Programming With the Operating System

4

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

Processes—Changes for Multithreading	*page 81*
Alarms, Interval Timers, and Profiling	*page 83*
Nonlocal Goto—setjmp(3C) and longjmp(3C)	*page 84*
Resource Limits	*page 84*
LWPs and Scheduling Classes	*page 84*
Extending Traditional Signals	*page 88*
I/O Issues	*page 98*

Processes—Changes for Multithreading

Duplicating Parent Threads

`fork`*(2)*

With the `fork`(2) and `fork1`(2) functions, you can choose between duplicating all parent threads in the child or only one parent thread in the child.

The `fork`() function duplicates the address space and all the threads (and LWPs) in the child. This is useful, for example, when the child process never calls `exec`(2) but does use its copy of the parent address space.

To illustrate, think about a thread in the parent process—other than the one that called `fork`()—that has locked a mutex. This mutex is copied into the child process in its locked state, but no thread is copied over to unlock the mutex. So, any thread in the child that tries to lock the mutex waits forever. To avoid this sort of situation, use `fork`() to duplicate all the threads in the process.

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

`fork1`*(2)*

The `fork1`(2)[1] function duplicates the complete address space in the child but duplicates only the thread that called `fork1`(). This is useful when the child process immediately calls `exec`(), which is what happens after most calls to `fork`(). In this case, the child process does not need a duplicate of any thread other than the one that called `fork`(2).

Do not call any library functions after calling `fork1`() and before calling `exec`()—one of the library functions might use a lock that is held by more than one thread.

Cautions for Both `fork`*(2) and* `fork1`*(2)*

For both `fork`() and `fork1`(), be careful when you use global state after a call to either.

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

1. Terminology will probably change when POSIX 1003.4a is adopted. What is currently called `fork`(2) will be called `forkall`(), and what is called `fork1`(2) will be called `fork`(). Also added in POSIX is the idea of the "fork cleanup handler"— you can call `pthread_atfork`() to register three functions to be called, respectively, just before the `fork`() takes place, and just after the `fork`() in both the parent and the child processes. These routines are to clean up locks and so on, although this is necessary only with the version of `fork`() that creates only one thread in the child process.

Also for both `fork`() and `fork1`(), be careful not to create locks that are held by both the parent and child processes. This can happen when locks are allocated in memory that is sharable (that is `mmap`(2)'ed with the `MAP_SHARED` flag).

`vfork`*(2)*

`vfork`(2) is like `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.

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

For example, it is dangerous to create new threads between the call to `vfork`() and the call to `exec`().

Executing Files and Terminating Processes

`exec`*(2) and* `exit`*(2)*

Both the `exec`(2) and `exit`(2) 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 (and so all active threads) are destroyed.

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

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

Alarms, Interval Timers, and Profiling

Each LWP has a unique realtime interval timer and alarm that a thread bound to the LWP can use. The timer or alarm delivers one signal to the thread when the timer or alarm expires.

Each LWP also has a virtual time or profile interval timer that a thread bound to the LWP can use. When the interval timer expires, either `SIGVTALRM` or `SIGPROF`, as appropriate, is sent to the LWP that owns the interval timer.

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

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

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

Resource Limits

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

LWPs and Scheduling Classes

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

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

Each scheduling class maps the priority of the LWP it is scheduling to an overall dispatching priority according to the configurable priority of the scheduling class.

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

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

- `PC_GETCID`—Get the class ID and class attributes for a specific class.
- `PC_GETCLINFO`—Get the class name and class attributes for a specific class.
- `PC_GETPARMS`—Get the class identifier and the class-specific scheduling parameters of a process, an LWP with a process, or a group of processes.
- `PC_SETPARMS`—Set the class identifier and the class-specific scheduling parameters of a process, an LWP with a process, or a group of processes.

Use `priocntl`() only on bound threads. To affect the priority of unbound threads, use `thr_setprio`(3T).

Timeshare Scheduling

Timeshare scheduling fairly distributes the processing resource to the set of processes. Other parts of the kernel can monopolize the processor for short intervals without degrading response time as seen by the user.

The `priocntl`(2) call sets the `nice`(2) level of one or more processes. `priocntl`() affects the `nice`() level of all the timesharing class LWPs in the process. The `nice`() level ranges from 0 to +20 normally and from -20 to +20 for processes with superuser privilege. The lower the value, the higher the priority.

The dispatch priority of time-shared LWPs is calculated from the instantaneous CPU use rate of the LWP and from its `nice`() level. The `nice`() level indicates the relative priority of the 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.

Realtime Scheduling

The realtime class (`RT`) can be applied to a whole process or to one or more LWPs in a process. This requires superuser privilege. Unlike the `nice`(2) level of the timeshare class, LWPs that are classified realtime can be assigned priorities either individually or jointly. A `priocntl`(2) call affects the attributes of all the realtime LWPs in the process.

The scheduler always dispatches the highest-priority realtime LWP. It preempts a lower-priority LWP when a higher-priority LWP becomes runable. A preempted LWP is placed at the head of its level queue. A realtime LWP retains control of a processor until it is preempted, it suspends, or its realtime priority is changed. LWPs in the `RT` class have absolute priority over processes in the `TS` class.

A new LWP inherits the scheduling class of the parent process or LWP. An `RT` class LWP inherits the parent's time slice, whether finite or infinite. An LWP with a finite time slice runs until it terminates, blocks (for example, to wait for an I/O event), is preempted by a higher-priority runable realtime process, or the time slice expires. An LWP with an infinite time slice ceases execution only when it terminates, blocks, or is preempted.

LWP Scheduling and Thread Binding

The threads library automatically adjusts the number of LWPs in the pool used to run unbound threads. Its objectives are:

- To prevent the program from being blocked by a lack of unblocked LWPs

  For example, if there are more runable 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.

Keep in mind that LWPs are time-sliced, not threads. This means that when there is only one LWP there is no time slicing within the process—threads run on the LWP until they block (through interthread synchronization), are preempted, or terminate.

You can assign priorities to threads with `thr_setprio`(3T); lower-priority unbound threads are assigned to LWPs only when no higher-priority unbound threads are available. Bound threads, of course, do not compete for LWPs because they have their own.

Bind threads to your LWPs to get precise control over whatever is being scheduled. This control is not possible when many unbound threads compete for an LWP.

Realtime threads are useful for getting a quick response to external stimuli. Consider a thread used for mouse tracking that must respond instantly to mouse clicks. By binding the thread to an LWP, you guarantee that there is an LWP available when it is needed. By assigning the LWP to the realtime scheduling class, you ensure that the LWP is scheduled quickly in response to mouse clicks.

SIGWAITING—Creating LWPs for Waiting Threads

The library usually ensures that there are enough LWPs in its pool for a program to proceed. When all the LWPs in the process are blocked in indefinite waits (such as blocked reading from a tty or network), the operating system sends the new signal, `SIGWAITING`, to the process. This signal is handled by the threads library. When the process contains a thread that is waiting to run, a new LWP is created and the appropriate waiting thread is assigned to it for execution.

The `SIGWAITING` mechanism does not ensure that an additional LWP is created when one or more threads are compute bound and another thread becomes runable. A compute-bound thread can prevent multiple runable threads from being started because of a shortage of LWPs. This can be prevented by calling `thr_setconcurrency`(3T) or by using `THR_NEW_LWP` in calls to `thr_create`(3T).

Aging LWPs

When the number of active threads is reduced, some of the LWPs in the pool are no longer needed. When there are more LWPs than active threads, the threads library destroys the unneeded ones. The library ages LWPs—they are deleted when they are unused for a “long” time, currently set at five minutes.

4

Extending Traditional Signals

The traditional UNIX signal model is extended to threads in a fairly natural way. The disposition of signals is established process-wide, using the traditional mechanisms (`signal`(2), `sigaction`(2), and so on).

When a signal handler is marked `SIG_DFL` or `SIG_IGN`, the action on receipt of the signal (exit, core dump, stop, continue, or ignore) is performed on the entire receiving process, affecting all threads in the process. See `signal`(5) for basic information about signals.

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

A thread in one process cannot send a signal to a specific thread in another process. A signal sent by `kill`(2) or `sigsend`(2) is to a process and is handled by any one of the receptive threads in the process.

Unbound threads cannot use alternate signal stacks. A bound thread can use an alternate stack because the state is associated with the execution resource. An alternate stack must be enabled for the signal through `sigaction`(2), and declared and enabled through `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 zero.

Signals are divided into two categories: traps and exceptions (synchronous signals) and interrupts (asynchronous signals).

As in traditional UNIX, if a signal is pending, additional occurrences of that signal have no additional effect—a pending signal is represented by a bit, not a counter.

As is the case with single-threaded processes, when a thread receives a signal while blocked in a system call, the thread might return early, either with the `EINTR` error code, or, in the case of I/O calls, with fewer bytes transferred than requested.

Of particular importance to multithreaded programs is the effect of signals on `cond_wait`(3T). This call usually returns in response to a `cond_signal`(3T) or a `cond_broadcast`(3T), but, if the waiting thread receives a UNIX signal, it returns with the error code `EINTR`. See "Interrupted Waits on Condition Variables" on page 96 for more information.

Synchronous Signals

Traps (such as `SIGILL`, `SIGFPE`, `SIGSEGV`) result from something a thread does to itself, such as dividing by zero or explicitly sending itself a signal. A trap is handled only by the thread that caused it. Several threads in a process can generate and handle the same type of trap simultaneously.

Extending the idea of signals to individual threads is easy for synchronous signals—the signal is dealt with by the thread that caused the problem. However, if the thread has not chosen to deal with the problem, such as by establishing a signal handler with `sigaction`(2), the entire process is terminated.

Because such a synchronous signal usually means that something is seriously wrong with the whole process, and not just with a thread, terminating the process is often a good choice.

Asynchronous Signals

Interrupts (such as `SIGINT` and `SIGIO`) are asynchronous with any thread and result from some action outside the process. They might be signals sent explicitly by other threads, or they might represent external actions such as a user typing Control-c. Dealing with asynchronous signals is more complicated than dealing with synchronous signals.

An interrupt can be handled by any thread whose signal mask allows it. When more than one thread is able to receive the interrupt, only one is chosen.

When multiple occurrences of the same signal are sent to a process, then each occurrence can be handled by a separate thread, as long as threads are available that do not have it masked. When all threads have the signal masked, then the signal is marked *pending* and the first thread to unmask the signal handles it.

4

Continuation Semantics

Continuation semantics are the traditional way to deal with signals. The idea is that when a signal handler returns, control resumes where it was at the time of the interruption. This is well suited for asynchronous signals in single-threaded processes, as shown in Code Example 4-1. This is also used as the exception-handling mechanism in some programming languages, such as PL/1.

Code Example 4-1 Continuation Semantics

```
unsigned int nestcount;

unsigned int A(int i, int j) {
    nestcount++;

    if (i==0)
        return(j+1)
    else if (j==0)
        return(A(i-1, 1));
    else
        return(A(i-1, A(i, j-1)));
}

void sig(int i) {
    printf("nestcount = %d\n", nestcount);
}

main() {
    sigset(SIGINT, sig);
    A(4,4);
}
```

New Operations on Signals

Several new signal operations for multithreaded programming have been added to the operating system.

`thr_sigsetmask`*(3T)*

`thr_sigsetmask`(3T) does for a thread what `sigprocmask`(2) does for a process—it sets the (thread's) signal mask. When a new thread is created, its initial mask is inherited from its creator.

Avoid using `sigprocmask`() in multithreaded programs because it sets the signal mask of the underlying LWP, and the thread that is affected by this can change over time.

Unlike `sigprocmask`(), `thr_sigsetmask`() is relatively inexpensive to call because it does not generally cause a system call, as does `sigprocmask`().

`thr_kill`*(3T)*

`thr_kill`(3T) is the thread analog of `kill`(2)—it sends a signal to a specific thread.

This, of course, is different from sending a signal to a process. When a signal is sent to a process, the signal can be handled by any thread in the process. A signal sent by `thr_kill`() can be handled only by the specified thread.

Note than you can use `thr_kill`() to send signals only to threads in the current process. This is because the thread identifier (type `thread_t`) is local in scope—it is not possible to name a thread in any process but your own.

`sigwait`*(2)*

`sigwait`(2) causes the calling thread to wait until any signal identified by its *set* argument is delivered to the thread. While the thread is waiting, signals identified by the *set* argument are unmasked, but the original mask is restored when the call returns.

Use `sigwait`() to separate threads from asynchronous signals. You can create one thread that is listening for asynchronous signals while your other threads are created to block any asynchronous signals that might be set to this process.

When the signal is delivered, `sigwait`() clears the pending signal and returns its number. Many threads can call `sigwait`() at the same time, but only one thread returns for each signal that is received.

With `sigwait`() you can treat asynchronous signals synchronously—a thread that deals with such signals simply calls `sigwait`() and returns as soon as a signal arrives. By ensuring that all threads (including the caller of `sigwait`()) have such signals masked, you can be sure that signals are handled only by the intended handler and that they are handled safely.

Usually, you use `sigwait`() to create one or more threads that wait for signals. Because `sigwait`() can retrieve even masked signals, be sure to block the signals of interest in all other threads so they are not accidentally delivered. When the signals arrive, a thread returns from `sigwait`(), handles the signal, and waits for more signals. The signal-handling thread is not restricted to using Async-Safe functions and can synchronize with other threads in the usual way. (The Async-Safe category is defined in "MT Interface Safety Levels" on page 105.)

Note – `sigwait`() should *never* be used with synchronous signals.

`sigtimedwait`(2)

`sigtimedwait`(2) is similar to `sigwait`(2) except that it fails and returns an error when a signal is not received in the indicated amount of time.

Thread-Directed Signals

The UNIX signal mechanism is extended with the idea of *thread-directed signals*. These are just like ordinary asynchronous signals, except that they are sent to a particular thread instead of to a process.

Waiting for asynchronous signals in a separate thread can be safer and easier than installing a signal handler and processing the signals there.

A better way to deal with asynchronous signals is to treat them synchronously. By calling `sigwait`(2), discussed on page 91, a thread can wait until a signal occurs.

Code Example 4-2 Asynchronous Signals and `sigwait`(2)

```
main() {
    sigset_t set;
    void runA(void);

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    thr_sigsetmask(SIG_BLOCK, &set, NULL);
    thr_create(NULL, 0, runA, NULL, THR_DETACHED, NULL);

    while (1) {
        sigwait(&set);
        printf("nestcount = %d\n", nestcount);
    }
}

void runA() {
    A(4,4);
    exit(0);
}
```

This example modifies the code of Code Example 4-1: the main routine masks the `SIGINT` signal, creates a child thread that calls the function *A* of the previous example, and finally issues `sigwait`s to handle the `SIGINT` signal.

Note that the signal is masked in the compute thread because the compute thread inherits its signal mask from the main thread. The main thread is protected from `SIGINT` while, and only while, it is not blocked inside of `sigwait`().

Also, note that there is never any danger of having system calls interrupted when you use `sigwait`().

Completion Semantics

Another way to deal with signals is with *completion semantics*. Use completion semantics when a signal indicates that something so catastrophic has happened that there is no reason to continue executing the current code block. The signal handler runs *instead of* the remainder of the block that had the problem. In other words, the signal handler *completes* the block.

In Code Example 4-3, the block in question is the body of the `then` part of the `if` statement. The call to `setjmp`(3C) saves the current register state of the program in `jbuf` and returns 0—thereby executing the block.

Code Example 4-3 Completion Semantics

```
sigjmp_buf jbuf;
void mult_divide(void) {
    int a, b, c, d;
    void problem();

    sigset(SIGFPE, problem);
    while (1) {
        if (sigsetjmp(&jbuf) == 0) {
            printf("Three numbers, please:\n");
            scanf("%d %d %d", &a, &b, &c);
            d = a*b/c;
            printf("%d*%d/%d = %d\n", a, b, c, d);
        }
    }
}

void problem(int sig) {
    printf("Couldn't deal with them, try again\n");
    siglongjmp(&jbuf, 1);
}
```

If a `SIGFPE` (a floating-point exception) occurs, the signal handler is invoked.

The signal handler calls `siglongjmp`(3C), which restores the register state saved in `jbuf`, causing the program to return from `sigsetjmp`() again (among the registers saved are the program counter and the stack pointer).

This time, however, `sigsetjmp`(3C) returns the second argument of `siglongjmp`(), which is 1. Notice that the block is skipped over, only to be executed during the next iteration of the `while` loop.

Note that you can use `sigsetjmp`(3C) and `siglongjmp`(3C) in multithreaded programs, but be careful that a thread never does a `siglongjmp`() using the results of another thread's `sigsetjmp()`. Also, `sigsetjmp`() and `siglongjmp`() save and restore the signal mask, but `setjmp`(3C) and `longjmp`(3C) do not. It is best to use `sigsetjmp`() and `siglongjmp`() when you work with signal handlers.

Completion semantics are often used to deal with exceptions. In particular, the Ada® programming language uses this model.

Note – Remember, `sigwait`(2) should *never* be used with synchronous signals.

Signal Handlers and Async Safety

A concept similar to thread safety is *async safety*. Async-Safe operations are guaranteed not to interfere with operations being interrupted.

The problem of async safety arises when the actions of a signal handler can interfere with the operation being interrupted. For example, suppose a program is in the middle of a call to `printf`(3S) and a signal occurs whose handler itself calls `printf`(): the output of the two `printf`() statements would be intertwined. To avoid this, the handler should not call `printf`() itself when `printf`() might be interrupted by a signal.

This problem cannot be solved by using synchronization primitives because any attempted synchronization between the signal handler and the operation being synchronized would produce immediate deadlock.

For example, suppose that `printf`() is to protect itself by using a mutex. Now suppose that a thread that is in a call to `printf`(), and so holds the lock on the mutex, is interrupted by a signal. If the handler (being called by the thread that is still inside of `printf`()) itself calls `printf`(), the thread that holds the lock on the mutex will attempt to take it again, resulting in an instant deadlock.

To avoid interference between the handler and the operation, either ensure that the situation never arises (perhaps by masking off signals at critical moments) or invoke only Async-Safe operations from inside signal handlers.

Because setting a thread's mask is an inexpensive user-level operation, you can inexpensively make functions or sections of code fit in the Async-Safe category.

Interrupted Waits on Condition Variables

When a signal is delivered to a thread while the thread is waiting on a condition variable, the old convention (assuming that the process is not terminated) is that interrupted calls return `EINTR`.

The ideal new condition would be that when `cond_wait`(3T) and `cond_timedwait`(3T) return, the lock has been retaken on the mutex.

This is what is done in Solaris threads: when a thread is blocked in `cond_wait`() or `cond_timedwait`() and an unmasked, caught signal is delivered to the thread, the handler is invoked and the call to `cond_wait`() or `cond_timedwait`() returns `EINTR` with the mutex locked.

This implies that the mutex is locked in the signal handler because the handler might have to clean up after the thread.

This is illustrated by Code Example 4-4.

Code Example 4-4 Condition Variables and Interrupted Waits

```
int sig_catcher() {
    sigset_t set;
    void hdlr();

    mutex_lock(&mut);

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    thr_sigsetmask(SIG_UNBLOCK, &set, 0);

    if (cond_wait(&cond, &mut) == EINTR) {
        /* signal occurred and lock is held */
        cleanup();
        mutex_unlock(&mut);
        return(0);
    }
    normal_processing();
    mutex_unlock(&mut);
    return(1);
}

void hdlr() {
    /* lock is held in the handler */
    ...
}
```

Assume that the `SIGINT` signal is blocked in all threads on entry to `sig_catcher`() and that `hdlr`() has been established (with a call to `sigaction`(2)) as the handler for the `SIGINT` signal.

When an unmasked and caught instance of the `SIGINT` signal is delivered to the thread while it is in `cond_wait`(), the thread first reacquires the lock on the mutex, then calls `hdlr`(), and then returns `EINTR` from `cond_wait`().

Note that whether `SA_RESTART` has been specified as a flag to `sigaction`() has no effect here—`cond_wait`(3T) is not a system call and is not automatically restarted. When a caught signal occurs while a thread is blocked in `cond_wait`(), the call always returns `EINTR`.

4

I/O Issues

One of the attractions of multithreaded programming is I/O performance. The traditional UNIX API gave the programmer little assistance in this area—you either used the facilities of the file system or bypassed the file system entirely.

This section shows how to use threads to get more flexibility through I/O concurrency and multibuffering. This section also discusses the differences and similarities between the approaches of synchronous I/O (with threads) and asynchronous I/O (with and without threads).

I/O as a Remote Procedure Call

In the traditional UNIX model, I/O appears to be *synchronous*, as if you were placing a remote procedure call to the I/O device. Once the call returns, then the I/O has completed (or at least it appears to have completed—a write request, for example, might merely result in the transfer of the data to a buffer in the operating system).

The advantage of this model is that it is easy to understand because programmers are very familiar with the concept of procedure calls.

An alternative approach not found in traditional UNIX systems is the *asynchronous* model, in which an I/O request merely starts an operation. The program must somehow discover when the operation completes.

This approach is not as simple as the synchronous model, but it has the advantage of allowing concurrent I/O and processing in traditional, single-threaded UNIX processes.

Tamed Asynchrony

You can get most of the benefits of asynchronous I/O by using synchronous I/O in a multithreaded program. Where, with asynchronous I/O, you would issue a request and check later to determine when it completes, you can instead have a separate thread perform the I/O synchronously. The main thread can then check (perhaps by calling `thr_join`(3T)) for the completion of the operation at some later time.

Asynchronous I/O

In most situations there is no need for asynchronous I/O, since its effects can be achieved with the use of threads, each doing synchronous I/O. However, in a few situations, threads cannot achieve what asynchronous I/O can.

The most straightforward example is writing to a tape drive to make the tape drive *stream*. This technique prevents the tape drive from stopping while it is being written to and moves the tape forward at high speed while supplying a constant stream of data that it writes to tape.

To do this, the tape driver in the kernel must issue a queued write request when the tape driver responds to an interrupt indicating that the previous tape-write operation has completed.

Threads cannot guarantee that asynchronous writes will be ordered because the order in which threads execute is indeterminate. Trying to order a write to a tape, for example, is not possible.

Asynchronous I/O Operations

```
#include <sys/asynch.h>

int aioread(int fildes, char *bufp, int bufs, off_t offset,
    int whence, aio_result_t *resultp);

int aiowrite(int filedes, const char *bufp, int bufs,
    off_t offset, int whence, aio_result_t *resultp);

aio_result_t *aiowait(const struct timeval *timeout);

int aiocancel(aio_result_t *resultp);
```

`aioread`(3) and `aiowrite`(3) are similar in form to `pread`(2) and `pwrite`(2), except for the addition of the last argument. Calls to `aioread`() and `aiowrite`() result in the initiation (or queueing) of an I/O operation.

The call returns without blocking, and the status of the call is returned in the structure pointed to by `resultp`. This is an item of type `aio_result_t` that contains

```
int aio_return;
int aio_errno;
```

When a call fails immediately, the failure code can be found in `aio_errno`. Otherwise, this field contains `AIO_INPROGRESS`, meaning that the operation has been successfully queued.

You can wait for an outstanding asynchronous I/O operation to complete by calling `aiowait`(3). This returns a pointer to the `aio_result_t` structure supplied with the original `aioread`(3) or `aiowrite`(3) call. This time `aio_result` contains whatever `read`(2) or `write`(2) would have returned if it had been called instead of the asynchronous versions, and `aio_errno` contains the error code, if any.

`aiowait`() takes a `timeout` argument, which indicates how long the caller is willing to wait. As usual, a NULL pointer here means that the caller is willing to wait indefinitely, and a pointer to a structure containing a zero value means that the caller is unwilling to wait at all.

You might start an asynchronous I/O operation, do some work, then call `aiowait`() to wait for the request to complete. Or you can use `SIGIO` to be notified, asynchronously, when the operation completes.

Finally, a pending asynchronous I/O operation can be cancelled by calling `aiocancel`(). This routine is called with the address of the result area as an argument. This result area identifies which operation is being cancelled.

Shared I/O and New I/O System Calls

When multiple threads are performing I/O operations at the same time with the same file descriptor, you might discover that the traditional UNIX I/O interface is not thread-safe. The problem occurs with nonsequential I/O. This uses the `lseek`(2) system call to set the file offset, which is then used in the next `read`(2) or `write`(2) call to indicate where in the file the operation should start. When two or more threads are issuing `lseek`(2)'s to the same file descriptor, a conflict results.

To avoid this conflict, use the new `pread`(2) and `pwrite`(2) system calls.

```
#include <sys/types.h>
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

ssize_t pwrite(int filedes, void *buf, size_t nbyte,
    off_t offset);
```

These behave just like `read`(2) and `write`(2) except that they take an additional argument, the *file offset*. With this argument, you specify the offset without using `lseek`(2), so multiple threads can use these routines safely for I/O on the same file descriptor.

Alternatives to `getc`*(3S) and* `putc`*(3S)*

An additional problem occurs with standard I/O. Programmers are accustomed to routines such as `getc`(3S) and `putc`(3S) being very quick—they are implemented as macros. Because of this, they can be used within the inner loop of a program with no concerns about efficiency.

However, when they are made thread safe they suddenly become more expensive—they now require (at least) two internal subroutine calls, to lock and unlock a mutex. To get around this problem, alternative versions of these routines are supplied—`getc_unlocked`(3S) and `putc_unlocked`(3S).

These do not acquire locks on a mutex and so are as quick as the originals, nonthread-safe versions of `getc`(3S) and `putc`(3S). However, to use them in a thread-safe way, you must explicitly lock and release the mutexes that protect the standard I/O streams, using `flockfile`(3S) and `funlockfile`(3S). The calls to these latter routines are placed outside the loop, and the calls to `getc_unlocked`() or `putc_unlocked`() are placed inside the loop.

4