and new object files	*Without* the `_REENTRANT` or `_POSIX_C_SOURCE` flag	Static storage	The traditional *errno*
New object files	*With* the `_REENTRANT` or `_POSIX_C_SOURCE` flag	`__errno`, the new binary entry point	The address of the thread's definition of `errno`
Programs using TLI in libnsl[1]	*With* the `_REENTRANT` or `_POSIX_C_SOURCE` flag (required)	`__t_errno`, a new entry point	The address of the thread's definition of `t_errno`.

1. Include `tiuser.h` to get the TLI global error variable.

Debugging a Multithreaded Program

Common Oversights

The following list points out some of the more frequent oversights that can cause bugs in multithreaded programs.

- Passing a pointer to the caller's stack as an argument to a new thread.
- Accessing global memory (shared changeable state) without the protection of a synchronization mechanism.
- Creating deadlocks caused by two threads trying to acquire rights to the same pair of global resources in alternate order (so that one thread controls the first resource and the other controls the second resource and neither can proceed until the other gives up).
- Trying to reacquire a lock already held (recursive deadlock).
- Creating a hidden gap in synchronization protection. This is caused when a code segment protected by a synchronization mechanism contains a call to a function that frees and then reacquires the synchronization mechanism before it returns to

the caller. The result is that it appears to the caller that the global data has been protected when it actually has not.

- Mixing UNIX signals with threads—it is better to use the `sigwait(2)` model for handling asynchronous signals.
- Using `setjmp(3B)` and `longjmp(3B)`, and then long-jumping away without releasing the mutex locks.
- Failing to reevaluate the conditions after returning from a call to `*_cond_wait(3T)` or `*_cond_timedwait(3T)`.
- Forgetting that default threads are created `PTHREAD_CREATE_JOINABLE` and must be reclaimed with `pthread_join(3T)`; note, `pthread_exit(3T)` does not free up its storage space.
- Making deeply nested, recursive calls and using large automatic arrays can cause problems because multithreaded programs have a more limited stack size than single-threaded programs.
- Specifying an inadequate stack size, or using non-default stacks.

And, note that multithreaded programs (especially those containing bugs) often behave differently in two successive runs, given identical inputs, because of differences in the thread scheduling order.

In general, multithreading bugs are statistical instead of deterministic. Tracing is usually a more effective method of finding order of execution problems than is breakpoint-based debugging.

Tracing and Debugging With the TNF Utilities

Use the TNF utilities (included as part of the Solaris system) to trace, debug, and gather performance analysis information from your applications and libraries. The TNF utilities integrate trace information from the kernel and from multiple user processes and threads, and so are especially useful for multithreaded code.

With the TNF utilities, you can easily trace and debug multithreaded programs. See the TNF utilities chapter in the *Programming Utilities Guide* for detailed information on using `prex(1)`, `tnfdump(1)`, and other TNF utilities.

Using truss(1)

See `truss(1)` in the *man Pages(1): User Commands* for information on tracing system calls and signals.

Using adb(1)

When you bind all threads in a multithreaded program, a thread and an LWP are synonymous. Then you can access each thread with the following `adb` commands that support multithreaded programming.

TABLE 7–3 MT `adb` Commands

`pid``:A`	Attaches to process # *pid*. This stops the process and all its LWPs.
`:R`	Detaches from process. This resumes the process and all its LWPs.
`$L`	Lists all active LWPs in the (stopped) process.
`n``:l`	Switches focus to LWP # *n*.
`$l`	Shows the LWP currently focused.
`num``:i`	Ignores signal number *num*.

These commands to set conditional breakpoints are often useful.

TABLE 7–4 Setting `adb` Breakpoints

`[`*label***`],[`***count***`]:b [`***expression***`]**	Breakpoint is detected when *expression* equals zero
foo,**`ffff:b <g7-0xabcdef`**	Stop at *foo* when *g7* = the hex value `0xABCDEF`

Using dbx

With the `dbx` utility you can debug and execute source programs written in C++, ANSI C, FORTRAN, and Pascal. `dbx` accepts the same commands as the SPARCworks™ Debugger but uses a standard terminal (TTY) interface. Both `dbx` and the Debugger support debugging multithreaded programs. For a full overview of `dbx` and Debugger features see the SunSoft Developer Products (formerly SunPro) `dbx(1)` reference manual page and the *Debugging a Program* user's guide.

All the `dbx` options listed in Table 7–5 can support multithreaded applications.

TABLE 7–5 `dbx` Options for MT Programs

Option	Meaning
`cont at line [sig signo id]`	Continues execution at *line* with signal *signo*. The *id*, if present, specifies which thread or LWP to continue. The default value is *all*.
`lwp`	Displays current LWP. Switches to given LWP [lwpid].
`lwps`	Lists all LWPs in the current process.
`next ... tid`	Steps the given thread. When a function call is skipped, all LWPs are implicitly resumed for the duration of that function call. Nonactive threads cannot be stepped.
`next ... lid`	Steps the given LWP. Does not implicitly resume all LWPs when skipping a function. The LWP on which the given thread is active. Does not implicitly resume all LWP when skipping a function.
`step... tid`	Steps the given thread. When a function call is skipped, all LWPs are implicitly resumed for the duration of that function call. Nonactive threads cannot be stepped.
`step... lid`	Steps the given LWP. Does not implicitly resume all LWPs when skipping a function.
`stepi... lid`	The given LWP.
`stepi... tid`	The LWP on which the given thread is active.
`thread`	Displays current thread. Switches to thread *tid*. In all the following variations, an optional tid implies the current thread.
`thread -info [ tid ]`	Prints everything known about the given thread.
`thread -locks [ tid ]`	Prints all locks held by the given thread.
`thread -suspend [ tid ]`	Puts the given thread into suspended state.
`thread -continue [ tid ]`	Unsuspends the given thread.

TABLE 7–5 `dbx` Options for MT Programs *(continued)*

`thread -hide [ tid ]`	*Hides* the given (or current) thread. It will not appear in the generic `threads` listing.
`thread -unhide [ tid ]`	*Unhides* the given (or current) thread.
`allthread-unhide`	*Unhides* all threads.
`threads`	Prints the list of all known threads.
`threads-all`	Prints threads that are not usually printed (zombies).
`all|filterthreads-mode`	Controls whether `threads` prints all threads or filters them by default.
`auto|manualthreads-mode`	Enables automatic updating of the thread listing in the SPARCworks Debugger.
`threads-mode`	Echoes the current modes. Any of the previous forms can be followed by a thread or LWP ID to get the traceback for the specified entity.

CHAPTER 8

Tools for Enhancing MT Programs

Sun provides several tools for enhancing the performance of MT programs. This chapter describes three of them.

Thread Analyzer

Thread Analyzer displays standard profiling information for each thread in your program. Additionally, Thread Analyzer displays metrics specific to a particular thread (such as Mutex Wait Time and Semaphore Wait Time). Thread Analyzer can be used with C, C++, and FORTRAN 77 programs.

LockLint

LockLint verifies the consistent use of mutex and readers/writer locks in multithreaded ANSI C programs.

LockLint performs a static analysis of the use of mutex and readers/writer locks, and looks for inconsistent use of these locking techniques. In looking for inconsistent use of locks, LockLint detects the most common causes of data races and deadlocks.

LoopTool

LoopTool, along with its companion program LoopReport, profiles loops for FORTRAN programs; it provides information about programs parallelized by SPARCompiler FORTRAN MP. LoopTool displays a graph of loop runtimes, shows which loops were parallelized, and provides compiler hints as to why a loop was not parallelized.

LoopReport creates a summary table of all loop runtimes correlated with compiler hints about why a loop was not parallelized.

This chapter presents scenarios showing how each tool is used:

- Scenario One looks at Mandelbrot, a C program that can be made to run much faster by making it multithreaded. The discussion analyzes the program with Thread Analyzer to see where performance bottlenecks take place, then threads it accordingly.

- Scenario Two (“Scenario: Checking a Program With LockLint” on page 145) shows the use of LockLint to check the Mandelbrot program’s use of locks.
- Scenario Three (“Scenario: Parallelizing Loops with LoopTool” on page 148) shows the use of LoopTool to parallelize portions of a library.

Scenario: Threading the Mandelbrot Program

This scenario shows

1. Threading a program to achieve better performance.
2. Examining the program with Thread Analyzer to determine why it hasn’t shown optimal speed-up.
3. Re-writing the program to take better advantage of threading.

Mandelbrot is a well-known program that plots vectors on the plane of complex numbers, producing an interesting pattern on the screen.

In the simplest, nonthreaded version of Mandelbrot, the program flow simply repeats this series:

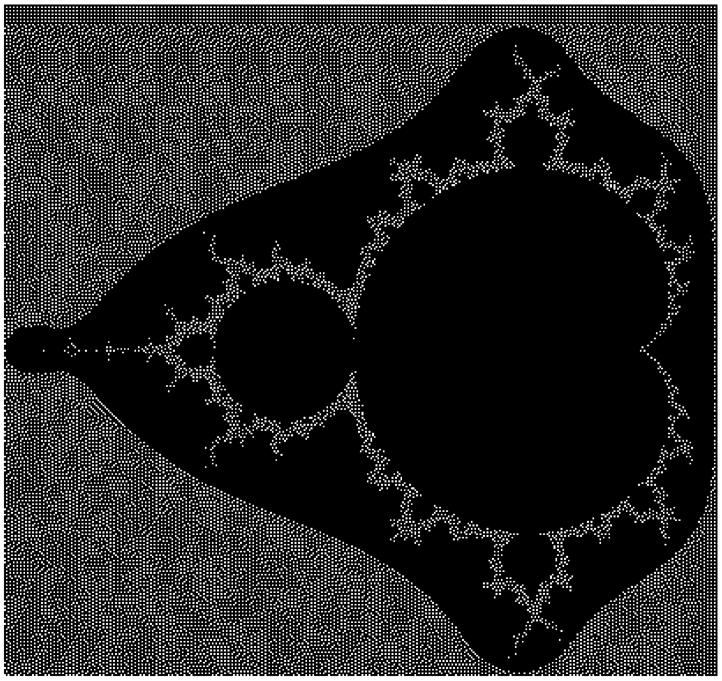
- Calculate each point.
- Display each point.

Obviously, on a multiprocessor machine this is not the most efficient way to run the program. Since each point can be calculated independently, the program is a good candidate for parallelization.

The program can be threaded to make it more efficient. In the threaded version, several threads (one for each processor) are running simultaneously. Each thread calculates and displays a row of points independently.

Thread One	**Thread Two**
Calculate row (vector 1)	Calculate row (vector 2)
Display row (vector 1)	Display row (vector 2)

However, even though the threaded Mandelbrot is faster than the unthreaded version, it doesn’t show the performance speedup that might be expected.



Using Thread Analyzer to Evaluate Mandelbrot

The Thread Analyzer is used to see where the performance bottlenecks are occurring. In our example, we chose to check which procedures were waiting on locks.

In our example, after recompiling the program to instrument it for Thread Analyzer, we displayed the main window. The main window shows the program's threads and the procedures they call.

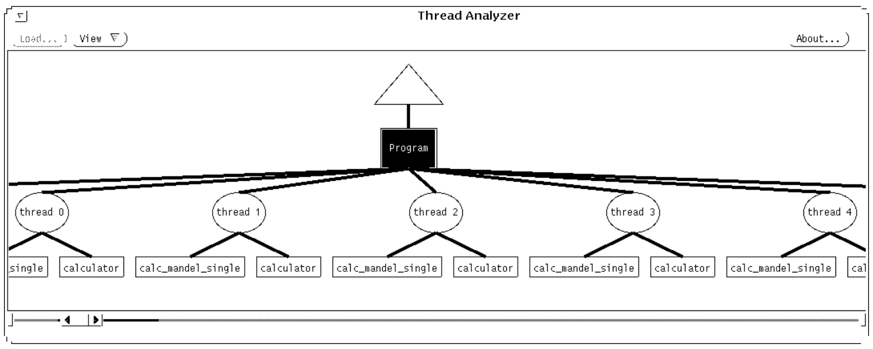


Figure 8–1 Thread Analyzer Main Window (partial)

Thread Analyzer allows you to view the program in many ways, including those listed in Table 8–1:

TABLE 8–1 Thread Analyzer Views

View	Meaning
Graph	Plot the value of selected metrics against wallclock time.
`gprof(1)` Table	Display call-graph profile data for threads and functions.
`prof(1)` Table	Display profile data for program, threads, and functions.
Sorted Metric Profile Table	Display a metric for a particular aspect of the program.
Metric Table	Show multiple metrics for a particular thread or function.
Filter Threads by CPU	Display the threads whose percent of CPU is equal to or above a designated threshold.
Filter Functions by CPU	Display the functions whose percent of CPU is equal to or above a designated threshold.

To look at wallclock and CPU times, choose the Graph view, and select CPU, Wallclock time, and Mutex Wait metrics. Figure 8–2 displays the Graph view of the wallclock and CPU times:

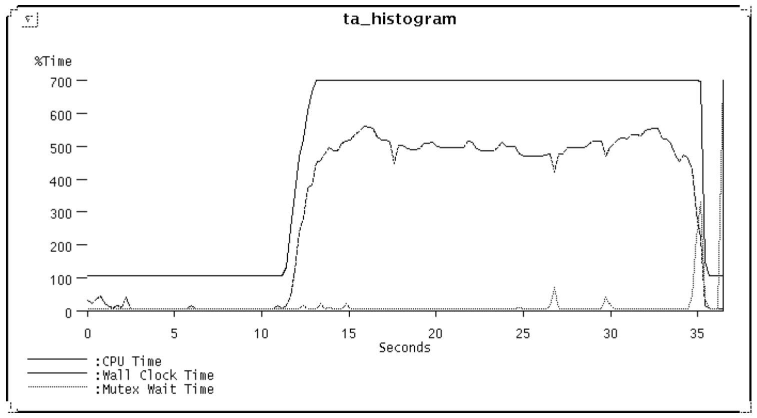
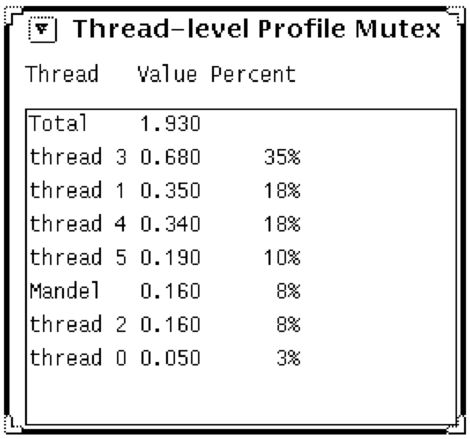


Figure 8–2 Thread Analyzer: Wallclock and CPU Time

According to this graph, CPU time is consistently below wallclock time. This indicates that fewer threads than were allocated are being used, because some threads are blocked (that is, contending for resources).

Look at mutex wait times to see which threads are blocked. To do this, you can select a thread node from the main window, and then Mutex Wait from the Sorted Metrics menu. The table in Figure 8–3 displays the amount of time each thread spent waiting on mutexes:



```
Thread    Value Percent

Total     1.930
thread 3  0.680     35%
thread 1  0.350     18%
thread 4  0.340     18%
thread 5  0.190     10%
Mandel    0.160      8%
thread 2  0.160      8%
thread 0  0.050      3%
```

Figure 8–3 Thread Analyzer: Mutex Wait Time

The various threads spend a lot of time waiting for each other to release locks. (In this example, Thread 3 waits so much more than the others because of randomness.) Because the display is a serial resource—a thread cannot display until another thread has finished displaying—the threads are probably waiting for other threads to give up the display lock.

Figure 8–4 shows what's happening.

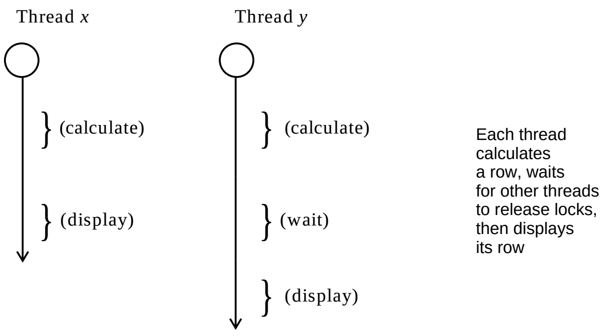

Thread *x*
Thread *y*
(calculate)
(display)
(calculate)
(wait)
(display)
Each thread calculates a row, waits for other threads to release locks, then displays its row

Figure 8–4 Mandelbrot Multithreaded: Each Thread Calculates and Displays

To speed things up, rewrite the code so that the calculations and the display are entirely separate. Figure 8–5 shows how the rewritten code uses several threads simultaneously to calculate rows of points and write th results into a buffer, while another thread reads from the buffer and displays rows:

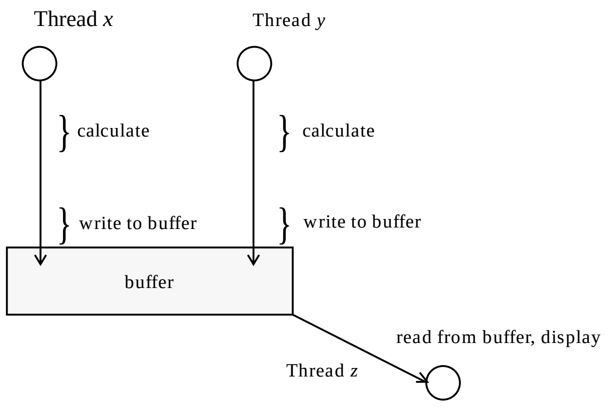
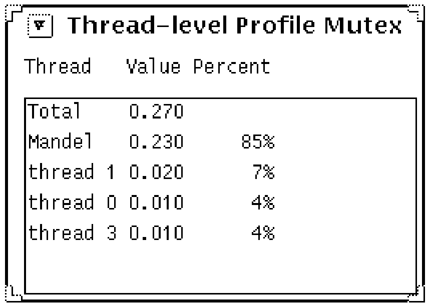


Figure 8–5 Mandelbrot Threaded (Separate Display Thread)

Now, instead of the display procedure of each thread waiting for another thread to calculate and display, only the display thread waits (for the current line of the buffer to be filled). While it waits, other threads are calculating and writing, so that there is little time spent waiting for the display lock.

Display the mutex wait times again to see the amount of time spent waiting on a mutex:



```
Thread   Value Percent

Total    0.270
Mandel   0.230    85%
thread 1 0.020     7%
thread 0 0.010     4%
thread 3 0.010     4%
```

Figure 8–6 Thread Analyzer: Mutex Wait Time (Separate Display Thread)

The program spends almost all of its time in the main loop (Mandel), and the time spent waiting for locks is reduced significantly. In addition, Mandelbrot runs noticeably faster.

Scenario: Checking a Program With LockLint

A program can run efficiently but still contain potential problems. One such problem occurs when two threads try to access the same data simultaneously. This can lead to:

- Deadlocks — when two threads are mutually waiting for the other to release a lock.
- Data races — when two or more threads have overlapping read / write access to data, causing unexpected data values. For example, suppose Thread A writes the variable *calc*, goes off and does something else, and then comes back to read *calc*; in the meantime Thread B writes to *calc* and changes its value to something Thread A does not “expect.”

Here’s how you can use LockLint to see if data is adequately protected.

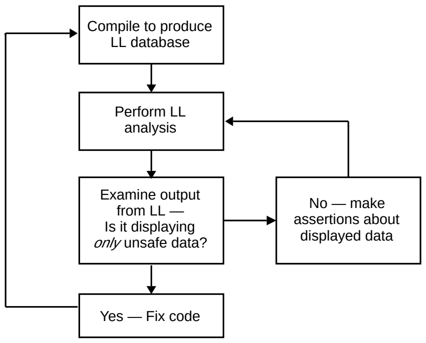

Compile to produce LL database
Perform LL analysis
Examine output from LL — Is it displaying *only* unsafe data?
No — make assertions about displayed data
Yes — Fix code

Figure 8–7 The LockLint Usage Flowchart

1. **Compile the program with LockLint instrumentation.**

   The compiler has an option to produce a version of the program that LockLint can use for analysis.

2. **Create a LockLint shell and load the instrumented program.**

   You can use this shell as you would any other, including running scripts.

3. **Save the executable’s state.**

   LockLint is designed to run iteratively. You run it over and over, making progressively stronger assertions about the data it is analyzing, until you find a problem or are satisfied that the data is safe.

Note - Analyzing the program with LockLint changes its state; that is, once you've done an analysis, you can't add further assertions. By saving and restoring the state, you can run the analysis over and over, with different assertions about the program's data.

---

4. **Analyze the program.**

   The `analyze` command performs consistency checks on the program's data.

5. **Search for unsafe data.**

   Having run the analysis, you can look for unprotected elements.

   The `held=[]` indicates that variables did not have locks consistently held on them while they were accessed. An asterisk preceding the `held=[]` indicates that these variables were written to. An asterisk, therefore, means that LockLint "believes" the data is not safe.

CODE EXAMPLE 8–1 Fragment of Initial LockLint Output

```
$ lock_lint analyze
   $ lock_lint vars -h | grep held
   :arrow_cursor        *held={ }
   :bottom_row        *held={ }
   :box_height        *held={ }
   :box_width        *held={ }
   :box_x        *held={ }
   :busy_cursor   *held={ }
   :c_text *held={ }
   :calc_mandel    *held={ }
   :calc_type  *held={ }
   :canvas *held={ }
   :canvas_proc/drag   *held={ }
   :canvas_proc/x  *held={ }
   [. . . ]
   :gap        *held={ }
   :gc        *held={ }
   :next_row        *held={ }
   :now.tv_sec        held={ }
   :now.tv_usec        held={ }
   :p_text        *held={ }
   :panel        *held={ }
   :picture_cols        *held={ }
   :picture_id        *held={ }
   :picture_rows        *held={ }
   :picture_state        *held={ }
   :pw        *held={ }
   :ramp.blue        *held={ }
   :ramp.green        *held={ }
   :ramp.red        *held={ }
   :rectangle_selected *held={ }
   :row_inc        *held={ }
   :run_button        *held={ }
   [ . . . ]
```

However, this analysis is limited in its usefulness because many of the variables displayed do not *need* to be protected (such as variables that are not written to, except when they're initialized). By excluding some data from consideration, and having LockLint repeat its analyses, you can narrow your search to the unprotected variables that you are interested in.

6. **Restore the program to its saved state.**

   To be able to run the analysis again, pop the state back to what it was before the program was last analyzed.

7. **Refine the analysis by excluding some data.**

   For example, you can ignore variables that aren't written to—since they don't change, they won't cause data races. And you can ignore places where the variables are initialized (if they're not visible to other threads).

   You can ignore the variables that you know are safe by making *assertions* about them. In the code example below, the following is done:

   - Initialization functions are ignored because no data is overwritten at initialization.
   - Some variables are asserted to be read-only.

   For illustration, this is done on the command line in verbose mode. After you become familiar with the command syntax, you can use aliases and shell scripts to make the task easier.

```
$ lock_lint ignore CreateXStuff run_proc canvas_proc main
   $ lock_lint assert read only bottom_row
   $ lock_lint assert read only calc_mandel
   etc.
```

8. **Analyze the program again, and search for unsafe data.**

   The list of unsafe data is considerably reduced.

CODE EXAMPLE 8–2 Unsafe Data Reported by LockLint

```
$ lock_lint vars -h | grep held
   :bottom_row        held={ }
   :calc_mandel        held={ }
   :colors        held={ }
   :corner_i        held={ }
   :corner_r        held={ }
   :display        held={ }
   :drawable        held={ }
   :frame        held={ }
   :gap        held={ }
   :gc        held={ }
   :next_row        held={
   mandel_display.c:next_row_lock }
   :picture_cols        held={ }
   :picture_id        held={ }
```

```
:picture_rows        *held={ }
:picture_state        *held={ }
:row_inc        held={ }
```

Only two variables were written to (*picture_rows* and *picture_state*) and are flagged by `LockLint` as inconsistently protected.

The analysis also flags the variable *next_row*, which the calculator threads use to find the next chunk of work to be done. However, as the analysis states, this variable is consistently protected.

Alter your source code to properly protect *picture_rows* and *picture_state*.

Scenario: Parallelizing Loops with LoopTool

IMSL™ is a popular math library used by many FORTRAN and C programmers.[1] One of its routines is a good candidate for parallelizing with LoopTool.

This example is a FORTRAN program called `l2trg.f()`. (It computes LU factorization of a single-precision general matrix.) The program is compiled without any parallelization, then checked to see how long it takes to run with the time(1) command.

CODE EXAMPLE 8–3 Original Times for `l2trg.f()` (Not Parallelized)

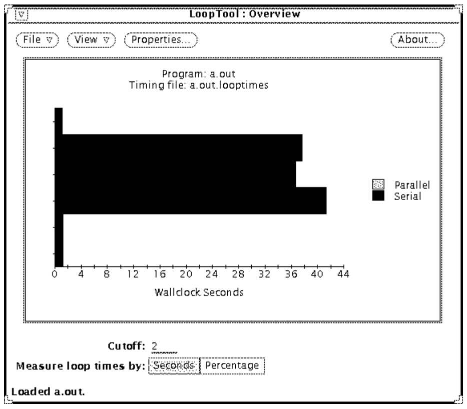
```
$ f77 l2trg.f -cg92 -03 -lmsl
$ /bin/time a.out
real   44.8
user   43.5
sys   1.0
```

To look at the program with LoopTool, recompile with the LoopTool instrumentation, using the –`Zlp` option.

```
$ f77 l2trg.f -cg92 -03 -Zlp -lmsl
```

Start LoopTool. Figure 8–8shows the initial Overview screen.

1. IMSL is a registered trademark of IMSL, Inc. This example is used with permission.



Cutoff: 2
Measure loop times by: [✓ Seconds] Percentage
Loaded a.out.

Figure 8–8 LoopTool View Before Parallelization

Most of the program's time is spent in three loops; each loop indicated by a horizontal bar.

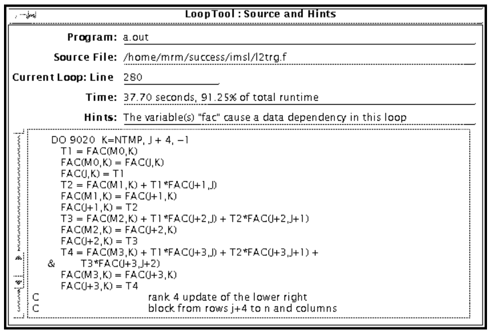
The LoopTool user interface brings up various screens triggerred by cursor movement and mouse actions. In the Overview window:

Put the cursor over a loop to get its line number.

Click on the loop to bring up a window that displays the loop's source code.

In our example, we clicked on the middle horizontal bar to look at the source code for the middle loop. The source code reveals that loops are nested.

Figure 8–9 shows the Source and Hints window for the middle loop.



Program: a.out
Source File: /home/mrm/success/imsl/l2trg.f
Current Loop: Line 280
Time: 37.70 seconds, 91.25% of total runtime
Hints: The variable(s) "fac" cause a data dependency in this loop

```
      DO 9020  K=NTMP, J + 4, -1
         T1 = FAC(M0,K)
         FAC(M0,K) = FAC(J,K)
         FAC(J,K) = T1
         T2 = FAC(M1,K) + T1*FAC(J+1,J)
         FAC(M1,K) = FAC(J+1,K)
         FAC(J+1,K) = T2
         T3 = FAC(M2,K) + T1*FAC(J+2,J) + T2*FAC(J+2,J+1)
         FAC(M2,K) = FAC(J+2,K)
         FAC(J+2,K) = T3
         T4 = FAC(M3,K) + T1*FAC(J+3,J) + T2*FAC(J+3,J+1) +
     &        T3*FAC(J+3,J+2)
         FAC(M3,K) = FAC(J+3,K)
         FAC(J+3,K) = T4
C                         rank 4 update of the lower right
C                         block from rows j+4 to n and columns
```

Figure 8–9 LoopTool (Source and Hints Window)

In this case, LoopTool gives the Hints message:

```
    The variable ''fac'' causes a data dependency in this loop
```

In the source code, you can see that *fac* is calculated in the nested, innermost loop (9030):

```
C                         update the remaining rectangular
C                         block of U, rows j to j+3 and
```

```
C                        columns j+4 to n

      DO 9020  K=NTMP, J + 4, -1
         T1 = FAC(M0,K)
         FAC(M0,K) = FAC(J,K)
         FAC(J,K) = T1
         T2 = FAC(M1,K) + T1*FAC(J+1,J)
         FAC(M1,K) = FAC(J+1,K)
         FAC(J+1,K) = T2
         T3 = FAC(M2,K) + T1*FAC(J+2,J) + T2*FAC(J+2,J+1)
         FAC(M2,K) = FAC(J+2,K)
         FAC(J+2,K) = T3
         T4 = FAC(M3,K) + T1*FAC(J+3,J) + T2*FAC(J+3,J+1) +
     &        T3*FAC(J+3,J+2)
         FAC(M3,K) = FAC(J+3,K)
         FAC(J+3,K) = T4
C                        rank 4 update of the lower right
C                        block from rows j+4 to n and columns
C                        j+4 to n
         DO 9030  I=KBEG, NTMP
            FAC(I,K) = FAC(I,K) + T1*FAC(I,J) + T2*FAC(I,J+1) +
     &                 T3*FAC(I,J+2) + T4*FAC(I,J+3)
 9030    CONTINUE
 9020 CONTINUE
```

The loop index, *I*, of the innermost loop is used to access rows of the array *fac*. So the innermost loop updates the *I*th row of *fac*. Since updating these rows does not depend on updates of any other rows of *fac*, it's safe to parallelize this loop.

The calculation of *fac* is speeded up by parallelizing loop 9030, so there should be a significant performance improvement. Force explicit parallelization by inserting a DOALL directive in front of loop 9030:

```
C$PAR DOALL
(Add DOALL directive here)
        DO 9030  I=KBEG, NTMP
            FAC(I,K) = FAC(I,K) + T1*FAC(I,J) + T2*FAC(I,J+1) +
     &                 T3*FAC(I,J+2) + T4*FAC(I,J+3)
 9030    CONTINUE
```

Now you can recompile the `FORTRAN` code, run the program, and compare the new time with the original times. More specifically, Code Example 8–4 shows the use of all the processors on the machine by setting the `PARALLEL` environment variable equal to 2, and forces explicit parallelization of that loop with the `–explicitpar` compiler option.

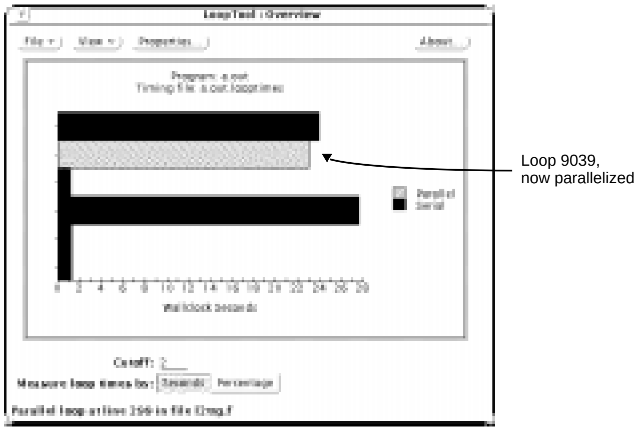
Finally, run the program and compare its time with that of the original times (shown in Code Example 8–3).

CODE EXAMPLE 8–4 Post-Parallelization Times for `l2trg.f()`

```
$ setenv PARALLEL 2
(2 is the # of processors on the machine)
$ f77 l2trg.f -cg92 -03 -explicitpar -imsl
$ /bin/time a.out
```

```
real   28.4
user   53.8
sys   1.1
```

The program now runs over a third faster. (The higher number for *user* reflects the fact that there are now two processes running.) Figure 8–10 shows the `LoopTool` Overview window. You see that, in fact, the innermost loop is now parallel.



Parallel loop at line 259 in file l2mg.f

Figure 8–10 `LoopTool` View After Parallelization

For More Information

To find out more about Solaris threads and related issues on the World Wide Web (WWW) see the following URL:

`http://www.sun.com/sunsoft/Products/Developer-products/sig/threads`

Also, the following manuals more information about multithreaded tools:

Thread Analyzer User's Guide 801-6691-10

LockLint User's Guide 801-6692-10

LoopTool User's Guide 801-6693-10

CHAPTER **9**

Programming with Solaris Threads

This chapter compares the Application Program Interface (API) for Solaris and POSIX threads, and explains the Solaris features that are not found in POSIX threads.

Comparing APIs for Solaris Threads and POSIX Threads

The Solaris threads API and the pthreads API are two solutions to the same problem: building parallelism into application software. Although each API is complete in itself, you can safely mix Solaris threads functions and pthread functions in the same program.

The two APIs do not match exactly, however. Solaris threads supports functions that are not found in pthreads, and pthreads includes functions that are not supported in the Solaris interface. For those functions that *do* match, the associated arguments might not, although the information content is effectively the same.

By combining the two APIs, you can use features not found in one to enhance the other. Similarly, you can run applications using Solaris threads, exclusively, with applications using pthreads, exclusively, on the same system.

Major API Differences

Solaris threads and pthreads are very similar in both API action and syntax. The major differences are listed in Table 9–1.

TABLE 9–1 Unique Solaris Threads and pthreads Features

Solaris Threads (libthread)	**POSIX Threads (libpthread)**
`thr_` prefix for threads function names; `sema_` prefix for semaphore function names	`pthread_` prefix for pthreads function names; `sem_` prefix for semaphore function names
Readers/Writer locks	Attribute objects (these replace many Solaris arguments or flags with pointers to pthreads attribute objects)
Ability to create “daemon” threads	Cancellation semantics
Suspending and continuing a thread	Scheduling policies
Setting concurrency (requesting a new LWP): determining concurrency level	

Function Comparison Table

The following table compares Solaris threads functions with pthreads functions. Note that even when Solaris threads and pthreads functions appear to be essentially the same, the arguments to the functions can differ.

When a comparable interface is not available either in pthreads or Solaris threads, a hyphen ‘-’ appears in the column. Entries in the pthreads column that are followed by “POSIX 1003.4” or “POSIX.4” are part of the POSIX Realtime standard specification and are not part of pthreads.

TABLE 9–2 Solaris Threads and POSIX pthreads Comparison

Solaris Threads (libthread)	pthreads (libpthread)
`thr_create()`	`pthread_create()`
`thr_exit()`	`pthread_exit()`
`thr_join()`	`pthread_join()`
`thr_yield()`	`sched_yield()` POSIX.4
`thr_self()`	`pthread_self()`
`thr_kill()`	`pthread_kill()`
`thr_sigsetmask()`	`pthread_sigmask()`
`thr_setprio()`	`pthread_setschedparam()`
`thr_getprio()`	`pthread_getschedparam()`
`thr_setconcurrency()`	-
`thr_getconcurrency()`	-
`thr_suspend()`	-
`thr_continue()`	-
`thr_keycreate()`	`pthread_key_create()`
-	`pthread_key_delete()`
`thr_setspecific()`	`pthread_setspecific()`
`thr_getspecific()`	`pthread_getspecific()`

TABLE 9–2 Solaris Threads and POSIX pthreads Comparison *(continued)*

Solaris Threads (libthread)	pthreads (libpthread)
-	`pthread_once()`
-	`pthread_equal()`
-	`pthread_cancel()`
-	`pthread_testcancel()`
-	`pthread_cleanup_push()`
-	`pthread_cleanup_pop()`
-	`pthread_setcanceltype()`
-	`pthread_setcancelstate()`
`mutex_lock()`	`pthread_mutex_lock()`
`mutex_unlock()`	`pthread_mutex_unlock()`
`mutex_trylock()`	`pthread_mutex_trylock()`
`mutex_init()`	`pthread_mutex_init()`
`mutex_destroy()`	`pthread_mutex_destroy()`
`cond_wait()`	`pthread_cond_wait()`
`cond_timedwait()`	`pthread_cond_timedwait()`
`cond_signal()`	`pthread_cond_signal()`
`cond_broadcast()`	`pthread_cond_broadcast()`

TABLE 9–2 Solaris Threads and POSIX pthreads Comparison *(continued)*

Solaris Threads (libthread)	**pthreads (libpthread)**
`cond_init( )`	`pthread_cond_init( )`
`cond_destroy( )`	`pthread_cond_destroy( )`
`rwlock_init( )`	-
`rwlock_destroy( )`	-
`rw_rdlock( )`	-
`rw_wrlock( )`	-
`rw_unlock( )`	-
`rw_tryrdlock( )`	-
`rw_trywrlock( )`	-
`sema_init( )`	`sem_init( )` POSIX 1003.4
`sema_destroy( )`	`sem_destroy( )` POSIX 1003.4
`sema_wait( )`	`sem_wait( )` POSIX 1003.4
`sema_post( )`	`sem_post( )` POSIX 1003.4
`sema_trywait( )`	`sem_trywait( )` POSIX 1003.4
`fork1( )`	`fork( )`
-	`pthread_atfork( )`
`fork( )` (multiple thread copy)	-

TABLE 9–2 Solaris Threads and POSIX pthreads Comparison *(continued)*

Solaris Threads (libthread)	pthreads (libpthread)
-	`pthread_mutexattr_init( )`
-	`pthread_mutexattr_destroy( )`
`type( )` argument in `cond_init( )`	`pthread_mutexattr_setpshared( )`
-	`pthread_mutxattr_getpshared( )`
-	`pthread_condattr_init( )`
-	`pthread_condattr_destroy( )`
`type( )` argument in `cond_init( )`	`pthread_condattr_setpshared( )`
-	`pthread_condattr_getpshared( )`
-	`pthread_attr_init( )`
-	`pthread_attr_destroy( )`
THR_BOUND flag in `thr_create( )`	`pthread_attr_setscope( )`
-	`pthread_attr_getscope( )`
`stack_size( )` argument in`thr_create( )`	`pthread_attr_setstacksize( )`
-	`pthread_attr_getstacksize( )`
`stack_addr( )` argument in`thr_create( )`	`pthread_attr_setstackaddr( )`
-	`pthread_attr_getstackaddr( )`

TABLE 9–2 Solaris Threads and POSIX pthreads Comparison *(continued)*

Solaris Threads (libthread)	**pthreads (libpthread)**
THR_DETACH flag in `thr_create()`	`pthread_attr_setdetachstate()`
-	`pthread_attr_getdetachstate()`
-	`pthread_attr_setschedparam()`
-	`pthread_attr_getschedparam()`
-	`pthread_attr_setinheritsched()`
-	`pthread_attr_getinheritsched()`
-	`pthread_attr_setsschedpolicy()`
-	`pthread_attr_getschedpolicy()`

To use the Solaris threads functions described in this chapter, you must link with the Solaris threads library `–lthread`).

Where functionality is virtually the same for both Solaris threads and for pthreads, (even though the function names or arguments might differ), only a brief example consisting of the correct include file and the function prototype is presented. Where return values are not given for the Solaris threads functions, see the appropriate pages in *man Pages(3): Library Routines* for the function return values.

For more information on Solaris related functions, see the related pthreads documentation for the similarly named function.

Where Solaris threads functions offer capabilities that are not available in pthreads, a full description of the functions is provided.

Unique Solaris Threads Functions

- "Suspend Thread Execution" on page 160
- "Continue a Suspended Thread" on page 161

- "Set Thread Concurrency Level" on page 161
- "Get Thread Concurrency" on page 163

Suspend Thread Execution

thr_suspend(3T)

`thr_suspend()` immediately suspends the execution of the thread specified by target_thread. On successful return from `thr_suspend()`, the suspended thread is no longer executing.

Once a thread is suspended, subsequent calls to `thr_suspend()` have no effect. Signals cannot awaken the suspended thread; they remain pending until the thread resumes execution.

```
#include <thread.h>

int thr_suspend(thread_t tid);
```

In the following synopsis, `pthread_t` *tid* as defined in pthreads is the same as `thread_t` *tid* in Solaris threads. *tid* values can be used interchangeably either by assignment or through the use of casts.

```
thread_t tid; /* tid from thr_create() */

/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create() */
pthread_t ptid;

int ret;

ret = thr_suspend(tid);

/* using pthreads ID variable with a cast */
ret = thr_suspend((thread_t) ptid);
```

Return Values

`thr_suspend()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_suspend()` fails and returns the corresponding value.

`ESRCH`

tid cannot be found in the current process.

Continue a Suspended Thread

thr_continue(3T)

`thr_continue()` resumes the execution of a suspended thread. Once a suspended thread is continued, subsequent calls to `thr_continue()` have no effect.

```
#include <thread.h>

int thr_continue(thread_t tid);
```

A suspended thread will not be awakened by a signal. The signal stays pending until the execution of the thread is resumed by `thr_continue()`.

`pthread_t` *tid* as defined in pthreads is the same as `thread_t` *tid* in Solaris threads. *tid* values can be used interchangeably either by assignment or through the use of casts.

```
thread_t tid; /* tid from thr_create()*/

/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create()*/
pthread_t ptid;

int ret;

ret = thr_continue(tid);

/* using pthreads ID variable with a cast */
ret = thr_continue((thread_t) ptid)
```

Return Values

`thr_continue()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_continue()` fails and returns the corresponding value.

`ESRCH`

tid cannot be found in the current process.

Set Thread Concurrency Level

By default, Solaris threads attempt to adjust the system execution resources (LWPs) used to run unbound threads to match the real number of active threads. While the Solaris threads package cannot make perfect decisions, it at least ensures that the process continues to make progress.

When you have some idea of the number of unbound threads that should be simultaneously active (executing code or system calls), tell the library through thr_setconcurrency( ). To get the number of threads being used, use thr_getconcurrency( ).

thr_setconcurrency(3T)

thr_setconcurrency( ) provides a hint to the system about the required level of concurrency in the application. The system ensures that a sufficient number of threads are active so that the process continues to make progress.

```
#include <thread.h>

int new_level;
int ret;

ret = thr_setconcurrency(new_level);
```

Unbound threads in a process might or might not be required to be simultaneously active. To conserve system resources, the threads system ensures by default that enough threads are active for the process to make progress, and that the process will not deadlock through a lack of concurrency.

Because this might not produce the most effective level of concurrency, thr_setconcurrency( ) permits the application to give the threads system a hint, specified by *new_level*, for the desired level of concurrency.

The actual number of simultaneously active threads can be larger or smaller than *new_level*.

Note that an application with multiple compute-bound threads can fail to schedule all the runnable threads if thr_setconcurrency( ) has not been called to adjust the level of execution resources.

You can also affect the value for the desired concurrency level by setting the THR_NEW_LWP flag in thr_create( ). This effectively increments the current level by one.

Return Values

Returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, thr_setconcurrency( ) fails and returns the corresponding value.

EAGAIN

The specified concurrency level would cause a system resource to be exceeded.

EINVAL

The value for *new_level* is negative.

Get Thread Concurrency

thr_getconcurrency(3T)

Use `thr_getconcurrency()` to get the current value of the concurrency level previously set by `thr_setconcurrency()`. Note that the actual number of simultaneously active threads can be larger or smaller than this number.

```
#include <thread.h>

int thr_getconcurrency(void)
```

Return Value

`thr_getconcurrency()` always returns the current value for the desired concurrency level.

Unique Solaris Synchronization Functions–Readers / Writer Locks

Readers / Writer locks allow simultaneous read access by many threads while restricting write access to only one thread at a time.

- "Initialize a Readers / Writer Lock" on page 164
- "Acquire a Read Lock" on page 165
- "Try to Acquire a Read Lock" on page 166
- "Acquire a Write Lock" on page 166
- "Try to Acquire a Write Lock" on page 167
- "Unlock a Readers / Writer Lock" on page 168
- "Destroy Readers / Writer Lock State" on page 168

When any thread holds the lock for reading, other threads can also acquire the lock for reading but must wait to acquire the lock for writing. If one thread holds the lock for writing, or is waiting to acquire the lock for writing, other threads must wait to acquire the lock for either reading or writing.

Readers / writer locks are slower than mutexes, but can improve performance when they protect data that are not frequently written but that are read by many concurrent threads.

Use readers / writer locks to synchronize threads in this process and other processes by allocating them in memory that is writable and shared among the cooperating processes (see mmap(2)) and by initializing them for this behavior.

By default, the acquisition order is not defined when multiple threads are waiting for a readers/writer lock. However, to avoid writer starvation, the Solaris threads package tends to favor writers over readers.

Readers/writer locks must be initialized before use.

Initialize a Readers/Writer Lock

rwlock_init(3T)

```
#include <synch.h>  (or #include <thread.h>)

int rwlock_init(rwlock_t *rwlp, int type, void * arg);
```

Use `rwlock_init()` to initialize the readers/writer lock pointed to by *rwlp* and to set the lock state to unlocked. *type* can be one of the following (note that *arg* is currently ignored).

- `USYNC_PROCESS` The readers/writer lock can be used to synchronize threads in this process and other processes. *arg* is ignored.
- `USYNC_THREAD` The readers/writer lock can be used to synchronize threads in this process, only. *arg* is ignored.

Multiple threads must not initialize the same readers/writer lock simultaneously. Readers/writer locks can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. A readers/writer lock must not be reinitialized while other threads might be using it.

Initializing Readers/Writer Locks with Intraprocess Scope

```
#include <thread.h>

rwlock_t rwlp;
int ret;

/* to be used within this process only */
ret = rwlock_init(&rwlp, USYNC_THREAD, 0);
```

Initializing Readers/Writer Locks with Interprocess Scope

```
#include <thread.h>

rwlock_t rwlp;
int ret;

/* to be used among all processes */
ret = rwlock_init(&rwlp, USYNC_PROCESS, 0);
```

Return Values

`rwlock_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

Invalid argument.

`EFAULT`

rwlp or *arg* points to an illegal address.

Acquire a Read Lock

rw_rdlock(3T)

`#include <synch.h>` *(or* `#include <thread.h>`*)*

`int rw_rdlock(rwlock_t *`*rwlp*`);`

Use `rw_rdlock()` to acquire a read lock on the readers/writer lock pointed to by *rwlp*. When the readers/writer lock is already locked for writing, the calling thread blocks until the write lock is released. Otherwise, the read lock is acquired.

Return Values

`rw_rdlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

Invalid argument.

`EFAULT`

rwlp points to an illegal address.

Try to Acquire a Read Lock

rw_tryrdlock(3T)

```
#include <synch.h>  (or #include <thread.h>)
```

```
int rw_tryrdlock(rwlock_t *rwlp);
```

Use `rw_tryrdlock()` to attempt to acquire a read lock on the readers/writer lock pointed to by *rwlp*. When the readers/writer lock is already locked for writing, it returns an error. Otherwise, the read lock is acquired.

Return Values

`rw_tryrdlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

Invalid argument.

`EFAULT`

rwlp points to an illegal address.

`EBUSY`

The readers/writer lock pointed to by *rwlp* was already locked.

Acquire a Write Lock

rw_wrlock(3T)

```
#include <synch.h>  (or #include <thread.h>)
```

```
int rw_wrlock(rwlock_t *rwlp);
```

Use `rw_wrlock()` to acquire a write lock on the readers/writer lock pointed to by *rwlp*. When the readers/writer lock is already locked for reading or writing, the calling thread blocks until all the read locks and write locks are released. Only one thread at a time can hold a write lock on a readers/writer lock.

Return Values

`rw_wrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

Invalid argument.

`EFAULT`

rwlp points to an illegal address.

Try to Acquire a Write Lock

rw_trywrlock(3T)

```
#include <synch.h>  (or #include <thread.h>)

int rw_trywrlock(rwlock_t *rwlp);
```

Use `rw_trywrlock()` to attempt to acquire a write lock on the readers/writer lock pointed to by *rwlp*. When the readers/writer lock is already locked for reading or writing, it returns an error.

Return Values

`rw_trywrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

Invalid argument.

`EFAULT`

rwlp points to an illegal address.

`EBUSY`

The readers/writer lock pointed to by *rwlp* was already locked.

Unlock a Readers / Writer Lock

rw_unlock(3T)

```
#include <synch.h>  (or #include <thread.h>)
```

```
int rw_unlock(rwlock_t *rwlp);
```

Use `rw_unlock( )` to unlock a readers / writer lock pointed to by *rwlp*. The readers / writer lock must be locked and the calling thread must hold the lock either for reading or writing. When any other threads are waiting for the readers / writer lock to become available, one of them is unblocked.

Return Values

`rw_unlock( )` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

Invalid argument.

`EFAULT`

rwlp points to an illegal address.

Destroy Readers / Writer Lock State

rwlock_destroy(3T)

```
#include <synch.h>  (or #include <thread.h>)
```

```
int rwlock_destroy(rwlock_t *rwlp);
```

Use `rwlock_destroy( )` to destroy any state associated with the readers / writer lock pointed to by *rlwp*. The space for storing the readers / writer lock is not freed.

Return Values

`rwlock_destroy( )` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

EINVAL

Invalid argument.

EFAULT

rwlp points to an illegal address.

Readers/Writer Lock Example

Code Example 9–1 uses a bank account to demonstrate readers / writer locks. While the program could allow multiple threads to have concurrent read-only access to the account balance, only a single writer is allowed. Note that the get_balance() function needs the lock to ensure that the addition of the checking and saving balances occurs atomically.

CODE EXAMPLE 9–1 Read / Write Bank Account

```
rwlock_t account_lock;
float checking_balance = 100.0;
float saving_balance = 100.0;
...
rwlock_init(&account_lock, 0, NULL);
...

float
get_balance() {
    float bal;

    rw_rdlock(&account_lock);
    bal = checking_balance + saving_balance;
    rw_unlock(&account_lock);
    return(bal);
}

void
transfer_checking_to_savings(float amount) {
    rw_wrlock(&account_lock);
    checking_balance = checking_balance - amount;
    saving_balance = saving_balance + amount;
    rw_unlock(&account_lock);
}
```

Similar Solaris Threads Functions

"Create a Thread" on page 171	"thr_create(3T) " on page 171
"Get the Minimal Stack Size" on page 173	"thr_min_stack(3T)" on page 173
"Get the Thread Identifier" on page 174	"thr_self(3T)" on page 174
"Yield Thread Execution" on page 174	"thr_yield(3T)" on page 174
"Send a Signal to a Thread" on page 174	"thr_kill(3T)" on page 174
"Access the Signal Mask of the Calling Thread" on page 175	"thr_sigsetmask(3T)" on page 175
"Terminate a Thread" on page 175	"thr_exit(3T)" on page 175
"Wait for Thread Termination" on page 175	"thr_join(3T)" on page 175
"Create a Thread-Specific Data Key" on page 176	"thr_keycreate(3T)" on page 176
"Set the Thread-Specific Data Key" on page 176	"thr_setspecific(3T)" on page 176
"Get the Thread-Specific Data Key" on page 177	"thr_getspecific(3T)" on page 177
"Set the Thread Priority " on page 177	"thr_setprio(3T)" on page 177
"Get the Thread Priority " on page 178	"thr_getprio(3T)" on page 178

Create a Thread

The `thr_create(3T)` routine is one of the most elaborate of all the Solaris threads library routines.

thr_create(3T)

Use `thr_create()` to add a new thread of control to the current process.

Note that the new thread does not inherit pending signals, but it does inherit priority and signal masks.

```
#include <thread.h>

int thr_create(void *stack_base, size_t stack_size,
    void *(*start_routine) (void *), void *arg, long flags,
    thread_t *new_thread);

size_t thr_min_stack(void);
```

stack_base—Contains the address for the stack that the new thread uses. If *stack_base* is `NULL` then `thr_create()` allocates a stack for the new thread with at least *stack_size* bytes.

stack_size—Contains the size, in number of bytes, for the stack that the new thread uses. If *stack_size* is zero, a default size is used. In most cases, a zero value works best. If *stack_size* is not zero, it must be greater than the value returned by `thr_min_stack()`.

There is no general need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the –`MAP_NORESERVE` option of `mmap()` to make the allocations.)

start_routine—Contains the function with which the new thread begins execution. When `start_routine()` returns, the thread exits with the exit status set to the value returned by *start_routine* (see "thr_exit(3T)" on page 175;).

arg—Can be anything that is described by `void`, which is typically any 4-byte value. Anything larger must be passed indirectly by having the argument point to it.

Note that you can supply only one argument. To get your procedure to take multiple arguments, encode them as one (such as by putting them in a structure).

flags`---()`Specifies attributes for the created thread. In most cases a zero value works best.

The value in *flags* is constructed from the bitwise inclusive OR of the following:

- `THR_SUSPENDED`—Suspends the new thread and does not execute *start_routine* until the thread is started by `thr_continue()`. Use this to operate on the thread (such as changing its priority) before you run it. The termination of a detached thread is ignored.

- THR_DETACHED—Detaches the new thread so that its thread ID and other resources can be reused as soon as the thread terminates. Set this when you do not want to wait for the thread to terminate.

Note - When there is no explicit synchronization to prevent it, an unsuspended, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `thr_create()`.

- THR_BOUND—Permanently binds the new thread to an LWP (the new thread is a *bound thread*).
- THR_NEW_LWP—Increases the concurrency level for unbound threads by one. The effect is similar to incrementing concurrency by one with `thr_setconcurrency(3T)`, although THR_NEW_LWP does not affect the level set through the `thr_setconcurrency()` function. Typically, THR_NEW_LWP adds a new LWP to the pool of LWPs running unbound threads.
- When you specify both THR_BOUND and THR_NEW_LWP, two LWPs are typically created—one for the bound thread and another for the pool of LWPs running unbound threads.
- THR_DAEMON—Marks the new thread as a daemon. The process exits when all nondaemon threads exit. Daemon threads do not affect the process exit status and are ignored when counting the number of thread exits.

  A process can exit either by calling exit() or by having every thread in the process that was not created with the THR_DAEMON flag call `thr_exit(3T)`. An application, or a library it calls, can create one or more threads that should be ignored (not counted) in the decision of whether to exit. The THR_DAEMON flag identifies threads that are not counted in the process exit criterion.

new_thread—Points to a location (when *new_thread* is not NULL) where the ID of the new thread is stored when `thr_create()` is successful. The caller is responsible for supplying the storage this argument points to. The ID is valid only within the calling process.

If you are not interested in this identifier, supply a zero value to *new_thread*.

Return Values

Returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_create()` fails and returns the corresponding value.

EAGAIN

A system limit is exceeded, such as when too many LWPs have been created.

ENOMEM

Not enough memory was available to create the new thread.

`EINVAL`

stack_base is not `NULL` and *stack_size* is less than the value returned by `thr_min_stack.()`

Stack Behavior

Stack behavior in Solaris threads is generally the same as that in pthreads. For more information about stack setup and operation, see "About Stacks" on page 49.

You can get the absolute minimum on stack size by calling `thr_min_stack()`, which returns the amount of stack space required for a thread that executes a null procedure. Useful threads need more than this, so be very careful when reducing the stack size.

You can specify a custom stack in two ways. The first is to supply a `NULL` for the stack location, thereby asking the runtime library to allocate the space for the stack, but to supply the desired size in the stacksize parameter to `thr_create()`.

The other approach is to take overall aspects of stack management and supply a pointer to the stack to `thr_create()`. This means that you are responsible not only for stack allocation but also for stack deallocation—when the thread terminates, you must arrange for the disposal of its stack.

When you allocate your own stack, be sure to append a red zone to its end by calling `mprotect(2)`.

Get the Minimal Stack Size

thr_min_stack(3T)

Use `thr_min_stack(3T)` to get the minimum stack size for a thread.

```
#include <thread.h>

size_t thr_min_stack(void);
```

`thr_min_stack()` returns the amount of space needed to execute a null thread (a null thread is a thread that is created to execute a null procedure).

A thread that does more than execute a null procedure should allocate a stack size greater than the size of `thr_min_stack()`.

When a thread is created with a user-supplied stack, the user must reserve enough space to run the thread. In a dynamically linked execution environment, it is difficult to know what the thread minimal stack requirements are.

Most users should not create threads with user-supplied stacks. User-supplied stacks exist only to support applications that require complete control over their execution environments.

Instead, users should let the threads library manage stack allocation. The threads library provides default stacks that should meet the requirements of any created thread.

Get the Thread Identifier

thr_self(3T)

Use `thr_self(3T)` to get the ID of the calling thread.

```
#include <thread.h>

thread_t thr_self(void);
```

Yield Thread Execution

thr_yield(3T)

`thr_yield()` causes the current thread to yield its execution in favor of another thread with the same or greater priority; otherwise it has no effect. There is no guarantee that a thread calling `thr_yield()` will do so.

```
#include <thread.h>

void thr_yield(void);
```

Send a Signal to a Thread

thr_kill(3T)

`thr_kill()` sends a signal to a thread.

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


int thr_kill(thread_t target_thread, int sig);
```

Access the Signal Mask of the Calling Thread

thr_sigsetmask(3T)

Use `thr_sigsetmask()` to change or examine the signal mask of the calling thread.

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


int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset);
```

Terminate a Thread

thr_exit(3T)

Use `thr_exit()` to terminate a thread.

```
#include <thread.h>

void thr_exit(void *status);
```

Wait for Thread Termination

thr_join(3T)

Use the `thr_join()` function to wait for a thread to terminate.

```
#include <thread.h>

int thr_join(thread_t tid, thread_t *departedid, void **status);
```

Join specific

```
#include <thread.h>

thread_t tid;
thread_t departedid;
int ret;
int status;

/* waiting to join thread "tid" with status */
ret = thr_join(tid, &departedid, (void**)&status);

/* waiting to join thread "tid" without status */
ret = thr_join(tid, &departedid, NULL);

/* waiting to join thread "tid" without return id and status */
ret = thr_join(tid, NULL, NULL);
```

When the *tid* is `(thread_t)0`, then `thread_join()` waits for any undetached thread in the process to terminate. In other words, when no thread identifier is specified, any undetached thread that exits causes `thread_join()` to return.

Join any

```
#include <thread.h>

thread_t tid;
thread_t departedid;
int ret;
int status;

/* waiting to join thread "tid" with status */
ret = thr_join(NULL, &departedid, (void **)&status);
```

By indicating `NULL` as thread id in the Solaris `thr_join()`, a join will take place when any non detached thread in the process exits. The departedid will indicate the thread ID of exiting thread.

Create a Thread-Specific Data Key

Except for the function names and arguments, thread specific data is the same for Solaris as it is for POSIX. The synopses for the Solaris functions are given in this section. The functions are explained in “Create a Thread-Specific Data Key” on page 176.

thr_keycreate(3T)

`thr_keycreate()` allocates a key that is used to identify thread-specific data in a process.

```
#include <thread.h>

int thr_keycreate(thread_key_t *keyp,
    void (*destructor) (void *value));
```

Set the Thread-Specific Data Key

thr_setspecific(3T)

`thr_setspecific()` binds *value* to the thread-specific data key, *key*, for the calling thread.

```
#include <thread.h>
```

```
int thr_setspecific(thread_key_t key, void *value);
```

Get the Thread-Specific Data Key

thr_getspecific(3T)

`thr_getspecific()` stores the current value bound to *key* for the calling thread into the location pointed to by *valuep*.

```
#include <thread.h>

int thr_getspecific(thread_key_t key, void **valuep);
```

Set the Thread Priority

In Solaris threads, if a thread is to be created with a priority other than that of its parent's, it is created in SUSPEND mode. While suspended, the threads priority is modified using the `thr_setprio(3T)` function call; then it is continued.

An unbound thread is usually scheduled only with respect to other threads in the process using simple priority levels with no adjustments and no kernel involvement. Its system priority is usually uniform and is inherited from the creating process.

thr_setprio(3T)

The function `thr_setprio()` changes the priority of the thread, specified by *tid*, within the current process to the priority specified by *newprio*.

```
#include <thread.h>

int thr_setprio(thread_t tid, int newprio)
```

By default, threads are scheduled based on fixed priorities that range from zero, the least significant, to the largest integer. The *tid* will preempt lower priority threads, and will yield to higher priority threads.

```
thread_t tid;
int ret;
int newprio = 20;

/* suspended thread creation */
ret = thr_create(NULL, NULL, func, arg, THR_SUSPEND, &tid);

/* set the new priority of suspended child thread */
ret = thr_setprio(tid, newprio);

/* suspended child thread starts executing with new priority */
ret = thr_continue(tid);
```

Get the Thread Priority

thr_getprio(3T)

Use `thr_getprio()` to get the current priority for the thread. Each thread inherits a priority from its creator. `thr_getprio()` stores the current priority, *tid*, in the location pointed to by *newprio*.

```
#include <thread.h>

int thr_getprio(thread_t tid, int *newprio)
```

Similar Synchronization Functions–Mutual Exclusion Locks

- "Initialize a Mutex" on page 178
- "Destroy a Mutex" on page 179
- "Acquire a Mutex" on page 179
- "Release a Mutex" on page 180
- "Try to Acquire a Mutex" on page 180

Initialize a Mutex

mutex_init(3T)

```
#include <synch.h>    (or
#include <thread.h>)

int mutex_init(mutex_t *mp, int type, void *arg));
```

Use `mutex_init()` to initialize the mutex pointed to by *mp*. The *type* can be one of the following (note that *arg* is currently ignored).

- `USYNC_PROCESS` The mutex can be used to synchronize threads in this and other processes.
- `USYNC_THREAD` The mutex can be used to synchronize threads in this process, only.

Mutexes can also be initialized by allocation in zeroed memory, in which case a *type* of `USYNC_THREAD` is assumed.

Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be reinitialized while other threads might be using it.

Mutexes with Intraprocess Scope

```
#include <thread.h>

mutex_t mp;
int ret;

/* to be used within this process only */
ret = mutex_init(&mp, USYNC_THREAD, 0);
```

Mutexes with Interprocess Scope

```
#include <thread.h>

mutex_t mp;
int ret;

/* to be used among all processes */
ret = mutex_init(&mp, USYNC_PROCESS, 0);
```

Destroy a Mutex

mutex_destroy(3T)

```
#include <thread.h>

int mutex_destroy (mutex_t *mp);
```

Use `mutex_destroy()` to destroy any state associated with the mutex pointed to by *mp*. Note that the space for storing the mutex is not freed.

Acquire a Mutex

mutex_lock(3T)

```
#include <thread.h>

int mutex_lock(mutex_t *mp);
```

Use `mutex_lock()` to lock the mutex pointed to by *mp*. When the mutex is already locked, the calling thread blocks until the mutex becomes available (blocked threads wait on a prioritized queue).

Release a Mutex

mutex_unlock(3T)

```
#include <thread.h>

int mutex_unlock(mutex_t *mp);
```

Use `mutex_unlock()` to unlock the mutex pointed to by *mp*. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner).

Try to Acquire a Mutex

mutex_trylock(3T)

```
#include <thread.h>

int mutex_trylock(mutex_t *mp);
```

Use `mutex_trylock()` to attempt to lock the mutex pointed to by *mp*. This function is a nonblocking version of `mutex_lock()`.

Similar Synchronization Functions–Condition Variables

- "Initialize a Condition Variable" on page 181
- "Destroy a Condition Variable" on page 181
- "Wait for a Condition" on page 182
- "Wait for an Absolute Time" on page 182
- "Signal One Condition Variable" on page 182
- "Signal All Condition Variables" on page 183

Initialize a Condition Variable

cond_init(3T)

```
#include <thread.h>

int cond_init(cond_t *cv, int type, int arg);
```

Use `cond_init( )` to initialize the condition variable pointed to by *cv*. The *type* can be one of the following (note that *arg* is currently ignored).

- `USYNC_PROCESS` The condition variable can be used to synchronize threads in this and other processes. *arg* is ignored.
- `USYNC_THREAD` The condition variable can be used to synchronize threads in this process, only. *arg* is ignored.

Condition variables can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed.

Multiple threads must not initialize the same condition variable simultaneously. A condition variable must not be reinitialized while other threads might be using it.

Condition Variables with Intraprocess Scope

```
#include <thread.h>

cond_t cv;
int ret;

/* to be used within this process only */
ret = cond_init(cv, USYNC_THREAD, 0);
```

Condition Variables with Interprocess Scope

```
#include <thread.h>

cond_t cv;
int ret;

/* to be used among all processes */
ret = cond_init(&cv, USYNC_PROCESS, 0);
```

Destroy a Condition Variable

cond_destroy(3T)

```
#include <thread.h>

int cond_destroy(cond_t *cv);
```

Use `cond_destroy( )` to destroy state associated with the condition variable pointed to by *cv*. The space for storing the condition variable is not freed.

Wait for a Condition

cond_wait(3T)

```
#include <thread.h>
```

```
int cond_wait(cond_t *cv, mutex_t *mp);
```

Use `cond_wait( )` to atomically release the mutex pointed to by *mp* and to cause the calling thread to block on the condition variable pointed to by *cv*. The blocked thread can be awakened by `cond_signal( )`, `cond_broadcast( )`, or when interrupted by delivery of a signal or a `fork( )`.

Wait for an Absolute Time

cond_timedwait(3T)

```
#include <thread.h>
```

```
int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime)
```

Use `cond_timedwait( )` as you would use `cond_wait( )`, except that `cond_timedwait( )` does not block past the time of day specified by *abstime*.

`cond_timedwait( )` always returns with the mutex locked and owned by the calling thread even when returning an error.

The `cond_timedwait( )` function blocks until the condition is signaled or until the time of day specified by the last argument has passed. The time-out is specified as a time of day so the condition can be retested efficiently without recomputing the time-out value.

Signal One Condition Variable

cond_signal(3T)

```
#include <thread.h>
```

```
int cond_signal(cond_t *cv);
```

Use `cond_signal()` to unblock one thread that is blocked on the condition variable pointed to by *cv*. Call this function under protection of the same mutex used with the condition variable being signaled. Otherwise, the condition could be signaled between its test and `cond_wait()`, causing an infinite wait.

Signal All Condition Variables

cond_broadcast(3T)

```
#include <thread.h>

int cond_broadcast(cond_t *cv);
```

Use `cond_broadcast()` to unblock all threads that are blocked on the condition variable pointed to by *cv*. When no threads are blocked on the condition variable then `cond_broadcast()` has no effect.

Similar Synchronization Functions–Semaphores

Semaphore operations are the same in both Solaris and POSIX. The function name changed from `sema_` in Solaris to `sem_` in pthreads.

- “Initialize a Semaphore” on page 183
- “Increment a Semaphore” on page 184
- “Block on a Semaphore Count” on page 185
- “Decrement a Semaphore Count” on page 185
- “Destroy the Semaphore State” on page 185

Initialize a Semaphore

sema_init(3T)

```
#include <thread.h>
```

```
int sema_init(sema_t *sp, unsigned int count, int type,
    void *arg);
```

Use `sema_init()` to initialize the semaphore variable pointed to by *sp* by *count* amount. *type* can be one of the following (note that *arg* is currently ignored).

`USYNC_PROCESS` The semaphore can be used to synchronize threads in this process and other processes. Only one process should initialize the semaphore. *arg* is ignored.

`USYNC_THREAD` The semaphore can be used to synchronize threads in this process, only. *arg* is ignored.

Multiple threads must not initialize the same semaphore simultaneously. A semaphore must not be reinitialized while other threads may be using it.

Semaphores with Intraprocess Scope

```
#include <thread.h>

sema_t sp;
int ret;
int count;
count = 4;

/* to be used within this process only */
ret = sema_init(&sp, count, USYNC_THREAD, 0);
```

Semaphores with Interprocess Scope

```
#include <thread.h>

sema_t sp;
int ret;
int count;
count = 4;

/* to be used among all the processes */
ret = sema_init (&sp, count, USYNC_PROCESS, 0);
```

Increment a Semaphore

sema_post(3T)

```
#include <thread.h>

int sema_post(sema_t *sp);
```

Use `sema_post()` to atomically increment the semaphore pointed to by *sp*. When any threads are blocked on the semaphore, one is unblocked.

Block on a Semaphore Count

sema_wait(3T)

```
#include <thread.h>

int sema_wait(sema_t *sp);
```

Use `sema_wait()` to block the calling thread until the count in the semaphore pointed to by *sp* becomes greater than zero, then atomically decrement it.

Decrement a Semaphore Count

sema_trywait(3T)

```
#include <thread.h>

int sema_trywait(sema_t *sp);
```

Use `sema_trywait()` to atomically decrement the count in the semaphore pointed to by *sp* when the count is greater than zero. This function is a nonblocking version of `sema_wait()`.

Destroy the Semaphore State

sema_destroy(3T)

```
#include <thread.h>

int sema_destroy(sema_t *sp);
```

Use `sema_destroy()` to destroy any state associated with the semaphore pointed to by *sp*. The space for storing the semaphore is not freed.

Synchronization Across Process Boundaries

Each of the synchronization primitives can be set up to be used across process boundaries. This is done quite simply by ensuring that the synchronization variable is located in a shared memory segment and by calling the appropriate `init` routine with type set to `USYNC_PROCESS`.

If this has been done, then the operations on the synchronization variables work just as they do when *type* is `USYNC_THREAD`.

```
mutex_init(&m, USYNC_PROCESS, 0);


rwlock_init(&rw, USYNC_PROCESS, 0);


cond_init(&cv, USYNC_PROCESS, 0);


sema_init(&s, count, USYNC_PROCESS, 0);
```

Using LWPs Between Processes

Using locks and condition variables between processes does not require using the threads library. The recommended approach is to use the threads library interfaces, but when this is not desirable, then the `_lwp_mutex_*` and `_lwp_cond_*` interfaces can be used as follows:

1. Allocate the locks and condition variables as usual in shared memory (either with shmop(2) or mmap(2)).
2. Then initialize the newly allocated objects appropriately with the `USYNC_PROCESS` type. Because no interface is available to perform the initialization (_lwp_mutex_init(2) and _lwp_cond_init(2) do not exist), the objects can be initialized using statically allocated and initialized dummy objects.

For example, to initialize `lockp`:

```
lwp_mutex_t *lwp_lockp;
lwp_mutex_t dummy_shared_mutex = SHAREDMUTEX;
 /* SHAREDMUTEX is defined in /usr/include/synch.h */
...
...
lwp_lockp = alloc_shared_lock();
*lwp_lockp = dummy_shared_mutex;
```

Similarly, for condition variables:

```
lwp_cond_t *lwp_condp;
lwp_cond_t dummy_shared_cv = SHAREDCV;
```

```
 /* SHAREDCV is defined in /usr/include/synch.h */
...
...
lwp_condp = alloc_shared_cv();
*lwp_condp = dummy_shared_cv;
```

Producer/Consumer Problem Example

Code Example 9–2 shows the producer/consumer problem with the producer and consumer in separate processes. The main routine maps zero-filled memory (that it shares with its child process) into its address space. Note that `mutex_init()` and `cond_init()` must be called because the type of the synchronization variables is `USYNC_PROCESS`.

A child process is created that runs the consumer. The parent runs the producer.

This example also shows the drivers for the producer and consumer. The `producer_driver`() simply reads characters from `stdin` and calls `producer`(). The `consumer_driver`() gets characters by calling `consumer`() and writes them to `stdout`.

The data structure for Code Example 9–2 is the same as that used for the solution with condition variables (see "Nested Locking with a Singly Linked List" on page 68).

CODE EXAMPLE 9–2 The Producer/Consumer Problem, Using USYNC_PROCESS

```
main() {
    int zfd;
    buffer_t *buffer;

    zfd = open(‘‘/dev/zero’’, O_RDWR);
    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
    buffer->occupied = buffer->nextin = buffer->nextout = 0;

    mutex_init(&buffer->lock, USYNC_PROCESS, 0);
    cond_init(&buffer->less, USYNC_PROCESS, 0);
    cond_init(&buffer->more, USYNC_PROCESS, 0);
    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}

void producer_driver(buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, ‘\0’);
            break;
        } else
            producer(b, (char)item);
    }
}
```

```
void consumer_driver(buffer_t *b) {
    char item;

    while (1) {
        if ((item = consumer(b)) == '\0')
            break;
        putchar(item);
    }
}
```

A child process is created to run the consumer; the parent runs the producer.

Special Issues for fork() and Solaris Threads

Solaris threads and POSIX threads define the behavior of `fork()` differently. See "Process Creation–exec(2)and exit(2) Issues" on page 102 for a thorough discussion of `fork()` issues.

Solaris `libthread` supports both `fork()` and `fork1()`. The `fork()` call has "fork-all" semantics—it duplicates everything in the process, including threads and LWPs, creating a true clone of the parent. The `fork1()` call creates a clone that has only one thread; the process state and address space are duplicated, but only the calling thread is cloned.

POSIX `libpthread` supports only `fork()`, which has the same semantics as `fork1()` in Solaris threads.

Whether `fork()` has "fork-all" semantics or "fork-one" semantics is dependent upon which library is used. Linking with `–lthread` assigns "fork-all" semantics to `fork()`, while linking with `–lpthread` assigns "fork-one" semantics to `fork()`.

See "Linking With libthread or libpthread" on page 131 for more details.

CHAPTER **10**

Programming Guidelines

This chapter gives some pointers on programming with threads. Most pointers apply to both Solaris and POSIX threads, but where functionality differs, it is noted. Changing from single-threaded thinking to multithreaded thinking is emphasized in this chapter.

Rethinking Global Variables

Historically, most code has been designed for single-threaded programs. This is especially true for most of the library routines called from C programs. The following implicit assumptions were made for single-threaded code:

- When you write into a global variable and then, a moment later, read from it, what you read is exactly what you just wrote.
- This is also true for nonglobal, static storage.
- You do not need synchronization because there is nothing to synchronize with.

The next few examples discuss some of the problems that arise in multithreaded programs because of these assumptions, and how you can deal with them.

Traditional, single-threaded C and UNIX have a convention for handling errors detected in system calls. System calls can return anything as a functional value (for example, `write()` returns the number of bytes that were transferred). However, the value `-1` is reserved to indicate that something went wrong. So, when a system call returns `-1`, you know that it failed.

CODE EXAMPLE 10–1 Global Variables and *errno*

```
extern int errno;
...
if (write(file_desc, buffer, size) == -1) {
    /* the system call failed */
    fprintf(stderr, ‘‘something went wrong, ‘‘
        ‘‘error code = %d\n’’, errno);
    exit(1);
}
...
```

Rather than return the actual error code (which could be confused with normal return values), the error code is placed into the global variable `errno`. When the system call fails, you can look in `errno` to find out what went wrong.

Now consider what happens in a multithreaded environment when two threads fail at about the same time, but with different errors. Both expect to find their error codes in `errno`, but one copy of `errno` cannot hold both values. This global variable approach simply does not work for multithreaded programs.

Threads solves this problem through a conceptually new storage class—thread-specific data. This storage is similar to global storage in that it can be accessed from any procedure in which a thread might be running. However, it is private to the thread—when two threads refer to the thread-specific data location of the same name, they are referring to two different areas of storage.

So, when using threads, each reference to `errno` is thread-specific because each thread has a private copy of `errno`. This is achieved in this implementation by making `errno` a macro that expands to a function call.

Providing for Static Local Variables

Code Example 10–2 shows a problem similar to the `errno` problem, but involving static storage instead of global storage. The function `gethostbyname(3N)` is called with the computer name as its argument. The return value is a pointer to a structure containing the required information for contacting the computer through network communications.

CODE EXAMPLE 10–2 The `gethostbyname()` Problem

```
struct hostent *gethostbyname(char *name) {
    static struct hostent result;
        /* Lookup name in hosts database */
        /* Put answer in result */
    return(&result);
}
```

Returning a pointer to a local variable is generally not a good idea, although it works in this case because the variable is static. However, when two threads call this variable at once with different computer names, the use of static storage conflicts.

Thread-specific data could be used as a replacement for static storage, as in the `errno` problem, but this involves dynamic allocation of storage and adds to the expense of the call.

A better way to handle this kind of problem is to make the caller of `gethostbyname()` supply the storage for the result of the call. This is done by having the caller supply an additional argument, an output argument, to the routine. This requires a new interface to `gethostbyname()`.

This technique is used in threads to fix many of these problems. In most cases, the name of the new interface is the old name with "`_r`" appended, as in `gethostbyname_r(3N)`.

Synchronizing Threads

The threads in an application must cooperate and synchronize when sharing the data and the resources of the process.

A problem arises when multiple threads call something that manipulates an object. In a single-threaded world, synchronizing access to such objects is not a problem, but as Code Example 10–3illustrates, this is a concern with multithreaded code. (Note that the `printf(3S)` function is safe to call for a multithreaded program; this example illustrates what could happen if `printf()` were not safe.)

CODE EXAMPLE 10–3 The `printf()` Problem

```
/* thread 1: */
    printf("go to statement reached");


/* thread 2: */
    printf("hello world");



printed on display:
    go to hello
```

Single-Threaded Strategy

One strategy is to have a single, application-wide mutex lock that is acquired whenever any thread in the application is running and is released before it must block. Since only one thread can be accessing shared data at any one time, each thread has a consistent view of memory.

Because this is effectively a single-threaded program, very little is gained by this strategy.

Reentrance

A better approach is to take advantage of the principles of modularity and data encapsulation. A reentrant function is one that behaves correctly if it is called simultaneously by several threads. Writing a reentrant function is a matter of understanding just what *behaves correctly* means for this particular function.

Functions that are callable by several threads must be made reentrant. This might require changes to the function interface or to the implementation.

Functions that access global state, like memory or files, have reentrance problems. These functions need to protect their use of global state with the appropriate synchronization mechanisms provided by threads.

The two basic strategies for making functions in modules reentrant are code locking and data locking.

Code Locking

Code locking is done at the function call level and guarantees that a function executes entirely under the protection of a lock. The assumption is that all access to data is done through functions. Functions that share data should execute under the same lock.

Some parallel programming languages provide a construct called a monitor that implicitly does code locking for functions that are defined within the scope of the monitor. A monitor can also be implemented by a mutex lock.

Functions under the protection of the same mutex lock or within the same monitor are guaranteed to execute atomically with respect to each other.

Data Locking

Data locking guarantees that access to a collection of data is maintained consistently. For data locking, the concept of locking code is still there, but code locking is around references to shared (global) data, only. For a mutual exclusion locking protocol, only one thread can be in the critical section for each collection of data.

Alternatively, in a multiple readers, single writer protocol, several readers can be allowed for each collection of data or one writer. Multiple threads can execute in a single module when they operate on different data collections and do not conflict on a single collection for the multiple readers, single writer protocol. So, data locking typically allows more concurrency than does code locking. (Note that Solaris threads has “Readers/Writer Lock” functionality built in.)

What strategy should you use when using locks (whether implemented with mutexes, condition variables, or semaphores) in a program? Should you try to achieve maximum parallelism by locking only when necessary and unlocking as soon as possible (fine-grained locking)? Or should you hold locks for long periods to minimize the overhead of taking and releasing them (coarse-grained locking)?

The granularity of the lock depends on the amount of data it protects. A very coarse-grained lock might be a single lock to protect all data. Dividing how the data is protected by the appropriate number of locks is very important. Too fine a grain of locking can degrade performance. The small cost associated with acquiring and releasing locks can add up when there are too many locks.

The common wisdom is to start with a coarse-grained approach, identify bottlenecks, and add finer-grained locking where necessary to alleviate the bottlenecks. This is reasonably sound advice, but use your own judgment about taking it to the extreme.

Invariants

For both code locking and data locking, *invariants* are important to control locking complexity. An invariant is a condition or relation that is always true.

The definition is modified somewhat for concurrent execution: an invariant is a condition or relation that is true when the associated lock is being set. Once the lock is set, the invariant can be false. However, the code holding the lock must reestablish the invariant before releasing the lock.

An invariant can also be a condition or relation that is true when a lock is being set. Condition variables can be thought of as having an invariant that is the condition.

CODE EXAMPLE 10–4 Testing the Invariant with assert(3X)

```
    mutex_lock(&lock);
    while((condition)==FALSE)
        cond_wait(&cv,&lock);
    assert((condition)==TRUE);
      .
      .
      .
    mutex_unlock(&lock);
```

The `assert()` statement is testing the invariant. The `cond_wait()` function does not preserve the invariant, which is why the invariant must be re-evaluated when the thread returns.

Another example is a module that manages a doubly linked list of elements. For each item on the list a good invariant is the forward pointer of the previous item on the list that should also point to the same thing as the backward pointer of the forward item.

Assume this module uses code-based locking and therefore is protected by a single global mutex lock. When an item is deleted or added the mutex lock is acquired, the correct manipulation of the pointers is made, and the mutex lock is released. Obviously, at some point in the manipulation of the pointers the invariant is false, but the invariant is reestablished before the mutex lock is released.

Avoiding Deadlock

Deadlock is a permanent blocking of a set of threads that are competing for a set of resources. Just because some thread can make progress does not mean that there is not a deadlock somewhere else.

The most common error causing deadlock is *self deadlock* or *recursive deadlock*: a thread tries to acquire a lock it is already holding. Recursive deadlock is very easy to program by mistake.

For example, if a code monitor has every module function grabbing the mutex lock for the duration of the call, then any call between the functions within the module protected by the mutex lock immediately deadlocks. If a function calls some code outside the module which, through some circuitous path, calls back into any method protected by the same mutex lock, then it will deadlock too.

The solution for this kind of deadlock is to avoid calling functions outside the module when you don't know whether they will call back into the module without reestablishing invariants and dropping all module locks before making the call. Of course, after the call completes and the locks are reacquired, the state must be verified to be sure the intended operation is still valid.

An example of another kind of deadlock is when two threads, thread 1 and thread 2, each acquires a mutex lock, A and B, respectively. Suppose that thread 1 tries to acquire mutex lock B and thread 2 tries to acquire mutex lock A. Thread 1 cannot proceed and it is blocked waiting for mutex lock B. Thread 2 cannot proceed and it is blocked waiting for mutex lock A. Nothing can change, so this is a permanent blocking of the threads, and a deadlock.

This kind of deadlock is avoided by establishing an order in which locks are acquired (a *lock hierarchy*). When all threads always acquire locks in the specified order, this deadlock is avoided.

Adhering to a strict order of lock acquisition is not always optimal. When thread 2 has many assumptions about the state of the module while holding mutex lock B, giving up mutex lock B to acquire mutex lock A and then reacquiring mutex lock B in order would cause it to discard its assumptions and reevaluate the state of the module.

The blocking synchronization primitives usually have variants that attempt to get a lock and fail if they cannot, such as `mutex_trylock()`. This allows threads to violate the lock hierarchy when there is no contention. When there is contention, the held locks must usually be discarded and the locks reacquired in order.

Deadlocks Related to Scheduling

Because there is no guaranteed order in which locks are acquired, a problem in threaded programs is that a particular thread never acquires a lock, even though it seems that it should.

This usually happens when the thread that holds the lock releases it, lets a small amount of time pass, and then reacquires it. Because the lock was released, it might seem that the other thread should acquire the lock. But, because nothing blocks the thread holding the lock, it continues to run from the time it releases the lock until it reacquires the lock, and so no other thread is run.

You can usually solve this type of problem by calling `thr_yield(3T)` just before the call to reacquire the lock. This allows other threads to run and to acquire the lock.

Because the time-slice requirements of applications are so variable, the threads library does not impose any. Use calls to `thr_yield()` to make threads share time as you require.

Locking Guidelines

Here are some simple guidelines for locking.

- Try not to hold locks across long operations like I/O where performance can be adversely affected.
- Don’t hold locks when calling a function that is outside the module and that might reenter the module.
- In general, start with a coarse-grained approach, identify bottlenecks, and add finer-grained locking where necessary to alleviate the bottlenecks. Most locks are held for short amounts of time and contention is rare, so fix only those locks that have measured contention.
- When using multiple locks, avoid deadlocks by making sure that all threads acquire the locks in the same order.

Following Some Basic Guidelines

- Know what you are importing and whether it is safe.

  A threaded program cannot arbitrarily enter nonthreaded code.
- Threaded code can safely refer to unsafe code only from the initial thread.

  This ensures that the static storage associated with the initial thread is used only by that thread.
- Sun-supplied libraries are defined to be *safe* unless explicitly documented as unsafe.

  If a reference manual entry does not say whether a function is MT-Safe, it is safe. All MT-unsafe functions are identified explicitly in the manual page.
- Use compilation flags to manage binary incompatible source changes. (See Chapter 7"for complete instructions.)
  - `–D_REENTRANT` enables multithreading with the Solaris threads -lthread library
  - `–D_POSIX_C_SOURCE` with `–lpthread` gives POSIX threads behavior
  - `–D_POSIX_PTHREADS_SEMANTICS` with `–lthread` gives both Solaris threads and pthreads interfaces with a preference given to the POSIX interfaces when the two interfaces conflict.
- When making a library safe for multithreaded use, do not thread global process operations.

  Do not change global operations (or actions with global side effects) to behave in a threaded manner. For example, if file I/O is changed to per-thread operation, threads cannot cooperate in accessing files.

  For thread-specific behavior, or *thread cognizant* behavior, use thread facilities. For example, when the termination of `main( )` should terminate only the thread that is exiting `main( )`, the end of `main( )` should be:

  ```
  thr_exit();
   /*NOTREACHED*/
  ```

Creating and Using Threads

The threads packages will cache the threads data structure, stacks, and LWPs so that the repetitive creation of unbound threads can be inexpensive.

Unbound thread creation is very inexpensive when compared to process creation or even to bound thread creation. In fact, the cost is similar to unbound thread synchronization when you include the context switches to stop one thread and start another.

So, creating and destroying threads as they are required is usually better than attempting to manage a pool of threads that wait for independent work.

A good example of this is an RPC server that creates a thread for each request and destroys it when the reply is delivered, instead of trying to maintain a pool of threads to service requests.

While thread creation is relatively inexpensive when compared to process creation, it is not inexpensive when compared to the cost of a few instructions. Create threads for processing that lasts at least a couple of thousand machine instructions.

Lightweight Processes

Figure 10–1 illustrates the relationship between LWPs and the user and kernel levels.

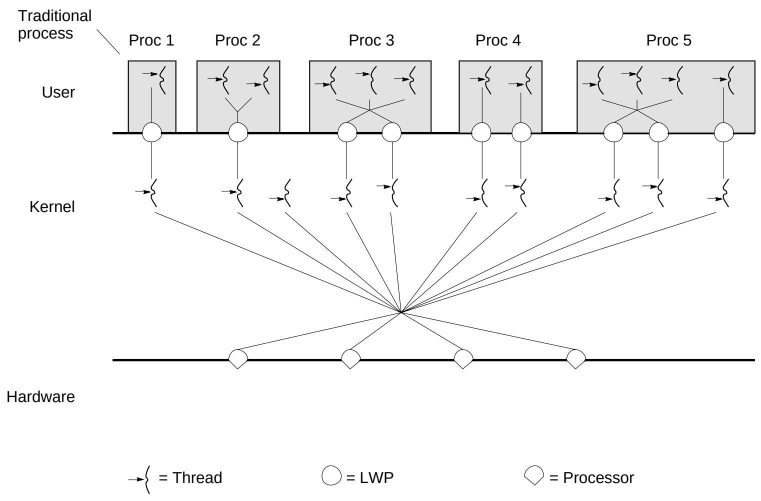


Figure 10–1 Multithreading Levels and Relationships

The user-level threads library, with help from the programmer and the operating system, ensures that the number of LWPs available is adequate for the currently active user-level threads. However, there is no one-to-one mapping between user threads and LWPs, and user-level threads can freely migrate from one LWP to another.

With Solaris threads, a programmer can tell the threads library how many threads should be “running” at the same time.

For example, if the programmer says that up to three threads should run at the same time, then at least three LWPs should be available. If there are three available processors, the threads run in parallel. If there is only one processor, then the operating system multiplexes the three LWPs on that one processor. If all the LWPs block, the threads library adds another LWP to the pool.

When a user thread blocks due to synchronization, its LWP transfers to another runnable thread. This transfer is done with a coroutine linkage and not with a system call.

The operating system decides which LWP should run on which processor and when. It has no knowledge about what user threads are or how many are active in each process.

The kernel schedules LWPs onto CPU resources according to their scheduling classes and priorities. The threads library schedules threads on the process pool of LWPs in much the same way.

Each LWP is independently dispatched by the kernel, performs independent system calls, incurs independent page faults, and runs in parallel on a multiprocessor system.

An LWP has some capabilities that are not exported directly to threads, such as a special scheduling class.

Unbound Threads

The library invokes LWPs as needed and assigns them to execute runnable threads. The LWP assumes the state of the thread and executes its instructions. If the thread becomes blocked on a synchronization mechanism, or if another thread should be run, the thread state is saved in process memory and the threads library assigns another thread to the LWP to run.

Bound Threads

Sometimes having more threads than LWPs, as can happen with unbound threads, is a disadvantage.

For example, a parallel array computation divides the rows of its arrays among different threads. If there is one LWP for each processor, but multiple threads for each LWP, each processor spends time switching between threads. In this case, it is better to have one thread for each LWP, divide the rows among a smaller number of threads, and reduce the number of thread switches.

A mixture of threads that are permanently bound to LWPs and unbound threads is also appropriate for some applications.

An example of this is a realtime application that has some threads with system-wide priority and realtime scheduling, and other threads that attend to background computations. Another example is a window system with unbound threads for most operations and a mouse serviced by a high-priority, bound, realtime thread.

When a user-level thread issues a system call, the LWP running the thread calls into the kernel and remains attached to the thread at least until the system call completes.

Bound threads are more expensive than unbound threads. Because bound threads can change the attributes of the underlying LWP, the LWPs are not cached when the bound threads exit. Instead, the operating system provides a new LWP when a bound thread is created and destroys it when the bound thread exits.

Use bound threads only when a thread needs resources that are available only through the underlying LWP, such as a virtual time interval timer or an alternate stack, or when the thread must be visible to the kernel to be scheduled with respect to all other active threads in the system, as in realtime scheduling.

Use unbound threads even when you expect all threads to be active simultaneously. This allows Solaris threads to efficiently cache LWP and thread resources so that thread creation and destruction are fast. Use `thr_setconcurrency(3T)` to tell Solaris threads how many threads you expect to be simultaneously active.

Thread Concurrency (Solaris Threads Only)

By default, Solaris threads attempts to adjust the system execution resources (LWPs) used to run unbound threads to match the real number of active threads. While the Solaris threads package cannot make perfect decisions, it at least ensures that the process continues to make progress.

When you have some idea of the number of unbound threads that should be simultaneously active (executing code or system calls), tell the library through `thr_setconcurrency`(3T).

For example:

- A database server that has a thread for each user should tell Solaris threads the expected number of simultaneously active users.
- A window server that has one thread for each client should tell Solaris threads the expected number of simultaneously active clients.
- A file copy program that has one reader thread and one writer thread should tell Solaris threads that the desired concurrency level is two.

Alternatively, the concurrency level can be incremented by one through the `THR_NEW_LWP` flag as each thread is created.

Include unbound threads blocked on interprocess (`USYNC_PROCESS`) synchronization variables as active when you compute thread concurrency. Exclude bound threads—they do not require concurrency support from Solaris threads because they are equivalent to LWPs.

Efficiency

A new thread is created with `thr_create(3T)` in less time than an existing thread can be restarted. This means that it is more efficient to create a new thread when one is needed and have it call `thr_exit(3T)` when it has completed its task than it would be to stockpile an idle thread and restart it.

Thread Creation Guidelines

Here are some simple guidelines for using threads.

- Use threads for independent activities that must do a meaningful amount of work.
- Use Solaris threads to take advantage of CPU concurrency.
- Use bound threads only when absolutely necessary, that is, when some facility of the underlying LWP is required.

Working with Multiprocessors

Multithreading lets you take advantage of multiprocessors, primarily through parallelism and scalability. Programmers should be aware of the differences between the memory models of a multiprocessor and a uniprocessor.

Memory consistency is always from the viewpoint of the processor interrogating memory. For uniprocessors, memory is obviously consistent because there is only one processor viewing memory.

To improve multiprocessor performance, memory consistency is relaxed.You cannot always assume that changes made to memory by one processor are immediately reflected in the other processors' views of that memory.

You can avoid this complexity by using synchronization variables when you use shared or global variables.

Barrier synchronization is sometimes an efficient way to control parallelism on multiprocessors. An example of barriers can be found in Appendix B."

Another multiprocessor issue is efficient synchronization when threads must wait until all have reached a common point in their execution.

Note - The issues discussed here are not important when the threads synchronization primitives are always used to access shared memory locations.

The Underlying Architecture

When threads synchronize access to shared storage locations using the threads synchronization routines, the effect of running a program on a shared-memory multiprocessor is identical to the effect of running the program on a uniprocessor.

However, in many situations a programmer might be tempted to take advantage of the multiprocessor and use "tricks" to avoid the synchronization routines. As Code Example 10–5and Code Example 10–6 show, such tricks can be dangerous.

Understanding the memory models supported by common multiprocessor architectures helps to understand the dangers.

The major multiprocessor components are:

- The processors themselves
- Store buffers, which connect the processors to their caches
- *Caches*, which hold the contents of recently accessed or modified storage locations
- memory, which is the primary storage (and is shared by all processors).

In the simple traditional model, the multiprocessor behaves as if the processors are connected directly to memory: when one processor stores into a location and another immediately loads from the same location, the second processor loads what was stored by the first.

Caches can be used to speed the average memory access, and the desired semantics can be achieved when the caches are kept consistent with one another.

A problem with this simple approach is that the processor must often be delayed to make certain that the desired semantics are achieved. Many modern multiprocessors use various techniques to prevent such delays, which, unfortunately, change the semantics of the memory model.

Two of these techniques and their effects are explained in the next two examples.

"Shared-Memory" Multiprocessors

Consider the purported solution to the producer/consumer problem shown in Code Example 10–5.

Although this program works on current SPARC-based multiprocessors, it assumes that all multiprocessors have strongly ordered memory. This program is therefore not portable.

CODE EXAMPLE 10–5 The Producer/Consumer Problem—Shared Memory Multiprocessors

```
                    char buffer[BSIZE];
                    unsigned int in = 0;
                    unsigned int out = 0;

void                                 char
producer(char item) {                  consumer(void) {
                                        char item;
    do
        ;/* nothing */                    do
    while                                     ;/* nothing */
        (in - out == BSIZE);               while
                                             (in - out == 0);
    buffer[in%BSIZE] = item;                item = buffer[out%BSIZE];
    in++;                                 out++;
}                                    }
```

When this program has exactly one producer and exactly one consumer and is run on a shared-memory multiprocessor, it appears to be correct. The difference between *in* and *out* is the number of items in the buffer.

The producer waits (by repeatedly computing this difference) until there is room for a new item, and the consumer waits until there is an item in the buffer.

For memory that is *strongly ordered* (for instance, a modification to memory on one processor is immediately available to the other processors), this solution is correct (it is correct even taking into account that *in* and *out* will eventually overflow, as long as `BSIZE` is less than the largest integer that can be represented in a word).

Shared-memory multiprocessors do not necessarily have strongly ordered memory. A change to memory by one processor is not necessarily available immediately to the other processors. When two changes to different memory locations are made by one processor, the other processors do not necessarily see the changes in the order in which they were made because changes to memory don't happen immediately.

First the changes are stored in *store buffers* that are not visible to the cache.

The processor looks at these store buffers to ensure that a program has a consistent view, but because store buffers are not visible to other processors, a write by one processor doesn't become visible until it is written to cache.

The synchronization primitives (see Chapter 4") use special instructions that flush the store buffers to cache. So, using locks around your shared data ensures memory consistency.

When memory ordering is very relaxed, Code Example 10–5has a problem because the consumer might see that *in* has been incremented by the producer before it sees the change to the corresponding buffer slot.

This is called *weak ordering* because stores made by one processor can appear to happen out of order by another processor (memory, however, is always consistent from the same processor). To fix this, the code should use mutexes to flush the cache.

The trend is toward relaxing memory order. Because of this, programmers are becoming increasingly careful to use locks around all global or shared data.

As demonstrated by Code Example 10–5and Code Example 10–6, locking is essential.

Peterson's Algorithm

The code in Code Example 10–6is an implementation of Peterson's Algorithm, which handles mutual exclusion between two threads. This code tries to guarantee that there is never more than one thread in the critical section and that, when a thread calls `mut_excl()`, it enters the critical section sometime "*soon*."

An assumption here is that a thread exits fairly quickly after entering the critical section.

CODE EXAMPLE 10–6 Mutual Exclusion for Two Threads?

```
void mut_excl(int me /* 0 or 1 */) {
    static int loser;
    static int interested[2] = {0, 0};
    int other; /* local variable */

    other = 1 - me;
    interested[me] = 1;
    loser = me;
    while (loser == me && interested[other])
        ;

    /* critical section */
    interested[me] = 0;
}
```

This algorithm works some of the time when it is assumed that the multiprocessor has strongly ordered memory.

Some multiprocessors, including some SPARC-based multiprocessors, have store buffers. When a thread issues a store instruction, the data is put into a store buffer. The buffer contents are eventually sent to the cache, but not necessarily right away. (Note that the caches on each of the processors maintain a consistent view of memory, but modified data does not reach the cache right away.)

When multiple memory locations are stored into, the changes reach the cache (and memory) in the correct order, but possibly after a delay. SPARC-based multiprocessors with this property are said to have total store order (TSO).

When one processor stores into location A and then loads from location B, and another processor stores into location B and loads from location A, the expectation is that either the first processor fetches the newly modified value in location B or the second processor fetches the newly modified value in location A, or both, but that the case in which both processors load the old values simply cannot happen.

However, with the delays caused by load and store buffers, the “impossible case” can happen.

What could happen with Peterson’s algorithm is that two threads running on separate processors each stores into its own slot of the interested array and then loads from the other slot. They both see the old values (0), assume that the other party is not present, and both enter the critical section. (Note that this is the sort of problem that might not show up when you test a program, but only much later.)

This problem is avoided when you use the threads synchronization primitives, whose implementations issue special instructions to force the writing of the store buffers to the cache.

Parallelizing a Loop on a Shared-Memory Parallel Computer

In many applications, and especially numerical applications, while part of the algorithm can be parallelized, other parts are inherently sequential, as shown in the following:

```
Thread1                                Thread2 through Threadn

while(many_iterations) {               while(many_iterations) {

    sequential_computation
    --- Barrier ---                        --- Barrier ---
    parallel_computation                   parallel_computation
}                                      }
```

For example, you might produce a set of matrices with a strictly linear computation, then perform operations on the matrices using a parallel algorithm, then use the results of these operations to produce another set of matrices, then operate on them in parallel, and so on.

The nature of the parallel algorithms for such a computation is that little synchronization is required during the computation, but synchronization of all the threads employed is required to ensure that the sequential computation is finished before the parallel computation begins.

The barrier forces all the threads that are doing the parallel computation to wait until all threads involved have reached the barrier. When they’ve reached the barrier, they are released and begin computing together.

Summary

This guide has covered a wide variety of important threads programming issues. Look in Appendix A”for a pthreads program example that uses many of the features and styles that have been discussed. Look in Appendix B”for a program example that uses Solaris threads.

Further Reading

For more in-depth information about multithreading, see the following book:

- *Programming with Threads* by Steve Kleiman, Devang Shah, and Bart Smaalders (Prentice-Hall, to be published in 1995)

APPENDIX **A**

Sample Application – Multithreaded `grep`

Description of `tgrep`

The `tgrep` sample program is a multithreaded version of find(1) combined with grep(1). `tgrep` supports all but the `–w` (word search) options of the normal `grep`, and a few exclusively available options.

By default, the `tgrep` searches are like the following command:

```
find . -exec grep [ options ] pattern {} \;
```

For large directory hierarchies, `tgrep` gets results more quickly than the `find` command, depending on the number of processors available. On uniprocessor machines it is about twice as fast, and on four processor machines it is about four times as fast.

The `–e` option changes the way `tgrep` interprets the pattern string. Ordinarily (without the `–e` option) `tgrep` uses a literal string match. With the `–e` option, `tgrep` uses an MT-Safe public domain version of a regular expression handler. The regular expression method is slower.

The `–B` option tells `tgrep` to use the value of the environment variable called `TGLIMIT` to limit the number of threads it will use during a search. This option has no affect if `TGLIMIT` is not set. Because `tgrep` can use a lot of system resources, this is a way to run it politely on a timesharing system.

Getting Online Source Code

Source for `tgrep` is included on the Catalyst Developer's CD. Contact your sales representative to find out how you can get a copy.

A copy might also be available on the World Wide Web (WWW) at the following URL:

http://www.sun.com/sunsoft/Products/Developer-products/sig/threads/

Only the multithreaded `main.c` module appears here. Other modules, including those for regular expression handling, plus documentation and Makefiles, might be available from the sources listed above.

CODE EXAMPLE A–1 Source Code for `tgrep` Program

```
/* Copyright (c) 1993, 1994  Ron Winacott                               */
/* This program may be used, copied, modified, and redistributed freely */
/* for ANY purpose, so long as this notice remains intact.              */

#define _REENTRANT

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
#include <sys/stat.h>
#include <dirent.h>

#include "version.h"

#include <fcntl.h>
#include <sys/uio.h>
#include <pthread.h>
#include <sched.h>

#ifdef MARK
#include <prof.h> /* to turn on MARK(), use -DMARK to compile (see man prof5)*/
#endif

#include "pmatch.h"

#define PATH_MAX                1024 /* max # of characters in a path name */
#define HOLD_FDS                6  /* stdin,out,err and a buffer */
#define UNLIMITED               99999 /* The default tglimit */
#define MAXREGEXP               10  /* max number of -e options */

#define FB_BLOCK                0x00001
#define FC_COUNT                0x00002
#define FH_HOLDNAME             0x00004
```

```
#define FI_IGNCASE              0x00008
#define FL_NAMEONLY             0x00010
#define FN_NUMBER               0x00020
#define FS_NOERROR              0x00040
#define FV_REVERSE              0x00080
#define FW_WORD                 0x00100
#define FR_RECUR                0x00200
#define FU_UNSORT               0x00400
#define FX_STDIN                0x00800
#define TG_BATCH                0x01000
#define TG_FILEPAT              0x02000
#define FE_REGEXP               0x04000
#define FS_STATS                0x08000
#define FC_LINE                 0x10000
#define TG_PROGRESS             0x20000

#define FILET                   1
#define DIRT                    2

typedef struct work_st {
    char                *path;
    int                 tp;
    struct work_st      *next;
} work_t;

typedef struct out_st {
    char                *line;
    int                 line_count;
    long                byte_count;
    struct out_st       *next;
} out_t;

#define ALPHASIZ        128
typedef struct bm_pattern {     /* Boyer - Moore pattern                */
        short            p_m;           /* length of pattern string     */
        short            p_r[ALPHASIZ]; /* "r" vector                   */
        short           *p_R;           /* "R" vector                   */
        char            *p_pat;         /* pattern string               */
} BM_PATTERN;

/* bmpmatch.c */
extern BM_PATTERN *bm_makepat(char *p);
extern char *bm_pmatch(BM_PATTERN *pat, register char *s);
extern void bm_freepat(BM_PATTERN *pattern);
BM_PATTERN      *bm_pat;  /* the global target read only after main */

/* pmatch.c */
extern char *pmatch(register PATTERN *pattern, register char *string, int *len);
extern PATTERN *makepat(char *string, char *metas);
extern void freepat(register PATTERN *pat);
extern void printpat(PATTERN *pat);
PATTERN         *pm_pat[MAXREGEXP];  /* global targets read only for pmatch */

#include "proto.h"  /* function prototypes of main.c */

/* local functions to POSIX only */
void pthread_setconcurrency_np(int con);
int pthread_getconcurrency_np(void);
void pthread_yield_np(void);
```

```
pthread_attr_t  detached_attr;
pthread_mutex_t output_print_lk;
pthread_mutex_t global_count_lk;

int             global_count = 0;

work_t          *work_q = NULL;
pthread_cond_t  work_q_cv;
pthread_mutex_t work_q_lk;
pthread_mutex_t debug_lock;

#include "debug.h"  /* must be included AFTER the
                        mutex_t debug_lock line */

work_t          *search_q = NULL;
pthread_mutex_t search_q_lk;
pthread_cond_t  search_q_cv;
int             search_pool_cnt = 0;    /* the count in the pool now */
int             search_thr_limit = 0;   /* the max in the pool */

work_t          *cascade_q = NULL;
pthread_mutex_t cascade_q_lk;
pthread_cond_t  cascade_q_cv;
int             cascade_pool_cnt = 0;
int             cascade_thr_limit = 0;

int             running = 0;
pthread_mutex_t running_lk;

pthread_mutex_t stat_lk;
time_t          st_start = 0;
int             st_dir_search = 0;
int             st_file_search = 0;
int             st_line_search = 0;
int             st_cascade = 0;
int             st_cascade_pool = 0;
int             st_cascade_destroy = 0;
int             st_search = 0;
int             st_pool = 0;
int             st_maxrun = 0;
int             st_worknull = 0;
int             st_workfds = 0;
int             st_worklimit = 0;
int             st_destroy = 0;

int             all_done = 0;
int             work_cnt = 0;
int             current_open_files = 0;
int             tglimit = UNLIMITED;    /* if -B limit the number of
                                   threads */
int             progress_offset = 1;
int             progress = 0;  /* protected by the print_lock ! */
unsigned int    flags = 0;
int             regexp_cnt = 0;
char            *string[MAXREGEXP];
int             debug = 0;
int             use_pmatch = 0;
char            file_pat[255];  /* file patten match */
PATTERN         *pm_file_pat; /* compiled file target string (pmatch()) */
```

```
/*
 * Main: This is where the fun starts
 */
int
main(int argc, char **argv)
{
    int         c,out_thr_flags;
    long        max_open_files = 0l, ncpus = 0l;
    extern int  optind;
    extern char *optarg;
    int         prio = 0;
    struct stat sbuf;
    pthread_t tid,dtid;
    void        *status;
    char        *e = NULL, *d = NULL; /* for debug flags */
    int         debug_file = 0;
    struct sigaction sigact;
    sigset_t    set,oset;
    int         err = 0, i = 0, pm_file_len = 0;
    work_t      *work;
    int         restart_cnt = 10;

    /* NO OTHER THREADS ARE RUNNING */
    flags = FR_RECUR;  /* the default */

    while ((c = getopt(argc, argv, "d:e:bchilnsvwruf:p:BCSZzHP:")) != EOF) {
        switch (c) {
#ifdef DEBUG
        case ’d’:
            debug = atoi(optarg);
            if (debug == 0)
                debug_usage();

            d = optarg;
            fprintf(stderr,"tgrep: Debug on at level(s) ");
            while (*d) {
                for (i=0; i<9; i++)
                    if (debug_set[i].level == *d) {
                        debug_levels |= debug_set[i].flag;
                        fprintf(stderr,"%c ",debug_set[i].level);
                        break;
                    }
                d++;
            }
            fprintf(stderr,"\n");
            break;
        case ’f’: debug_file = atoi(optarg); break;
#endif      /* DEBUG */

        case ’B’:
            flags |= TG_BATCH;
#ifndef __lock_lint
        /* locklint complains here, but there are no other threads */
            if ((e = getenv("TGLIMIT"))) {
                tglimit = atoi(e);
            }
            else {
                if (!(flags & FS_NOERROR))  /* order dependent! */
                    fprintf(stderr,"env TGLIMIT not set, overriding -B\n");
                flags &= ~TG_BATCH;
            }
#endif
            break;
        case ’p’:
            flags |= TG_FILEPAT;
            strcpy(file_pat,optarg);
            pm_file_pat = makepat(file_pat,NULL);
            break;
        case ’P’:
            flags |= TG_PROGRESS;
            progress_offset = atoi(optarg);
            break;
        case ’S’: flags |= FS_STATS;    break;
        case ’b’: flags |= FB_BLOCK;    break;
        case ’c’: flags |= FC_COUNT;    break;
        case ’h’: flags |= FH_HOLDNAME; break;
        case ’i’: flags |= FI_IGNCASE;  break;
        case ’l’: flags |= FL_NAMEONLY; break;
        case ’n’: flags |= FN_NUMBER;   break;
        case ’s’: flags |= FS_NOERROR;  break;
        case ’v’: flags |= FV_REVERSE;  break;
        case ’w’: flags |= FW_WORD;     break;
        case ’r’: flags &= ~FR_RECUR;   break;
        case ’C’: flags |= FC_LINE;     break;
        case ’e’:
            if (regexp_cnt == MAXREGEXP) {
                fprintf(stderr,"Max number of regexp’s (%d) exceeded!\n",
                        MAXREGEXP);
                exit(1);
            }
            flags |= FE_REGEXP;
            if ((string[regexp_cnt] =(char *)malloc(strlen(optarg)+1))==NULL){
                fprintf(stderr,"tgrep: No space for search string(s)\n");
                exit(1);
            }
            memset(string[regexp_cnt],0,strlen(optarg)+1);
            strcpy(string[regexp_cnt],optarg);
            regexp_cnt++;
            break;
        case ’z’:
        case ’Z’: regexp_usage();
            break;
        case ’H’:
        case ’?’:
        default : usage();
        }
    }
    if (flags & FS_STATS)
        st_start = time(NULL);

    if (!(flags & FE_REGEXP)) {
       if (argc - optind < 1) {
            fprintf(stderr,"tgrep: Must supply a search string(s) "
                    "and file list or directory\n");
            usage();
        }
        if ((string[0]=(char *)malloc(strlen(argv[optind])+1))==NULL){
            fprintf(stderr,"tgrep: No space for search string(s)\n");
            exit(1);
        }
        memset(string[0],0,strlen(argv[optind])+1);
        strcpy(string[0],argv[optind]);
        regexp_cnt=1;
        optind++;
    }

    if (flags & FI_IGNCASE)
        for (i=0; i<regexp_cnt; i++)
            uncase(string[i]);

    if (flags & FE_REGEXP) {
        for (i=0; i<regexp_cnt; i++)
            pm_pat[i] = makepat(string[i],NULL);
        use_pmatch = 1;
    }
    else {
        bm_pat = bm_makepat(string[0]); /* only one allowed */
    }

    flags |= FX_STDIN;


    max_open_files = sysconf(_SC_OPEN_MAX);
    ncpus = sysconf(_SC_NPROCESSORS_ONLN);
    if ((max_open_files - HOLD_FDS - debug_file) < 1) {
        fprintf(stderr,"tgrep: You MUST have at least ONE fd "
                "that can be used, check limit (>10)\n");
        exit(1);
    }
    search_thr_limit = max_open_files - HOLD_FDS - debug_file;
    cascade_thr_limit = search_thr_limit / 2;
    /* the number of files that can be open */
    current_open_files = search_thr_limit;

    pthread_attr_init(&detached_attr);
    pthread_attr_setdetachstate(&detached_attr,
        PTHREAD_CREATE_DETACHED);

    pthread_mutex_init(&global_count_lk,NULL);
    pthread_mutex_init(&output_print_lk,NULL);
    pthread_mutex_init(&work_q_lk,NULL);
    pthread_mutex_init(&running_lk,NULL);
    pthread_cond_init(&work_q_cv,NULL);
    pthread_mutex_init(&search_q_lk,NULL);
    pthread_cond_init(&search_q_cv,NULL);
    pthread_mutex_init(&cascade_q_lk,NULL);
    pthread_cond_init(&cascade_q_cv,NULL);

    if ((argc == optind) && ((flags & TG_FILEPAT) || (flags & FR_RECUR))) {
        add_work(".",DIRT);
        flags = (flags & ~FX_STDIN);
    }
    for ( ; optind < argc; optind++) {
        restart_cnt = 10;
        flags = (flags & ~FX_STDIN);
      STAT_AGAIN:
        if (stat(argv[optind], &sbuf)) {
            if (errno == EINTR) { /* try again !, restart */
                if (--restart_cnt)
                    goto STAT_AGAIN;
            }
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can’t stat file/dir %s, %s\n",
                        argv[optind], strerror(errno));
            continue;
        }
        switch (sbuf.st_mode & S_IFMT) {
        case S_IFREG :
            if (flags & TG_FILEPAT) {
                if (pmatch(pm_file_pat, argv[optind], &pm_file_len))
                    DP(DLEVEL1,("File pat match %s\n",argv[optind]));
                    add_work(argv[optind],FILET);
            }
            else {
                add_work(argv[optind],FILET);
            }
            break;
        case S_IFDIR :
            if (flags & FR_RECUR) {
                add_work(argv[optind],DIRT);
            }
            else {
                if (!(flags & FS_NOERROR))
                    fprintf(stderr,"tgrep: Can’t search directory %s, "
                            "-r option is on. Directory ignored.\n",
                            argv[optind]);
            }
            break;
        }
    }

    pthread_setconcurrency_np(3);

    if (flags & FX_STDIN) {
        fprintf(stderr,"tgrep: stdin option is not coded at this time\n");
        exit(0);                        /* XXX Need to fix this SOON */
        search_thr(NULL);
        if (flags & FC_COUNT) {
            pthread_mutex_lock(&global_count_lk);
            printf("%d\n",global_count);
            pthread_mutex_unlock(&global_count_lk);
        }
        if (flags & FS_STATS)
            prnt_stats();
        exit(0);
    }

    pthread_mutex_lock(&work_q_lk);
    if (!work_q) {
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: No files to search.\n");
        exit(0);
    }
    pthread_mutex_unlock(&work_q_lk);

    DP(DLEVEL1,("Starting to loop through the work_q for work\n"));

    /* OTHER THREADS ARE RUNNING */
    while (1) {
        pthread_mutex_lock(&work_q_lk);
```

```
while ((work_q == NULL || current_open_files == 0 || tglimit <= 0) &&
       all_done == 0) {
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        if (work_q == NULL)
            st_worknull++;
        if (current_open_files == 0)
            st_workfds++;
        if (tglimit <= 0)
            st_worklimit++;
        pthread_mutex_unlock(&stat_lk);
    }
    pthread_cond_wait(&work_q_cv,&work_q_lk);
}
if (all_done != 0) {
    pthread_mutex_unlock(&work_q_lk);
    DP(DLEVEL1,("All_done was set to TRUE\n"));
    goto OUT;
}
work = work_q;
work_q = work->next;  /* maybe NULL */
work->next = NULL;
current_open_files--;
pthread_mutex_unlock(&work_q_lk);

tid = 0;
switch (work->tp) {
case DIRT:
    pthread_mutex_lock(&cascade_q_lk);
    if (cascade_pool_cnt) {
        if (flags & FS_STATS) {
            pthread_mutex_lock(&stat_lk);
            st_cascade_pool++;
            pthread_mutex_unlock(&stat_lk);
        }
        work->next = cascade_q;
        cascade_q = work;
        pthread_cond_signal(&cascade_q_cv);
        pthread_mutex_unlock(&cascade_q_lk);
        DP(DLEVEL2,("Sent work to cascade pool thread\n"));
    }
    else {
        pthread_mutex_unlock(&cascade_q_lk);
        err = pthread_create(&tid,&detached_attr,cascade,(void *)work);
        DP(DLEVEL2,("Sent work to new cascade thread\n"));
        if (flags & FS_STATS) {
            pthread_mutex_lock(&stat_lk);
            st_cascade++;
            pthread_mutex_unlock(&stat_lk);
        }
    }
    break;
case FILET:
    pthread_mutex_lock(&search_q_lk);
    if (search_pool_cnt) {
        if (flags & FS_STATS) {
            pthread_mutex_lock(&stat_lk);
            st_pool++;
            pthread_mutex_unlock(&stat_lk);
        }
```

```
                work->next = search_q;  /* could be null */
                search_q = work;
                pthread_cond_signal(&search_q_cv);
                pthread_mutex_unlock(&search_q_lk);
                DP(DLEVEL2,("Sent work to search pool thread\n"));
            }
            else {
                pthread_mutex_unlock(&search_q_lk);
                err = pthread_create(&tid,&detached_attr,
                                     search_thr,(void *)work);
                pthread_setconcurrency_np(pthread_getconcurrency_np()+1);
                DP(DLEVEL2,("Sent work to new search thread\n"));
                if (flags & FS_STATS) {
                    pthread_mutex_lock(&stat_lk);
                    st_search++;
                    pthread_mutex_unlock(&stat_lk);
                }
            }
            break;
        default:
            fprintf(stderr,"tgrep: Internal error, work_t->tp not valid\n");
            exit(1);
        }
        if (err) {  /* NEED TO FIX THIS CODE. Exiting is just wrong */
            fprintf(stderr,"Could not create new thread!\n");
            exit(1);
        }
    }

 OUT:
    if (flags & TG_PROGRESS) {
        if (progress)
            fprintf(stderr,".\n");
        else
            fprintf(stderr,"\n");
    }
    /* we are done, print the stuff. All other threads are parked */
    if (flags & FC_COUNT) {
        pthread_mutex_lock(&global_count_lk);
        printf("%d\n",global_count);
        pthread_mutex_unlock(&global_count_lk);
    }
    if (flags & FS_STATS)
        prnt_stats();
    return(0); /* should have a return from main */
}

/*
 * Add_Work: Called from the main thread, and cascade threads to add file
 * and directory names to the work Q.
 */
int
add_work(char *path,int tp)
{
    work_t      *wt,*ww,*wp;

    if ((wt = (work_t *)malloc(sizeof(work_t))) == NULL)
        goto ERROR;
    if ((wt->path = (char *)malloc(strlen(path)+1)) == NULL)
        goto ERROR;
    strcpy(wt->path,path);
    wt->tp = tp;
    wt->next = NULL;
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        if (wt->tp == DIRT)
            st_dir_search++;
        else
            st_file_search++;
        pthread_mutex_unlock(&stat_lk);
    }
    pthread_mutex_lock(&work_q_lk);
    work_cnt++;
    wt->next = work_q;
    work_q = wt;
    pthread_cond_signal(&work_q_cv);
    pthread_mutex_unlock(&work_q_lk);
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Could not add %s to work queue. Ignored\n",
                path);
    return(-1);
}

/*
 * Search thread: Started by the main thread when a file name is found
 * on the work Q to be serached. If all the needed resources are ready
 * a new search thread will be created.
 */
void *
search_thr(void *arg) /* work_t *arg */
{
    FILE        *fin;
    char        fin_buf[(BUFSIZ*4)];  /* 4 Kbytes */
    work_t      *wt,std;
    int         line_count;
    char        rline[128];
    char        cline[128];
    char        *line;
    register char *p,*pp;
    int            pm_len;
    int         len = 0;
    long        byte_count;
    long        next_line;
    int         show_line;  /* for the -v option */
    register int slen,plen,i;
    out_t       *out = NULL;    /* this threads output list */

    pthread_yield_np();
    wt = (work_t *)arg; /* first pass, wt is passed to use. */

    /* len = strlen(string);*/  /* only set on first pass */

    while (1) {  /* reuse the search threads */
        /* init all back to zero */
        line_count = 0;
        byte_count = 0l;
        next_line = 0l;
```

```
show_line = 0;

pthread_mutex_lock(&running_lk);
running++;
pthread_mutex_unlock(&running_lk);
pthread_mutex_lock(&work_q_lk);
tglimit--;
pthread_mutex_unlock(&work_q_lk);
DP(DLEVEL5,("searching file (STDIO) %s\n",wt->path));

if ((fin = fopen(wt->path,"r")) == NULL) {
    if (!(flags & FS_NOERROR)) {
        fprintf(stderr,"tgrep: %s. File \"%s\" not searched.\n",
                strerror(errno),wt->path);
    }
    goto ERROR;
}
setvbuf(fin,fin_buf,_IOFBF,(BUFSIZ*4));  /* XXX */
DP(DLEVEL5,("Search thread has opened file %s\n",wt->path));
while ((fgets(rline,127,fin)) != NULL) {
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        st_line_search++;
        pthread_mutex_unlock(&stat_lk);
    }
    slen = strlen(rline);
    next_line += slen;
    line_count++;
    if (rline[slen-1] == '\n')
        rline[slen-1] = '\0';
    /*
    ** If the uncase flag is set, copy the read in line (rline)
    ** To the uncase line (cline) Set the line pointer to point at
    ** cline.
    ** If the case flag is NOT set, then point line at rline.
    ** line is what is compared, rline is what is printed on a
    ** match.
    */
    if (flags & FI_IGNCASE) {
        strcpy(cline,rline);
        uncase(cline);
        line = cline;
    }
    else {
        line = rline;
    }
    show_line = 1;  /* assume no match, if -v set */
    /* The old code removed */
    if (use_pmatch) {
        for (i=0; i<regexp_cnt; i++) {
            if (pmatch(pm_pat[i], line, &pm_len)) {
                if (!(flags & FV_REVERSE)) {
                    add_output_local(&out,wt,line_count,
                                     byte_count,rline);
                    continue_line(rline,fin,out,wt,
                                  &line_count,&byte_count);
                }
                else {
                    show_line = 0;
                } /* end of if -v flag if / else block */
```

```
                    /*
                    ** if we get here on ANY of the regexp targets
                    ** jump out of the loop, we found a single
                    ** match so do not keep looking!
                    ** If name only, do not keep searcthing the same
                    ** file, we found a single match, so close the file,
                    ** print the file name and move on to the next file.
                    */
                    if (flags & FL_NAMEONLY)
                        goto OUT_OF_LOOP;
                    else
                        goto OUT_AND_DONE;
                } /* end found a match if block */
            } /* end of the for pat[s] loop */
        }
        else {
            if (bm_pmatch( bm_pat, line)) {
                if (!(flags & FV_REVERSE)) {
                    add_output_local(&out,wt,line_count,byte_count,rline);
                    continue_line(rline,fin,out,wt,
                                  &line_count,&byte_count);
                }
                else {
                    show_line = 0;
                }
                if (flags & FL_NAMEONLY)
                    goto OUT_OF_LOOP;
            }
        }
      OUT_AND_DONE:
        if ((flags & FV_REVERSE) && show_line) {
            add_output_local(&out,wt,line_count,byte_count,rline);
            show_line = 0;
        }
        byte_count = next_line;
    }
  OUT_OF_LOOP:
    fclose(fin);
    /*
    ** The search part is done, but before we give back the FD,
    ** and park this thread in the search thread pool, print the
    ** local output we have gathered.
    */
    print_local_output(out,wt);  /* this also frees out nodes */
    out = NULL; /* for the next time around, if there is one */
ERROR:
    DP(DLEVEL5,("Search done for %s\n",wt->path));
    free(wt->path);
    free(wt);

    notrun();
    pthread_mutex_lock(&search_q_lk);
    if (search_pool_cnt > search_thr_limit) {
        pthread_mutex_unlock(&search_q_lk);
        DP(DLEVEL5,("Search thread exiting\n"));
        if (flags & FS_STATS) {
            pthread_mutex_lock(&stat_lk);
            st_destroy++;
            pthread_mutex_unlock(&stat_lk);
        }
```

```
            return(0);
        }
        else {
            search_pool_cnt++;
            while (!search_q)
                pthread_cond_wait(&search_q_cv,&search_q_lk);
            search_pool_cnt--;
            wt = search_q;  /* we have work to do! */
            if (search_q->next)
                search_q = search_q->next;
            else
                search_q = NULL;
            pthread_mutex_unlock(&search_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Continue line: Special case search with the -C flag set. If you are
 * searching files like Makefiles, some lines might have escape char’s to
 * contine the line on the next line. So the target string can be found, but
 * no data is displayed. This function continues to print the escaped line
 * until there are no more "\" chars found.
 */
int
continue_line(char *rline, FILE *fin, out_t *out, work_t *wt,
              int *lc, long *bc)
{
    int len;
    int cnt = 0;
    char *line;
    char nline[128];

    if (!(flags & FC_LINE))
        return(0);

    line = rline;
  AGAIN:
    len = strlen(line);
    if (line[len-1] == ’\\’) {
        if ((fgets(nline,127,fin)) == NULL) {
            return(cnt);
        }
        line = nline;
        len = strlen(line);
        if (line[len-1] == ’\n’)
            line[len-1] = ’\0’;
        *bc = *bc + len;
        *lc++;
        add_output_local(&out,wt,*lc,*bc,line);
        cnt++;
        goto AGAIN;
    }
    return(cnt);
}

/*
 * cascade: This thread is started by the main thread when directory names
 * are found on the work Q. The thread reads all the new file, and directory
 * names from the directory it was started when and adds the names to the
 * work Q. (it finds more work!)
 */

void *
cascade(void *arg)  /* work_t *arg */
{
    char        fullpath[1025];
    int         restart_cnt = 10;
    DIR         *dp;

    char        dir_buf[sizeof(struct dirent) + PATH_MAX];
    struct dirent *dent = (struct dirent *)dir_buf;
    struct stat   sbuf;
    char        *fpath;
    work_t      *wt;
    int         fl = 0, dl = 0;
    int         pm_file_len = 0;

    pthread_yield_np();  /* try toi give control back to main thread */
    wt = (work_t *)arg;

    while(1) {
        fl = 0;
        dl = 0;
        restart_cnt = 10;
        pm_file_len = 0;

        pthread_mutex_lock(&running_lk);
        running++;
        pthread_mutex_unlock(&running_lk);
        pthread_mutex_lock(&work_q_lk);
        tglimit--;
        pthread_mutex_unlock(&work_q_lk);

        if (!wt) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Bad work node passed to cascade\n");
            goto DONE;
        }
        fpath = (char *)wt->path;
        if (!fpath) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Bad path name passed to cascade\n");
            goto DONE;
        }
        DP(DLEVEL3,("Cascading on %s\n",fpath));
        if (( dp = opendir(fpath)) == NULL) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can’t open dir %s, %s. Ignored.\n",
                        fpath,strerror(errno));
            goto DONE;
        }
        while ((readdir_r(dp,dent)) != NULL) {
            restart_cnt = 10;  /* only try to restart the interupted 10 X */

            if (dent->d_name[0] == ’.’) {
                if (dent->d_name[1] == ’.’ && dent->d_name[2] == ’\0’)
                    continue;
                if (dent->d_name[1] == ’\0’)
                    continue;
            }

            fl = strlen(fpath);
            dl = strlen(dent->d_name);
            if ((fl + 1 + dl) > 1024) {
                fprintf(stderr,"tgrep: Path %s/%s is too long. "
                        "MaxPath = 1024\n",
                        fpath, dent->d_name);
                continue;  /* try the next name in this directory */
            }
            strcpy(fullpath,fpath);
            strcat(fullpath,"/");
            strcat(fullpath,dent->d_name);

          RESTART_STAT:
            if (stat(fullpath,&sbuf)) {
                if (errno == EINTR) {
                    if (--restart_cnt)
                        goto RESTART_STAT;
                }
                if (!(flags & FS_NOERROR))
                    fprintf(stderr,"tgrep: Can’t stat file/dir %s, %s. "
                            "Ignored.\n",
                            fullpath,strerror(errno));
                goto ERROR;
            }

            switch (sbuf.st_mode & S_IFMT) {
            case S_IFREG :
                if (flags & TG_FILEPAT) {
                    if (pmatch(pm_file_pat, dent->d_name, &pm_file_len)) {
                        DP(DLEVEL3,("file pat match (cascade) %s\n",
                                    dent->d_name));
                        add_work(fullpath,FILET);
                    }
                }
                else {
                    add_work(fullpath,FILET);
                    DP(DLEVEL3,("cascade added file (MATCH) %s to Work Q\n",
                                fullpath));
                }
                break;

            case S_IFDIR :
                DP(DLEVEL3,("cascade added dir %s to Work Q\n",fullpath));
                add_work(fullpath,DIRT);
                break;
            }
        }

      ERROR:
        closedir(dp);

      DONE:
        free(wt->path);
        free(wt);
        notrun();
        pthread_mutex_lock(&cascade_q_lk);
        if (cascade_pool_cnt > cascade_thr_limit) {
            pthread_mutex_unlock(&cascade_q_lk);
            DP(DLEVEL5,("Cascade thread exiting\n"));
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                st_cascade_destroy++;
                pthread_mutex_unlock(&stat_lk);
            }
            return(0); /* pthread_exit */
        }
        else {
            DP(DLEVEL5,("Cascade thread waiting in pool\n"));
            cascade_pool_cnt++;
            while (!cascade_q)
                pthread_cond_wait(&cascade_q_cv,&cascade_q_lk);
            cascade_pool_cnt--;
            wt = cascade_q;  /* we have work to do! */
            if (cascade_q->next)
                cascade_q = cascade_q->next;
            else
                cascade_q = NULL;
            pthread_mutex_unlock(&cascade_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Print Local Output: Called by the search thread after it is done searching
 * a single file. If any oputput was saved (matching lines), the lines are
 * displayed as a group on stdout.
 */
int
print_local_output(out_t *out, work_t *wt)
{
    out_t       *pp, *op;
    int         out_count = 0;
    int         printed = 0;

    pp = out;
    pthread_mutex_lock(&output_print_lk);
    if (pp && (flags & TG_PROGRESS)) {
        progress++;
        if (progress >= progress_offset) {
            progress = 0;
            fprintf(stderr,".");
        }
    }
    while (pp) {
        out_count++;
        if (!(flags & FC_COUNT)) {
            if (flags & FL_NAMEONLY) {  /* Pint name ONLY ! */
                if (!printed) {
                    printed = 1;
                    printf("%s\n",wt->path);
                }
            }
            else {  /* We are printing more then just the name */
                if (!(flags & FH_HOLDNAME))
                    printf("%s :",wt->path);
                if (flags & FB_BLOCK)
```

```
                    printf("%ld:",pp->byte_count/512+1);
                if (flags & FN_NUMBER)
                    printf("%d:",pp->line_count);
                printf("%s\n",pp->line);
            }
        }
        op = pp;
        pp = pp->next;
        /* free the nodes as we go down the list */
        free(op->line);
        free(op);
    }

    pthread_mutex_unlock(&output_print_lk);
    pthread_mutex_lock(&global_count_lk);
    global_count += out_count;
    pthread_mutex_unlock(&global_count_lk);
    return(0);
}

/*
 * add output local: is called by a search thread as it finds matching lines.
 * the matching line, its byte offset, line count, etc. are stored until the
 * search thread is done searching the file, then the lines are printed as
 * a group. This way the lines from more then a single file are not mixed
 * together.
 */

int
add_output_local(out_t **out, work_t *wt,int lc, long bc, char *line)
{
    out_t       *ot,*oo, *op;

    if (( ot = (out_t *)malloc(sizeof(out_t))) == NULL)
        goto ERROR;
    if (( ot->line = (char *)malloc(strlen(line)+1)) == NULL)
        goto ERROR;

    strcpy(ot->line,line);
    ot->line_count = lc;
    ot->byte_count = bc;

    if (!*out) {
        *out = ot;
        ot->next = NULL;
        return(0);
    }
    /* append to the END of the list; keep things sorted! */
    op = oo = *out;
    while(oo) {
        op = oo;
        oo = oo->next;
    }
    op->next = ot;
    ot->next = NULL;
    return(0);

 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Output lost. No space. "
                "[%s: line %d byte %d match : %s\n",
                wt->path,lc,bc,line);
    return(1);
}

/*
 * print stats: If the -S flag is set, after ALL files have been searched,
 * main thread calls this function to print the stats it keeps on how the
 * search went.
 */

void
prnt_stats(void)
{
    float a,b,c;
    float t = 0.0;
    time_t  st_end = 0;
    char    tl[80];

    st_end = time(NULL); /* stop the clock */
    printf("\n----------------- Tgrep Stats. --------------------\n");
    printf("Number of directories searched:           %d\n",st_dir_search);
    printf("Number of files searched:                 %d\n",st_file_search);
    c = (float)(st_dir_search + st_file_search) / (float)(st_end - st_start);
    printf("Dir/files per second:                     %3.2f\n",c);
    printf("Number of lines searched:                 %d\n",st_line_search);
    printf("Number of matching lines to target:       %d\n",global_count);

    printf("Number of cascade threads created:        %d\n",st_cascade);
    printf("Number of cascade threads from pool:      %d\n",st_cascade_pool);
    a = st_cascade_pool; b = st_dir_search;
    printf("Cascade thread pool hit rate:             %3.2f%%\n",((a/b)*100));
    printf("Cascade pool overall size:                %d\n",cascade_pool_cnt);
    printf("Number of search threads created:         %d\n",st_search);
    printf("Number of search threads from pool:       %d\n",st_pool);
    a = st_pool; b = st_file_search;
    printf("Search thread pool hit rate:              %3.2f%%\n",((a/b)*100));
    printf("Search pool overall size:                 %d\n",search_pool_cnt);
    printf("Search pool size limit:                   %d\n",search_thr_limit);
    printf("Number of search threads destroyed:       %d\n",st_destroy);

    printf("Max # of threads running concurenly:      %d\n",st_maxrun);
    printf("Total run time, in seconds.               %d\n",
           (st_end - st_start));

    /* Why did we wait ? */
    a = st_workfds; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    printf("Work stopped due to no FD’s:  (%.3d)       %d Times, %3.2f%%\n",
           search_thr_limit,st_workfds,c);
    a = st_worknull; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    printf("Work stopped due to no work on Q:         %d Times, %3.2f%%\n",
           st_worknull,c);
    if (tglimit == UNLIMITED)
        strcpy(tl,"Unlimited");
    else
        sprintf(tl,"   %.3d   ",tglimit);
    a = st_worklimit; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    printf("Work stopped due to TGLIMIT:  (%.9s) %d Times, %3.2f%%\n",
           tl,st_worklimit,c);
    printf("Work continued to be handed out:          %3.2f%%\n",100.00-t);
    printf("----------------------------------------------------\n");
}
/*
 * not running: A glue function to track if any search threads or cascade
 * threads are running. When the count is zero, and the work Q is NULL,
 * we can safely say, WE ARE DONE.
 */
void
notrun (void)
{
    pthread_mutex_lock(&work_q_lk);
    work_cnt--;
    tglimit++;
    current_open_files++;
    pthread_mutex_lock(&running_lk);
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        if (running > st_maxrun) {
            st_maxrun = running;
            DP(DLEVEL6,("Max Running has increased to %d\n",st_maxrun));
        }
        pthread_mutex_unlock(&stat_lk);
    }
    running--;
    if (work_cnt == 0 && running == 0) {
        all_done = 1;
        DP(DLEVEL6,("Setting ALL_DONE flag to TRUE.\n"));
    }
    pthread_mutex_unlock(&running_lk);
    pthread_cond_signal(&work_q_cv);
    pthread_mutex_unlock(&work_q_lk);
}

/*
 * uncase: A glue function. If the -i (case insensitive) flag is set, the
 * target strng and the read in line is converted to lower case before
 * comparing them.
 */
void
uncase(char *s)
{
    char        *p;

    for (p = s; *p != NULL; p++)
        *p = (char)tolower(*p);
}

/*
 * usage: Have to have one of these.
 */

void
usage(void)
{
    fprintf(stderr,"usage: tgrep <options> pattern <{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Where:\n");
```

```
#ifdef DEBUG
    fprintf(stderr,"Debug     -d = debug level -d <levels> (-d0 for usage)\n");
    fprintf(stderr,"Debug     -f = block fd’s from use (-f #)\n");
#endif
    fprintf(stderr,"          -b = show block count (512 byte block)\n");
    fprintf(stderr,"          -c = print only a line count\n");
    fprintf(stderr,"          -h = Do NOT print file names\n");
    fprintf(stderr,"          -i = case insensitive\n");
    fprintf(stderr,"          -l = print file name only\n");
    fprintf(stderr,"          -n = print the line number with the line\n");
    fprintf(stderr,"          -s = Suppress error messages\n");
    fprintf(stderr,"          -v = print all but matching lines\n");
#ifdef NOT_IMP
    fprintf(stderr,"          -w = search for a \"word\"\n");
#endif
    fprintf(stderr,"          -r = Do not search for files in all "
                                "sub-directories\n");
    fprintf(stderr,"          -C = show continued lines (\"\\\")\n");
    fprintf(stderr,"          -p = File name regexp pattern. (Quote it)\n");
    fprintf(stderr,"          -P = show progress. -P 1 prints a DOT on stderr\n"
                   "               for each file it finds, -P 10 prints a DOT\n"
                   "               on stderr for each 10 files it finds, etc...\n");
    fprintf(stderr,"          -e = expression search.(regexp) More then one\n");
    fprintf(stderr,"          -B = limit the number of threads to TGLIMIT\n");
    fprintf(stderr,"          -S = Print thread stats when done.\n");
    fprintf(stderr,"          -Z = Print help on the regexp used.\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Notes:\n");
    fprintf(stderr,"      If you start tgrep with only a directory name\n");
    fprintf(stderr,"      and no file names, you must not have the -r option\n");
    fprintf(stderr,"      set or you will get no output.\n");
    fprintf(stderr,"      To search stdin (piped input), you must set -r\n");
    fprintf(stderr,"      Tgrep will search ALL files in ALL \n");
    fprintf(stderr,"      sub-directories. (like */* */*/* */*/*/* etc..)\n");
    fprintf(stderr,"      if you supply a directory name.\n");
    fprintf(stderr,"      If you do not supply a file, or directory name,\n");
    fprintf(stderr,"      and the -r option is not set, the current \n");
    fprintf(stderr,"      directory \".\" will be used.\n");
    fprintf(stderr,"      All the other options should work \"like\" grep\n");
    fprintf(stderr,"      The -p patten is regexp; tgrep will search only\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"      Copy Right By Ron Winacott, 1993-1995.\n");
    fprintf(stderr,"\n");
    exit(0);
}

/*
 * regexp usage: Tell the world about tgrep custom (THREAD SAFE) regexp!
 */
int
regexp_usage (void)
{
    fprintf(stderr,"usage: tgrep <options> -e \"pattern\" <-e ...> "
            "<{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"metachars:\n");
    fprintf(stderr,"    . - match any character\n");
    fprintf(stderr,"    * - match 0 or more occurrences of previous char\n");
    fprintf(stderr,"    + - match 1 or more occurrences of previous char.\n");
    fprintf(stderr,"    ^ - match at beginning of string\n");
    fprintf(stderr,"    $ - match end of string\n");
    fprintf(stderr,"    [ - start of character class\n");
    fprintf(stderr,"    ] - end of character class\n");
    fprintf(stderr,"    ( - start of a new pattern\n");
    fprintf(stderr,"    ) - end of a new pattern\n");
    fprintf(stderr,"    @(n)c - match <c> at column <n>\n");
    fprintf(stderr,"    | - match either pattern\n");
    fprintf(stderr,"    \\ - escape any special characters\n");
    fprintf(stderr,"    \\c - escape any special characters\n");
    fprintf(stderr,"    \\o - turn on any special characters\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"To match two diffrerent patterns in the same command\n");
    fprintf(stderr,"Use the or function. \n"
            "ie: tgrep -e \"(pat1)|(pat2)\" file\n"
            "This will match any line with \"pat1\" or \"pat2\" in it.\n");
    fprintf(stderr,"You can also use up to %d -e expressions\n",MAXREGEXP);
    fprintf(stderr,"RegExp Pattern matching brought to you by Marc Staveley\n");
    exit(0);
}

/*
 * debug usage: If compiled with -DDEBUG, turn it on, and tell the world
 * how to get tgrep to print debug info on different threads.
 */

#ifdef DEBUG
void
debug_usage(void)
{
    int i = 0;

    fprintf(stderr,"DEBUG usage and levels:\n");
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"Level                   code\n");
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"0                 This message.\n");
    for (i=0; i<9; i++) {
        fprintf(stderr,"%d                 %s\n",i+1,debug_set[i].name);
    }
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"You can or the levels together like -d134 for levels\n");
    fprintf(stderr,"1 and 3 and 4.\n");
    fprintf(stderr,"\n");
    exit(0);
}
#endif

/* Pthreads NP functions */

#ifdef __sun
void
pthread_setconcurrency_np(int con)
{
    thr_setconcurrency(con);
}

int
pthread_getconcurrency_np(void)
{
    return(thr_getconcurrency());
```

```
}

void
pthread_yield_np(void)
{
/*     In Solaris 2.4, these functions always return - 1 and set errno to ENOSYS */
    if (sched_yield())  /* call UI interface if we are older then 2.5 */
        thr_yield();
}

#else
void
pthread_setconcurrency_np(int con)
{
    return;
}

int
pthread_getconcurrency_np(void)
{
    return(0);
}

void
pthread_yield_np(void)
{
    return;
}
#endif
```

APPENDIX **B**

Solaris Threads Example: `barrier.c`

The `barrier.c` program demonstrates an implementation of a barrier for Solaris threads. (See "Parallelizing a Loop on a Shared-Memory Parallel Computer" on page 204 for a definition of barriers.)

CODE EXAMPLE B–1 Solaris Threads Example: `barrier.c`

```
#define _REENTRANT

/* Include Files        */

#include <thread.h>
#include <errno.h>

/* Constants & Macros   *

/* Data Declarations    */

typedef struct {
     int     maxcnt;     /* maximum number of runners */
     struct _sb {
        cond_t  wait_cv;   /* cv for waiters at barrier */
        mutex_t wait_lk;   /* mutex for waiters at barrier */
        int     runners;   /* number of running threads */
     } sb[2];
     struct _sb   *sbp;    /* current sub-barrier */
} barrier_t;




/*
 * barrier_init - initialize a barrier variable.
 *
 */

int
barrier_init( barrier_t *bp, int count, int type, void *arg ) {
     int n;
     int i;

     if (count < 1)
         return(EINVAL);

     bp->maxcnt = count;
     bp->sbp = &bp->sb[0];

     for (i = 0; i < 2; ++i) {
#if defined(__cplusplus)
     struct barrier_t::_sb *sbp = &( bp->sb[i] );
#else
     struct _sb *sbp = &( bp->sb[i] );
#endif
     sbp->runners = count;

     if (n = mutex_init(&sbp->wait_lk, type, arg))
             return(n);

     if (n = cond_init(&sbp->wait_cv, type, arg))
             return(n);
     }
     return(0);
}

/*
 * barrier_wait - wait at a barrier for everyone to arrive.
 *
 */

int
barrier_wait(register barrier_t *bp) {
#if defined(__cplusplus)
     register struct barrier_t::_sb *sbp = bp->sbp;
#else
     register struct _sb *sbp = bp->sbp;
#endif
     mutex_lock(&sbp->wait_lk);

     if (sbp->runners == 1) {   /* last thread to reach barrier */
           if (bp->maxcnt != 1) {
           /* reset runner count and switch sub-barriers */
                  sbp->runners = bp->maxcnt;
                  bp->sbp = (bp->sbp == &bp->sb[0])
                              ? &bp->sb[1] : &bp->sb[0];

                  /* wake up the waiters          */
                  cond_broadcast(&sbp->wait_cv);
            }
        } else {
           sbp->runners--;         /* one less runner  */

           while (sbp->runners != bp->maxcnt)
                  cond_wait( &sbp->wait_cv, &sbp->wait_lk);
        }

     mutex_unlock(&sbp->wait_lk);

     return(0);
}
```

```
/*
 * barrier_destroy - destroy a barrier variable.
 *
 */

int
barrier_destroy(barrier_t *bp) {
        int     n;
        int     i;

        for (i=0; i < 2; ++ i) {
                if (n = cond_destroy(&bp->sb[i].wait_cv))
                        return( n );

                if (n = mutex_destroy( &bp->sb[i].wait_lk))
                        return(n);
        }

        return(0);
}


#define NTHR    4
#define NCOMPUTATION 2
#define NITER   1000
#define NSQRT   1000

        void *
compute(barrier_t *ba )
{
 int count = NCOMPUTATION;

 while (count--) {
  barrier_wait( ba );
  /* do parallel computation */
 }
}

main( int argc, char *argv[] ) {
        int             i;
        int             niter;
        int             nthr;
        barrier_t       ba;
        double          et;
        thread_t        *tid;

        switch ( argc ) {
          default:
          case 3 :      niter   = atoi( argv[1] );
                        nthr    = atoi( argv[2] );
                        break;

          case 2 :      niter   = atoi( argv[1] );
                        nthr    = NTHR;
                        break;

          case 1 :      niter   = NITER;
                        nthr    = NTHR;
                        break;
        }

        barrier_init( &ba, nthr + 1, USYNC_THREAD, NULL );
        tid = (thread_t *) calloc(nthr, sizeof(thread_t));

        for (i = 0; i < nthr; ++i) {
                int     n;

                if (n = thr_create(NULL, 0,
                        (void *(*)( void *)) compute,
                        &ba, NULL, &tid[i])) {
                            errno = n;
                            perror("thr_create");
                            exit(1);
                        }
        }

 for (i = 0; i < NCOMPUTATION; i++) {
         barrier_wait(&ba );
  /* do parallel algorithm */
 }

 for (i = 0; i < nthr; i++) {
  thr_join(tid[i], NULL, NULL);
 }

}
```

APPENDIX **C**

MT Safety Levels: Library Interfaces

Table C–1 lists the safety levels for interfaces from Section 3 of the *man Pages(3): Library Routines* (see “MT Interface Safety Levels” on page 124 for explanations of the safety categories).

TABLE C–1 MT Safety Levels of Library Routines

`a64l(3C)`	MT-Safe
`abort(3C)`	Safe
`abs(3C)`	MT-Safe
`accept(3N)`	Safe
`acos(3M)`	MT-Safe
`acosh(3M)`	MT-Safe
`addch(3X)`	Unsafe
`addchnstr(3X)`	Unsafe
`addchstr(3X)`	Unsafe
`addnstr(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`addnwstr(3X)`	Unsafe
`addsev(3C)`	MT-safe
`addseverity(3C)`	Safe
`addstr(3X)`	Unsafe
`addwch(3X)`	Unsafe
`addwchnstr(3X)`	Unsafe
`addwchstr(3X)`	Unsafe
`addwstr(3X)`	Unsafe
`adjcurspos(3X)`	Unsafe
`advance(3G)`	MT-Safe
`aiocancel(3)`	Unsafe
`aioread(3)`	Unsafe
`aiowait(3)`	Unsafe
`aiowrite(3)`	Unsafe
`aio_cancel(3R)`	MT-Safe
`aio_error(3R)`	Async-Signal-Safe
`aio_fsync(3R)`	MT-Safe
`aio_read(3R)`	MT-Safe
`aio_return(3R)`	Async-Signal-Safe
`aio_suspend(3R)`	Async-Signal-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`aio_write(3R)`	MT-Safe
`alloca(3C)`	Safe
`arc(3)`	Safe
`ascftime(3C)`	MT-Safe
`asctime(3C)`	Unsafe, use `asctime_r()`
`asin(3M)`	MT-Safe
`asinh(3M)`	MT-Safe
`assert(3C)`	Safe
`atan(3M)`	MT-Safe
`atan2(3M)`	MT-Safe
`atanh(3M)`	MT-Safe
`atexit(3C)`	Safe
`atof(3C)`	MT-Safe
`atoi(3C)`	MT-Safe
`atol(3C)`	MT-Safe
`atoll(3C)`	MT-Safe
`attroff(3X)`	Unsafe
`attron(3X)`	Unsafe
`attrset(3X)`	Unsafe
`authdes_create(3N)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`authdes_getucred(3N)`	MT-Safe
`authdes_seccreate(3N)`	MT-Safe
`authkerb_getucred(3N)`	Unsafe
`authkerb_seccreate(3N)`	Unsafe
`authnone_create(3N)`	MT-Safe
`authsys_create(3N)`	MT-Safe
`authsys_create_default(3N)`	MT-Safe
`authunix_create(3N)`	Unsafe
`authunix_create_default(3N)`	Unsafe
`auth_destroy(3N)`	MT-Safe
`au_close(3)`	Safe
`au_open(3)`	Safe
`au_user_mask(3)`	MT-Safe
`au_write(3)`	Safe
`basename(3G)`	MT-Safe
`baudrate(3X)`	Unsafe
`beep(3X)`	Unsafe
`bessel(3M)`	MT-Safe
`bgets(3G)`	MT-Safe
`bind(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`bindtextdomain(3I)`	Safe with exceptions
`bkgd(3X)`	Unsafe
`bkgdset(3X)`	Unsafe
`border(3X)`	Unsafe
`bottom_panel(3X)`	Unsafe
`box(3)`	Safe
`box(3X)`	Unsafe
`bsearch(3C)`	Safe
`bufsplit(3G)`	MT-Safe
`byteorder(3N)`	Safe
`calloc(3C)`	Safe
`calloc(3X)`	Safe
`callrpc(3N)`	Unsafe
`cancellation(3T)`	MT-Safe
`can_change_color(3X)`	Unsafe
`catclose(3C)`	MT-Safe
`catgets(3C)`	MT-Safe
`catopen(3C)`	MT-Safe
`cbc_crypt(3)`	MT-Safe
`cbreak(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`cbrt(3M)`	MT-Safe
`ceil(3M)`	MT-Safe
`cfgetispeed(3)`	MT-Safe, Async-Signal-Safe
`cfgetospeed(3)`	MT-Safe, Async-Signal-Safe
`cfree(3X)`	Safe
`cfsetispeed(3)`	MT-Safe, Async-Signal-Safe
`cfsetospeed(3)`	MT-Safe, Async-Signal-Safe
`cftime(3C)`	MT-Safe
`circle(3)`	Safe
`clear(3X)`	Unsafe
`clearerr(3S)`	MT-Safe
`clearok(3X)`	Unsafe
`clntraw_create(3N)`	Unsafe
`clnttcp_create(3N)`	Unsafe
`clntudp_bufcreate(3N)`	Unsafe
`clntudp_create(3N)`	Unsafe
`clnt_broadcast(3N)`	Unsafe
`clnt_call(3N)`	MT-Safe
`clnt_control(3N)`	MT-Safe
`clnt_create(3N)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`clnt_create_timed(3N)`	MT-Safe
`clnt_create_vers(3N)`	MT-Safe
`clnt_destroy(3N)`	MT-Safe
`clnt_dg_create(3N)`	MT-Safe
`clnt_freeres(3N)`	MT-Safe
`clnt_geterr(3N)`	MT-Safe
`clnt_pcreateerror(3N)`	MT-Safe
`clnt_perrno(3N)`	MT-Safe
`clnt_perror(3N)`	MT-Safe
`clnt_raw_create(3N)`	MT-Safe
`clnt_spcreateerror(3N)`	MT-Safe
`clnt_sperrno(3N)`	MT-Safe
`clnt_sperror(3N)`	MT-Safe
`clnt_tli_create(3N)`	MT-Safe
`clnt_tp_create(3N)`	MT-Safe
`clnt_tp_create_timed(3N)`	MT-Safe
`clnt_vc_create(3N)`	MT-Safe
`clock(3C)`	MT-Safe
`clock_gettime(3R)`	Async-Signal-Safe
`closedir(3C)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`closelog(3)`	Safe
`closepl(3)`	Safe
`closevt(3)`	Safe
`clrtobot(3X)`	Unsafe
`clrtoeol(3X)`	Unsafe
`color_content(3X)`	Unsafe
`compile(3G)`	MT-Safe
`condition(3T)`	MT-Safe
`cond_broadcast(3T)`	MT-Safe
`cond_destroy(3T)`	MT-Safe
`cond_init(3T)`	MT-Safe
`cond_signal(3T)`	MT-Safe
`cond_timedwait(3T)`	MT-Safe
`cond_wait(3T)`	MT-Safe
`confstr(3C)`	MT-Safe
`connect(3N)`	Safe
`cont(3)`	Safe
`conv(3C)`	MT-Safe with exceptions
`copylist(3G)`	MT-Safe
`copysign(3M)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`copywin(3X)`	Unsafe
`cos(3M)`	MT-Safe
`cosh(3M)`	MT-Safe
`crypt(3C)`	Safe
`crypt(3X)`	Unsafe
`cset(3I)`	MT-Safe with exceptions
`csetcol(3I)`	MT-Safe with exceptions
`csetlen(3I)`	MT-Safe with exceptions
`csetno(3I)`	MT-Safe with exceptions
`ctermid(3S)`	Unsafe, use `ctermid_r()`
`ctime(3C)`	Unsafe, use `ctime_r()`
`ctype(3C)`	MT-Safe with exceptions
`current_field(3X)`	Unsafe
`current_item(3X)`	Unsafe
`curses(3X)`	Unsafe
`curs_addch(3X)`	Unsafe
`curs_addchstr(3X)`	Unsafe
`curs_addstr(3X)`	Unsafe
`curs_addwch(3X)`	Unsafe
`curs_addwchstr(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`curs_addwstr(3X)`	Unsafe
`curs_alecompat(3X)`	Unsafe
`curs_attr(3X)`	Unsafe
`curs_beep(3X)`	Unsafe
`curs_bkgd(3X)`	Unsafe
`curs_border(3X)`	Unsafe
`curs_clear(3X)`	Unsafe
`curs_color(3X)`	Unsafe
`curs_delch(3X)`	Unsafe
`curs_deleteln(3X)`	Unsafe
`curs_getch(3X)`	Unsafe
`curs_getstr(3X)`	Unsafe
`curs_getwch(3X)`	Unsafe
`curs_getwstr(3X)`	Unsafe
`curs_getyx(3X)`	Unsafe
`curs_inch(3X)`	Unsafe
`curs_inchstr(3X)`	Unsafe
`curs_initscr(3X)`	Unsafe
`curs_inopts(3X)`	Unsafe
`curs_insch(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`curs_insstr(3X)`	Unsafe
`curs_instr(3X)`	Unsafe
`curs_inswch(3X)`	Unsafe
`curs_inswstr(3X)`	Unsafe
`curs_inwch(3X)`	Unsafe
`curs_inwchstr(3X)`	Unsafe
`curs_inwstr(3X)`	Unsafe
`curs_kernel(3X)`	Unsafe
`curs_move(3X)`	Unsafe
`curs_outopts(3X)`	Unsafe
`curs_overlay(3X)`	Unsafe
`curs_pad(3X)`	Unsafe
`curs_printw(3X)`	Unsafe
`curs_refresh(3X)`	Unsafe
`curs_scanw(3X)`	Unsafe
`curs_scroll(3X)`	Unsafe
`curs_scr_dump(3X)`	Unsafe
`curs_set(3X)`	Unsafe
`curs_slk(3X)`	Unsafe
`curs_termattrs(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`curs_termcap(3X)`	Unsafe
`curs_terminfo(3X)`	Unsafe
`curs_touch(3X)`	Unsafe
`curs_util(3X)`	Unsafe
`curs_window(3X)`	Unsafe
`cuserid(3S)`	MT-Safe
`data_ahead(3X)`	Unsafe
`data_behind(3X)`	Unsafe
`dbm_clearerr(3)`	Unsafe
`dbm_close(3)`	Unsafe
`dbm_delete(3)`	Unsafe
`dbm_error(3)`	Unsafe
`dbm_fetch(3)`	Unsafe
`dbm_firstkey(3)`	Unsafe
`dbm_nextkey(3)`	Unsafe
`dbm_open(3)`	Unsafe
`dbm_store(3)`	Unsafe
`db_add_entry(3N)`	Unsafe
`db_checkpoint(3N)`	Unsafe
`db_create_table(3N)`	Unsafe

`db_destroy_table(3N)`	Unsafe
`db_first_entry(3N)`	Unsafe
`db_free_result(3N)`	Unsafe
`db_initialize(3N)`	Unsafe
`db_list_entries(3N)`	Unsafe
`db_next_entry(3N)`	Unsafe
`db_remove_entry(3N)`	Unsafe
`db_reset_next_entry(3N)`	Unsafe
`db_standby(3N)`	Unsafe
`db_table_exists(3N)`	Unsafe
`db_unload_table(3N)`	Unsafe
`dcgettext(3I)`	Safe with exceptions
`decimal_to_double(3)`	MT-Safe
`decimal_to_extended(3)`	MT-Safe
`decimal_to_floating(3)`	MT-Safe
`decimal_to_quadruple(3)`	MT-Safe
`decimal_to_single(3)`	MT-Safe
`def_prog_mode(3X)`	Unsafe
`def_shell_mode(3X)`	Unsafe
`delay_output(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`delch(3X)`	Unsafe
`deleteln(3X)`	Unsafe
`delscreen(3X)`	Unsafe
`delwin(3X)`	Unsafe
`del_curterm(3X)`	Unsafe
`del_panel(3X)`	Unsafe
`derwin(3X)`	Unsafe
`des_crypt(3)`	MT-Safe
`DES_FAILED(3)`	MT-Safe
`des_failed(3)`	MT-Safe
`des_setparity(3)`	MT-Safe
`dgettext(3I)`	Safe with exceptions
`dial(3N)`	Unsafe
`difftime(3C)`	MT-Safe
`dirname(3G)`	MT-Safe
`div(3C)`	MT-Safe
`dladdr(3X)`	MT-Safe
`dlclose(3X)`	MT-Safe
`dlerror(3X)`	MT-Safe
`dlopen(3X)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`dlsym(3X)`	MT-Safe
`dn_comp(3N)`	Unsafe
`dn_expand(3N)`	Unsafe
`doconfig(3N)`	Unsafe
`double_to_decimal(3)`	MT-Safe
`doupdate(3X)`	Unsafe
`drand48(3C)`	Safe
`dup2(3C)`	Unsafe, Async-Signal-Safe
`dupwin(3X)`	Unsafe
`dup_field(3X)`	Unsafe
`dynamic_field_info(3X)`	Unsafe
`ecb_crypt(3)`	MT-Safe
`echo(3X)`	Unsafe
`echochar(3X)`	Unsafe
`echowchar(3X)`	Unsafe
`econvert(3)`	MT-Safe
`ecvt(3)`	MT-Safe
`ecvt(3C)`	Unsafe
`el(32_fsize.3E)`	Unsafe
`el(32_getehdr.3E)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`el(32_getshdr.3E)`	Unsafe
`el(32_newehdr.3E)`	Unsafe
`el(32_newphdr.3E)`	Unsafe
`el(32_xlatetof.3E)`	Unsafe
`el(32_xlatetom.3E)`	Unsafe
`elf(3E)`	Unsafe
`elf_begin(3E)`	Unsafe
`elf_cntl(3E)`	Unsafe
`elf_end(3E)`	Unsafe
`elf_errmsg(3E)`	Unsafe
`elf_errno(3E)`	Unsafe
`elf_fill(3E)`	Unsafe
`elf_flagdata(3E)`	Unsafe
`elf_flagehdr(3E)`	Unsafe
`elf_flagelf(3E)`	Unsafe
`elf_flagphdr(3E)`	Unsafe
`elf_flagscn(3E)`	Unsafe
`elf_flagshdr(3E)`	Unsafe
`elf_getarhdr(3E)`	Unsafe
`elf_getarsym(3E)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`elf_getbase(3E)`	Unsafe
`elf_getdata(3E)`	Unsafe
`elf_getident(3E)`	Unsafe
`elf_getscn(3E)`	Unsafe
`elf_hash(3E)`	Unsafe
`elf_kind(3E)`	Unsafe
`elf_memory(3E)`	Unsafe
`elf_ndxscn(3E)`	Unsafe
`elf_newdata(3E)`	Unsafe
`elf_newscn(3E)`	Unsafe
`elf_next(3E)`	Unsafe
`elf_nextscn(3E)`	Unsafe
`elf_rand(3E)`	Unsafe
`elf_rawdata(3E)`	Unsafe
`elf_rawfile(3E)`	Unsafe
`elf_strptr(3E)`	Unsafe
`elf_update(3E)`	Unsafe
`elf_version(3E)`	Unsafe
`encrypt(3C)`	Safe
`endac(3)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`endauclass(3)`	MT-Safe
`endauevent(3)`	MT-Safe
`endauuser(3)`	MT-Safe
`endnetconfig(3N)`	MT-Safe
`endnetpath(3N)`	MT-Safe
`endutent(3C)`	Unsafe
`endutxent(3C)`	Unsafe
`endwin(3X)`	Unsafe
`erand48(3C)`	Safe
`erase(3)`	Safe
`erase(3X)`	Unsafe
`erasechar(3X)`	Unsafe
`erf(3M)`	MT-Safe
`erfc(3M)`	MT-Safe
`errno(3C)`	MT-Safe
`ethers(3N)`	MT-Safe
`ether_aton(3N)`	MT-Safe
`ether_hostton(3N)`	MT-Safe
`ether_line(3N)`	MT-Safe
`ether_ntoa(3N)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`ether_ntohost(3N)`	MT-Safe
`euccol(3I)`	Safe
`euclen(3I)`	Safe
`eucscol(3I)`	Safe
`exit(3C)`	Safe
`exp(3M)`	MT-Safe
`expm1(3M)`	MT-Safe
`extended_to_decimal(3)`	MT-Safe
`fabs(3M)`	MT-Safe
`fattach(3C)`	MT-Safe
`fclose(3S)`	MT-Safe
`fconvert(3)`	MT-Safe
`fcvt(3)`	MT-Safe
`fcvt(3C)`	Unsafe
`fdatasync(3R)`	Async-Signal-Safe
`fdetach(3C)`	Unsafe
`fdopen(3S)`	MT-Safe
`feof(3S)`	MT-Safe
`ferror(3S)`	MT-Safe
`fflush(3S)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`ffs(3C)`	MT-Safe
`fgetc(3S)`	MT-Safe
`fgetgrent(3C)`	Unsafe, use `fgetgrent_r()`
`fgetpos(3C)`	MT-Safe
`fgetpwent(3C)`	Unsafe, use `fgetpwent_r()`
`fgets(3S)`	MT-Safe
`fgetspent(3C)`	Unsafe, use `fgetspent_r()`
`fgetwc(3I)`	MT-Safe
`fgetws(3I)`	MT-Safe
`field_arg(3X)`	Unsafe
`field_back(3X)`	Unsafe
`field_buffer(3X)`	Unsafe
`field_count(3X)`	Unsafe
`field_fore(3X)`	Unsafe
`field_index(3X)`	Unsafe
`field_info(3X)`	Unsafe
`field_init(3X)`	Unsafe
`field_just(3X)`	Unsafe
`field_opts(3X)`	Unsafe
`field_opts_off(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`field_opts_on(3X)`	Unsafe
`field_pad(3X)`	Unsafe
`field_status(3X)`	Unsafe
`field_term(3X)`	Unsafe
`field_type(3X)`	Unsafe
`field_userptr(3X)`	Unsafe
`fileno(3S)`	MT-Safe
`file_to_decimal(3)`	MT-Safe
`filter(3X)`	Unsafe
`finite(3C)`	MT-Safe
`flash(3X)`	Unsafe
`floating_to_decimal(3)`	MT-Safe
`flockfile(3S)`	MT-Safe
`floor(3M)`	MT-Safe
`flushinp(3X)`	Unsafe
`fmod(3M)`	MT-Safe
`fmtmsg(3C)`	Safe
`fnmatch(3C)`	MT-Safe
`fn_attribute_add(3N)`	Safe
`fn_attribute_assign(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_attribute_copy(3N)`	Safe
`fn_attribute_create(3N)`	Safe
`fn_attribute_destroy(3N)`	Safe
`fn_attribute_first(3N)`	Safe
`fn_attribute_identifier(3N)`	Safe
`fn_attribute_next(3N)`	Safe
`fn_attribute_remove(3N)`	Safe
`fn_attribute_syntax(3N)`	Safe
`FN_attribute_t(3N)`	Safe
`fn_attribute_valuecount(3N)`	Safe
`fn_attrmodlist_add(3N)`	Safe
`fn_attrmodlist_assign(3N)`	Safe
`fn_attrmodlist_copy(3N)`	Safe
`fn_attrmodlist_count(3N)`	Safe
`fn_attrmodlist_create(3N)`	Safe
`fn_attrmodlist_destroy(3N)`	Safe
`fn_attrmodlist_first(3N)`	Safe
`fn_attrmodlist_next(3N)`	Safe
`FN_attrmodlist_t(3N)`	Safe
`fn_attrset_add(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_attrset_assign(3N)`	Safe
`fn_attrset_copy(3N)`	Safe
`fn_attrset_count(3N)`	Safe
`fn_attrset_create(3N)`	Safe
`fn_attrset_destroy(3N)`	Safe
`fn_attrset_first(3N)`	Safe
`fn_attrset_get(3N)`	Safe
`fn_attrset_next(3N)`	Safe
`fn_attrset_remove(3N)`	Safe
`FN_attrset_t(3N)`	Safe
`fn_attr_get(3N)`	Safe
`fn_attr_get_ids(3N)`	Safe
`fn_attr_get_values(3N)`	Safe
`fn_attr_modify(3N)`	Safe
`fn_attr_multi_get(3N)`	Safe
`fn_attr_multi_modify(3N)`	Safe
`fn_bindinglist_destroy(3N)`	Safe
`fn_bindinglist_next(3N)`	Safe
`FN_bindinglist_t(3N)`	Safe
`fn_composite_name_append_comp(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_composite_name_append_name(3N)`	Safe
`fn_composite_name_assign(3N)`	Safe
`fn_composite_name_copy(3N)`	Safe
`fn_composite_name_count(3N)`	Safe
`fn_composite_name_create(3N)`	Safe
`fn_composite_name_delete_comp(3N)`	Safe
`fn_composite_name_destroy(3N)`	Safe
`fn_composite_name_first(3N)`	Safe
`fn_composite_name_from_string(3N)`	Safe
`fn_composite_name_insert_comp(3N)`	Safe
`fn_composite_name_insert_name(3N)`	Safe
`fn_composite_name_is_empty(3N)`	Safe
`fn_composite_name_is_equal(3N)`	Safe
`fn_composite_name_is_prefix(3N)`	Safe
`fn_composite_name_is_suffix(3N)`	Safe
`fn_composite_name_last(3N)`	Safe
`fn_composite_name_next(3N)`	Safe
`fn_composite_name_prefix(3N)`	Safe
`fn_composite_name_prepend_comp(3N)`	Safe
`fn_composite_name_prepend_name(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_composite_name_prev(3N)`	Safe
`fn_composite_name_suffix(3N)`	Safe
`FN_composite_name_t(3N)`	Safe
`fn_compound_name_append_comp(3N)`	Safe
`fn_compound_name_assign(3N)`	Safe
`fn_compound_name_copy(3N)`	Safe
`fn_compound_name_count(3N)`	Safe
`fn_compound_name_delete_all(3N)`	Safe
`fn_compound_name_delete_comp(3N)`	Safe
`fn_compound_name_destroy(3N)`	Safe
`fn_compound_name_first(3N)`	Safe
`fn_compound_name_from_syntax_attrs`	Safe
`fn_compound_name_get_syntax_attrs(3N)`	Safe
`fn_compound_name_insert_comp(3N)`	Safe
`fn_compound_name_is_empty(3N)`	Safe
`fn_compound_name_is_equal(3N)`	Safe
`fn_compound_name_is_prefix(3N)`	Safe
`fn_compound_name_is_suffix(3N)`	Safe
`fn_compound_name_last(3N)`	Safe
`fn_compound_name_next(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_compound_name_prefix(3N)`	Safe
`fn_compound_name_prepend_comp(3N)`	Safe
`fn_compound_name_prev(3N)`	Safe
`fn_compound_name_suffix(3N)`	Safe
`FN_compound_name_t(3N)`	Safe
`fn_ctx_bind(3N)`	Safe
`fn_ctx_create_subcontext(3N)`	Safe
`fn_ctx_destroy_subcontext(3N)`	Safe
`fn_ctx_get_ref(3N)`	Safe
`fn_ctx_get_syntax_attrs(3N)`	Safe
`fn_ctx_handle_destroy(3N)`	Safe
`fn_ctx_handle_from_initial(3N)`	MT-Safe
`fn_ctx_handle_from_ref(3N)`	Safe
`fn_ctx_list_bindings(3N)`	Safe
`fn_ctx_list_names(3N)`	Safe
`fn_ctx_lookup(3N)`	Safe
`fn_ctx_lookup_link(3N)`	Safe
`fn_ctx_rename(3N)`	Safe
`FN_ctx_t(3N)`	Safe
`fn_ctx_unbind(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_multigetlist_destroy(3N)`	Safe
`fn_multigetlist_next(3N)`	Safe
`FN_multigetlist_t(3N)`	Safe
`fn_namelist_destroy(3N)`	Safe
`fn_namelist_next(3N)`	Safe
`FN_namelist_t(3N)`	Safe
`fn_ref_addrcount(3N)`	Safe
`fn_ref_addr_assign(3N)`	Safe
`fn_ref_addr_copy(3N)`	Safe
`fn_ref_addr_create(3N)`	Safe
`fn_ref_addr_data(3N)`	Safe
`fn_ref_addr_description(3N)`	Safe
`fn_ref_addr_destroy(3N)`	Safe
`fn_ref_addr_length(3N)`	Safe
`FN_ref_addr_t(3N)`	Safe
`fn_ref_addr_type(3N)`	Safe
`fn_ref_append_addr(3N)`	Safe
`fn_ref_assign(3N)`	Safe
`fn_ref_copy(3N)`	Safe
`fn_ref_create(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_ref_create_link(3N)`	Safe
`fn_ref_delete_addr(3N)`	Safe
`fn_ref_delete_all(3N)`	Safe
`fn_ref_description(3N)`	Safe
`fn_ref_destroy(3N)`	Safe
`fn_ref_first(3N)`	Safe
`fn_ref_insert_addr(3N)`	Safe
`fn_ref_is_link(3N)`	Safe
`fn_ref_link_name(3N)`	Safe
`fn_ref_next(3N)`	Safe
`fn_ref_prepend_addr(3N)`	Safe
`FN_ref_t(3N)`	Safe
`fn_ref_type(3N)`	Safe
`fn_status_advance_by_name(3N)`	Safe
`fn_status_append_remaining_name(3N)`	Safe
`fn_status_append_resolved_name(3N)`	Safe
`fn_status_assign(3N)`	Safe
`fn_status_code(3N)`	Safe
`fn_status_copy(3N)`	Safe
`fn_status_create(3N)`	Safe

`fn_status_description(3N)`	Safe
`fn_status_destroy(3N)`	Safe
`fn_status_diagnostic_message(3N)`	Safe
`fn_status_is_success(3N)`	Safe
`fn_status_link_code(3N)`	Safe
`fn_status_link_diagnostic_message(3N)`	Safe
`fn_status_link_remaining_name(3N)`	Safe
`fn_status_link_resolved_name(3N)`	Safe
`fn_status_link_resolved_ref(3N)`	Safe
`fn_status_remaining_name(3N)`	Safe
`fn_status_resolved_name(3N)`	Safe
`fn_status_resolved_ref(3N)`	Safe
`fn_status_set(3N)`	Safe
`fn_status_set_code(3N)`	Safe
`fn_status_set_diagnostic_message(3N)`	Safe
`fn_status_set_link_code(3N)`	Safe
`fn_status_set_link_diagnostic_message`	Safe
`fn_status_set_link_remaining_name(3N)`	Safe
`fn_status_set_link_resolved_name(3N)`	Safe
`fn_status_set_link_resolved_ref(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_status_set_remaining_name(3N)`	Safe
`fn_status_set_resolved_name(3N)`	Safe
`fn_status_set_resolved_ref(3N)`	Safe
`fn_status_set_success(3N)`	Safe
`FN_status_t(3N)`	Safe
`fn_string_assign(3N)`	Safe
`fn_string_bytecount(3N)`	Safe
`fn_string_charcount(3N)`	Safe
`fn_string_code_set(3N)`	Safe
`fn_string_compare(3N)`	Safe
`fn_string_compare_substring(3N)`	Safe
`fn_string_contents(3N)`	Safe
`fn_string_copy(3N)`	Safe
`fn_string_create(3N)`	Safe
`fn_string_destroy(3N)`	Safe
`fn_string_from_composite_name(3N)`	Safe
`fn_string_from_compound_name(3N)`	Safe
`fn_string_from_contents(3N)`	Safe
`fn_string_from_str(3N)`	Safe
`fn_string_from_strings(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fn_string_from_str_n(3N)`	Safe
`fn_string_from_substring(3N)`	Safe
`fn_string_is_empty(3N)`	Safe
`fn_string_next_substring(3N)`	Safe
`fn_string_prev_substring(3N)`	Safe
`fn_string_str(3N)`	Safe
`FN_string_t(3N)`	Safe
`fn_valuelist_destroy(3N)`	Safe
`fn_valuelist_next(3N)`	Safe
`FN_valuelist_t(3N)`	Safe
`fopen(3S)`	MT-Safe
`forms(3X)`	Unsafe
`form_cursor(3X)`	Unsafe
`form_data(3X)`	Unsafe
`form_driver(3X)`	Unsafe
`form_field(3X)`	Unsafe
`form_fields(3X)`	Unsafe
`form_fieldtype(3X)`	Unsafe
`form_field_attributes(3X)`	Unsafe
`form_field_buffer(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`form_field_info(3X)`	Unsafe
`form_field_just(3X)`	Unsafe
`form_field_new(3X)`	Unsafe
`form_field_opts(3X)`	Unsafe
`form_field_userptr(3X)`	Unsafe
`form_field_validation(3X)`	Unsafe
`form_hook(3X)`	Unsafe
`form_init(3X)`	Unsafe
`form_new(3X)`	Unsafe
`form_new_page(3X)`	Unsafe
`form_opts(3X)`	Unsafe
`form_opts_off(3X)`	Unsafe
`form_opts_on(3X)`	Unsafe
`form_page(3X)`	Unsafe
`form_post(3X)`	Unsafe
`form_sub(3X)`	Unsafe
`form_term(3X)`	Unsafe
`form_userptr(3X)`	Unsafe
`form_win(3X)`	Unsafe
`fpclass(3C)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`fpgetmask(3C)`	MT-Safe
`fpgetround(3C)`	MT-Safe
`fpgetsticky(3C)`	MT-Safe
`fprintf(3S)`	MT-Safe except with `setlocale()`
`fpsetmask(3C)`	MT-Safe
`fpsetround(3C)`	MT-Safe
`fpsetsticky(3C)`	MT-Safe
`fputc(3S)`	MT-Safe
`fputs(3S)`	MT-Safe
`fputwc(3I)`	MT-Safe
`fputws(3I)`	MT-Safe
`fread(3S)`	MT-Safe
`free(3C)`	Safe
`free(3X)`	Safe
`freenetconfigent(3N)`	MT-Safe
`free_field(3X)`	Unsafe
`free_fieldtype(3X)`	Unsafe
`free_form(3X)`	Unsafe
`free_item(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`free_menu(3X)`	Unsafe
`freopen(3S)`	MT-Safe
`frexp(3C)`	MT-Safe
`fscanf(3S)`	MT-Safe
`fseek(3S)`	MT-Safe
`fsetpos(3C)`	MT-Safe
`fsync(3C)`	Async-Signal-Safe
`ftell(3S)`	MT-Safe
`ftok(3C)`	MT-Safe
`ftruncate(3C)`	MT-Safe
`ftrylockfile(3S)`	MT-Safe
`ftw(3C)`	Safe
`func_to_decimal(3)`	MT-Safe
`funlockfile(3S)`	MT-Safe
`fwrite(3S)`	MT-Safe
`gconvert(3)`	MT-Safe
`gcvt(3)`	MT-Safe
`gcvt(3C)`	Unsafe
`getacdir(3)`	Safe
`getacflg(3)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`getacinfo(3)`	Safe
`getacmin(3)`	Safe
`getacna(3)`	Safe
`getauclassent(3)`	Unsafe
`getauclassent_r(3)`	MT-Safe
`getauclassnam(3)`	Unsafe
`getauclassnam_r(3)`	MT-Safe
`getauditflags(3)`	MT-Safe
`getauditflagsbin(3)`	MT-Safe
`getauditflagschar(3)`	MT-Safe
`getauevent(3)`	Unsafe
`getauevent_r(3)`	MT-Safe
`getauevnam(3)`	Unsafe
`getauevnam_r(3)`	MT-Safe
`getauevnonam(3)`	MT-Safe
`getauevnum(3)`	Unsafe
`getauevnum_r(3)`	MT-Safe
`getauuserent(3)`	Unsafe
`getauusernam(3)`	Unsafe
`getbegyx(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`getc(3S)`	MT-Safe
`getch(3X)`	Unsafe
`getchar(3S)`	MT-Safe
`getcwd(3C)`	Safe
`getdate(3C)`	MT-Safe
`getenv(3C)`	Safe
`getfauditflags(3)`	MT-Safe
`getgrent(3C)`	Unsafe, use `getgrent_r()`
`getgrgid(3C)`	Unsafe, use `getgrgid_r()`
`getgrnam(3C)`	Unsafe, use `getgrnam_r()`
`gethostbyaddr(3N)`	Unsafe, use `gethostbyaddr_r()`
`gethostbyname(3N)`	Unsafe, use `gethostbyname_r()`
`gethrtime(3C)`	MT-Safe
`gethrvtime(3C)`	MT-Safe
`getlogin(3C)`	Unsafe, use `getlogin_r()`
`getmaxyx(3X)`	Unsafe
`getmntany(3C)`	Safe
`getmntent(3C)`	Safe
`getnetbyaddr(3N)`	Unsafe, use `getnetbyaddr_r()`

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`getnetbyname(3N)`	Unsafe, use `getnetbyname_r()`
`getnetconfig(3N)`	MT-Safe
`getnetconfigent(3N)`	MT-Safe
`getnetgrent(3N)`	Unsafe, use `getnetgrent_r()`
`getnetname(3N)`	MT-Safe
`getnetpath(3N)`	MT-Safe
`getnwstr(3X)`	Unsafe
`getopt(3C)`	Unsafe
`getparyx(3X)`	Unsafe
`getpass(3C)`	Unsafe
`getpeername(3N)`	Safe
`getprotobyname(3N)`	Unsafe, use `getprotobyname_r()`
`getprotobynumber(3N)`	Unsafe, use `getprotobynumber_r()`
`getprotoent(3N)`	Unsafe, use `getprotoent_r()`
`getpublickey(3N)`	Safe
`getpw(3C)`	Safe
`getpwent(3C)`	Unsafe, use `getpwent_r()`
`getpwnam(3C)`	Unsafe, use `getpwnam_r()`
`getpwuid(3C)`	Unsafe, use `getpwuid_r()`

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`getrpcbyname(3N)`	Unsafe, use `getrpcbyname_r()`
`getrpcbynumber(3N)`	Unsafe, use `getrpcbynumber_r()`
`getrpcent(3N)`	Unsafe, use `getrpcent_r()`
`getrpcport(3N)`	Unsafe
`gets(3S)`	MT-Safe
`getsecretkey(3N)`	Safe
`getservbyname(3N)`	Unsafe, use `getservbyname_r()`
`getservbyport(3N)`	Unsafe, use `getservbyport_r()`
`getservent(3N)`	Unsafe, use `getservent_r()`
`getsockname(3N)`	Safe
`getsockopt(3N)`	Safe
`getspent(3C)`	Unsafe, use `getspent_r()`
`getspnam(3C)`	Unsafe, use `getspnam_r()`
`getstr(3X)`	Unsafe
`getsubopt(3C)`	MT-Safe
`getsyx(3X)`	Unsafe
`gettext(3I)`	Safe with exceptions
`gettimeofday(3C)`	MT-Safe
`gettxt(3C)`	Safe with exceptions

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`getutent(3C)`	Unsafe
`getutid(3C)`	Unsafe
`getutline(3C)`	Unsafe
`getutmp(3C)`	Unsafe
`getutmpx(3C)`	Unsafe
`getutxent(3C)`	Unsafe
`getutxid(3C)`	Unsafe
`getutxline(3C)`	Unsafe
`getvfsany(3C)`	Safe
`getvfsent(3C)`	Safe
`getvfsfile(3C)`	Safe
`getvfsspec(3C)`	Safe
`getw(3S)`	MT-Safe
`getwc(3I)`	MT-Safe
`getwch(3X)`	Unsafe
`getwchar(3I)`	MT-Safe
`getwidth(3I)`	MT-Safe with exceptions
`getwin(3X)`	Unsafe
`getws(3I)`	MT-Safe
`getwstr(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`getyx(3X)`	Unsafe
`get_myaddress(3N)`	Unsafe
`gmatch(3G)`	MT-Safe
`gmtime(3C)`	Unsafe, use `gmtime_r()`
`grantpt(3C)`	Safe
`gsignal(3C)`	Unsafe
`halfdelay(3X)`	Unsafe
`hasmntopt(3C)`	Safe
`has_colors(3X)`	Unsafe
`has_ic(3X)`	Unsafe
`has_il(3X)`	Unsafe
`havedisk(3N)`	MT-Safe
`hcreate(3C)`	Safe
`hdestroy(3C)`	Safe
`hide_panel(3X)`	Unsafe
`host2netname(3N)`	MT-Safe
`hsearch(3C)`	Safe
`htonl(3N)`	Safe
`htons(3N)`	Safe
`hyperbolic(3M)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`hypot(3M)`	MT-Safe
`iconv(3)`	MT-Safe
`iconv_close(3)`	MT-Safe
`iconv_open(3)`	MT-Safe
`idcok(3X)`	Unsafe
`idlok(3X)`	Unsafe
`ieee_functions(3M)`	MT-Safe
`ieee_test(3M)`	MT-Safe
`ilogb(3M)`	MT-Safe
`immedok(3X)`	Unsafe
`inch(3X)`	Unsafe
`inchnstr(3X)`	Unsafe
`inchstr(3X)`	Unsafe
`inet(3N)`	Safe
`inet_addr(3N)`	Safe
`inet_lnaof(3N)`	Safe
`inet_makeaddr(3N)`	Safe
`inet_netof(3N)`	Safe
`inet_network(3N)`	Safe
`inet_ntoa(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`initgroups(3C)`	Unsafe
`initscr(3X)`	Unsafe
`init_color(3X)`	Unsafe
`init_pair(3X)`	Unsafe
`innstr(3X)`	Unsafe
`innwstr(3X)`	Unsafe
`insch(3X)`	Unsafe
`insdelln(3X)`	Unsafe
`insertln(3X)`	Unsafe
`insnstr(3X)`	Unsafe
`insnwstr(3X)`	Unsafe
`insque(3C)`	Unsafe
`insstr(3X)`	Unsafe
`instr(3X)`	Unsafe
`inswch(3X)`	Unsafe
`inswstr(3X)`	Unsafe
`intrflush(3X)`	Unsafe
`inwch(3X)`	Unsafe
`inwchnstr(3X)`	Unsafe
`inwchstr(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`inwstr(3X)`	Unsafe
`isalnum(3C)`	MT-Safe with exceptions
`isalpha(3C)`	MT-Safe with exceptions
`isascii(3C)`	MT-Safe with exceptions
`isastream(3C)`	MT-Safe
`iscntrl(3C)`	MT-Safe with exceptions
`isdigit(3C)`	MT-Safe with exceptions
`isencrypt(3G)`	MT-Safe
`isendwin(3X)`	Unsafe
`isenglish(3I)`	MT-Safe with exceptions
`isgraph(3C)`	MT-Safe with exceptions
`isideogram(3I)`	MT-Safe with exceptions
`islower(3C)`	MT-Safe with exceptions
`isnan(3C)`	MT-Safe
`isnan(3M)`	MT-Safe
`isnand(3C)`	MT-Safe
`isnanf(3C)`	MT-Safe
`isnumber(3I)`	MT-Safe with exceptions
`isphonogram(3I)`	MT-Safe with exceptions
`isprint(3C)`	MT-Safe with exceptions

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`ispunct(3C)`	MT-Safe with exceptions
`isspace(3C)`	MT-Safe with exceptions
`isspecial(3I)`	MT-Safe with exceptions
`isupper(3C)`	MT-Safe with exceptions
`iswalnum(3I)`	MT-Safe with exceptions
`iswalpha(3I)`	MT-Safe with exceptions
`iswascii(3I)`	MT-Safe with exceptions
`iswcntrl(3I)`	MT-Safe with exceptions
`iswctype(3I)`	MT-Safe
`iswdigit(3I)`	MT-Safe with exceptions
`iswgraph(3I)`	MT-Safe with exceptions
`iswlower(3I)`	MT-Safe with exceptions
`iswprint(3I)`	MT-Safe with exceptions
`iswpunct(3I)`	MT-Safe with exceptions
`iswspace(3I)`	MT-Safe with exceptions
`iswupper(3I)`	MT-Safe with exceptions
`iswxdigit(3I)`	MT-Safe with exceptions
`isxdigit(3C)`	MT-Safe with exceptions
`is_linetouched(3X)`	Unsafe
`is_wintouched(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`item_count(3X)`	Unsafe
`item_description(3X)`	Unsafe
`item_index(3X)`	Unsafe
`item_init(3X)`	Unsafe
`item_name(3X)`	Unsafe
`item_opts(3X)`	Unsafe
`item_opts_off(3X)`	Unsafe
`item_opts_on(3X)`	Unsafe
`item_term(3X)`	Unsafe
`item_userptr(3X)`	Unsafe
`item_value(3X)`	Unsafe
`item_visible(3X)`	Unsafe
`j0(3M)`	MT-Safe
`j1(3M)`	MT-Safe
`jn(3M)`	MT-Safe
`jrand48(3C)`	Safe
`kerberos(3N)`	Unsafe
`kerberos_rpc(3N)`	Unsafe
`keyname(3X)`	Unsafe
`keypad(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`key_decryptsession(3N)`	MT-Safe
`key_encryptsession(3N)`	MT-Safe
`key_gendes(3N)`	MT-Safe
`key_secretkey_is_set(3N)`	MT-Safe
`key_setsecret(3N)`	MT-Safe
`killchar(3X)`	Unsafe
`krb_get_admhst(3N)`	Unsafe
`krb_get_cred(3N)`	Unsafe
`krb_get_krbhst(3N)`	Unsafe
`krb_get_lrealm(3N)`	Unsafe
`krb_get_phost(3N)`	Unsafe
`krb_kntoln(3N)`	Unsafe
`krb_mk_err(3N)`	Unsafe
`krb_mk_req(3N)`	Unsafe
`krb_mk_safe(3N)`	Unsafe
`krb_net_read(3N)`	Unsafe
`krb_net_write(3N)`	Unsafe
`krb_rd_err(3N)`	Unsafe
`krb_rd_req(3N)`	Unsafe
`krb_rd_safe(3N)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`krb_realmofhost(3N)`	Unsafe
`krb_recvauth(3N)`	Unsafe
`krb_sendauth(3N)`	Unsafe
`krb_set_key(3N)`	Unsafe
`krb_set_tkt_string(3N)`	Unsafe
`kvm_close(3K)`	Unsafe
`kvm_getcmd(3K)`	Unsafe
`kvm_getproc(3K)`	Unsafe
`kvm_getu(3K)`	Unsafe
`kvm_kread(3K)`	Unsafe
`kvm_kwrite(3K)`	Unsafe
`kvm_nextproc(3K)`	Unsafe
`kvm_nlist(3K)`	Unsafe
`kvm_open(3K)`	Unsafe
`kvm_read(3K)`	Unsafe
`kvm_setproc(3K)`	Unsafe
`kvm_uread(3K)`	Unsafe
`kvm_uwrite(3K)`	Unsafe
`kvm_write(3K)`	Unsafe
`l64a(3C)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`label(3)`	Safe
`labs(3C)`	MT-Safe
`lckpwdf(3C)`	MT-Safe
`lcong48(3C)`	Safe
`ldexp(3C)`	MT-Safe
`ldiv(3C)`	MT-Safe
`leaveok(3X)`	Unsafe
`lfind(3C)`	Safe
`lfmt(3C)`	MT-safe
`lgamma(3M)`	Unsafe, use `lgamma_r()`
`libpthread(3T)`	Fork1-Safe,MT-Safe,Async-Signal-Safe
`libthread(3T)`	Fork1-Safe,MT-Safe,Async-Signal-Safe
`line(3)`	Safe
`link_field(3X)`	Unsafe
`link_fieldtype(3X)`	Unsafe
`linmod(3)`	Safe
`lio_listio(3R)`	MT-Safe
`listen(3N)`	Safe
`llabs(3C)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`lldiv(3C)`	MT-Safe
`lltostr(3C)`	MT-Safe
`localeconv(3C)`	Safe with exceptions
`localtime(3C)`	Unsafe, use `localtime_r()`
`lockf(3C)`	MT-Safe
`log(3M)`	MT-Safe
`log10(3M)`	MT-Safe
`log1p(3M)`	MT-Safe
`logb(3C)`	MT-Safe
`logb(3M)`	MT-Safe
`longjmp(3C)`	Unsafe
`longname(3X)`	Unsafe
`lrand48(3C)`	Safe
`lsearch(3C)`	Safe
`madvise(3)`	MT-Safe
`maillock(3X)`	Unsafe
`major(3C)`	MT-Safe
`makecontext(3C)`	MT-Safe
`makedev(3C)`	MT-Safe
`mallinfo(3X)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`malloc(3C)`	Safe
`malloc(3X)`	Safe
`mallopt(3X)`	Safe
`mapmalloc(3X)`	Safe
`matherr(3M)`	MT-Safe
`mbchar(3C)`	MT-Safe with exceptions
`mblen(3C)`	MT-Safe with exceptions
`mbstowcs(3C)`	MT-Safe with exceptions
`mbstring(3C)`	MT-Safe with exceptions
`mbtowc(3C)`	MT-Safe with exceptions
`media_findname(3X)`	MT-Unsafe
`media_getattr(3X)`	MT-Safe
`media_setattr(3X)`	MT-Safe
`memalign(3C)`	Safe
`memccpy(3C)`	MT-Safe
`memchr(3C)`	MT-Safe
`memcmp(3C)`	MT-Safe
`memcpy(3C)`	MT-Safe
`memmove(3C)`	MT-Safe
`memory(3C)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`memset(3C)`	MT-Safe
`menus(3X)`	Unsafe
`menu_attributes(3X)`	Unsafe
`menu_back(3X)`	Unsafe
`menu_cursor(3X)`	Unsafe
`menu_driver(3X)`	Unsafe
`menu_fore(3X)`	Unsafe
`menu_format(3X)`	Unsafe
`menu_grey(3X)`	Unsafe
`menu_hook(3X)`	Unsafe
`menu_init(3X)`	Unsafe
`menu_items(3X)`	Unsafe
`menu_item_current(3X)`	Unsafe
`menu_item_name(3X)`	Unsafe
`menu_item_new(3X)`	Unsafe
`menu_item_opts(3X)`	Unsafe
`menu_item_userptr(3X)`	Unsafe
`menu_item_value(3X)`	Unsafe
`menu_item_visible(3X)`	Unsafe
`menu_mark(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`menu_new(3X)`	Unsafe
`menu_opts(3X)`	Unsafe
`menu_opts_off(3X)`	Unsafe
`menu_opts_on(3X)`	Unsafe
`menu_pad(3X)`	Unsafe
`menu_pattern(3X)`	Unsafe
`menu_post(3X)`	Unsafe
`menu_sub(3X)`	Unsafe
`menu_term(3X)`	Unsafe
`menu_userptr(3X)`	Unsafe
`menu_win(3X)`	Unsafe
`meta(3X)`	Unsafe
`minor(3C)`	MT-Safe
`mkdirp(3G)`	MT-Safe
`mkfifo(3C)`	MT-Safe, Async-Signal-Safe
`mktemp(3C)`	Safe
`mktime(3C)`	Unsafe
`mlock(3C)`	MT-Safe
`monitor(3C)`	Safe
`move(3)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`move(3X)`	Unsafe
`movenextch(3X)`	Unsafe
`moveprevch(3X)`	Unsafe
`move_field(3X)`	Unsafe
`move_panel(3X)`	Unsafe
`mq_close(3R)`	MT-Safe
`mq_getattr(3R)`	MT-Safe
`mq_notify(3R)`	MT-Safe
`mq_open(3R)`	MT-Safe
`mq_receive(3R)`	MT-Safe
`mq_send(3R)`	MT-Safe
`mq_setattr(3R)`	MT-Safe
`mq_unlink(3R)`	MT-Safe
`mrand48(3C)`	Safe
`msync(3C)`	MT-Safe
`munlock(3C)`	MT-Safe
`munlockall(3C)`	MT-Safe
`mutex(3T)`	MT-Safe
`mutex_destroy(3T)`	MT-Safe
`mutex_init(3T)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`mutex_lock(3T)`	MT-Safe
`mutex_trylock(3T)`	MT-Safe
`mutex_unlock(3T)`	MT-Safe
`mvaddch(3X)`	Unsafe
`mvaddchnstr(3X)`	Unsafe
`mvaddchstr(3X)`	Unsafe
`mvaddnstr(3X)`	Unsafe
`mvaddnwstr(3X)`	Unsafe
`mvaddstr(3X)`	Unsafe
`mvaddwch(3X)`	Unsafe
`mvaddwchnstr(3X)`	Unsafe
`mvaddwchstr(3X)`	Unsafe
`mvaddwstr(3X)`	Unsafe
`mvcur(3X)`	Unsafe
`mvdelch(3X)`	Unsafe
`mvderwin(3X)`	Unsafe
`mvgetch(3X)`	Unsafe
`mvgetnwstr(3X)`	Unsafe
`mvgetstr(3X)`	Unsafe
`mvgetwch(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`mvgetwstr(3X)`	Unsafe
`mvinch(3X)`	Unsafe
`mvinchnstr(3X)`	Unsafe
`mvinchstr(3X)`	Unsafe
`mvinnstr(3X)`	Unsafe
`mvinnwstr(3X)`	Unsafe
`mvinsch(3X)`	Unsafe
`mvinsnstr(3X)`	Unsafe
`mvinsnwstr(3X)`	Unsafe
`mvinsstr(3X)`	Unsafe
`mvinstr(3X)`	Unsafe
`mvinswch(3X)`	Unsafe
`mvinswstr(3X)`	Unsafe
`mvinwch(3X)`	Unsafe
`mvinwchnstr(3X)`	Unsafe
`mvinwchstr(3X)`	Unsafe
`mvinwstr(3X)`	Unsafe
`mvprintw(3X)`	Unsafe
`mvscanw(3X)`	Unsafe
`mvwaddch(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`mvwaddchnstr(3X)`	Unsafe
`mvwaddchstr(3X)`	Unsafe
`mvwaddnstr(3X)`	Unsafe
`mvwaddnwstr(3X)`	Unsafe
`mvwaddstr(3X)`	Unsafe
`mvwaddwch(3X)`	Unsafe
`mvwaddwchnstr(3X)`	Unsafe
`mvwaddwchstr(3X)`	Unsafe
`mvwaddwstr(3X)`	Unsafe
`mvwdelch(3X)`	Unsafe
`mvwgetch(3X)`	Unsafe
`mvwgetnwstr(3X)`	Unsafe
`mvwgetstr(3X)`	Unsafe
`mvwgetwch(3X)`	Unsafe
`mvwgetwstr(3X)`	Unsafe
`mvwin(3X)`	Unsafe
`mvwinch(3X)`	Unsafe
`mvwinchnstr(3X)`	Unsafe
`mvwinchstr(3X)`	Unsafe
`mvwinnstr(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`mvwinnwstr(3X)`	Unsafe
`mvwinsch(3X)`	Unsafe
`mvwinsnstr(3X)`	Unsafe
`mvwinsnwstr(3X)`	Unsafe
`mvwinsstr(3X)`	Unsafe
`mvwinstr(3X)`	Unsafe
`mvwinswch(3X)`	Unsafe
`mvwinswstr(3X)`	Unsafe
`mvwinwch(3X)`	Unsafe
`mvwinwchnstr(3X)`	Unsafe
`mvwinwchstr(3X)`	Unsafe
`mvwinwstr(3X)`	Unsafe
`mvwprintw(3X)`	Unsafe
`mvwscanw(3X)`	Unsafe
`nanosleep(3R)`	MT-Safe
`napms(3X)`	Unsafe
`nc_perror(3N)`	MT-Safe
`nc_sperror(3N)`	MT-Safe
`ndbm(3)`	Unsafe
`netdir(3N)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`netdir_free(3N)`	MT-Safe
`netdir_getbyaddr(3N)`	MT-Safe
`netdir_getbyname(3N)`	MT-Safe
`netdir_mergeaddr(3N)`	MT-Safe
`netdir_options(3N)`	MT-Safe
`netdir_perror(3N)`	MT-Safe
`netdir_sperror(3N)`	MT-Safe
`netname2host(3N)`	MT-Safe
`netname2user(3N)`	MT-Safe
`newpad(3X)`	Unsafe
`newterm(3X)`	Unsafe
`newwin(3X)`	Unsafe
`new_field(3X)`	Unsafe
`new_fieldtype(3X)`	Unsafe
`new_form(3X)`	Unsafe
`new_item(3X)`	Unsafe
`new_menu(3X)`	Unsafe
`new_page(3X)`	Unsafe
`new_panel(3X)`	Unsafe
`nextafter(3C)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`nextafter(3M)`	MT-Safe
`nftw(3C)`	Safe with exceptions
`nis_add(3N)`	MT-Safe
`nis_clone_object(3N)`	Safe
`nis_creategroup(3N)`	MT-Safe
`nis_db(3N)`	Unsafe
`nis_destroygroup(3N)`	MT-Safe
`nis_destroy_object(3N)`	Safe
`nis_dir_cmp(3N)`	Safe
`nis_domain_of(3N)`	Safe
`nis_error(3N)`	Safe
`nis_first_entry(3N)`	MT-Safe
`nis_freenames(3N)`	Safe
`nis_freeresult(3N)`	MT-Safe
`nis_freeservlist(3N)`	MT-Safe
`nis_freetags(3N)`	MT-Safe
`nis_getnames(3N)`	Safe
`nis_getservlist(3N)`	MT-Safe
`nis_groups(3N)`	MT-Safe
`nis_ismember(3N)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`nis_leaf_of(3N)`	Safe
`nis_lerror(3N)`	Safe
`nis_list(3N)`	MT-Safe
`nis_local_directory(3N)`	MT-Safe
`nis_local_group(3N)`	MT-Safe
`nis_local_host(3N)`	MT-Safe
`nis_local_names(3N)`	MT-Safe
`nis_local_principal(3N)`	MT-Safe
`nis_lookup(3N)`	MT-Safe
`nis_map_group(3N)`	MT-Safe
`nis_mkdir(3N)`	MT-Safe
`nis_modify(3N)`	MT-Safe
`nis_modify_entry(3N)`	MT-Safe
`nis_names(3N)`	MT-Safe
`nis_name_of(3N)`	Safe
`nis_next_entry(3N)`	MT-Safe
`nis_perror(3N)`	Safe
`nis_ping(3N)`	MT-Safe
`nis_print_group_entry(3N)`	MT-Safe
`nis_print_object(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`nis_remove(3N)`	MT-Safe
`nis_removemember(3N)`	MT-Safe
`nis_remove_entry(3N)`	MT-Safe
`nis_rmdir(3N)`	MT-Safe
`nis_server(3N)`	MT-Safe
`nis_servstate(3N)`	MT-Safe
`nis_sperrno(3N)`	Safe
`nis_sperror(3N)`	Safe
`nis_sperror_r(3N)`	Safe
`nis_stats(3N)`	MT-Safe
`nis_subr(3N)`	Safe
`nis_tables(3N)`	MT-Safe
`nis_verifygroup(3N)`	MT-Safe
`nl(3X)`	Unsafe
`nlist(3E)`	Safe
`nlsgetcall(3N)`	Unsafe
`nlsprovider(3N)`	Unsafe
`nlsrequest(3N)`	Unsafe
`nl_langinfo(3C)`	Safe with exceptions
`nocbreak(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`nodelay(3X)`	Unsafe
`noecho(3X)`	Unsafe
`nonl(3X)`	Unsafe
`noqiflush(3X)`	Unsafe
`noraw(3X)`	Unsafe
`NOTE(3X)`	Safe
`notimeout(3X)`	Unsafe
`nrand48(3C)`	Safe
`ntohl(3N)`	Safe
`ntohs(3N)`	Safe
`offsetof(3C)`	MT-Safe
`opendir(3C)`	Safe
`openlog(3)`	Safe
`openpl(3)`	Safe
`openvt(3)`	Safe
`overlay(3X)`	Unsafe
`overwrite(3X)`	Unsafe
`p2close(3G)`	Unsafe
`p2open(3G)`	Unsafe
`pair_content(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`panels(3X)`	Unsafe
`panel_above(3X)`	Unsafe
`panel_below(3X)`	Unsafe
`panel_hidden(3X)`	Unsafe
`panel_move(3X)`	Unsafe
`panel_new(3X)`	Unsafe
`panel_show(3X)`	Unsafe
`panel_top(3X)`	Unsafe
`panel_update(3X)`	Unsafe
`panel_userptr(3X)`	Unsafe
`panel_window(3X)`	Unsafe
`pathfind(3G)`	MT-Safe
`pclose(3S)`	Unsafe
`pechochar(3X)`	Unsafe
`pechowchar(3X)`	Unsafe
`perror(3C)`	MT-Safe
`pfmt(3C)`	MT-safe
`plot(3)`	Safe
`pmap_getmaps(3N)`	Unsafe
`pmap_getport(3N)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`pmap_rmtcall(3N)`	Unsafe
`pmap_set(3N)`	Unsafe
`pmap_unset(3N)`	Unsafe
`pnoutrefresh(3X)`	Unsafe
`point(3)`	Safe
`popen(3S)`	Unsafe
`post_form(3X)`	Unsafe
`post_menu(3X)`	Unsafe
`pos_form_cursor(3X)`	Unsafe
`pos_menu_cursor(3X)`	Unsafe
`pow(3M)`	MT-Safe
`prefresh(3X)`	Unsafe
`printf(3S)`	MT-Safe except with `setlocale()`
`printw(3X)`	Unsafe
`psiginfo(3C)`	Safe
`psignal(3C)`	Safe
`pthreads(3T)`	Fork1-Safe,MT-Safe,Async-Signal-Safe
`pthread_atfork(3T)`	MT-Safe
`pthread_attr_destroy(3T)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`pthread_attr_getdetachstate(3T)`	MT-Safe
`pthread_attr_getinheritsched(3T)`	MT-Safe
`pthread_attr_getschedparam(3T)`	MT-Safe
`pthread_attr_getschedpolicy(3T)`	MT-Safe
`pthread_attr_getscope(3T)`	MT-Safe
`pthread_attr_getstackaddr(3T)`	MT-Safe
`pthread_attr_getstacksize(3T)`	MT-Safe
`pthread_attr_init(3T)`	MT-Safe
`pthread_attr_setdetachstate(3T)`	MT-Safe
`pthread_attr_setscope(3T)`	MT-Safe
`pthread_attr_setstackaddr(3T)`	MT-Safe
`pthread_attr_setstacksize(3T)`	MT-Safe
`pthread_cancel(3T)`	MT-Safe
`pthread_cleanup_pop(3T)`	MT-Safe
`pthread_cleanup_push(3T)`	MT-Safe
`pthread_condattr_destroy(3T)`	MT-Safe
`pthread_condattr_getpshared(3T)`	MT-Safe
`pthread_condattr_init(3T)`	MT-Safe
`pthread_condattr_setpshared(3T)`	MT-Safe
`pthread_cond_broadcast(3T)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`pthread_cond_destroy(3T)`	MT-Safe
`pthread_cond_init(3T)`	MT-Safe
`pthread_cond_signal(3T)`	MT-Safe
`pthread_cond_timedwait(3T)`	MT-Safe
`pthread_cond_wait(3T)`	MT-Safe
`pthread_create(3T)`	MT-Safe
`pthread_detach(3T)`	MT-Safe
`pthread_equal(3T)`	MT-Safe
`pthread_exit(3T)`	MT-Safe
`pthread_getschedparam(3T)`	MT-Safe
`pthread_getspecific(3T)`	MT-Safe
`pthread_join(3T)`	MT-Safe
`pthread_key_create(3T)`	MT-Safe
`pthread_key_delete(3T)`	MT-Safe
`pthread_kill(3T)`	MT-Safe, Async-Signal-Safe
`pthread_mutexattr_destroy(3T)`	MT-Safe
`pthread_mutexattr_getprioceiling(3T)`	MT-Safe
`pthread_mutexattr_getprotocol(3T)`	MT-Safe
`pthread_mutexattr_getpshared(3T)`	MT-Safe
`pthread_mutexattr_init(3T)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

pthread_mutexattr_setprioceiling(3T)	MT-Safe
`pthread_mutexattr_setprotocol(3T)`	MT-Safe
`pthread_mutexattr_setpshared(3T)`	MT-Safe
`pthread_mutex_destroy(3T)`	MT-Safe
`pthread_mutex_getprioceiling(3T)`	MT-Safe
pthread_mutex_init(3T)	MT-Safe
`pthread_mutex_lock(3T)`	MT-Safe
`pthread_mutex_setprioceiling(3T)`	MT-Safe
`pthread_mutex_trylock(3T)`	MT-Safe
`pthread_mutex_unlock(3T)`	MT-Safe
`pthread_once(3T)`	MT-Safe
`pthread_self(3T)`	MT-Safe
`pthread_setcancelstate(3T)`	MT-Safe
`pthread_setcanceltype(3T)`	MT-Safe
`pthread_setschedparam(3T)`	MT-Safe
`pthread_setspecific(3T)`	MT-Safe
`pthread_sigmask(3T)`	MT-Safe, Async-Signal-Safe
`pthread_testcancel(3T)`	MT-Safe
`ptsname(3C)`	Safe
`publickey(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`putc(3S)`	MT-Safe
`putchar(3S)`	MT-Safe
`putenv(3C)`	Safe
`putmntent(3C)`	Safe
`putp(3X)`	Unsafe
`putpwent(3C)`	Unsafe
`puts(3S)`	MT-Safe
`putspent(3C)`	Unsafe
`pututline(3C)`	Unsafe
`pututxline(3C)`	Unsafe
`putw(3S)`	MT-Safe
`putwc(3I)`	MT-Safe
`putwchar(3I)`	MT-Safe
`putwin(3X)`	Unsafe
`putws(3I)`	MT-Safe
`qeconvert(3)`	MT-Safe
`qfconvert(3)`	MT-Safe
`qgconvert(3)`	MT-Safe
`qiflush(3X)`	Unsafe
`qsort(3C)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`quadruple_to_decimal(3)`	MT-Safe
`rac_drop(3N)`	Unsafe
`rac_poll(3N)`	Unsafe
`rac_recv(3N)`	Unsafe
`rac_send(3N)`	Unsafe
`raise(3C)`	MT-Safe
`rand(3C)`	Unsafe, use `rand_r()`
`random(3C)`	Unsafe
`raw(3X)`	Unsafe
`rcmd(3N)`	Unsafe
`readdir(3C)`	Unsafe, use `readdir_r()`
`read_vtoc(3X)`	Unsafe
`realloc(3C)`	Safe
`realloc(3X)`	Safe
`realpath(3C)`	MT-Safe
`recv(3N)`	Safe
`recvfrom(3N)`	Safe
`recvmsg(3N)`	Safe
`redrawwin(3X)`	Unsafe
`refresh(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`regcmp(3G)`	MT-Safe
`regcomp(3C)`	MT-Safe
`regerror(3C)`	MT-Safe
`regex(3G)`	MT-Safe
`regexec(3C)`	MT-Safe
`regexpr(3G)`	MT-Safe
`regfree(3C)`	MT-Safe
`registerrpc(3N)`	Unsafe
`remainder(3M)`	MT-Safe
`remove(3C)`	MT-Safe
`remque(3C)`	Unsafe
`replace_panel(3X)`	Unsafe
`resetty(3X)`	Unsafe
`reset_prog_mode(3X)`	Unsafe
`reset_shell_mode(3X)`	Unsafe
`resolver(3N)`	Unsafe
`restartterm(3X)`	Unsafe
`res_init(3N)`	Unsafe
`res_mkquery(3N)`	Unsafe
`res_search(3N)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`res_send(3N)`	Unsafe
`rewind(3S)`	MT-Safe
`rewinddir(3C)`	Safe
`rexec(3N)`	Unsafe
`rint(3M)`	MT-Safe
`ripoffline(3X)`	Unsafe
`rmdirp(3G)`	MT-Safe
`rnusers(3N)`	MT-Safe
`rpc(3N)`	MT-Safe with exceptions
`rpcbind(3N)`	MT-Safe
`rpcb_getaddr(3N)`	MT-Safe
`rpcb_getmaps(3N)`	MT-Safe
`rpcb_gettime(3N)`	MT-Safe
`rpcb_rmtcall(3N)`	MT-Safe
`rpc_broadcast_exp(3N)`	MT-Safe
`rpc_call(3N)`	MT-Safe
`rpc_clnt_auth(3N)`	MT-Safe
`rpc_clnt_calls(3N)`	MT-Safe
`rpc_clnt_create(3N)`	MT-Safe
`rpc_control(3N)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`rpc_createerr(3N)`	MT-Safe
`rpc_rac(3N)`	Unsafe
`rpc_reg(3N)`	MT-Safe
`rpc_soc(3N)`	Unsafe
`rpc_svc_create(3N)`	MT-Safe
`rpc_svc_err(3N)`	MT-Safe
`rpc_svc_reg(3N)`	MT-Safe
`rpc_xdr(3N)`	Safe
`rresvport(3N)`	Unsafe
`rstat(3N)`	MT-Safe
`ruserok(3N)`	Unsafe
`rusers(3N)`	MT-Safe
`rwall(3N)`	MT-Safe
`rwlock(3T)`	MT-Safe
`rwlock_destroy(3T)`	MT-Safe
`rwlock_init(3T)`	MT-Safe
`rw_rdlock(3T)`	MT-Safe
`rw_tryrdlock(3T)`	MT-Safe
`rw_trywrlock(3T)`	MT-Safe
`rw_unlock(3T)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`rw_wrlock(3T)`	MT-Safe
`savetty(3X)`	Unsafe
`scalb(3C)`	MT-Safe
`scalb(3M)`	MT-Safe
`scalbn(3M)`	MT-Safe
`scale_form(3X)`	Unsafe
`scale_menu(3X)`	Unsafe
`scanf(3S)`	MT-Safe
`scanw(3X)`	Unsafe
`sched_getparam(3R)`	MT-Safe
`sched_getscheduler(3R)`	MT-Safe
`sched_get_priority_max(3R)`	MT-Safe
`sched_get_priority_min(3R)`	MT-Safe
`sched_rr_get_interval(3R)`	MT-Safe
`sched_setparam(3R)`	MT-Safe
`sched_setscheduler(3R)`	MT-Safe
`sched_yield(3R)`	MT-Safe
`scrl(3X)`	Unsafe
`scroll(3X)`	Unsafe
`scrollok(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`scr_dump(3X)`	Unsafe
`scr_init(3X)`	Unsafe
`scr_restore(3X)`	Unsafe
`scr_set(3X)`	Unsafe
`seconvert(3)`	MT-Safe
`secure_rpc(3N)`	MT-Safe
`seed48(3C)`	Safe
`seekdir(3C)`	Safe
`select(3C)`	MT-Safe
`sema_destroy(3T)`	MT-Safe
`sema_init(3T)`	MT-Safe
`sema_post(3T)`	MT-Safe, Async-Signal-Safe
`sema_trywait(3T)`	MT-Safe
`sema_wait(3T)`	MT-Safe
`sem_close(3R)`	MT-Safe
`sem_destroy(3R)`	MT-Safe
`sem_getvalue(3R)`	MT-Safe
`sem_init(3R)`	MT-Safe
`sem_open(3R)`	MT-Safe
`sem_post(3R)`	Async-Signal-Safe

`sem_trywait(3R)`	MT-Safe
`sem_unlink(3R)`	MT-Safe
`sem_wait(3R)`	MT-Safe
`send(3N)`	Safe
`sendmsg(3N)`	Safe
`sendto(3N)`	Safe
`setac(3)`	Safe
`setauclass(3)`	MT-Safe
`setauevent(3)`	MT-Safe
`setauuser(3)`	MT-Safe
`setbuf(3S)`	MT-Safe
`setcat(3C)`	MT-safe
`setjmp(3C)`	Unsafe
`setkey(3C)`	Safe
`setlabel(3C)`	MT-safe
`setlocale(3C)`	Safe with exceptions
`setlogmask(3)`	Safe
`setnetconfig(3N)`	MT-Safe
`setnetpath(3N)`	MT-Safe
`setscrreg(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`setsockopt(3N)`	Safe
`setsyx(3X)`	Unsafe
`setterm(3X)`	Unsafe
`settimeofday(3C)`	MT-Safe
`setupterm(3X)`	Unsafe
`setutent(3C)`	Unsafe
`setvbuf(3S)`	MT-Safe
`set_current_field(3X)`	Unsafe
`set_current_item(3X)`	Unsafe
`set_curterm(3X)`	Unsafe
`set_fieldtype_arg(3X)`	Unsafe
`set_fieldtype_choice(3X)`	Unsafe
`set_field_back(3X)`	Unsafe
`set_field_buffer(3X)`	Unsafe
`set_field_fore(3X)`	Unsafe
`set_field_init(3X)`	Unsafe
`set_field_just(3X)`	Unsafe
`set_field_opts(3X)`	Unsafe
`set_field_pad(3X)`	Unsafe
`set_field_status(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`set_field_term(3X)`	Unsafe
`set_field_type(3X)`	Unsafe
`set_field_userptr(3X)`	Unsafe
`set_form_fields(3X)`	Unsafe
`set_form_init(3X)`	Unsafe
`set_form_opts(3X)`	Unsafe
`set_form_page(3X)`	Unsafe
`set_form_sub(3X)`	Unsafe
`set_form_term(3X)`	Unsafe
`set_form_userptr(3X)`	Unsafe
`set_form_win(3X)`	Unsafe
`set_item_init(3X)`	Unsafe
`set_item_opts(3X)`	Unsafe
`set_item_term(3X)`	Unsafe
`set_item_userptr(3X)`	Unsafe
`set_item_value(3X)`	Unsafe
`set_max_field(3X)`	Unsafe
`set_menu_back(3X)`	Unsafe
`set_menu_init(3X)`	Unsafe
`set_menu_items(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`set_menu_mark(3X)`	Unsafe
`set_menu_opts(3X)`	Unsafe
`set_menu_pad(3X)`	Unsafe
`set_menu_pattern(3X)`	Unsafe
`set_menu_sub(3X)`	Unsafe
`set_menu_term(3X)`	Unsafe
`set_menu_userptr(3X)`	Unsafe
`set_menu_win(3X)`	Unsafe
`set_new_page(3X)`	Unsafe
`set_panel_userptr(3X)`	Unsafe
`set_term(3X)`	Unsafe
`set_top_row(3X)`	Unsafe
`sfconvert(3)`	MT-Safe
`sgconvert(3)`	MT-Safe
`shm_open(3R)`	MT-Safe
`shm_unlink(3R)`	MT-Safe
`show_panel(3X)`	Unsafe
`shutdown(3N)`	Safe
`sigaddset(3C)`	MT-Safe, Async-Signal-Safe
`sigdelset(3C)`	MT-Safe, Async-Signal-Safe

`sigemptyset(3C)`	MT-Safe, Async-Signal-Safe
`sigfillset(3C)`	MT-Safe, Async-Signal-Safe
`sigfpe(3)`	Safe
`sigismember(3C)`	MT-Safe, Async-Signal-Safe
`siglongjmp(3C)`	Unsafe
`significand(3M)`	MT-Safe
`sigqueue(3R)`	Async-Signal-Safe
`sigsetjmp(3C)`	Unsafe
`sigsetops(3C)`	MT-Safe, Async-Signal-Safe
`sigtimedwait(3R)`	Async-Signal-Safe
`sigwaitinfo(3R)`	Async-Signal-Safe
`sin(3M)`	MT-Safe
`single_to_decimal(3)`	MT-Safe
`sinh(3M)`	MT-Safe
`sleep(3B)`	Async-Signal-Safe
`sleep(3C)`	Safe
`slk_attroff(3X)`	Unsafe
`slk_attron(3X)`	Unsafe
`slk_attrset(3X)`	Unsafe
`slk_clear(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`slk_init(3X)`	Unsafe
`slk_label(3X)`	Unsafe
`slk_noutrefresh(3X)`	Unsafe
`slk_refresh(3X)`	Unsafe
`slk_restore(3X)`	Unsafe
`slk_set(3X)`	Unsafe
`slk_touch(3X)`	Unsafe
`socket(3N)`	Safe
`socketpair(3N)`	Safe
`space(3)`	Safe
`spray(3N)`	Unsafe
`sprintf(3S)`	MT-Safe
`sqrt(3M)`	MT-Safe
`srand(3C)`	Unsafe
`srand48(3C)`	Safe
`srandom(3C)`	Unsafe
`sscanf(3S)`	MT-Safe
`ssignal(3C)`	Unsafe
`standend(3X)`	Unsafe
`standout(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`start_color(3X)`	Unsafe
`step(3G)`	MT-Safe
`str(3G)`	MT-Safe
`strcadd(3G)`	MT-Safe
`strcasecmp(3C)`	Safe
`strcat(3C)`	Safe
`strccpy(3G)`	MT-Safe
`strchr(3C)`	Safe
`strcmp(3C)`	Safe
`strcoll(3C)`	Safe with exceptions
`strcpy(3C)`	Safe
`strcspn(3C)`	Safe
`strdup(3C)`	Safe
`streadd(3G)`	MT-Safe
`strecpy(3G)`	MT-Safe
`strerror(3C)`	Safe
`strfind(3G)`	MT-Safe
`strfmon(3C)`	MT-Safe
`strftime(3C)`	MT-Safe
`string(3C)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`string_to_decimal(3)`	MT-Safe
`strlen(3C)`	Safe
`strncasecmp(3C)`	Safe
`strncat(3C)`	Safe
`strncmp(3C)`	Safe
`strncpy(3C)`	Safe
`strpbrk(3C)`	Safe
`strptime(3C)`	MT-Safe
`strrchr(3C)`	Safe
`strrspn(3G)`	MT-Safe
`strsignal(3C)`	Safe
`strspn(3C)`	Safe
`strstr(3C)`	Safe
`strtod(3C)`	MT-Safe
`strtok(3C)`	Unsafe, use `strtok_r()`
`strtol(3C)`	MT-Safe
`strtoll(3C)`	MT-Safe
`strtoul(3C)`	MT-Safe
`strtoull(3C)`	MT-Safe
`strtrns(3G)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`strxfrm(3C)`	Safe with exceptions
`subpad(3X)`	Unsafe
`subwin(3X)`	Unsafe
`svcerr_auth(3N)`	MT-Safe
`svcerr_decode(3N)`	MT-Safe
`svcerr_noproc(3N)`	MT-Safe
`svcerr_noprog(3N)`	MT-Safe
`svcerr_progvers(3N)`	MT-Safe
`svcerr_systemerr(3N)`	MT-Safe
`svcerr_weakauth(3N)`	MT-Safe
`svcfd_create(3N)`	Unsafe
`svcraw_create(3N)`	Unsafe
`svctcp_create(3N)`	Unsafe
`svcudp_bufcreate(3N)`	Unsafe
`svcudp_create(3N)`	Unsafe
`svc_auth_reg(3N)`	MT-Safe
`svc_control(3N)`	MT-Safe
`svc_create(3N)`	MT-Safe
`svc_destroy(3N)`	MT-Safe
`svc_dg_create(3N)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`svc_fds(3N)`	Unsafe
`svc_fd_create(3N)`	MT-Safe
`svc_getcaller(3N)`	Unsafe
`svc_reg(3N)`	MT-Safe
`svc_register(3N)`	Unsafe
`svc_tli_create(3N)`	MT-Safe
`svc_tp_create(3N)`	MT-Safe
`svc_unreg(3N)`	MT-Safe
`svc_unregister(3N)`	Unsafe
`svc_vc_create(3N)`	MT-Safe
`swab(3C)`	MT-Safe
`swapcontext(3C)`	MT-Safe
`syncok(3X)`	Unsafe
`sysconf(3C)`	MT-Safe, Async-Signal-Safe
`syslog(3)`	Safe
`system(3S)`	MT-Safe
`taddr2uaddr(3N)`	MT-Safe
`tan(3M)`	MT-Safe
`tanh(3M)`	MT-Safe
`tcdrain(3)`	MT-Safe, Async-Signal-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`tcflow(3)`	MT-Safe, Async-Signal-Safe
`tcflush(3)`	MT-Safe, Async-Signal-Safe
`tcgetattr(3)`	MT-Safe, Async-Signal-Safe
`tcgetpgrp(3)`	MT-Safe, Async-Signal-Safe
`tcgetsid(3)`	MT-Safe
`tcsendbreak(3)`	MT-Safe, Async-Signal-Safe
`tcsetattr(3)`	MT-Safe, Async-Signal-Safe
`tcsetpgrp(3)`	MT-Safe, Async-Signal-Safe
`tcsetpgrp(3C)`	MT-Safe
`tdelete(3C)`	Safe
`telldir(3C)`	Safe
`tempnam(3S)`	Safe
`termattrs(3X)`	Unsafe
`termname(3X)`	Unsafe
`textdomain(3I)`	Safe with exceptions
`tfind(3C)`	Safe
`tgetent(3X)`	Unsafe
`tgetflag(3X)`	Unsafe
`tgetnum(3X)`	Unsafe
`tgetstr(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`tgoto(3X)`	Unsafe
`threads(3T)`	Fork1-Safe,MT-Safe,Async-Signal-Safe
`thr_continue(3T)`	MT-Safe
`thr_create(3T)`	MT-Safe
`thr_exit(3T)`	MT-Safe
`thr_getconcurrency(3T)`	MT-Safe
`thr_getprio(3T)`	MT-Safe
`thr_getspecific(3T)`	MT-Safe
`thr_join(3T)`	MT-Safe
`thr_keycreate(3T)`	MT-Safe
`thr_kill(3T)`	MT-Safe, Async-Signal-Safe
`thr_main(3T)`	MT-Safe
`thr_min_stack(3T)`	MT-Safe
`thr_self(3T)`	MT-Safe
`thr_setconcurrency(3T)`	MT-Safe
`thr_setprio(3T)`	MT-Safe
`thr_setspecific(3T)`	MT-Safe
`thr_sigsetmask(3T)`	MT-Safe, Async-Signal-Safe
`thr_stksegment(3T)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`thr_suspend(3T)`	MT-Safe
`thr_yield(3T)`	MT-Safe
`tigetflag(3X)`	Unsafe
`tigetnum(3X)`	Unsafe
`tigetstr(3X)`	Unsafe
`timeout(3X)`	Unsafe
`timer_create(3R)`	MT-Safe with exceptions
`timer_delete(3R)`	MT-Safe with exceptions
`timer_getoverrun(3R)`	Async-Signal-Safe
`timer_gettime(3R)`	Async-Signal-Safe
`timer_settime(3R)`	Async-Signal-Safe
`tmpfile(3S)`	Safe
`tmpnam(3S)`	Unsafe, use `tmpnam_r()`
`TNF_DECLARE_RECORD(3X)`	MT-Safe
`TNF_DEFINE_RECORD(3.3X)`	MT-Safe
`TNF_DEFINE_RECORD_1(3X)`	MT-Safe
`TNF_DEFINE_RECORD_2(3X)`	MT-Safe
`TNF_DEFINE_RECORD_4(3X)`	MT-Safe
`TNF_DEFINE_RECORD_5(3X)`	MT-Safe
`TNF_PROBE(3.3X)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`TNF_PROBE(3X)`	MT-Safe
`TNF_PROBE_0(3X)`	MT-Safe
`TNF_PROBE_1(3X)`	MT-Safe
`TNF_PROBE_2(3X)`	MT-Safe
`TNF_PROBE_4(3X)`	MT-Safe
`TNF_PROBE_5(3X)`	MT-Safe
`tnf_process_disable(3X)`	MT-Safe
`tnf_process_enable(3X)`	MT-Safe
`tnf_thread_disable(3X)`	MT-Safe
`tnf_thread_enable(3X)`	MT-Safe
`toascii(3C)`	MT-Safe with exceptions
`tolower(3C)`	MT-Safe with exceptions
`top_panel(3X)`	Unsafe
`top_row(3X)`	Unsafe
`touchline(3X)`	Unsafe
`touchwin(3X)`	Unsafe
`toupper(3C)`	MT-Safe with exceptions
`towlower(3I)`	MT-Safe with exceptions
`towupper(3I)`	MT-Safe with exceptions
`tparm(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`tputs(3X)`	Unsafe
`trig(3M)`	MT-Safe
`truncate(3C)`	MT-Safe
`tsearch(3C)`	Safe
`ttyname(3C)`	Unsafe, use `ttyname_r()`
`ttyslot(3C)`	Safe
`twalk(3C)`	Safe
`typeahead(3X)`	Unsafe
`t_accept(3N)`	MT-Safe
`t_alloc(3N)`	MT-Safe
`t_bind(3N)`	MT-Safe
`t_close(3N)`	MT-Safe
`t_connect(3N)`	MT-Safe
`t_error(3N)`	MT-Safe
`t_free(3N)`	MT-Safe
`t_getinfo(3N)`	MT-Safe
`t_getstate(3N)`	MT-Safe
`t_listen(3N)`	MT-Safe
`t_look(3N)`	MT-Safe
`t_open(3N)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`t_optmgmt(3N)`	MT-Safe
`t_rcv(3N)`	MT-Safe
`t_rcvconnect(3N)`	MT-Safe
`t_rcvdis(3N)`	MT-Safe
`t_rcvrel(3N)`	MT-Safe
`t_rcvudata(3N)`	MT-Safe
`t_rcvuderr(3N)`	MT-Safe
`t_snd(3N)`	MT-Safe
`t_snddis(3N)`	MT-Safe
`t_sync(3N)`	MT-Safe
`t_unbind(3N)`	MT-Safe
`uaddr2taddr(3N)`	MT-Safe
`ulckpwdf(3C)`	MT-Safe
`ulltostr(3C)`	MT-Safe
`unctrl(3X)`	Unsafe
`ungetc(3S)`	MT-Safe
`ungetch(3X)`	Unsafe
`ungetwc(3I)`	MT-Safe
`ungetwch(3X)`	Unsafe
`unlockpt(3C)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`unordered(3C)`	MT-Safe
`unpost_form(3X)`	Unsafe
`unpost_menu(3X)`	Unsafe
`untouchwin(3X)`	Unsafe
`update_panels(3X)`	Unsafe
`updwtmp(3C)`	Unsafe
`updwtmpx(3C)`	Unsafe
`user2netname(3N)`	MT-Safe
`use_env(3X)`	Unsafe
`utmpname(3C)`	Unsafe
`utmpxname(3C)`	Unsafe
`valloc(3C)`	Safe
`vfprintf(3S)`	Async-Signal-Safe
`vidattr(3X)`	Unsafe
`vidputs(3X)`	Unsafe
`vlfmt(3C)`	MT-safe
`volmgt_check(3X)`	MT-Safe
`volmgt_inuse(3X)`	MT-Safe
`volmgt_root(3X)`	MT-Safe
`volmgt_running(3X)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`volmgt_symdev(3X)`	MT-Safe
`volmgt_symname(3X)`	MT-Safe
`vpfmt(3C)`	MT-safe
`vprintf(3S)`	Async-Signal-Safe
`vsprintf(3S)`	MT-Safe
`vsyslog(3)`	Safe
`vwprintw(3X)`	Unsafe
`vwscanw(3X)`	Unsafe
`waddch(3X)`	Unsafe
`waddchnstr(3X)`	Unsafe
`waddchstr(3X)`	Unsafe
`waddnstr(3X)`	Unsafe
`waddnwstr(3X)`	Unsafe
`waddstr(3X)`	Unsafe
`waddwch(3X)`	Unsafe
`waddwchnstr(3X)`	Unsafe
`waddwchstr(3X)`	Unsafe
`waddwstr(3X)`	Unsafe
`wadjcurspos(3X)`	Unsafe
`watof(3I)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`watoi(3I)`	MT-Safe
`watol(3I)`	MT-Safe
`watoll(3I)`	MT-Safe
`wattroff(3X)`	Unsafe
`wattron(3X)`	Unsafe
`wattrset(3X)`	Unsafe
`wbkgd(3X)`	Unsafe
`wbkgdset(3X)`	Unsafe
`wborder(3X)`	Unsafe
`wclear(3X)`	Unsafe
`wclrtobot(3X)`	Unsafe
`wclrtoeol(3X)`	Unsafe
`wconv(3I)`	MT-Safe with exceptions
`wcscat(3I)`	MT-Safe
`wcschr(3I)`	MT-Safe
`wcscmp(3I)`	MT-Safe
`wcscoll(3I)`	MT-Safe
`wcscpy(3I)`	MT-Safe
`wcscspn(3I)`	MT-Safe
`wcsetno(3I)`	MT-Safe with exceptions

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`wcslen(3I)`	MT-Safe
`wcsncat(3I)`	MT-Safe
`wcsncmp(3I)`	MT-Safe
`wcsncpy(3I)`	MT-Safe
`wcspbrk(3I)`	MT-Safe
`wcsrchr(3I)`	MT-Safe
`wcsspn(3I)`	MT-Safe
`wcstod(3I)`	MT-Safe
`wcstok(3I)`	MT-Safe
`wcstol(3I)`	MT-Safe
`wcstombs(3C)`	MT-Safe with exceptions
`wcstoul(3I)`	MT-Safe
`wcstring(3I)`	MT-Safe
`wcswcs(3I)`	MT-Safe
`wcswidth(3I)`	MT-Safe
`wcsxfrm(3I)`	MT-Safe
`wctomb(3C)`	MT-Safe with exceptions
`wctype(3I)`	MT-Safe
`wcursyncup(3X)`	Unsafe
`wcwidth(3I)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`wdelch(3X)`	Unsafe
`wdeleteln(3X)`	Unsafe
`wechochar(3X)`	Unsafe
`wechowchar(3X)`	Unsafe
`werase(3X)`	Unsafe
`wgetch(3X)`	Unsafe
`wgetnstr(3X)`	Unsafe
`wgetnwstr(3X)`	Unsafe
`wgetstr(3X)`	Unsafe
`wgetwch(3X)`	Unsafe
`wgetwstr(3X)`	Unsafe
`whline(3X)`	Unsafe
`winch(3X)`	Unsafe
`winchnstr(3X)`	Unsafe
`winchstr(3X)`	Unsafe
`windex(3I)`	MT-Safe
`winnstr(3X)`	Unsafe
`winnwstr(3X)`	Unsafe
`winsch(3X)`	Unsafe
`winsdelln(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`winsertln(3X)`	Unsafe
`winsnstr(3X)`	Unsafe
`winsnwstr(3X)`	Unsafe
`winsstr(3X)`	Unsafe
`winstr(3X)`	Unsafe
`winswch(3X)`	Unsafe
`winswstr(3X)`	Unsafe
`winwch(3X)`	Unsafe
`winwchnstr(3X)`	Unsafe
`winwchstr(3X)`	Unsafe
`winwstr(3X)`	Unsafe
`wmove(3X)`	Unsafe
`wmovenextch(3X)`	Unsafe
`wmoveprevch(3X)`	Unsafe
`wprintw(3X)`	Unsafe
`wredrawln(3X)`	Unsafe
`wrefresh(3X)`	Unsafe
`wrindex(3I)`	MT-Safe
`write_vtoc(3X)`	Unsafe
`wscanw(3X)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`wscasecmp(3I)`	MT-Safe
`wscat(3I)`	MT-Safe
`wschr(3I)`	MT-Safe
`wscmp(3I)`	MT-Safe
`wscol(3I)`	MT-Safe
`wscoll(3I)`	MT-Safe
`wscpy(3I)`	MT-Safe
`wscrl(3X)`	Unsafe
`wscspn(3I)`	MT-Safe
`wsdup(3I)`	MT-Safe
`wsetscrreg(3X)`	Unsafe
`wslen(3I)`	MT-Safe
`wsncasecmp(3I)`	MT-Safe
`wsncat(3I)`	MT-Safe
`wsncmp(3I)`	MT-Safe
`wsncpy(3I)`	MT-Safe
`wspbrk(3I)`	MT-Safe
`wsprintf(3I)`	MT-Safe
`wsrchr(3I)`	MT-Safe
`wsscanf(3I)`	MT-Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`wsspn(3I)`	MT-Safe
`wstandend(3X)`	Unsafe
`wstandout(3X)`	Unsafe
`wstod(3I)`	MT-Safe
`wstok(3I)`	MT-Safe
`wstol(3I)`	MT-Safe
`wstring(3I)`	MT-Safe
`wsxfrm(3I)`	MT-Safe
`wsyncdown(3X)`	Unsafe
`wsyncup(3X)`	Unsafe
`wtimeout(3X)`	Unsafe
`wtouchln(3X)`	Unsafe
`wvline(3X)`	Unsafe
`xdr(3N)`	Safe
`xdrmem_create(3N)`	MT-Safe
`xdrrec_create(3N)`	MT-Safe
`xdrrec_endofrecord(3N)`	Safe
`xdrrec_eof(3N)`	Safe
`xdrrec_readbytes(3N)`	Safe
`xdrrec_skiprecord(3N)`	Safe

`xdrstdio_create(3N)`	MT-Safe
`xdr_accepted_reply(3N)`	Safe
`xdr_admin(3N)`	Safe
`xdr_array(3N)`	Safe
`xdr_authsys_parms(3N)`	Safe
`xdr_authunix_parms(3N)`	Unsafe
`xdr_bool(3N)`	Safe
`xdr_bytes(3N)`	Safe
`xdr_callhdr(3N)`	Safe
`xdr_callmsg(3N)`	Safe
`xdr_char(3N)`	Safe
`xdr_complex(3N)`	Safe
`xdr_control(3N)`	Safe
`xdr_create(3N)`	MT-Safe
`xdr_destroy(3N)`	MT-Safe
`xdr_double(3N)`	Safe
`xdr_enum(3N)`	Safe
`xdr_float(3N)`	Safe
`xdr_free(3N)`	Safe
`xdr_getpos(3N)`	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

xdr_hyper(3N)	Safe
xdr_inline(3N)	Safe
xdr_int(3N)	Safe
xdr_long(3N)	Safe
xdr_longlong_t(3N)	Safe
xdr_opaque(3N)	Safe
xdr_opaque_auth(3N)	Safe
xdr_pointer(3N)	Safe
xdr_quadruple(3N)	Safe
xdr_reference(3N)	Safe
xdr_rejected_reply(3N)	Safe
xdr_replymsg(3N)	Safe
xdr_setpos(3N)	Safe
xdr_short(3N)	Safe
xdr_simple(3N)	Safe
xdr_sizeof(3N)	Safe
xdr_string(3N)	Safe
xdr_union(3N)	Safe
xdr_u_char(3N)	Safe
xdr_u_hyper(3N)	Safe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`xdr_u_int(3N)`	Safe
`xdr_u_long(3N)`	Safe
`xdr_u_longlong_t(3N)`	Safe
`xdr_u_short(3N)`	Safe
`xdr_vector(3N)`	Safe
`xdr_void(3N)`	Safe
`xdr_wrapstring(3N)`	Safe
`xprt_register(3N)`	MT-Safe
`xprt_unregister(3N)`	MT-Safe
`y0(3M)`	MT-Safe
`y1(3M)`	MT-Safe
`yn(3M)`	MT-Safe
`ypclnt(3N)`	Unsafe
`yperr_string(3N)`	Unsafe
`ypprot_err(3N)`	Unsafe
`yp_all(3N)`	Unsafe
`yp_bind(3N)`	Unsafe
`yp_first(3N)`	Unsafe
`yp_get_default_domain(3N)`	Unsafe
`yp_master(3N)`	Unsafe

TABLE C–1 MT Safety Levels of Library Routines *(continued)*

`yp_match(3N)`	Unsafe
`yp_next(3N)`	Unsafe
`yp_order(3N)`	Unsafe
`yp_unbind(3N)`	Unsafe
`yp_update(3N)`	Unsafe
`_NOTE(3X)`	Safe
`_tolower(3C)`	MT-Safe with exceptions
`_toupper(3C)`	MT-Safe with exceptions
`__nis_map_group(3N)`	MT-Safe

Index