Please
Recycle

Contents

Code Examples

Preface

The *Guide to Multithread Programming* describes the primary facilities for implementing multithreaded applications. It also guides the programmer in the use of these facilities.

Audience

Any programmer who must:

- convert an existing single-thread application to a multithreaded application
- design a multithreaded application
- implement a multithreaded application
- maintain a multithreaded application on the SunOS™ 5.x operating system platform

The manual is more easily understood if you have experience with multi-tasked applications (applications composed of two or more cooperating processes). The manual assumes basic competence in programming, a working familiarity with the C programming language, a working familiarity with the UNIX® operating system, and a working familiarity with interprocess communication facilities.

Organization of the Manual

The services and capabilities of the multithread library are described in the following chapters.

Chapter 1, “Introduction to Multithread,” introduces multithreading through analogous concepts from other areas of computing and identifies some of the benefits of multithreaded techniques.

Chapter 2, “Threads Basics,” describes basic elements of multithread programming and the Solaris® model. It also identifies some issues about the use of synchronization mechanisms.

Chapter 3, “Thread Creation and Scheduling,” describes the facilities and techniques of starting, scheduling, and terminating threads.

Chapter 4, “Synchronization,” describes the synchronization mechanisms of the threads library.

Chapter 5, “Signals,” describes the differences between signals in a conventional UNIX environment and in the threads environment. Cautions and precautions are emphasized.

Chapter 6, “Threads and Process Resources,” describes creating and destroying processes, use of alarms and timers, and the Thread Specific Data mechanism.

Chapter 7, “Advanced Topics,” describes lightweight processes, their relation to threads, and scheduling of bound threads. It also describes other issues of asynchronous processing in a common memory space.

Chapter 8, “Using Threads,” describes strategies and considerations for using threads and the threads library. It also provides pointers on selecting the most appropriate synchronization mechanism in various situations.

Chapter 9, “The Threads Environment,” identifies the environmental requirements and options to compile, link, and run threads applications successfully.

Chapter 10, “Debugging Techniques,” describes debugging techniques that have been found to be effective. Use of threads `dbx` and other tools is described.

Appendix A, “Example Applications,” presents four examples of using the threads library mechanisms.

Typographic Conventions

This manual uses a number of typographic conventions:

- *This font* is used for emphasis, command arguments, variables in a command or code sequence, and book titles. For example:

  You must also type the filename argument as described in the *SunOS 5.x Reference Manual*.

- `This font` is used for program listings, command names, file names, and system names. For example:

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

  It is also used for text that the system displays on the screen.

- **`This font`** indicates text that you type in a screen example sequence. For example:

  ```
  $ cc printmsg.c -o printmsg
  ```

Introduction to Multithread 1

This manual describes the application programmer's interface to the multithread library in the SunOS 5.3 component of the Solaris 2.3 system. The multithread library interface is defined in section 3T of the *SunOS 5.3 Reference Manual*. The manual also describes concurrent programming concepts and practices that are suitable to Solaris Threads.

This chapter is an introduction to multithreaded programs and to some of the basic concepts of the Solaris Threads implementation.

What Are Threads?

All programmers are used to programming a single thread of control. A thread is a single sequence of execution steps, performed by a program or by a series of programs. There is only one point of execution in a thread at any instant. A single threaded program is always synchronous in that control is sometimes in the program and sometimes in the system. The thread runs through the program, the system-provided functions, the kernel, and back to the program.

A traditional UNIX process is an example of a program with a single thread of control. The traditional process includes a thread that has sole possession of its address space. Any point of execution in the process can usually assume access to any element of the address space that is free of interference from any other program.

Many programmers have designed and developed programs having two or more threads of control even though they may not think of the result as a multithreaded program. Any program that uses two or more processes running concurrently (or pseudo-concurrently) and sharing data through shared memory or other form of interprocess communication is multithreaded.

A networked UNIX environment contains many more examples of multithreaded processing with each thread occupying a separate process. A server process that spawns a copy of itself to service each individual client request is an example of multithreading. Each spawned process is a thread of control.

Access to global memory by multiple threads is analogous in use to several processes using `mmap()` to share all of their memory writable or to traditional System V IPC shared memory. Each thread has its own stack for local memory and calls/returns.

There are more examples of multithreading that may be familiar. For example, a device driver usually contains at least two threads of control, an initiator routine and an interrupt response routine. Even though only one thread is executed at a time, the routines perform asynchronous functionality, so are separate threads. Similarly, in most of the programs that have one, a signal handler is a separate thread of control in a single process.

The Solaris Threads Model

Using multitasking to develop a multithreaded application is distinguished by the basic separation of the threads. The inherent separation of processes may require a major effort by the programmer to communicate between the threads or to synchronize their actions. But, maintaining thread separation is effortless.

Multiple threads of control in a single process exchanges the easy issues and the difficult ones of multiple threads in multiple processes. Multithreading in a single process means that communication between the threads is easy. Separation of the threads is now what requires the programmer's attention.

Solaris Threads is a subset of the UNIX International threads interface. (This interface is similar to the POSIX 1003.4a draft.) Solaris Threads lets many threads of control share a single UNIX process and its address space. A multithreaded UNIX process is no longer a thread of control in itself. One thread can start other threads. The threads can be scheduled within the process (via threads library facilities) or you can have them scheduled by the kernel.

Threads execute independently. There is no general way to predict how the execution of instructions by different threads is interleaved. On machines with more than one processor, more than one thread in a process executes simultaneously. Threads make operating system requests independently. When a thread waits for completion of an operating system service, it does not prevent other threads in the process from executing. In general, the number or identities of threads that an application process applies to a problem is invisible from outside the process.

The address space of a process is equally accessible by all threads. Threads in a single UNIX process share most of its state, including file descriptors, signal handlers, memory, and so on.

The Benefits of Threads

Throughput

Unless it uses asynchronous I/O, a single threaded program must wait each time it requests a service from the operating system. On a uniprocessor, threads let a process overlap processing with one or more I/O requests as though it were using asynchronous I/O. Unlike asynchronous I/O, threads allow this overlap while using higher level services. Threads provide this overlap of I/O requests although each request is coded in the usual synchronous style. The thread making a request waits, but another thread in the process can continue. This lets a process have many I/O requests outstanding.

An example is a program that does several remote procedure calls (RPCs) to different hosts to obtain a result. In a single-threaded program the RPCs must be done serially. The program must send a request to a host and wait for the response before sending the next request. Figure 1-1 illustrates this sequence.

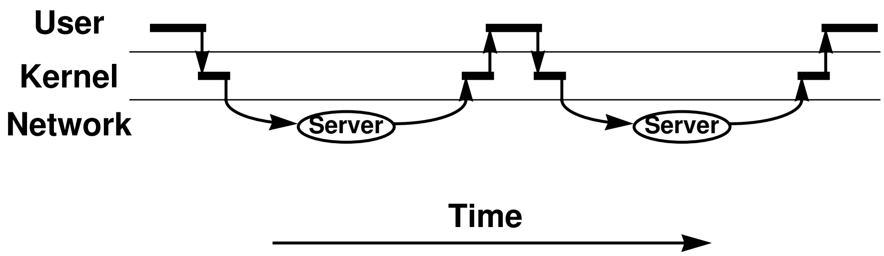

Figure 1-1 Single-thread RPCs

In a multithreaded program, a thread can be created to issue each RPC request. Each thread makes the request serially (synchronously) to a host. But the threads wait concurrently for the replies as illustrated in Figure 1-2.

.

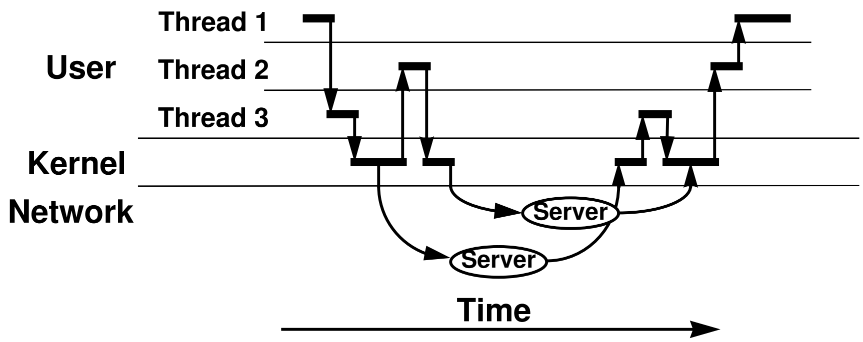

Figure 1-2 Multithread RPCs

On a multiprocessor, multiple threads let a single process take advantage of more than one processor at a time, and allow many I/O requests to be pending.

Communication

A multitasked application that communicates large amounts of data between its processes through traditional IPC facilities (for example, pipes or sockets) can benefit from multithreading. Much of the communication between processes that requires extensive use of system resources is done implicitly in a multithreaded application through memory that is global to all the threads.

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 are single threaded programs.

An application can create hundreds or even thousands of threads, one for each asynchronous task, with only minor impact on system resources. Solaris threads only use system resources in proportion to the number of threads actually running rather than to the total number of threads. Multiple threads in a single process allow finer grained division with greater resulting flexibility.

Concurrent processing is a particularly beneficial structure in some programs. It lets the program defer some processing to another thread. In addition, the client request processing code can concentrate on processing a single request at a time without worrying about other pending requests. This usually results in more understandable and maintainable code. An example of this is a database server. If each client application has a unique thread, clients requiring lengthy processing do not necessarily block other clients.

Multiple Processors

Computers with more than one processor provide multiple simultaneous points of execution. Multiple threads are an efficient way for application developers to utilize the parallelism of the hardware.

A good example of this is a process that does a matrix multiplication. A thread can be created for each available processor. The threads each repeatedly compute an unique element of the result matrix by repeatedly doing the appropriate vector multiplication.

Availability

This is related to concurrency. The blocking of one part of a process should not block the whole process. For example, single-threaded applications that do a time-consuming operation when a button is pressed typically display a “please wait” cursor and freeze while the operation is in progress. If such an application were multithreaded, long operations could be done by independent threads, enabling the application to remain active and for its user interface to remain active.

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 makes each process much more expensive than a thread in both time and space. In addition, the inherent separation between processes may require a major effort by the programmer to communicate between the threads in different processes or to synchronize their actions.

Some Considerations

Some applications are not appropriate users of multithread techniques. No compute-bound program can benefit from multithreading when run on uniprocessor hardware. For some I/O bound programs, the system may already provide the required degree of I/O concurrency (for example, read-ahead and write-behind to regular files in the primary file system). Compute bound programs with coarse-grained parallelism can benefit from multithreading when run on multiprocessor hardware. While the overhead involved in synchronizing and context switching threads is significantly smaller than using processes, it is not zero. Programs in which each thread does not execute enough code between synchronizations or context switches will have performance problems.

Even appropriate multithreaded applications can be difficult to design and to debug. The shared memory and process state of a multithreaded program can be very intimate. Because of the ease of access to shared data and other process resources, programming with multiple threads requires more care and discipline than does single-threaded programming.

Threads Basics

2

This chapter describes the basic concepts and functional elements of multithread programming with the Solaris Threads library.

Multithread Processes

A multithreaded process is not a thread of control itself. Instead it contains one or more threads of control.

Each thread is executed independently and asynchronously. The threads have execution priorities that can influence the relative rate of execution. Each thread can take page faults independently and make arbitrary system calls.

Some system calls affect the process as a whole. For example, if one thread calls `exit()`, the process and all the threads it contains is destroyed. If one thread must wait for the completion of a system call or page fault, other threads in the process can still execute.

The number and identities of threads in a process are invisible to other processes.

Shared Resources

Most process resources are equally accessible to all the threads in the process. All process virtual memory is shared. A change in shared data by one thread is available to the other threads in the process. Threads share most of the operating system state of a process. Each sees the same open files. For example, when one thread opens a file, the other threads can read it. For another example, if two or more threads read the same file sequentially, they advance through the file as though one thread were reading it.

Because threads share so much of the process state, they can affect each other in sometimes surprising ways. Programming with threads requires more care and discipline than single-threaded programming because there is no system-enforced protection between threads.

Per-Thread Resources

Each thread has a thread identifier, a set of registers (including program counter and stack pointer), the signal mask, execution priority, a stack, and thread-specific data. Even though there is a separate stack for each thread, any thread can access any stack since stacks are allocated in the global process memory.

Each thread has its own copy of thread-local variables. The C library variable `errno` is a good example. This lets threads interleave execution without corrupting `errno` in another thread.

Each thread has some storage that is unique (besides the stack and thread-local variables) called *thread-specific data*. Each thread-specific data item contains a pointer and an optional pointer to a function to call to destroy the item. Typically the pointer is used to store a reference to memory allocated out of the (shared) heap. Each item is associated with a *global key*. The global key is used to access the item associated with the thread of the caller.

The Solaris Implementation

The threads library uses execution resources (called *lightweight processes*) that are scheduled on processors by the operating system (see Chapter 7, “Advanced Topics”). The threads library schedules runnable threads on the execution resources. Threads scheduled this way are called *unbound* threads. A process usually has more threads than execution resources. The library usually ensures that there are enough execution resources for a program to proceed. If all the active threads in the process are blocked in indefinite waits and the process contains a runnable thread, the threads library allocates the additional execution resource needed to run the thread. (An application with multiple compute-bound threads can fail to schedule all of the runnable threads if `thr_setconcurrency`(3T) has not been called to adjust the amount of execution resources. See "thr_setconcurrency(3T)" on page 15 and "Thread Library Scheduling" on page 37.)

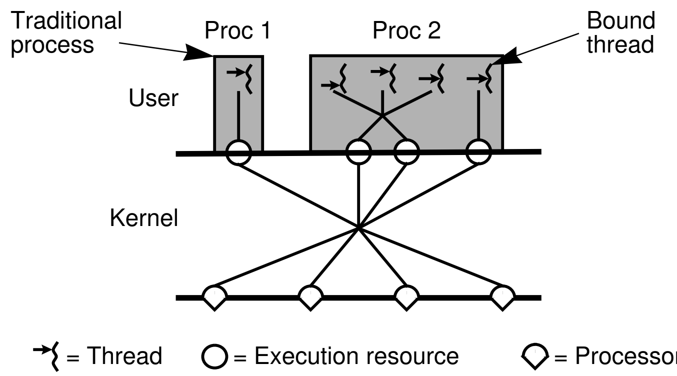

Figure 2-1 General Architecture of Threads Interface

Synchronization Primitives

Threads share access to the process address space. So, they need methods of synchronizing their accesses to data. For example, if one thread writes a variable and another thread reads it, the reader cannot determine whether the data is an old or new value without some means of synchronizing the accesses.

Solaris Threads provides a variety of synchronization facilities for threads to cooperate in accessing shared data. Several synchronization facilities with different semantics support different frequencies and styles of interaction:

- Mutual exclusion (mutex) locks
- Condition variables
- Counting semaphores
- Multiple readers, single writer locks

Each synchronization facility has a data structure (called a *synchronization variable*) that is allocated in the address space, and various synchronization functions. Synchronization variables can be statically or dynamically allocated. Specify the implementation variant of the synchronization semantic when the variable is initialized. If the variable is to be initialized to zero, use the default implementation.

The synchronization variables can also be placed in shared memory and used by threads in different processes to coordinate access to shared data, as shown in Figure 2-2. There is an implementation version of each synchronization type that lets the variable synchronize threads in the processes sharing the variable.

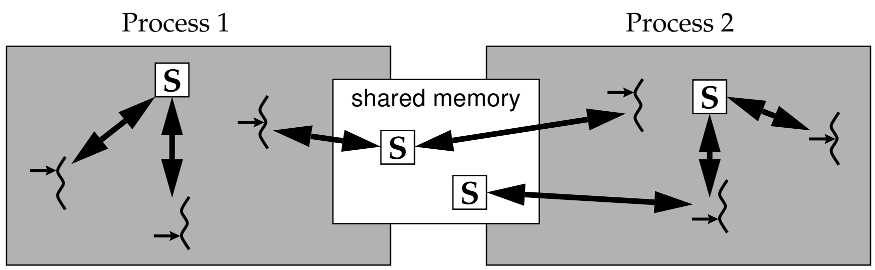

Figure 2-2 Synchronization Variables in Shared Memory

Synchronization variables can also be stored in mapped files and have lifetimes beyond that of the creating process. For example, suppose a file contains database records. Each record contains a mutual exclusion lock synchronization variable that controls access to the record. Mutual exclusion locks only allow one thread to obtain the lock at a time. A process can map the file and a thread in it obtains the lock of a record and changes the record. When the modification is done, the thread releases the lock and unmaps the file. 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.

Synchronization Primitives and Memory

In older multiprocessors, access to memory by the processors is strictly multiplexed. This means that a processor executing program code reads or writes memory in the order specified by the code. All memory operations of all processors in the system appear to happen in some global order, though the operation histories of different processors are interleaved arbitrarily. The memory operations of such machines are called strongly ordered. In this environment, threads can synchronize using ordinary memory operations.

For example, a producer thread and a consumer thread can synchronize access to a circular data buffer as follows:

```
int rdptr = 0;
int wrptr = 0
data_t buf[BUFSIZE];
```

Thread 1	Thread 2

`while (work_to_do) {`
`    buf[wrptr] = produce();`
`    wrptr = (wrptr + 1) % BUFSIZE;`
`    while (wrptr == rdptr)`
`        ;`
`}`
	`while (work_to_do) {`
	`    while (rdptr == wrptr)`
	`        ;`
	`    consume(buf[rdptr]);`
	`    rdptr = (rdptr + 1) % BUFSIZE;`
	`}`

Code Example 2-1 Producer/consumer example for strongly ordered memory

In current multiprocessors, the ordering is relaxed for greater performance. So, if one processor stores values in location A and then in location B, other processors loading data from location B and then location A may see the new value of B but the old value of A. The memory operations of such machines are called weakly ordered. On these machines the circular buffer technique shown in Code Example 2-1 fails because the consumer may see the new value of `wrptr` but the old value of the data in the buffer.

In such machines, synchronization is only achieved through the use of special instructions that force an order on memory operations. Most high level language compilers generate only ordinary memory operations to take advantage of the increased performance. They usually cannot determine when memory operation order is important and generate the special ordering instructions. They rely on the programmer to correctly use synchronization primitives to ensure that modifications to a location in memory are ordered with respect to modifications and/or access to the same location in other threads. Access to read-only data need not be synchronized. The resulting program is said to be data-race free.

Synchronization is still important even when accessing a single primitive variable (for example. an integer). On machines where the integer may not be aligned to the bus data width or be larger than the data width, a single memory load may require multiple memory cycles. This means that is may be possible for some parts of the integer to have an old value while other parts have a newer value. On some processor architectures this cannot happen (for example, SPARC®) but portable programs can't rely on this.

In summary, a portable multithreaded program *must* use the synchronization primitives to synchronize data access. It cannot rely on modifications to memory being observed by other threads in the order written in the program or even on modifications of a single variable being seen atomically.

Thread Creation and Scheduling 3

This chapter describes the facilities and techniques of starting, ending, and scheduling threads.

Thread Control Functions

This section introduces the Solaris Threads functions that create threads and control their execution. The functions are described in the next sections.

`thr_create(3T)`

The function creates and starts a new thread. The new thread executes the function specified in the call. The identity of the new thread is returned in a location provided by the caller. The thread identifiers are only valid within the process. Optional flags provide some control of the thread's behavior.

When the new thread is started, its execution begins by a procedure call to the specified function. The thread executes on a stack created by the threads library or one specified by the caller. The initial thread priority and signal mask are set to the same values as its creator. If the function returns, the thread exits.

Specifying a NULL stack base and a stack size of zero causes the threads library to allocate a default sized stack. Stack allocation is further discussed in Chapter 7, "Advanced Topics."

The scheduling options can be specified when creating a thread:

THR_SUSPENDED	The thread is immediately suspended after it is created. Use this to operate on the thread (such as changing its priority) before you run it.
THR_DETACHED	The new thread is detached. Set this when you do not want to wait for the thread to terminate. The termination of a detached thread is ignored.
THR_BOUND	The thread is created bound. See Chapter 7, "Advanced Topics."
THR_NEW_LWP	The degree of concurrency is incremented. No binding is implied. See "thr_setconcurrency(3T)" on page 15

thr_exit(3T)

The function terminates the invoking thread and sets the exit status to the specified value. If the thread was not detached, its identifier and status are retained until thr_join() is executed by another thread. If the thread is the only one in the process, calling thr_exit() destroys the process with a zero exit status.

thr_self(3T)

The function returns the thread identifier structure of the caller.

thr_join(3T)

The function blocks until the specified thread exits. If no thread identifier is specified, any undetached thread that exits causes thr_join() to return. If a stack was specified when the thread was created, it can be reclaimed when thr_join() returns. A successful thr_join() returns the identifier of the thread that exited if the wait is successful. After thr_join() returns successfully, the returned thread_id is unusable in any subsequent thread operation.

Waiting for a detached thread, waiting for the current thread, or multiple thr_join()s on the same thread are all errors.

Thread Library Scheduling

Scheduling of threads is influenced by the Solaris Threads library functions thr_setconcurrency() and thr_setprio(). Normally, threads are scheduled only with respect to other threads in the process using simple priority levels with no adjustments and no kernel involvement. (Bound threads are treated differently, as described in Chapter 7, "Advanced Topics.") Their system priority is usually uniform and is inherited from the creating process. (An application with multiple compute-bound threads can fail to schedule all of the runnable threads if thr_setconcurrency(3T) has not been called to adjust the level of execution resources. See "thr_setconcurrency(3T)" on page 15 and "Thread Library Scheduling" on page 37.)

thr_setconcurrency(3T)

The function indicates the desired level of concurrency that threads in the application require. View this as the number of threads that can be simultaneously active (executing user code or system calls). If zero is specified (the default), Solaris Threads ensures that the process continues to progress without threads waiting indefinitely for execution resources. This uses system resources efficiently but may not produce the most effective concurrency level. The actual number of simultaneously active threads may be more or less than the number specified. (Threads created with THR_NEW_LWP simply increment the degree of concurrency, as described in Chapter 7, "Advanced Topics.")

thr_setprio(3T)

The function sets the priority of the specified thread. The priority must be greater than or equal to zero. A greater value specifies an increased thread scheduling priority.

thr_getprio(3T)

The function gets the thread priority of the specified thread.

thr_yield(3T)

The function yields the caller's executing status to any thread with the same or a higher priority.

Synchronization 4

This chapter describes the synchronization mechanisms of Solaris Threads. It also provides pointers on selecting the most appropriate synchronization tool in various situations.

Synchronization Variables

The thread synchronization facilities synchronize threads both within a process and between processes. Any synchronization variable can be statically or dynamically allocated and must be allocated in memory that is globally accessible. Initializing the synchronization variable to zero specifies the default implementation variant in the default initial state. You can usually specify variations on the synchronization operation such as interprocess.

Interprocess Sychronization

Threads in two or more processes can use a single synchronization variable jointly. This requires that the variable be allocated in memory that is shared by the processes through mapping files or System V shared memory. Refer to `mmap`(2) and `shmop`(2). Sharing the synchronization variable must also be specified when it is initialized by specifying `USYNC_PROCESS`. (For a local synchronization variable, specify `USYNC_THREAD`.) The variable can be initialized only by the process that first uses it.

Interprocess synchronization variables can also be placed in files. Once initialized, subsequent processes can then map the file in and use the synchronization variable. For example, an initialized interprocess mutual exclusion lock synchronization variable (see below) can be stored in a file. Processes can map the file into their address space, acquire the mutual exclusion lock when they need to access it, access the file, release the lock, and unmap the file. The interprocess mutual exclusion lock ensures that other processes doing the same thing concurrently do not access or modify the data in the file while the process that has acquired the mutual exclusion lock holds it.

Mutex Lock

Mutual exclusion (mutex) locks let one thread at a time hold the lock. They are typically used to ensure that only one thread at a time executes a section of code (called a critical section) that accesses or modifies some shared data. They use little memory and are fast. Use mutex locks to limit access to a resource to one thread at a time. Mutex locks can also be used to preserve code that is single threaded. Mutex locks let only the thread that sets a lock release it. See the `mutex`(3T) manpages.

`mutex_init(3T)`

The function initializes the mutual exclusion lock to unlocked. Arguments specify the mutex variable and its type. Mutex locks can also be initialized to the default variant type (with `USYNC_THREAD`) by allocating them in zeroed memory.

`mutex_destroy(3T)`

The function destroys the specified mutex lock, initialized by `mutex_init()`.

`mutex_lock(3T)`

The function acquires the lock or blocks the calling thread if the lock is already held. Blocked threads wait on a prioritized queue.

mutex_trylock(3T)

The function enables asynchronous polling of the specified lock. It acquires the lock if it is free and does not block (returns an error) if the lock is held. `mutex_trylock()` can be used to avoid deadlock in operations that would normally violate the lock hierarchy.

mutex_unlock(3T)

The function unlocks the mutex lock. `mutex_unlock()` must be called by the thread that holds the mutex lock, which must be locked. If other threads are waiting for the lock, the thread at the head of the queue is unblocked.

Mutex locks can be used to synchronize access to data and to ensure that changes to data are seen atomically in all threads (an atomic memory access preserves sequentiality):

```
mutex_t count_mutex;
int count;

increment_count()
{
    mutex_lock(&count_mutex);
    count = count + 1;
    mutex_unlock(&count_mutex);
}

int
get_count()
{
    int c;

    mutex_lock(&count_mutex);
    c = count;
    mutex_unlock(&count_mutex);
    return (c);
}
```

Code Example 4-1 Mutex Lock Example

The two functions in the example use the mutex lock for different purposes. `increment_count()` uses the mutex lock simply to assure an atomic update of the shared variable. `get_count()` uses the mutex lock to guarantee that memory is synchronized when it refers to `count`.

Condition Variable

Condition variables are used to wait until a particular condition is true. A condition variable must be used in conjunction with a mutex lock. See also the `condition`(3T) manpages.

`cond_init(3T)`

The function initializes the condition variable. Arguments specify the condition variable and its type. Condition variables can also be initialized to the default type (with `USYNC_THREAD`) by allocating them in zeroed memory.

`cond_destroy(3T)`

The function destroys the state of the specified condition variable, initialized by `cond_init`().

`cond_wait(3T)`

The function blocks until the condition is signaled. It atomically releases the associated mutex lock before blocking, and atomically reacquires it before returning. In typical use, a condition expression is evaluated under the protection of a mutex lock. If 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 try to reacquire the mutex lock. Since reacquiring the mutex lock can be blocked by other waiting threads, the condition that caused the wait must be retested after the mutex lock has been acquired.

Since the condition is protected by the mutex lock, it can be a nontrivial expression. There is no guaranteed order of acquisition if more than one thread blocks on the condition variable.

`cond_signal(3T)`

The function signals one of the threads blocked in `cond_wait`().

```
mutex_t count_lock;
cond_t count_nonzero;
unsigned int count;
```

```
decrement_count()
{
    mutex_lock(&count_lock);
    while (count == 0)
        cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    mutex_unlock(&count_lock);
}
increment_count()
{
    mutex_lock(&count_lock);
    if (count == 0)
        cond_signal(&count_nonzero);
    count = count + 1;
    mutex_unlock(&count_lock);
}
```

Code Example 4-2 Condition Variable Use

cond_broadcast(3T)

The function wakes all the threads blocked in `cond_wait()`. Since `cond_broadcast()` causes all threads blocked on the condition to contend again for the mutex lock, it should be used with care. For example, you can use `cond_broadcast()` to allow threads to contend for variable amounts of resources when the resources are freed.

```
mutex_t rsrc_lock;
cond_t rsrc_add;
unsigned int resources, waiting;

get_resources(int amount)
{
    mutex_lock(&rsrc_lock);
    while (resources < amount) {
        waiting++;
        cond_wait(&rsrc_add, &rsrc_lock);
    }
    resources -= amount;
    mutex_unlock(&rsrc_lock);
}

add_resources(int amount)
{
```

```
    mutex_lock(&rsrc_lock);
    resources += amount;
    if (waiting > 0) {
        waiting = 0;
        cond_broadcast(&rsrc_add);
    }
    mutex_unlock(&rsrc_lock);
}
```

Code Example 4-3 Condition Variable Broadcast

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

Lost Wake Up Problem

Calling `cond_signal()` or `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 signal or broadcast has been sent but a thread is waiting on the condition variable even though the condition is true. This happens when the thread that calls `cond_signal()` does not hold the mutex locally. If the thread calls `cond_signal()` when another thread is between the test of the condition and the call to `cond_wait()` there are no waiting threads and the signal has no effect.

`cond_timedwait(3T)`

The 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 that the condition can be retested efficiently without recomputing the time-out value, as follows

```
timestruc_t to;
mutex_t m;
cond_t c;
...
mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
    err = cond_timedwait(&c, &m, &to);
    if (err == ETIME) {
        /* timeout do something */
        break;
```

```
    }
}
mutex_unlock(&m);
```

Code Example 4-4 Timed condition wait example

Semaphores

Solaris Threads provides conventional counting semaphores. A semaphore is a nonnegative integer count that is incremented and decremented. It is not as efficient as a mutex lock. They need not be acquired and released by the same thread, so they may be used for asynchronous event notification (such as in signal handlers). Since they contain state, they can be used asynchronously without acquiring a mutex lock as required by condition variables. See the `semaphore`(3T) manpage.

`sema_init(3T)`

The function initializes the semaphore variable. Arguments specify the semaphore variable, its initial state, and its type.

`sema_destroy(3T)`

The function destroy the state of the specified semaphore, initialized by `sema_init`().

`sema_wait(3T)`

The function blocks the thread until the semaphore becomes greater than zero, then decrements it.

`sema_trywait(3T)`

The function decrements the semaphore if its count is greater than zero. Otherwise it returns an error.

`sema_post(3T)`

The function increments the semaphore, potentially unblocking a waiting thread.

The following code is an example of a producer thread and a consumer thread that synchronize through a semaphore:

```
int rdptr = 0;
int wrptr = 0
data_t buf[BUFSIZE];
sema_t sem;
```

Thread 1

```
while (work_to_do) {
    buf[wrptr] = produce();
    wrptr = (wrptr + 1) % BUFSIZE;
    sema_post(&sem);
}
```

Thread 2

```
while (work_to_do) {
    sema_wait(&sem);
    consume(buf[rdptr]);
    rdptr = (rdptr + 1) % BUFSIZE;
}
```

Code Example 4-5 Producer/Consumer Example with Semaphore

Read/Write Locks

A multiple-reader, single-writer lock allow many threads simultaneous read-only access to a protected object. It also allows a single thread write access to the object while excluding any readers. This type of lock is also called a read/write lock and is usually used to protect data that is read more often than it is written. Like mutex locks, read/write locks must bracket. See the `rw_lock`(3T) manpage.

`rwlock_init(3T)`

The function initializes the read/write lock and sets its state to unlocked. Arguments specify the lock variable and type. Read/writer locks can also be initialized to the default type (with `USYNC_THREAD`) by allocating them in zeroed memory.

`rwlock_destroy(3T)`

The function destroys the state of the specified read/write lock, initialized by `rwlock_init`().

`rw_rdlock(3T)`

The function acquires a read lock. It blocks if a writer holds the lock.

rw_tryrdlock(3T)

The function acquires a read lock or returns an error.

rw_wrlock(3T)

The function acquires a write lock. It blocks if a writer or any readers hold the lock.

rw_trywrlock(3T)

The function acquires a write lock or returns an error.

rw_unlock(3T)

The function unlocks a read/write lock if the caller holds either the read or write lock.

The following example is of a bank account. 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:

```
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;
```

```
    savings_balance = savings_balance + amount;
    rw_unlock(&account_lock);
}
```

Code Example 4-6 Read/Write: Bank account

Comparing Primitives

The most basic synchronization primitive in Solaris 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 Solaris Threads is the condition variable. The basic use of a condition variable is to block on a change of state. Remember that a mutex lock must be acquired before blocking on a condition variable and must be unlocked after returning from `cond_wait()` and after changing the state of the variable.

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. Semaphores do not require that the caller that set a state is the one that changes the state.

The multiple-reader, single-writer lock is the most complex Solaris Threads synchronization mechanism. This means that the read/write lock is most efficiently used with a much coarser granularity than is effective with the other synchronization primitives. The basic use of a read/write lock is a resource whose contents are searched more often than they are changed.

Signals 5

This chapter describes signal handling in a multithreaded process. It also describes some differences between signal handling in a single-threaded process and in a multithreaded process.

Signal Handling

Each thread has its own signal mask. This lets a thread block some signals while it uses memory or other state that is also used by a signal handler. All threads in a process share the set of signal handlers set up by `signal`(2) and its variants, as usual.

If 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. So, it affects all the threads in the process.

Signals are divided into two categories: traps and interrupts.

Traps (such as `SIGILL`, `SIGFPE`, `SIGSEGV`) result from execution of a specific thread and are handled only by the thread that caused them. Several threads in a process could generate and handle the same type of trap simultaneously.

Interrupts (such as `SIGINT`, `SIGIO`) are asynchronous with any thread and result from some action outside the process. An interrupt can be handled by any thread whose signal mask has it enabled. If more than one thread is able to receive the interrupt, only one is chosen. One result is that several threads can be in the process of handling the same kind of signal simultaneously. If all threads mask a signal, it will remain pending on the process until some thread enables the signal. As in single-threaded processes, the number of signals received by the process is less than or equal to the number sent. For example, an application can enable several threads to handle a particular I/O interrupt. As each new interrupt comes in, another thread is chosen to handle the signal until all the enabled threads are active. Any additional signals remain pending until a thread completes processing and re-enables the signal.

Threads can send signals to other threads in the process through `thr_kill`(3T). A signal initiated through `thr_kill`() behaves like a trap and is handled only by the specified thread. As usual, a signal is sent to another process by `kill`(2) or `sigsend`(2). There is no direct way for a thread in one process to send a signal to a specific thread in another process.

Unbound threads are not allowed to use alternate signal stacks. A thread that is bound can use an alternate stack because the state is associated with the execution resource. (Bound threads are described in Chapter 7, "Advanced Topics.") An alternate stack must be enabled for the signal, through `sigaction`(2), and declared and enabled, through `sigaltstack`(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 0.)

Non-Local Goto

The scope of `setjmp`() and `longjmp`() is limited to one thread. In most threads, this is not an issue. However, this means that a thread that handles a signal can only perform a `longjmp`() if the `setjmp`() was performed in the same thread.

Async-Safe Functions

Functions that may be called from signal handlers are said to be *async-safe*. In the Solaris threads library the following functions are async-safe in addition to those defined by POSIX (IEEE Std 1003.1-1990, 3.3.1.3 (3)(f), page 55):

- `sema_post`()
- `thr_sigsetmask`()
- `thr_kill`()

Thread Signal Interface

Each thread has its own signal mask. A thread's signal mask applies equally to intra-process signals (signals sent between threads) and to interprocess signals.

thr_sigsetmask(3T)

The function sets the thread's signal mask. A thread's initial signal mask is inherited from the parent thread.

thr_kill(3T)

The function causes the specified signal to be sent to a specified thread.

Waiting for Signals

A new way to handle asynchronous signals is to wait for them in a separate thread. This is may be safer and easier than installing a signal handler and processing them there.

sigwait(2)

The function waits for a pending signal from the set specified by the argument, regardless of the signal mask. `sigwait`() clears the pending signal and returns its number. If more than one thread is waiting for the same signal than one thread is chosen and it returns from `sigwait`().

`sigwait`() is typically used by creating one or more threads to wait for signals. Since `sigwait`() can even retrieve masked signals, the signals of interest are usually blocked 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-safe functions and can synchronize with other threads in the usual way.

```
mutex_t m;
int hup = 0;

main()
{
    thread_t t;
```

```
    int finishup = 0;
    sigset_t set;
    ...
    sigemptyset(&set);
    sigaddset(&set, SIG_... );
    thr_sigsetmask(SIG_BLOCK, &set, NULL);
    thr_create(NULL, 0, handle_hup, NULL, THR_DETACHED, &t);
    do {
        stuff();
        mutex_lock(&m);
        if (hup)
            finishup = 1;
        mutex_unlock(&m);
    } while (finishup == 0);
}

handle_hup()
{
    sigset_t set;

    sigemptyset(&set);
    sigaddset(&set, SIGHUP);
    sigwait(&set);
    mutex_lock(&m);
    hup = 1;
    mutex_unlock(&m);
}
```

Code Example 5-1 Waiting for Signals with `sigwait()`

Threads and Process Resources 6

This chapter discusses how threads affect process creation and destruction. Then it describes how threads interact with other process resources, such as timers and resource limits.

Process Creation and Destruction

When a process executes the `fork`(2) system call, it creates an exact duplicate of the process. The address space and all the threads (and LWPs, see "Lightweight Processes" on page 35) are duplicated in the child. When one thread in a process calls `fork`(), it can cause interruptible system calls in the other threads to return `EINTR`.

A new system call, `fork1`(2), `duplicates` the complete address space but only the invoking thread. No other thread or LWP from the original process is duplicated in the new process.

`vfork`(2) behaves similarly to `fork1`() 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.

Both the `exit`(2) and `exec`() system calls work as they do in single thread processes except that they destroy all the threads in the address space. Both calls block until all the execution resources (hence all active threads) are destroyed. When `exec`() rebuilds the process, it creates a single LWP. The process start-up code builds the initial thread. As usual, if the initial thread returns, it calls `exit`() and the process is destroyed.

If all the threads in a process exit, then the process itself exits with a status of zero.

Differences Between `fork()` *and* `fork1()`

`fork()` has two basic uses. It is used to duplicate the entire process (the BSD `dump` program uses this technique). Or it is used to create a new process before calling `exec(2)`. For the latter purpose, `fork1()` is much more efficient than `fork()` because only one LWP and thread are duplicated.

Having `fork()` completely duplicate the process is most similar to the single-threaded `fork()`. It presents fewer pitfalls. Having `fork1()` fork only one thread provides an optimized `fork` and that can be immediately followed by `exec()`.

Hazards of `forking`

There are a number of hazards to using the variations of `fork`.

- `fork1()`: The child is a single threaded clone of its parent. If the parent process has more than one thread that are holding locks, the child process contains locks that are held by nonexistent threads. If the child tries to use one of these locks, it deadlocks.
- `fork1()`: Before invoking `exec()`, call no library functions, since one may use a lock that is held by more than one thread.
- `fork()` and `fork1()`: Be careful when using global state after a call to `fork()`. For example, if a thread reads a file serially and another thread in the process calls `fork()`, there is then a thread in both processes reading the files after the `fork()` completes. Since the seek pointer for a file descriptor is shared after a `fork()`, the thread in the parent will get some data while the thread in the child will get the rest.
- `fork()` and `fork1()`: Locks that are allocated in memory that is sharable (that is `mmap()`'ed with the `MAP_SHARED` flag) can cause the lock to be held in both processes, unless care is taken to avoid this.
- `vfork()`: The thread in the child process must be careful not to change memory before it calls `exec(2)`. Remember that vfork() gives the parents 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()`.

popen(3S) and dup2(3C) both call fork(), whose behavior must be accounted for.

Thread-Specific Data

Thread-specific data (TSD) 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.

thr_keycreate(3T)

The function allocates a global key value. The caller can optionally specify a function to be called when a thread that has a non-NULL value associated with the allocated key exits. The function is called with the current value bound to the key for the exiting thread as an argument. The function, called a destructor function, typically frees data pointed to by the bound value.

thr_setspecific(3T)

The function binds a value to the key for the calling thread.

thr_getspecific(3T)

The function retrieves the current value bound to the key for the calling thread.

Use

When thread-specific data is used in a function that can be called at any time, the function must detect the first call to initialize the key, as follows:

```
thread_key_t my_key;
int once = 0;
mutex_t keylock;

my_func()
{
    my_data_t *ptr;
```

```
    mutex_lock(&keylock);
    if (!once) {
        once = 1;
        thr_keycreate(&my_key, free);
    }
    mutex_unlock(&keylock);
    /* no locking beyond here since data is per-thread */
    thr_getspecific(my_key, &ptr);
    if (ptr == NULL) {
        /* first time called in this thread
        ptr = (my_data_t *)malloc(sizeof(my_data_t));
        thr_setspecific(my_key, ptr);
    }
    my_use_it(ptr);
}
```

Code Example 6-1 Self-initializing Thread-specific Data

Note that the key allocation and storage must be protected by a lock so if two threads call the same function simultaneously, they do not both allocate keys.

Resource Limits

The resource limits are set on the entire process (in other words, the sum of the resource use of all the LWPs in the process). When a soft resource limit is exceeded, the offending LWP is sent the appropriate signal. The sum of the resource use (including CPU use) for all LWPs (and thus threads) in the process is available through `getrusage()`.

Advanced Topics 7

The threads library is the programmer's interface for multithreading. Earlier chapters explained that threads were scheduled onto execution resources. These resources are kernel threads, called lightweight processes (LWP). Only programmers who need to design and implement a threads library of their own will use the LWP interface directly. For all others, the Solaris Threads library performs all needed use of the LWP interface.

This chapter describes LWPs and how threads relate to them. Most of the details in this chapter are implementation dependent.

Lightweight Processes

Solaris Threads are implemented as a library, using underlying threads of control, called lightweight processes, that are supported by the kernel. An LWP can be thought of as a virtual CPU that executes code or system calls. The threads library schedules threads on a pool of LWPs in the process, in much the same way as the kernel schedules LWPs on a pool of processors. Each LWP is independently dispatched by the kernel, performs independent system calls, incurs independent page faults, and runs in parallel on a multiprocessor system. The LWPs in the system are scheduled by the kernel onto the available CPU resources according to their scheduling class and priority.

Not 4.0 LWP

SunOS Release 4.0 introduced a lightweight process library. The SunOS 4.0 LWP library supports a form of multithreaded programming with no kernel support. While Solaris Threads is generically similar to the SunOS 4.0 LWP library, the interface is completely different. There are other significant differences in implementation.

Solaris 2.2 contains neither support for nor emulation of the SunOS 4.0 LWP library. Any application that uses the 4.0 LWP library must be ported to Solaris Threads before it can be run under Solaris 2.2

Solaris Threads Implementation

The Solaris Threads library schedules threads onto lightweight processes (LWPs) to execute program code and system calls. Solaris Threads provides two ways for LWPs to execute threads: bound and unbound.

Unbound Threads

Threads that are scheduled on the pool of LWPs are called *unbound threads.* The library invokes LWPs as needed and assigns them to execute runnable threads. After assuming the state of the thread, the LWP executes its instructions. If the thread becomes blocked on a synchronization mechanism, or if another thread should be run, the state of the thread is saved in process memory. The threads library then assigns another thread to the LWP to run.

Bound Threads

In some cases, a thread may require a special capability. For example, a thread may need to be scheduled on a real-time basis in strict priority with respect to all the other active threads in the system. Since unbound threads are only scheduled within a process they are not scheduled with respect to threads outside the process. A programmer can take advantage of the extra capabilities while retaining use of all the thread interfaces and capabilities by specifying that the thread is to be permanently bound to an LWP when it is created. Such threads are called *bound threads*.

Thread Library Scheduling

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 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 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.

The library usually ensures that there are enough LWPs in its pool for a program to proceed. If all of the LWPs in the process are blocked in indefinite waits (for example 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. If 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()` (see page 15) or by using `THR_NEW_LWP` in calls to `thr_create()` (see page 13 and page 38).

A contraction in the number of active threads can cause an excessive number of LWPs in the pool of a process. If LWPs in the pool remain idle for a certain period (that is, there are more LWPs than active threads) the threads library destroys the unneeded ones. The library “ages” LWPs. They are deleted if they are unused for a “long” time, currently set at 5 minutes.

Scheduling Interface

Scheduling of threads is controlled through the Solaris Threads library functions `thr_setconcurrency()`,`thr_setprio()`, and the `THR_NEW_LWP` option of `thr_create()`. The default number of LWPs created by the library is not always the optimum number for performance. The library may create too few or too many LWPs. You can influence the number of LWPs by creating them with the `THR_NEW_LWP` flag or with `thr_setconcurrency`().

Unbound thread scheduling uses simple priority levels with no adjustments and no kernel involvement. Thread scheduling regulates only how threads are assigned to LWPs. It has no effect on scheduling of LWPs by the kernel. The LWP's system priority is usually uniform and is inherited from the creating process.

thr_create(3T)

The function creates and starts a new thread as described in Chapter 3, "Thread Creation and Scheduling. There is a range of options between purely Solaris Threads library based scheduling and purely kernel-based scheduling. The options are selected by the `THR_BOUND` and `THR_NEW_LWP` flags.

Any combination of the options shown can be specified in the `flags` argument:

`THR_SUSPENDED`	The thread is immediately suspended after it is created. The thread does not execute the specified function until another thread executes `thr_continue`() to start it. If `THR_SUSPENDED` is not specified, the thread is runnable immediately.
`THR_DETACHED`	The new thread is detached. When a detached thread exits, its identifier, status, and other resources may be immediately reused. Other threads cannot wait for this thread to exit. A detached runnable thread may run and exit before `thr_create()` returns unless other synchronization prevents it.
`THR_BOUND`	The new thread is permanently bound to a dedicated LWP.
`THR_NEW_LWP`	A new LWP is created along with the thread. The new LWP is added to the pool of LWPs used to execute threads. This is the same as incrementing the concurrency level by calling `thr_setconcurrency()`.

When to Bind

Bind a thread to obtain one of the following properties of an LWP:

- Have the thread scheduled globally (on a system-wide basis instead of on a process basis), for example real time
- Give the thread an alternate signal stack
- Give the thread an unique alarm or timer

thr_suspend(3T)

The function immediately suspends execution of the specified thread. thr_suspend() returns when the specified thread is stopped. A thread that specifies itself to thr_suspend() must be restarted by some other thread.

thr_continue(3T)

The function restarts a thread that has been suspended via thr_suspend() or that was created suspended.

LWP Scheduling

The Solaris 2.2 kernel has three classes of process scheduling. Scheduling class is maintained for each LWP. 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. The highest priority scheduling class is real time (RT). The middle priority scheduling class is system. The system scheduling class cannot be applied to a user process. The lowest and default priority scheduling class is timeshare (TS).

When a process is created, its one initial LWP inherits the scheduling class and priority of 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.

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.

The scheduling class is set by the `priocntl`(2) system call. It can affect just the calling LWP or all the LWPs of one or more processes, as specified by the first two arguments. The third argument of `priocntl`() is the command, which may be one of:

`PC_GETCID`	Ignores the first and second arguments. The fourth argument points to a `pcinfo_t` structure, defined in `<sys/priocntl.h>`. If the name field of the structure contains a recognizable class name (`RT` or `TS`), the class id and an array of class attribute data are returned.
`PC_GETCLINFO`	Ignores the first and second arguments. The fourth argument points to a `pcinfo_t` structure, defined in `<sys/priocntl.h>`. If the id field of the structure contains a recognizable class identifier (`RT` or `TS`), the class name and an array of class attribute data are returned.
`PC_GETPARMS`	Returns the class identifier and/or the class specific scheduling parameters of the process(es) specified by the first two arguments. The results are returned in the `pcparms_t` structure pointed to by the fourth argument.
`PC_SETPARMS`	Sets the class identifier and/or the class specific scheduling parameters of the process(es) specified by the first two arguments. The results are returned in the `pcparms_t` structure pointed to by the fourth argument.

Timeshare Scheduling

Timeshare scheduling is designed to provide a fair distribution of the processing resource to the set of processes. Other parts of the kernel can monopolize the processor for an interval commensurate with human reaction times, without degrading response time as seen by the user.

The dispatch priority of time-shared LWPs is calculated from the instantaneous CPU use rate of the LWP, and from its `nice`(1) level. The `nice`() level indicates the relative priority of the processes 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. In general, this biases priority in favor of I/O bound processes since they need few CPU cycles to keep going.

The `priocntl`(2) system call sets the `nice`() level of one or more processes. `priocntl`() affects the `nice`() level of all the timesharing class LWPs in the process. The `nice`() level ranges from -19 to +20. The less the value, the higher the priority.

Real-time Scheduling

The real-time class (RT) can be applied to a whole process or to one or more LWPs in a process. Unlike the `nice`() level of the timeshare class, LWPs that are classified real-time can be assigned priorities individually or jointly.

The scheduler always dispatches the highest priority real-time process or LWP. It preempts a lower-priority process or LWP when a higher-priority process or LWP becomes runnable. A preempted LWP is placed at the head of its level queue. A real-time LWP retains control of a processor until it is preempted, it suspends, or its real-time priority is changed. Processes and LWPs in the `RT` class have absolute priority over processes in the `TS` class. Given the latency of the system in recognizing interrupts and dispatching the highest priority process, the worst case behavior of the processes can be calculated.

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 real-time process, or the time slice expires. An LWP with an infinite time slice ceases execution only when it terminates, blocks, or is preempted.

Programmer Control of Real-time Scheduling

The `priocntl`() system call sets the real-time priority and time-quantum of one or more processes. A `priocntl`() call affects the attributes of all the real-time LWPs in the process.

Alarms, Interval Timers, and Profiling

Solaris Threads provides an unique real-time interval timer and alarm per LWP. This results in an interval timer for each bound thread. The timer or alarm delivers one signal to the thread when the timer or alarm expires.

Unbound threads do not have virtual time or profile interval timers (this can be expensive to maintain for threads). If this is required, threads can be bound to an LWP and use the underlying LWPs virtual time or profile interval timer. When these interval timers expire either `SIGVTALRM` or `SIGPROF`, as appropriate, is sent to the LWP that owns the interval timer.

Profiling is enabled for each LWP individually (see `profil`(2)). Each LWP can have a separate profiling buffer. A profiling buffer can be shared by two or more LWPs if accumulated data is desired. Profiling data is updated at each clock tick in LWP user time. The state of profiling is inherited from the creating LWP.

Stacks

There is no general need to allocate stack space for threads. The threads library allocates a 1 megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of `mmap`(2) to do the allocations.)

Each thread stack created by the threads library has a red zone. A stack red zone is made by rounding the stack size to the next page boundary and appending an invalid page. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

Don't specify stacks or their sizes to `thr_create`() unless you're absolutely certain you know it's correct. There are very few occasions when it is sensible to specify a stack, its size or both to `thr_create`(). It is difficult even for an expert to know if the size was specified right. The problem is that an ABI compliant program can't determine its stack size statically. Its size is dependent on the needs of the particular runtime in which it executes.

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

thr_min_stack(3T)

The function specifies the minimum stack size for all threads. The size of any stack specified to thr_create() must be greater than or equal to the minimum. This is equally true for stacks passed to the threads library and for stacks allocated by the threads library.

thr_min_stack() should be used only when it is essential to create a thread with a special stack. The value returned is the amount of stack space needed to execute a null procedure.

Using Threads 8

This chapter provides some guidelines to how to use threads.

Concurrency Strategies

As stated in Chapter 2, “Threads Basics,” the threads in an application must cooperate and synchronize when sharing the data and the resources of the process.

Single-Lock 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. This strategy can be quite effective on a uniprocessor, provided care is taken to ensure that shared memory is put into a consistent state before the lock is released and that the lock is released often enough so that more than one thread can run. In fact, many operating system implementations have taken this approach.

Unfortunately, this approach cannot take advantage of more than one processor and will reduce uniprocessor concurrency if the lock is not dropped during most I/O. It also requires cooperation from all the modules and libraries in the system to synchronize on the single lock.

Reentrance

An alternate approach is to let each module or library independently synchronize itself. This allows module synchronization to be hidden beneath the module interface. In other words, several threads may call functions in a module simultaneously and it is up to the function code to handle this. Such functions are said to be reentrant (also referred to as mt-safe). The assumption is that data shared between threads is only accessed by module functions. If mutable global data is available to clients of a module then appropriate locks must also be made visible in the interface, and the module function cannot be made reentrant unless the clients are assumed to use the locks at the appropriate times.

Functions, such as `sin()`, that access no global data or access only read-only data and use no other global state (for example files) are trivially reentrant. Trivially reentrant functions can be entered by any number of threads simultaneously.

Functions that do access global state must be made reentrant by restricting potential concurrency.

The greatest performance benefit in multithreaded programs is obtained when asynchronous processing is maximized. In other words, minimum synchronization of threads produces the greatest to throughput. Code that is executed asynchronously must be reentrant. For purposes of asynchronous execution of Solaris Threads, reentrant code has the following characteristics:

- Never changes any element of global memory
- Never changes the state of a file or device
- Refers to an element of global memory only in special circumstances

Any point in a multithreaded program that performs an action counter to any of these rules must be protected by one of the synchronizing mechanisms provided by Solaris Threads.

Locking Strategies

There are two basic strategies for making functions in modules reentrant: code locking and data locking.

Code Locking

Code locking guarantees that at most one thread is in module code at any one time. The code covered by the lock (usually a mutex lock) is called the *critical section* for the lock. There is one lock in the process for the module code. Multiple threads can be active in the process when they are executing different modules code. This is sometimes called a monitor.

Data Locking

Data locking guarantees that access to a *collection* of data that must be maintained consistently is synchronized. For data locking, there is still the concept of a critical section for the lock, but now there can be several threads in the critical section concurrently. For a mutual exclusion locking protocol, only one thread may be in the critical section per collection of data. Alternatively, in a multiple readers, single writers protocol, several readers may be allowed per collection of data or one writer. Multiple threads can execute in a single module provided they operate on different data collections or 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.

The collection of data associated with a lock can be any collection of data items that must be maintained consistently as a group. An extreme application of this principle would always lock the minimum consistent collection. While this may appear to maximize concurrency, it can cause severe performance degradation. The locking primitives are relatively cheap to use when there is no contention, but they are not free. Too fine a grain of data locking may cause locks to be acquired and released every few instructions.

Invariants

For both code locking and data locking *invariants* are important as a means 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 not set. When the lock is set, the invariant can be false. However, the code holding the lock must re-establish the invariant before releasing the lock.

For example, in a module that manages a doubly-linked list of elements a good invariant is, for each item on the list, the forward pointer to the next item on the list and the backward pointer to the previous item. Assume the 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, there is a point in the manipulation of the pointers where the invariant is false. However the invariant is reestablished before the mutex lock is released.

In both code-based and data-based locking, you must still be careful not to hold locks across long operations. For example, if the client RPC code was locked with single global mutex lock that was held for the duration of the call, the client application could only have one RPC in progress no matter how many threads it used. The usual technique is to drop all module locks before starting the long operation or the I/O. When the operation completes, the lock(s) can be reacquired. If everything is working, the invariant will be true after the lock(s) are reacquired but the *state* of the module may have changed. For example, the item that the function was about to delete from a list may have already been deleted. After the lock(s) are reacquired, the state must be reverified to be sure the intended operation is still valid.

This illustrates another reason why excessive attempts at increasing concurrency can actually degrade performance. Every time the lock(s) are dropped the module state must potentially be reevaluated.

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 you write a code monitor in which every module function grabs the mutex lock for the duration of the call, then any call between functions protected by the mutex lock will immediately deadlock. If a function calls some code outside the module which, through some devious 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 where 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 lock(s) are reacquired, the state must be reverified 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 acquire 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 hence a deadlock.

This kind of deadlock is avoided by establishing an order in which locks are acquired (called a *lock hierarchy*). If all threads always acquire locks in the specified order, then this deadlock is avoided.

Adhering to a strict order of lock acquisition is not always desirable. If thread 2 has built up a considerable number of assumptions about the state of the module while holding mutex lock B, giving up mutex lock B to acquire mutex lock A then reacquire mutex lock B in order would cause it to discard its assumptions and reevaluate the state of the module. The blocking synchronization primitive usually have variants that attempt to get a lock and fail if they cannot, e.g. `mutex_trylock()`. This allows threads to violate the lock hierarchy in cases where there is no contention. Where there is contention, the held locks must usually be discarded and the locks reacquired in order.

Scheduling Deadlocks

A common problem in threaded programs is that a particular thread never acquires a lock (usually a condition variable), even though it seems that the thread should acquire the lock. This usually happens when the thread that holds the lock releases it, lets a small amount of time pass, and then reacquires it. Since the lock was released, one would think 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 no other thread is run.

This type of problem can usually be solved by putting a call to 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 none. 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 duration operations like I/O where this can impact performance.
- Don't hold locks when calling a function outside the module that may reenter the module.
- Don't try for excessive processor concurrency. Without intervening system calls or I/O operation, locks are usually held for short amounts of time and contention is rare. Only attack those locks that actually have measured contention. In data based locking, first try a lock for each object, if the code is structured that way. This is usually sufficient.
- When using multiple locks, be sure that all threads acquire the locks in the same order to avoid deadlocks.

Thread Creation

Solaris Threads caches threads data structure, stacks, and LWPs so repetitive creation of unbound threads can be cheap. Unbound thread creation is very inexpensive when compared to process creation or even to bound thread creation. It is roughly the same order of magnitude as is unbound thread synchronization, when you include the context switches to stop one thread and start another. So, creating and destroying threads as required is usually better than attempting to manage a pool of threads that wait for independent things to do. In other words, create a thread to do an asynchronous job that must maintain an execution history for the duration of the job being processed. A good example 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 cheap when compared to process creation, it is not cheap when compared to the cost of a few instructions. Threads should be created for processing that lasts at least a couple of thousand machine instructions.

Thread Concurrency

As stated previously, 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. It cannot make perfect decisions, but it at least ensures that the process continues to make progress. If the application has some idea of the number of (unbound) threads that should be simultaneously active (executing code or system calls), it should tell the library via `thr_setconcurrency`(). For example, a database server that has a thread per-user should tell Solaris Threads the expected number of simultaneously active users. A window server that has one thread per 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 once, via the `THR_NEW_LWP` flag, as each thread is created.

For the purpose of computing concurrency, unbound threads blocked on interprocess (`USYNC_PROCESS`) synchronization variables are considered active. Active bound threads should not be counted.

Efficiency

A new thread is created via `thr_create`() 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`() when it has completed its task than it would be to stockpile an idle thread and restart it.

Bound Threads

Bound threads are more expensive than unbound threads. Since bound threads may change the attributes of the underlying LWP, Solaris Threads does not cache the LWPs when the bound threads exit. Solaris Threads asks the operating system for a new LWP when a bound thread is created and destroys it when the bound thread exits. Bound threads should only be used when a thread needs resources that are only available through the underlying LWP, such as a virtual time interval timer or an alternate stack, or the thread must be visible to the kernel to be scheduled with respect to all other active threads in the system, as in real-time scheduling.

Unbound threads should be used even when you expect all threads to be simultaneously active. This allows Solaris Threads to cache LWP and thread resources efficiently to make thread creation and destruction fast.

Thread Creation Guidelines

Some simple guidelines for using threads:

- Use threads for independent jobs that must maintain state for the life of the job.
- Don't use threads for very short jobs.
- Use threads to take advantage of CPU concurrency.
- Only use bound threads when absolutely necessary, that is, when some facility of the underlying LWP is required.
- Use `thr_setconcurrency()` to tell Solaris Threads how many threads you expect to be simultaneously active.

Guidelines for Threaded Programs and Libraries

These guidelines have been found important in developing threaded function libraries at Sun. Many of them apply also to developing threaded programs, in general.

- You must know what you import and whether it is safe.

  A threaded program cannot arbitrarily enter non-threaded code.

- Threaded code may refer to unsafe code only from the main thread.

  This ensures that the static storage associated with the main thread is used only by that thread. See "Compatibility Issues With Global Error Variables" on page 55

- Sun-supplied libraries are defined to be *safe* unless explicitly documented 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.

- The initial thread must protect shared data structures referred to by unsafe code.

Having only the initial thread call unsafe code is not enough to protect shared data structures. An example of this is the `stdio` buffers in the C library. The initial thread must either ensure there is no concurrent access to `stdio` buffers when unsafe code is active or it must explicitly lock those buffers (via `flockfile()`) while unsafe code is active.

- Compilation flags must be used to manage binary incompatible source changes.

  To enable programs that meet the portability constraints inherent in current ABIs, new facilities exhibiting a different binary behavior must be explicitly enabled. Always either specify `-D _REENTRANT` when compiling or be sure that `_REENTRANT` is defined before any header file is included.

- To maintain binary compatibility in libraries, old static storage with per-thread semantics must be used for the main thread.

  Where a pre-thread static storage cell requires per-thread use, the main thread must use the preexisting storage cell. This allows linking both threaded and nonthreaded object files (subject to the other use rules defined here). A good convention is to provide a function of the name *__<cell>()* where *<cell>* is the name of the previous instance. Header files declaring *<cell>* must be protected by a flag, such as `_REENTRANT`, as described above. When the flag is set, the expression *<cell>* should expand to an expression that results in an *lvalue*. An example of this is the definition of `errno`:

```
#ifdef _REENTRANT
#define errno *(int *)__errno()
#else
extern int errno;
#endif
```

  TSD is used to implement per-thread static storage.

- 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 had been changed to per thread operation, threads would not be able to cooperate in accessing files.

  For thread-specific behavior, or "thread cognizant" behavior, use thread facilities. For example, if the termination of `main()` should terminate only the thread exiting `main()`, the end of `main()` should be:

```
thr_exit();
/*NOTREACHED*/
```

Libraries

mt-Safe Libraries

All routines that can potentially be called by a thread from an MT program must be mt-safe. This means that two, or more, activations of a single routine must be able to *correctly* execute concurrently. (Another way of saying this is that each of at least two threads must be able to correctly execute the same library function in parallel.) So, every library that an MT program links (that is, uses) must provide interfaces that are all mt-safe.

Not all libraries are now mt-safe. Some of the more commonly used libraries have been made mt-safe. These are:

- `lib/libc` - `get`*XX*`by`*YY* interfaces (such as `gethostbyname`(3N)) are mt-safe
- `lib/libm` - mt-safe only when compiled for the shared library, not mt-safe when linked with the archived library
- `lib/libw`
- `lib/libintl`
- `lib/libmalloc`
- `lib/libmapmalloc`
- `lib/libnsl` - including the TLI interface, XDR, RPC clients, netdir, and netselect (does not include RPC servers)
- `lib/libsocket`
- `lib/nametoaddr`
- `lib/nsswitch`
- `lib/libdl_stubs` (to support static switch compiling)
- `lib/libresolv` (thread-specific `errno` support)

Additional libraries, particularly the remainder of the networking libraries and the windows libraries, will also be modified to be mt-safe.

Warning – For this release, `libnsl` is mostly converted to mt-safe. RPC servers have not yet been made mt-safe.

---

mt-Safe Interfaces

Functions in the mt-safe libraries fall into four groups:

- Functions that have an mt-safe interface that have always been or have been modified to be completely mt-safe.
- Functions that are not mt-safe because their size would have increased too much.
- Functions that have an mt-unsafe interface. For example, returning a pointer to a buffer in the stack.
- Functions that are the equivalent of the third group with interfaces that have been modified to make them mt-safe. Functions in this group are identified by the suffix "_r".

Fully mt-safe functions are not changed in the mt-safe libraries. Use of these functions is transparent. Most functions with mt-safe interfaces and a mt-unsafe implementation have been modified to make them fully mt-safe. Use of these functions is also transparent.

There are still some small, mt-unsafe library functions that are left unchanged. The reference manual entry of each such function identifies it.

Caution – There is no way to be certain that a function whose name does not end in "_r" is mt-safe other than checking its reference manual page. Use of a function identified as not mt-safe must be protected by a synchronizing device or restricted to the initial thread.

For most functions with mt-unsafe interfaces, an mt-safe version of the routine has been developed for use in the multithread context. The name of the new mt-safe routine is always the name of the old mt-unsafe routine with "_r" appended. Any time you find an "_r" version of a familiar routine, use it in any multithreaded program.

Compatibility Issues With Global Error Variables

In nonthreaded applications, the variable `errno` has been used to return error codes from `libc` function library calls and is traditionally allocated in static, global storage. The same is true of `t_errno` for TLI functions. A global, shared error variable will not work in a multithread environment. So, one error variable is allocated and managed in per-thread storage for each thread via the thread specific data (TSD) facilities of libthread. You must include errno.h for the libc global error variable and tiuser.h for the TLI global error variable.

At a binary level, this results in a situation in which:

- Old object files (non-threaded), or new object files generated without the _REENTRANT flag, reference the static storage labelled errno;
- New object files generated with _REENTRANT reference a new binary entry point, __errno, a function that returns the address of the storage cell for that thread's definition of errno.
- Programs using TLI in libnsl must also be compiled with the _REENTRANT flag and similarly reference a new entry point, __t_errno, a function that returns the address of the thread's definition of t_errno.

The main point is that new threaded object modules should be linked with old object modules only with great caution.

Unsafe Libraries

Routines in libraries that are not guaranteed mt-safe can be safely called by multithreaded programs if such calls are made only from the initial thread.

Another method of using routines that are not mt-safe is by synchronizing the threads' access to these routines. The synchronized protection must include access to implicit elements of the interface, such as error code globals. This method may have some cost in performance.

The Threads Environment 9

This chapter describes the environmental requirements and compilation and linkage options required to use threads successfully.

Compilers

To compile and link a multithreaded program requires:

- Include files:
  - `thread.h`
  - `synch.h`
  - `errno.h`
- The standard SunPro™ C compiler
- The standard Solaris linker
- The threads library (`libthread`)
- MT-safe libraries (`libc`, `libm`, `libw`, `libintl`, `libmalloc`, `libmapmalloc`, `libnsl`, etc.)

Multithread programs must be compiled using the `-D _REENTRANT` flag. This applies for every module of a new application. If the `-D_REENTRANT` flag is omitted, the old definitions for `errno`, `stdio`, and so on, are used. The default is to create single-threaded programs compatible with previous releases.

There are some cases in which the flag is turned on by header files for multithreaded programs like `thread.h` and `synch.h`.

All calls to `libthread` instances are effectively no-ops if the application does not link `libthread`. `libc` has defined a set of `libthread` stubs that are `null` procedures. True procedures are interposed by `libthread` when the application links both `libc` and `libthread`. A threaded program must be constructed to ensure that `libthread` physically interposes on the C library. The behavior of the C library is undefined if a program is constructed with an `ld` command line that includes the fragment:

```
.o's ... -lc -lthread ...
```

A correct result is obtained when the command line only `-lthread` and no `-lc`.

Do not link a single-threaded program with `-lthread`. Doing so causes some mechanism that are required by multithreaded programs to be established at link time and initiated at run time. This wastes resources and produced misleading results when debugging.

Debugging Techniques 10

Common Bugs

Multithread programming differs enough from conventional synchronous programming that it is worth while to identify some of the most common oversights that can result in a bug.

- Accessing global memory (shared changeable state) without the protection of a synchronization mechanism.
- 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).
- Hidden gaps in synchronization protection. This is caused when a segment of code that is 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.
- Multithread programs have a more limited stack size than single-threaded programs. Deeply nested recursive calls and large automatic arrays can cause problems. Providing stack through means other than thread library calls can also cause problems.
- Multithread programs (especially buggy ones!) often behave differently in two successive runs given identical inputs, because differences in the order of thread scheduling. In general, multithreading bugs are statistical in stead of deterministic in character. Tracing is usually more effective in finding problems in the order of execution than is breakpoint-based debugging.

Using adb

The commands that have been added to `adb` for use with multithreaded programs are:

adb Command	Interpretation
pid`:A`	Attach to process # *pid*. This stops the process and all its LWPs.
`:R`	Detach from process. This resumes the process and all its LWPs.
`$L`	List all active LWPs in the (stopped) process.
n`:l`	Switch focus to LWP # *n*
`$l`	Show the LWP currently focused
num`:i`	Ignore signal number *num*

An example of the use of these commands is:

- Determine the process identifier of the multithread program you want to debug
- Invoke `adb` on the program
- Use *pid*`:A` to attach to the hung process
- Use `$L` to list all LWPs
- If there are more than one LWP, use *n*`:l` to switch to each LWP and determine its state at the time the process stopped using the standard `adb` commands `$c`, `$r`, and so on. For example `3:l` switches focus to LWP #3. Then `$c` displays the stack trace for LWP # 3 and a `$r` displays its registers.

You can have `adb` ignore signals using *num*`:i`. If the signal number *num* is received, the signal is passed to the process being debugged but is ignored by `adb`.

You can bind all threads in a multithread program. Then, a thread and an LWP are synonymous. Now when you use `adb`, each thread is accessible using the above commands.

Installing MT dbx

A `dbx` directory contains a man page (`dbx-man.cat`) a `dbx`, and a new shared library, `libthread_db.so.0.0`. The new MT commands are described in the man page. `dbx` can be installed anywhere as long as it can find `libthread_db.so.0.0`. You can place `libthread_db.so.0.0` in `/lib` (the default location of all shared libraries). Or you can set the environment variable `LD_LIBRARY_PATH` to a path where the linker can find this new shared library. To check if the linker can find this library, type `lld` *pathname*/`dbx`. The environment variable `LD_LIBRARY_PATH` includes the pathname where you've put `libthread_db.so.0.0`.

Example Applications A

File Copy

Generating several I/O requests at once so that the I/O access time can be overlapped is often advantageous. A simple example of this is file copying. If the input and output files are on different devices, the read access for the next block can be overlapped with the write access for the last block. Code Example A-1 shows some of the code.

The main routine creates two threads: one to read the input, one to write the output. Each `thr_create()` also adds an LWP to the pool on which threads are scheduled (`THREAD_NEW_LWP`), since the application requires full system resources for each thread. (Chapter 7, “Advanced Topics,” explains this subject.) The LWPs are not permanently bound to the thread so the threads package can destroy any that are not utilized.

The reader thread reads from the input and places the data in a double buffer. The writer thread gets the data from the buffer and continuously writes it out. The threads synchronize using two counting semaphores; one that counts the number of buffers emptied by the writer and one that counts the number of buffers filled by the reader.

```
sema_t emptybuf_sem, fullbuf_sem;

/* double buffer */
struct {
    char data[BSIZE];
    int size;
} buf[2];
```

```
reader()
{
    int i = 0;

    sema_init(&emptybuf_sem, 2, 0, NULL);
    while (1) {
        sema_wait(&emptybuf_sem);
        buf[i].size = read(0, buf[i].data, BSIZE);
        sema_post(&fullbuf_sem);
        if (buf[i].size <= 0)
            break;
        i ^= 1;
    }
}

writer()
{
    int i = 0;

    sema_init(&fulbuf_sem, 0, 0, NULL);
    while (1) {
        sema_wait(&fullbuf_sem);
        if (buf[i].size <= 0)
            break;

        write(1, buf[i].data, buf[i].size);
        sema_post(&emptybuf_sem);
        i ^= 1;
    }
}

main()
{
    thread_t twriter;

    thr_setconcurrency(2);
    (void)thr_create( NULL, NULL, reader, NULL, THR_DETACHED, NULL;
    (void)thr_create( NULL, NULL, writer, NULL, , &twriter );
    thr_join(twriter, NULL, NULL);
}
```

Code Example A-1 File Copy Example with Semaphore

The example is a bit contrived since the system already generates asynchronous read-ahead and write behind requests when accessing regular files. The example is still useful if the files to be copied are raw devices, since raw device access is synchronous.

Matrix Multiply

Computationally intensive applications benefit from the use of all available processors. Matrix multiplication is a good example of this.

When the matrix multiply is called, it acquires a mutex lock to ensure that only one matrix multiply is in progress. This relies on mutex locks that are statically initialized to zero. The requesting thread then checks whether its worker threads have been created. If not, it creates one for each CPU. Once the worker threads are created, the requesting thread sets up a counter of work to do and signals the workers via a condition variable. Each worker picks off a row and column from the input matrices, then updates the counter of work so that the next worker will get the next item. It then releases the mutex lock so that computing the vector product can proceed in parallel. When the results are ready, the worker reacquires the mutex lock and updates the counter of work completed. The worker that completes the last bit of work signals the requesting thread.

```
struct {
    mutex_t lock;
    condvar_t start_cond, done_cond;
    int (*m1)[SZ][SZ], (*m2)[SZ][SZ], (*m3)[SZ][SZ];
    int row, col;
    int todo, notdone, workers;
} work;
mutex_t mul_lock;

matmul(int (*m1)[SZ][SZ], int (*m2)[SZ][SZ], int (*m3)[SZ][SZ]);
{
    int i;

    mutex_lock(&mul_lock);
    mutex_lock(&work.lock);
    if (work.workers == 0) {
        for (i = 0; i < NCPU; i++) {
            thread_create(NULL, NULL, worker, (void *)NULL,
                          THREAD_NEW_LWP);
        }
```

```
            work.workers = NCPU;
        }

        work.m1=m1; work.m2=m2; work.m3=m3;
        work.row = work.col = 0;
        work.todo = work.notdone = SZ*SZ;
        cv_broadcast(&work.start_cond);
        while (work.notdone)
            cv_wait(&work.done_cond, &work.lock);
        mutex_unlock(&work.lock);
        mutex_unlock(&mul_lock);
}

worker()
{
        int (*m1)[SZ][SZ], (*m2)[SZ][SZ], (*m3)[SZ][SZ];
        int row, col, i, result;

        while (1) {
            mutex_lock(&work.lock);
            while (work.todo == 0)
                cv_wait(&work.start_cond, &work.lock);
            work.todo--;
            m1=work.m1; m2=work.m2; m3=work.m3;
                row = work.row; col = work.col;
            work.col++;
            if (work.col == SZ) {
                work.col = 0;
                work.row++;
                if (work.row == SZ)
                    work.row = 0;
            }
            mutex_unlock(&work.lock);
            result = 0;
            for (i = 0; i < SZ; i++)
                result += (*m1)[row][i] * (*m2)[i][col];
            (*m3)[row][col] = result;
            mutex_lock(&work.lock);
            work.notdone--;
            if (work.notdone == 0)
                cv_signal(&work.done_cond);
            mutex_unlock(&work.lock);
        }
}
```

Code Example A-2 Matrix Multiply

Note that each iteration computed the results of one entry in the result matrix. In some cases this amount of work is not sufficient to justify the overhead of synchronizing. In these cases it is better to give each worker more work per synchronization. For example, each worker could compute an entire row of the output matrix.

Window System Server

A networked window system server tries to handle each client application as independently as possible. Each application should get a fair share of the machine resources, and any blocking on I/O should affect only the connection that caused it. This can be done by allocating a bound thread for each client application. While this would work, it is wasteful since more than a small subset of the clients are rarely active at any one time. Allocating an LWP for each connection ties up large amounts of kernel resources basically for waiting. On a busy desktop, this can be several dozen LWPs. (A window system server that is designed to run with a single-vel threads model would have different considerations about kernel resources and could be designed quite differently.)

The code shown in Code Example A-3 takes a different approach. It allocates two unbound threads for each client connection, one to process display requests and one to write out results. This allows further input to be processed while the results are being sent, yet it maintains strict serialization within the connection. A single control thread looks for requests on the network. The relationship between threads is shown in Figure A-1.

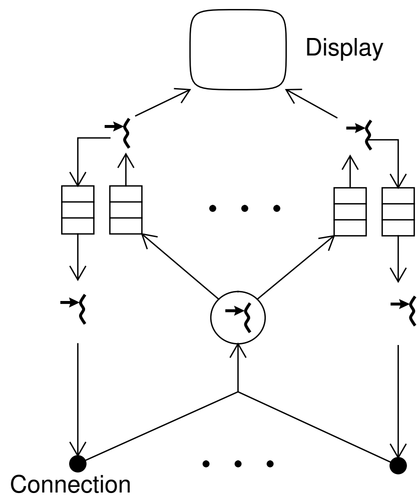

Figure A-1 Window Server Threads

With this arrangement, an LWP is used for the control thread and whatever threads happen to be active concurrently. The threads synchronize via queues. Each queue has its own mutex lock to maintain serialization, and a condition variable to inform waiting threads when something is placed on the queue.

```
main()
{
    /* set up server and listen port */
    for(;;) {
        poll(&fds, nfds, 0);
        for (i = 0; i < nfds; i++) {
            if (fds[i].revents & POLLIN)
                checkfd(fds[i].fd)
        }
    }
}

checkfd(int fd)
{
    struct connection *connp;

    if (fd == listenfd) {
        /* new connection request */
```

```
            connp = create_new_connection();
            thread_create(NULL, NULL, svc_requests, connp, 0);
            thread_create(NULL, NULL, send_replies, connp, 0);
        } else {
            requestp = new_msg();
            requestp->len =
                t_rcv(fd, requestp->data, BUFSZ, &flags);
            connp = find_connection(fd);
            put_q(connp->input_q, requestp);
        }
}

send_replies(struct connection *connp)
{
    struct msg *replyp;

    while (1) {
        replyp = get_q(connp->output_q);
        t_snd(connp->fd, replyp->data, replyp->len, &flags);
    }
}

svc_requests(struct connection *connp)
{
    struct msg *requestp, *replyp;

    while (1) {
        requestp = get_q(connp->input_q);
        replyp = do_request(requestp);
        if (replyp)
            put_q(connp->output_q, replyp);
    }
}

put_q(struct queue *qp, struct msg *msgp)
{
    mutex_lock(&qp->lock);
    if (list_empty(qp->list))
        cv_signal(&qp->notempty_cond);
    add_to_tail(msgp, &qp->list);
    mutex_unlock(&qp->lock);
}

struct msg *
get_q(struct queue *qp)
{
```

```
    struct msg *msgp;

    mutex_lock(&qp->lock);
    while (list_empty(qp->list))
        cv_wait(&qp->notempty_cond, &qp->lock);
    msgp = get_from_head(&qp->list);
    mutex_unlock(&qp->lock);
    return (msgp);
}
```

Code Example A-3 Window Server

Index