```
                return(NULL);
            }
            idx = HASH(fp);
            pthread_mutex_lock(&hashlock);
            fp->f_next = fh[idx];
            fh[idx] = fp->f_next;
            pthread_mutex_lock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            /* ... continue initialization ... */
        }
        return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
        pthread_mutex_lock(&hashlock);
        fp->f_count++;
        pthread_mutex_unlock(&hashlock);
}

struct foo *
foo_find(int id) /* find a existing object */
{
        struct foo  *fp;
        int         idx;

        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        for (fp = fh[idx]; fp != NULL; fp = fp->f_next) {
            if (fp->f_id == id) {
                fp->f_count++;
                break;
            }
        }
        pthread_mutex_unlock(&hashlock);
        return(fp);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
        struct foo  *tfp;
        int         idx;

        pthread_mutex_lock(&hashlock);
        if (--fp->f_count == 0) { /* last reference, remove from list */
            idx = HASH(fp);
            tfp = fh[idx];
            if (tfp == fp) {
                fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&hashlock);
    }
}
```

Figure 11.12 Simplified locking

Note how much simpler the program in Figure 11.12 is compared to the program in Figure 11.11. The lock-ordering issues surrounding the hash list and the reference count go away when we use the same lock for both purposes. Multithreaded software design involves these types of tradeoffs. If your locking granularity is too coarse, you end up with too many threads blocking behind the same locks, with little improvement possible from concurrency. If your locking granularity is too fine, then you suffer bad performance from excess locking overhead, and you end up with complex code. As a programmer, you need to find the correct balance between code complexity and performance, and still satisfy your locking requirements. □

Reader–Writer Locks

Reader–writer locks are similar to mutexes, except that they allow for higher degrees of parallelism. With a mutex, the state is either locked or unlocked, and only one thread can lock it at a time. Three states are possible with a reader–writer lock: locked in read mode, locked in write mode, and unlocked. Only one thread at a time can hold a reader–writer lock in write mode, but multiple threads can hold a reader–writer lock in read mode at the same time.

When a reader–writer lock is write-locked, all threads attempting to lock it block until it is unlocked. When a reader–writer lock is read-locked, all threads attempting to lock it in read mode are given access, but any threads attempting to lock it in write mode block until all the threads have relinquished their read locks. Although implementations vary, reader–writer locks usually block additional readers if a lock is already held in read mode and a thread is blocked trying to acquire the lock in write mode. This prevents a constant stream of readers from starving waiting writers.

Reader–writer locks are well suited for situations in which data structures are read more often than they are modified. When a reader–writer lock is held in write mode, the data structure it protects can be modified safely, since only one thread at a time can hold the lock in write mode. When the reader–writer lock is held in read mode, the data structure it protects can be read by multiple threads, as long as the threads first acquire the lock in read mode.

Reader–writer locks are also called shared–exclusive locks. When a reader–writer lock is read-locked, it is said to be locked in shared mode. When it is write-locked, it is said to be locked in exclusive mode.

As with mutexes, reader–writer locks must be initialized before use and destroyed before freeing their underlying memory.

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

                        Both return: 0 if OK, error number on failure
```

A reader–writer lock is initialized by calling `pthread_rwlock_init`. We can pass a null pointer for *attr* if we want the reader–writer lock to have the default attributes. We discuss reader–writer lock attributes in Section 12.4.

Before freeing the memory backing a reader–writer lock, we need to call `pthread_rwlock_destroy` to clean it up. If `pthread_rwlock_init` allocated any resources for the reader–writer lock, `pthread_rwlock_destroy` frees those resources. If we free the memory backing a reader–writer lock without first calling `pthread_rwlock_destroy`, any resources assigned to the lock will be lost.

To lock a reader–writer lock in read mode, we call `pthread_rwlock_rdlock`. To write-lock a reader–writer lock, we call `pthread_rwlock_wrlock`. Regardless of how we lock a reader–writer lock, we can call `pthread_rwlock_unlock` to unlock it.

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

                        All return: 0 if OK, error number on failure
```

Implementations might place a limit on the number of times a reader–writer lock can be locked in shared mode, so we need to check the return value of `pthread_rwlock_rdlock`. Even though `pthread_rwlock_wrlock` and `pthread_rwlock_unlock` have error returns, we don't need to check them if we design our locking properly. The only error returns defined are when we use them improperly, such as with an uninitialized lock, or when we might deadlock by attempting to acquire a lock we already own.

The Single UNIX Specification also defines conditional versions of the reader–writer locking primitives.

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

                        Both return: 0 if OK, error number on failure
```

When the lock can be acquired, these functions return 0. Otherwise, they return the error EBUSY. These functions can be used in situations in which conforming to a lock hierarchy isn't enough to avoid a deadlock, as we discussed previously.

Example

The program in Figure 11.13 illustrates the use of reader–writer locks. A queue of job requests is protected by a single reader–writer lock. This example shows a possible implementation of Figure 11.1, whereby multiple worker threads obtain jobs assigned to them by a single master thread.

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

struct job {
    struct job *j_next;
    struct job *j_prev;
    pthread_t   j_id;   /* tells which thread handles this job */
    /* ... more stuff here ... */
};

struct queue {
    struct job      *q_head;
    struct job      *q_tail;
    pthread_rwlock_t q_lock;
};

/*
 * Initialize a queue.
 */
int
queue_init(struct queue *qp)
{
    int err;

    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (err != 0)
        return(err);

    /* ... continue initialization ... */

    return(0);
}

/*
 * Insert a job at the head of the queue.
 */
void
```

```
job_insert(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
    if (qp->q_head != NULL)
        qp->q_head->j_prev = jp;
    else
        qp->q_tail = jp;    /* list was empty */
    qp->q_head = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Append a job on the tail of the queue.
 */
void
job_append(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
    jp->j_prev = qp->q_tail;
    if (qp->q_tail != NULL)
        qp->q_tail->j_next = jp;
    else
        qp->q_head = jp;    /* list was empty */
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Remove the given job from a queue.
 */
void
job_remove(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
        if (qp->q_tail == jp)
            qp->q_tail = NULL;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        if (qp->q_head == jp)
            qp->q_head = NULL;
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    pthread_rwlock_unlock(&qp->q_lock);
}
```

```
/*
 * Find a job for the given thread ID.
 */
struct job *
job_find(struct queue *qp, pthread_t id)
{
    struct job *jp;

    if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
        return(NULL);

    for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
        if (pthread_equal(jp->j_id, id))
            break;

    pthread_rwlock_unlock(&qp->q_lock);
    return(jp);
}
```

Figure 11.13 Using reader–writer locks

In this example, we lock the queue's reader–writer lock in write mode whenever we need to add a job to the queue or remove a job from the queue. Whenever we search the queue, we grab the lock in read mode, allowing all the worker threads to search the queue concurrently. Using a reader–writer lock will improve performance in this case only if threads search the queue much more frequently than they add or remove jobs.

The worker threads take only those jobs that match their thread ID off the queue. Since the job structures are used only by one thread at a time, they don't need any extra locking. □

Condition Variables

Condition variables are another synchronization mechanism available to threads. Condition variables provide a place for threads to rendezvous. When used with mutexes, condition variables allow threads to wait in a race-free way for arbitrary conditions to occur.

The condition itself is protected by a mutex. A thread must first lock the mutex to change the condition state. Other threads will not notice the change until they acquire the mutex, because the mutex must be locked to be able to evaluate the condition.

Before a condition variable is used, it must first be initialized. A condition variable, represented by the `pthread_cond_t` data type, can be initialized in two ways. We can assign the constant `PTHREAD_COND_INITIALIZER` to a statically-allocated condition variable, but if the condition variable is allocated dynamically, we can use the `pthread_cond_init` function to initialize it.

We can use the `pthread_mutex_destroy` function to deinitialize a condition variable before freeing its underlying memory.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
                      pthread_condattr_t *restrict attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Both return: 0 if OK, error number on failure

Unless you need to create a conditional variable with nondefault attributes, the *attr* argument to `pthread_cond_init` can be set to NULL. We will discuss condition variable attributes in Section 12.4.

We use `pthread_cond_wait` to wait for a condition to be true. A variant is provided to return an error code if the condition hasn't been satisfied in the specified amount of time.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict timeout);
```

Both return: 0 if OK, error number on failure

The mutex passed to `pthread_cond_wait` protects the condition. The caller passes it locked to the function, which then atomically places the calling thread on the list of threads waiting for the condition and unlocks the mutex. This closes the window between the time that the condition is checked and the time that the thread goes to sleep waiting for the condition to change, so that the thread doesn't miss a change in the condition. When `pthread_cond_wait` returns, the mutex is again locked.

The `pthread_cond_timedwait` function works the same as the `pthread_cond_wait` function with the addition of the timeout. The timeout value specifies how long we will wait. It is specified by the `timespec` structure, where a time value is represented by a number of seconds and partial seconds. Partial seconds are specified in units of nanoseconds:

```
struct timespec {
        time_t tv_sec;     /* seconds */
        long   tv_nsec;    /* nanoseconds */
};
```

Using this structure, we need to specify how long we are willing to wait as an absolute time instead of a relative time. For example, if we are willing to wait 3 minutes, instead of translating 3 minutes into a `timespec` structure, we need to translate now+3 minutes into a `timespec` structure.

We can use `gettimeofday` (Section 6.10) to get the current time expressed as a `timeval` structure and translate this into a `timespec` structure. To obtain the absolute time for the timeout value, we can use the following function:

```
void
maketimeout(struct timespec *tsp, long minutes)
{
    struct timeval now;

    /* get the current time */
    gettimeofday(&now);
    tsp->tv_sec = now.tv_sec;
    tsp->tv_nsec = now.tv_usec * 1000; /* usec to nsec */
    /* add the offset to get timeout value */
    tsp->tv_sec += minutes * 60;
}
```

If the timeout expires without the condition occurring, pthread_cond_timedwait will reacquire the mutex and return the error ETIMEDOUT. When it returns from a successful call to pthread_cond_wait or pthread_cond_timedwait, a thread needs to reevaluate the condition, since another thread might have run and already changed the condition.

There are two functions to notify threads that a condition has been satisfied. The pthread_cond_signal function will wake up one thread waiting on a condition, whereas the pthread_cond_broadcast function will wake up all threads waiting on a condition.

The POSIX specification allows for implementations of pthread_cond_signal to wake up more than one thread, to make the implementation simpler.

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

                    Both return: 0 if OK, error number on failure
```

When we call pthread_cond_signal or pthread_cond_broadcast, we are said to be *signaling* the thread or condition. We have to be careful to signal the threads only after changing the state of the condition.

Example

Figure 11.14 shows an example of how to use condition variables and mutexes together to synchronize threads.

```
#include <pthread.h>

struct msg {
    struct msg *m_next;
    /* ... more stuff here ... */
};
```

```
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* now process the message mp */
    }
}

void
enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

Figure 11.14 Using condition variables

The condition is the state of the work queue. We protect the condition with a mutex and evaluate the condition in a `while` loop. When we put a message on the work queue, we need to hold the mutex, but we don't need to hold the mutex when we signal the waiting threads. As long as it is okay for a thread to pull the message off the queue before we call `cond_signal`, we can do this after releasing the mutex. Since we check the condition in a `while` loop, this doesn't present a problem: a thread will wake up, find that the queue is still empty, and go back to waiting again. If the code couldn't tolerate this race, we would need to hold the mutex when we signal the threads. □

11.7 Summary

In this chapter, we introduced the concept of threads and discussed the POSIX.1 primitives available to create and destroy them. We also introduced the problem of thread synchronization. We discussed three fundamental synchronization mechanisms—mutexes, reader–writer locks, and condition variables—and we saw how to use them to protect shared resources.

Exercises

11.1 Modify the example shown in Figure 11.4 to pass the structure between the threads properly.

11.2 In the example shown in Figure 11.13, what additional synchronization (if any) is necessary to allow the master thread to change the thread ID associated with a pending job? How would this affect the `job_remove` function?

11.3 Apply the techniques shown in Figure 11.14 to the worker thread example (Figure 11.1 and Figure 11.13) to implement the worker thread function. Don't forget to update the `queue_init` function to initialize the condition variable and change the the `job_insert` and `job_append` functions to signal the worker threads. What difficulties arise?

11.4 Which sequence of steps is correct?

1. Lock a mutex (`pthread_mutex_lock`).
2. Change the condition protected by the mutex.
3. Signal threads waiting on the condition (`pthread_cond_broadcast`).
4. Unlock the mutex (`pthread_mutex_unlock`).

or

1. Lock a mutex (`pthread_mutex_lock`).
2. Change the condition protected by the mutex.
3. Unlock the mutex (`pthread_mutex_unlock`).
4. Signal threads waiting on the condition (`pthread_cond_broadcast`).

12

Thread Control

12.1 Introduction

In Chapter 11, we learned the basics about threads and thread synchronization. In this chapter, we will learn the details of controlling thread behavior. We will look at thread attributes and synchronization primitive attributes, which we ignored in the previous chapter in favor of the default behaviors.

We will follow this with a look at how threads can keep data private from other threads in the same process. Then we will wrap up the chapter with a look at how some process-based system calls interact with threads.

12.2 Thread Limits

We discussed the `sysconf` function in Section 2.5.4. The Single UNIX Specification defines several limits associated with the operation of threads, which we didn't show in Figure 2.10. As with other system limits, the thread limits can be queried using `sysconf`. Figure 12.1 summarizes these limits.

As with the other limits reported by `sysconf`, use of these limits is intended to promote application portability among different operating system implementations. For example, if your application requires that you create four threads for every file you manage, you might have to limit the number of files you can manage concurrently if the system won't let you create enough threads.

Name of limit	Description	*name* argument
`PTHREAD_DESTRUCTOR_ITERATIONS`	maximum number of times an implementation will try to destroy the thread-specific data when a thread exits (Section 12.6)	`_SC_THREAD_DESTRUCTOR_ITERATIONS`
`PTHREAD_KEYS_MAX`	maximum number of keys that can be created by a process (Section 12.6)	`_SC_THREAD_KEYS_MAX`
`PTHREAD_STACK_MIN`	minimum number of bytes that can be used for a thread's stack (Section 12.3)	`_SC_THREAD_STACK_MIN`
`PTHREAD_THREADS_MAX`	maximum number of threads that can be created in a process (Section 12.3)	`_SC_THREAD_THREADS_MAX`

Figure 12.1 Thread limits and *name* arguments to `sysconf`

Figure 12.2 shows the values of the thread limits for the four implementations described in this book. When the implementation doesn't define the corresponding `sysconf` symbol (starting with `_SC_`), "no symbol" is listed. If the implementation's limit is indeterminate, "no limit" is listed. This doesn't mean that the value is unlimited, however. An "unsupported" entry means that the implementation defines the corresponding `sysconf` limit symbol, but the `sysconf` function doesn't recognize it.

Note that although an implementation may not provide access to these limits, that doesn't mean that the limits don't exist. It just means that the implementation doesn't provide us with a way to get at them using `sysconf`.

Limit	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
`PTHREAD_DESTRUCTOR_ITERATIONS`	no symbol	unsupported	no symbol	no limit
`PTHREAD_KEYS_MAX`	no symbol	unsupported	no symbol	no limit
`PTHREAD_STACK_MIN`	no symbol	unsupported	no symbol	4,096
`PTHREAD_THREADS_MAX`	no symbol	unsupported	no symbol	no limit

Figure 12.2 Examples of thread configuration limits

12.3 Thread Attributes

In all the examples in which we called `pthread_create` in Chapter 11, we passed in a null pointer instead of passing in a pointer to a `pthread_attr_t` structure. We can use the `pthread_attr_t` structure to modify the default attributes, and associate these attributes with threads that we create. We use the `pthread_attr_init` function to initialize the `pthread_attr_t` structure. After calling `pthread_attr_init`, the `pthread_attr_t` structure contains the default values for all the thread attributes supported by the implementation. To change individual attributes, we need to call other functions, as described later in this section.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);

                    Both return: 0 if OK, error number on failure
```

To deinitialize a `pthread_attr_t` structure, we call `pthread_attr_destroy`. If an implementation of `pthread_attr_init` allocated any dynamic memory for the attribute object, `pthread_attr_destroy` will free that memory. In addition, `pthread_attr_destroy` will initialize the attribute object with invalid values, so if it is used by mistake, `pthread_create` will return an error.

The `pthread_attr_t` structure is opaque to applications. This means that applications aren't supposed to know anything about its internal structure, thus promoting application portability. Following this model, POSIX.1 defines separate functions to query and set each attribute.

The thread attributes defined by POSIX.1 are summarized in Figure 12.3. POSIX.1 defines additional attributes in the real-time threads option, but we don't discuss those here. In Figure 12.3, we also show which platforms support each thread attribute. If the attribute is accessible through an obsolete interface, we show **ob** in the table entry.

Name	Description	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
detachstate	detached thread attribute	•	•	•	•
guardsize	guard buffer size in bytes at end of thread stack		•	•	•
stackaddr	lowest address of thread stack	**ob**	•	•	**ob**
stacksize	size in bytes of thread stack	•	•	•	•

Figure 12.3 POSIX.1 thread attributes

In Section 11.5, we introduced the concept of detached threads. If we are no longer interested in an existing thread's termination status, we can use `pthread_detach` to allow the operating system to reclaim the thread's resources when the thread exits.

If we know that we don't need the thread's termination status at the time we create the thread, we can arrange for the thread to start out in the detached state by modifying the *detachstate* thread attribute in the `pthread_attr_t` structure. We can use the `pthread_attr_setdetachstate` function to set the *detachstate* thread attribute to one of two legal values: `PTHREAD_CREATE_DETACHED` to start the thread in the detached state or `PTHREAD_CREATE_JOINABLE` to start the thread normally, so its termination status can be retrieved by the application.

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                int *detachstate);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

                         Both return: 0 if OK, error number on failure
```

We can call `pthread_attr_getdetachstate` to obtain the current *detachstate* attribute. The integer pointed to by the second argument is set to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`, depending on the value of the attribute in the given `pthread_attr_t` structure.

Example

Figure 12.4 shows a function that can be used to create a thread in the detached state.

```
#include "apue.h"
#include <pthread.h>

int
makethread(void *(*fn)(void *), void *arg)
{
    int             err;
    pthread_t       tid;
    pthread_attr_t  attr;

    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

Figure 12.4 Creating a thread in the detached state

Note that we ignore the return value from the call to `pthread_attr_destroy`. In this case, we initialized the thread attributes properly, so `pthread_attr_destroy` shouldn't fail. Nonetheless, if it does fail, cleaning up would be difficult: we would have to destroy the thread we just created, which is possibly already running, asynchronous to the execution of this function. By ignoring the error return from `pthread_attr_destroy`, the worst that can happen is that we leak a small amount of memory if `pthread_attr_init` allocated any. But if `pthread_attr_init` succeeded in initializing the thread attributes and then `pthread_attr_destroy` failed to clean up, we have no recovery strategy anyway, because the attributes structure is opaque to the application. The only interface defined to clean up the structure is `pthread_attr_destroy`, and it just failed. □

Support for thread stack attributes is optional for a POSIX-conforming operating system, but is required if the system is to conform to the XSI. At compile time, you can check whether your system supports each thread stack attribute using the _POSIX_THREAD_ATTR_STACKADDR and _POSIX_THREAD_ATTR_STACKSIZE symbols. If one is defined, then the system supports the corresponding thread stack attribute. You can also check at runtime, by using the _SC_THREAD_ATTR_STACKADDR and _SC_THREAD_ATTR_STACKSIZE parameters to the sysconf function.

POSIX.1 defines several interfaces to manipulate thread stack attributes. Two older functions, pthread_attr_getstackaddr and pthread_attr_setstackaddr, are marked as obsolete in Version 3 of the Single UNIX Specification, although many pthreads implementations still provide them. The preferred way to query and modify a thread's stack attributes is to use the newer functions pthread_attr_getstack and pthread_attr_setstack. These functions clear up ambiguities present in the definition of the older interfaces.

```
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(const pthread_attr_t *attr,
                          void *stackaddr, size_t *stacksize);

                        Both return: 0 if OK, error number on failure
```

These two functions are used to manage both the *stackaddr* and the *stacksize* thread attributes.

With a process, the amount of virtual address space is fixed. Since there is only one stack, its size usually isn't a problem. With threads, however, the same amount of virtual address space must be shared by all the thread stacks. You might have to reduce your default thread stack size if your application uses so many threads that the cumulative size of their stacks exceeds the available virtual address space. On the other hand, if your threads call functions that allocate large automatic variables or call functions many stack frames deep, you might need more than the default stack size.

If you run out of virtual address space for thread stacks, you can use malloc or mmap (see Section 14.9) to allocate space for an alternate stack and use pthread_attr_setstack to change the stack location of threads you create. The address specified by the *stackaddr* parameter is the lowest addressable address in the range of memory to be used as the thread's stack, aligned at the proper boundary for the processor architecture.

The *stackaddr* thread attribute is defined as the lowest memory address for the stack. This is not necessarily the start of the stack, however. If stacks grow from higher address to lower addresses for a given processor architecture, the *stackaddr* thread attribute will be the end of the stack instead of the beginning.

The drawback with pthread_attr_getstackaddr and pthread_attr_setstackaddr is that the *stackaddr* parameter was underspecified. It could have been interpreted as the start of the stack or as the lowest memory address of the memory extent to use as the stack. On architectures in which the stacks grow down from higher memory addresses to lower addresses, if the *stackaddr* parameter is the lowest memory address of the stack, then you need to know the stack size to determine the start of the stack. The `pthread_attr_getstack` and `pthread_attr_setstack` functions correct these shortcomings.

An application can also get and set the *stacksize* thread attribute using the `pthread_attr_getstacksize` and `pthread_attr_setstacksize` functions.

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

                         Both return: 0 if OK, error number on failure
```

The `pthread_attr_setstacksize` function is useful when you want to change the default stack size but don't want to deal with allocating the thread stacks on your own.

The *guardsize* thread attribute controls the size of the memory extent after the end of the thread's stack to protect against stack overflow. By default, this is set to `PAGESIZE` bytes. We can set the *guardsize* thread attribute to 0 to disable this feature: no guard buffer will be provided in this case. Also, if we change the *stackaddr* thread attribute, the system assumes that we will be managing our own stacks and disables stack guard buffers, just as if we had set the *guardsize* thread attribute to 0.

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                              size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);

                         Both return: 0 if OK, error number on failure
```

If the *guardsize* thread attribute is modified, the operating system might round it up to an integral multiple of the page size. If the thread's stack pointer overflows into the guard area, the application will receive an error, possibly with a signal.

The Single UNIX Specification defines several other optional thread attributes as part of the real-time threads option. We will not discuss them here.

More Thread Attributes

Threads have other attributes not represented by the `pthread_attr_t` structure:

- The cancelability state (discussed in Section 12.7)
- The cancelability type (also discussed in Section 12.7)
- The concurrency level

The concurrency level controls the number of kernel threads or processes on top of which the user-level threads are mapped. If an implementation keeps a one-to-one mapping between kernel-level threads and user-level threads, then changing the concurrency level will have no effect, since it is possible for all user-level threads to be scheduled. If the implementation multiplexes user-level threads on top of kernel-level threads or processes, however, you might be able to improve performance by increasing the number of user-level threads that can run at a given time. The `pthread_setconcurrency` function can be used to provide a hint to the system of the desired level of concurrency.

```
#include <pthread.h>

int pthread_getconcurrency(void);

                                    Returns: current concurrency level

int pthread_setconcurrency(int level);

                                Returns: 0 if OK, error number on failure
```

The `pthread_getconcurrency` function returns the current concurrency level. If the operating system is controlling the concurrency level (i.e., if no prior call to `pthread_setconcurrency` has been made), then `pthread_getconcurrency` will return 0.

The concurrency level specified by `pthread_setconcurrency` is only a hint to the system. There is no guarantee that the requested concurrency level will be honored. You can tell the system that you want it to decide for itself what concurrency level to use by passing a *level* of 0. Thus, an application can undo the effects of a prior call to `pthread_setconcurrency` with a nonzero value of *level* by calling it again with *level* set to 0.

12.4 Synchronization Attributes

Just as threads have attributes, so too do their synchronization objects. In this section, we discuss the attributes of mutexes, reader–writer locks, and condition variables.

Mutex Attributes

We use `pthread_mutexattr_init` to initialize a `pthread_mutexattr_t` structure and `pthread_mutexattr_destroy` to deinitialize one.

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

                           Both return: 0 if OK, error number on failure
```

The pthread_mutexattr_init function will initialize the pthread_mutexattr_t structure with the default mutex attributes. Two attributes of interest are the *process-shared* attribute and the *type* attribute. Within POSIX.1, the *process-shared* attribute is optional; you can test whether a platform supports it by checking whether the _POSIX_THREAD_PROCESS_SHARED symbol is defined. You can also check at runtime by passing the _SC_THREAD_PROCESS_SHARED parameter to the sysconf function. Although this option is not required to be provided by POSIX-conforming operating systems, the Single UNIX Specification requires that XSI-conforming operating systems do support this option.

Within a process, multiple threads can access the same synchronization object. This is the default behavior, as we saw in Chapter 11. In this case, the *process-shared* mutex attribute is set to PTHREAD_PROCESS_PRIVATE.

As we shall see in Chapters 14 and 15, mechanisms exist that allow independent processes to map the same extent of memory into their independent address spaces. Access to shared data by multiple processes usually requires synchronization, just as does access to shared data by multiple threads. If the *process-shared* mutex attribute is set to PTHREAD_PROCESS_SHARED, a mutex allocated from a memory extent shared between multiple processes may be used for synchronization by those processes.

We can use the pthread_mutexattr_getpshared function to query a pthread_mutexattr_t structure for its *process-shared* attribute. We can change the *process-shared* attribute with the pthread_mutexattr_setpshared function.

```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
                                 restrict attr,
                                 int *restrict pshared);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);

                              Both return: 0 if OK, error number on failure
```

The *process-shared* mutex attribute allows the pthread library to provide more efficient mutex implementations when the attribute is set to PTHREAD_PROCESS_PRIVATE, which is the default case with multithreaded applications. Then the pthread library can restrict the more expensive implementation to the case in which mutexes are shared among processes.

The *type* mutex attribute controls the characteristics of the mutex. POSIX.1 defines four types. The PTHREAD_MUTEX_NORMAL type is a standard mutex that doesn't do any special error checking or deadlock detection. The PTHREAD_MUTEX_ERRORCHECK mutex type provides error checking.

The PTHREAD_MUTEX_RECURSIVE mutex type allows the same thread to lock it multiple times without first unlocking it. A recursive mutex maintains a lock count and isn't released until it is unlocked the same number of times it is locked. So if you lock a recursive mutex twice and then unlock it, the mutex remains locked until it is unlocked a second time.

Finally, the PTHREAD_MUTEX_DEFAULT type can be used to request default semantics. Implementations are free to map this to one of the other types. On Linux, for example, this type is mapped to the normal mutex type.

The behavior of the four types is shown in Figure 12.5. The "Unlock when not owned" column refers to one thread unlocking a mutex that was locked by a different thread. The "Unlock when unlocked" column refers to what happens when a thread unlocks a mutex that is already unlocked, which usually is a coding mistake.

Mutex type	Relock without unlock?	Unlock when not owned?	Unlock when unlocked?
PTHREAD_MUTEX_NORMAL	deadlock	undefined	undefined
PTHREAD_MUTEX_ERRORCHECK	returns error	returns error	returns error
PTHREAD_MUTEX_RECURSIVE	allowed	returns error	returns error
PTHREAD_MUTEX_DEFAULT	undefined	undefined	undefined

Figure 12.5 Mutex type behavior

We can use pthread_mutexattr_gettype to get the mutex *type* attribute and pthread_mutexattr_settype to change the mutex *type* attribute.

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *
                              restrict attr, int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Both return: 0 if OK, error number on failure

Recall from Section 11.6 that a mutex is used to protect the condition that is associated with a condition variable. Before blocking the thread, the pthread_cond_wait and the pthread_cond_timedwait functions release the mutex associated with the condition. This allows other threads to acquire the mutex, change the condition, release the mutex, and signal the condition variable. Since the mutex must be held to change the condition, it is not a good idea to use a recursive mutex. If a recursive mutex is locked multiple times and used in a call to pthread_cond_wait, the condition can never be satisfied, because the unlock done by pthread_cond_wait doesn't release the mutex.

Recursive mutexes are useful when you need to adapt existing single-threaded interfaces to a multithreaded environment, but can't change the interfaces to your functions because of compatibility constraints. However, using recursive locks can be tricky, and they should be used only when no other solution is possible.

Example

Figure 12.6 illustrates a situation in which a recursive mutex might seem to solve a concurrency problem. Assume that func1 and func2 are existing functions in a library whose interfaces can't be changed, because applications exist that call them, and the applications can't be changed.

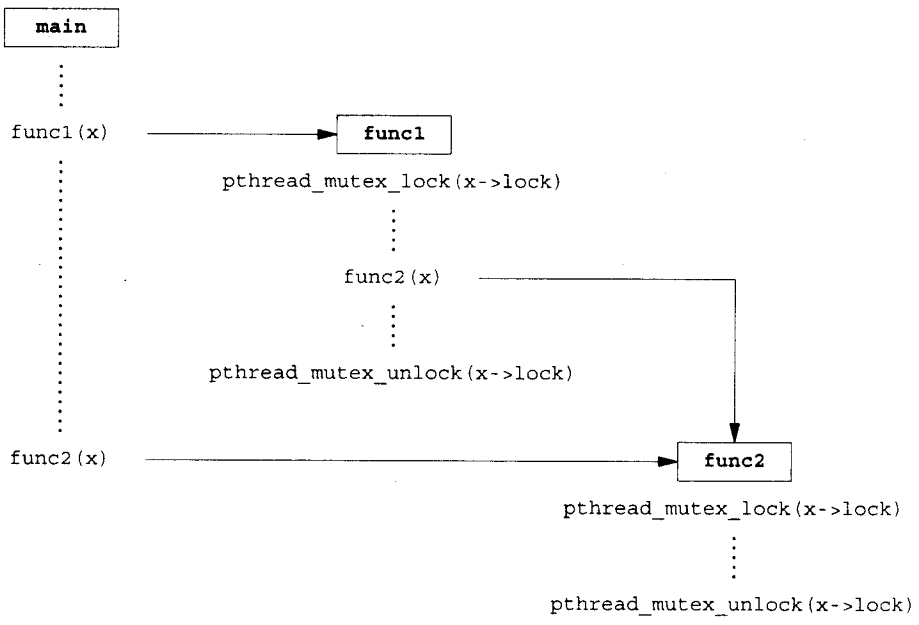
main
func1(x)
func1
pthread_mutex_lock(x->lock)
func2(x)
pthread_mutex_unlock(x->lock)
func2(x)
func2
pthread_mutex_lock(x->lock)
pthread_mutex_unlock(x->lock)

Figure 12.6 Recursive locking opportunity

To keep the interfaces the same, we embed a mutex in the data structure whose address (x) is passed in as an argument. This is possible only if we have provided an allocator function for the structure, so the application doesn't know about its size (assuming we must increase its size when we add a mutex to it).

This is also possible if we originally defined the structure with enough padding to allow us now to replace some pad fields with a mutex. Unfortunately, most programmers are unskilled at predicting the future, so this is not a common practice.

If both `func1` and `func2` must manipulate the structure and it is possible to access it from more than one thread at a time, then `func1` and `func2` must lock the mutex before manipulating the data. If `func1` must call `func2`, we will deadlock if the mutex type is not recursive. We could avoid using a recursive mutex if we could release the mutex before calling `func2` and reacquire it after `func2` returns, but this opens a window where another thread can possibly grab control of the mutex and change the data structure in the middle of `func1`. This may not be acceptable, depending on what protection the mutex is intended to provide.

Figure 12.7 shows an alternative to using a recursive mutex in this case. We can leave the interfaces to `func1` and `func2` unchanged and avoid a recursive mutex by providing a private version of `func2`, called `func2_locked`. To call `func2_locked`, we must hold the mutex embedded in the data structure whose address we pass as the argument. The body of `func2_locked` contains a copy of `func2`, and `func2` now simply acquires the mutex, calls `func2_locked`, and then releases the mutex.

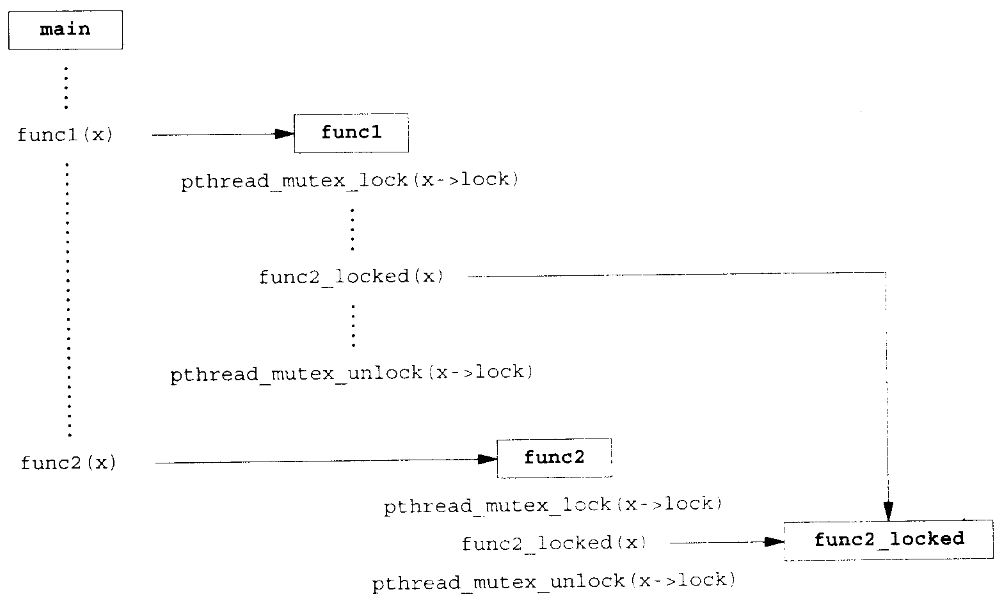
main
func1(x)
func1
pthread_mutex_lock(x->lock)
func2_locked(x)
pthread_mutex_unlock(x->lock)
func2(x)
func2
pthread_mutex_lock(x->lock)
func2_locked(x)
pthread_mutex_unlock(x->lock)
func2_locked

Figure 12.7 Avoiding a recursive locking opportunity

If we didn't have to leave the interfaces to the library functions unchanged, we could have added a second parameter to each function to indicate whether the structure is locked by the caller. It is usually better to leave the interfaces unchanged if we can, however, instead of polluting it with implementation artifacts.

The strategy of providing locked and unlocked versions of functions is usually applicable in simple situations. In more complex situations, such as when the library needs to call a function outside the library, which then might call back into the library, we need to rely on recursive locks. □

Example

The program in Figure 12.8 illustrates another situation in which a recursive mutex is necessary. Here, we have a "timeout" function that allows us to schedule another function to be run at some time in the future. Assuming that threads are an inexpensive resource, we can create a thread for each pending timeout. The thread waits until the time has been reached, and then it calls the function we've requested.

The problem arises when we can't create a thread or when the scheduled time to run the function has already passed. In these cases, we simply call the requested function now, from the current context. Since the function acquires the same lock that we currently hold, a deadlock will occur unless the lock is recursive.

```
#include "apue.h"
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

extern int makethread(void *(*)(void *), void *);

struct to_info {
    void    (*to_fn)(void *);    /* function */
    void    *to_arg;             /* argument */
    struct timespec to_wait;     /* time to wait */
};

#define SECTONSEC  1000000000    /* seconds to nanoseconds */
#define USECTONSEC 1000          /* microseconds to nanoseconds */

void *
timeout_helper(void *arg)
{
    struct to_info  *tip;

    tip = (struct to_info *)arg;
    nanosleep(&tip->to_wait, NULL);
    (*tip->to_fn)(tip->to_arg);
    return(0);
}

void
timeout(const struct timespec *when, void (*func)(void *), void *arg)
{
    struct timespec now;
    struct timeval  tv;
    struct to_info  *tip;
    int             err;

    gettimeofday(&tv, NULL);
    now.tv_sec = tv.tv_sec;
    now.tv_nsec = tv.tv_usec * USECTONSEC;
    if ((when->tv_sec > now.tv_sec) ||
      (when->tv_sec == now.tv_sec && when->tv_nsec > now.tv_nsec)) {
        tip = malloc(sizeof(struct to_info));
        if (tip != NULL) {
            tip->to_fn = func;
            tip->to_arg = arg;
            tip->to_wait.tv_sec = when->tv_sec - now.tv_sec;
            if (when->tv_nsec >= now.tv_nsec) {
                tip->to_wait.tv_nsec = when->tv_nsec - now.tv_nsec;
            } else {
                tip->to_wait.tv_sec--;
                tip->to_wait.tv_nsec = SECTONSEC - now.tv_nsec +
                  when->tv_nsec;
            }
            err = makethread(timeout_helper, (void *)tip);
            if (err == 0)
                return;
        }
    }

    /*
     * We get here if (a) when <= now, or (b) malloc fails, or
     * (c) we can't make a thread, so we just call the function now.
     */
    (*func)(arg);
}

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void
retry(void *arg)
{
    pthread_mutex_lock(&mutex);
    /* perform retry steps ... */
    pthread_mutex_unlock(&mutex);
}

int
main(void)
{
    int             err, condition, arg;
    struct timespec when;

    if ((err = pthread_mutexattr_init(&attr)) != 0)
        err_exit(err, "pthread_mutexattr_init failed");
    if ((err = pthread_mutexattr_settype(&attr,
      PTHREAD_MUTEX_RECURSIVE)) != 0)
        err_exit(err, "can't set recursive type");
    if ((err = pthread_mutex_init(&mutex, &attr)) != 0)
        err_exit(err, "can't create recursive mutex");
    /* ... */
    pthread_mutex_lock(&mutex);
    /* ... */
    if (condition) {
        /* calculate target time "when" */
        timeout(&when, retry, (void *)arg);
    }
    /* ... */
    pthread_mutex_unlock(&mutex);
    /* ... */
    exit(0);
}
```

Figure 12.8 Using a recursive mutex

We use the makethread function from Figure 12.4 to create a thread in the detached state. We want the function to run in the future, and we don't want to wait around for the thread to complete.

We could call sleep to wait for the timeout to expire, but that gives us only second granularity. If we want to wait for some time other than an integral number of seconds, we need to use nanosleep(2), which provides similar functionality.

Although nanosleep is required to be implemented only in the real-time extensions of the Single UNIX Specification, all the platforms discussed in this text support it.

The caller of timeout needs to hold a mutex to check the condition and to schedule the retry function as an atomic operation. The retry function will try to lock the same mutex. Unless the mutex is recursive, a deadlock will occur if the timeout function calls retry directly. □

Reader–Writer Lock Attributes

Reader–writer locks also have attributes, similar to mutexes. We use pthread_rwlockattr_init to initialize a pthread_rwlockattr_t structure and pthread_rwlockattr_destroy to deinitialize the structure.

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);

                              Both return: 0 if OK, error number on failure
```

The only attribute supported for reader–writer locks is the *process-shared* attribute. It is identical to the mutex *process-shared* attribute. Just as with the mutex *process-shared* attributes, a pair of functions is provided to get and set the *process-shared* attributes of reader–writer locks.

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
                                  restrict attr,
                                  int *restrict pshared);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared);

                              Both return: 0 if OK, error number on failure
```

Although POSIX defines only one reader–writer lock attribute, implementations are free to define additional, nonstandard ones.

Condition Variable Attributes

Condition variables have attributes, too. There is a pair of functions for initializing and deinitializing them, similar to mutexes and reader–writer locks.

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);

                           Both return: 0 if OK, error number on failure
```

Just as with the other synchronization primitives, condition variables support the *process-shared* attribute.

```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *
                                 restrict attr,
                                 int *restrict pshared);

int pthread_condattr_setpshared(pthread_condattr_t *attr,
                                 int pshared);

                           Both return: 0 if OK, error number on failure
```

12.5 Reentrancy

We discussed reentrant functions and signal handlers in Section 10.6. Threads are similar to signal handlers when it comes to reentrancy. With both signal handlers and threads, multiple threads of control can potentially call the same function at the same time.

If a function can be safely called by multiple threads at the same time, we say that the function is *thread-safe*. All functions defined in the Single UNIX Specification are guaranteed to be thread-safe, except those listed in Figure 12.9. In addition, the `ctermid` and `tmpnam` functions are not guaranteed to be thread-safe if they are passed a null pointer. Similarly, there is no guarantee that `wcrtomb` and `wcsrtombs` are thread-safe when they are passed a null pointer for their `mbstate_t` argument.

Implementations that support thread-safe functions will define the `_POSIX_THREAD_SAFE_FUNCTIONS` symbol in `<unistd.h>`. Applications can also use the `_SC_THREAD_SAFE_FUNCTIONS` argument with `sysconf` to check for support of thread-safe functions at runtime. All XSI-conforming implementations are required to support thread-safe functions.

When it supports the thread-safe functions feature, an implementation provides alternate, thread-safe versions of some of the POSIX.1 functions that aren't thread-safe. Figure 12.10 lists the thread-safe versions of these functions. Many functions are not

asctime	ecvt	gethostent	getutxline	putc_unlocked
basename	encrypt	getlogin	gmtime	putchar_unlocked
catgets	endgrent	getnetbyaddr	hcreate	putenv
crypt	endpwent	getnetbyname	hdestroy	pututxline
ctime	endutxent	getnetent	hsearch	rand
dbm_clearerr	fcvt	getopt	inet_ntoa	readdir
dbm_close	ftw	getprotobyname	l64a	setenv
dbm_delete	gcvt	getprotobynumber	lgamma	setgrent
dbm_error	getc_unlocked	getprotoent	lgammaf	setkey
dbm_fetch	getchar_unlocked	getpwent	lgammal	setpwent
dbm_firstkey	getdate	getpwnam	localeconv	setutxent
dbm_nextkey	getenv	getpwuid	localtime	strerror
dbm_open	getgrent	getservbyname	lrand48	strtok
dbm_store	getgrgid	getservbyport	mrand48	ttyname
dirname	getgrnam	getservent	nftw	unsetenv
dlerror	gethostbyaddr	getutxent	nl_langinfo	wcstombs
drand48	gethostbyname	getutxid	ptsname	wctomb

Figure 12.9 Functions *not* guaranteed to be thread-safe by POSIX.1

thread-safe, because they return data stored in a static memory buffer. They are made thread-safe by changing their interfaces to require that the caller provide its own buffer.

The functions listed in Figure 12.10 are named the same as their non-thread-safe relatives, but with an `_r` appended at the end of the name, signifying that these versions are reentrant.

If a function is reentrant with respect to multiple threads, we say that it is thread-safe. This doesn't tell us, however, whether the function is reentrant with respect to signal handlers. We say that a function that is safe to be reentered from an asynchronous signal handler is *async-signal safe*. We saw the async-signal safe functions in Figure 10.4 when we discussed reentrant functions in Section 10.6.

acstime_r	gmtime_r
ctime_r	localtime_r
getgrgid_r	rand_r
getgrnam_r	readdir_r
getlogin_r	strerror_r
getpwnam_r	strtok_r
getpwuid_r	ttyname_r

Figure 12.10 Alternate thread-safe functions

In addition to the functions listed in Figure 12.10, POSIX.1 provides a way to manage `FILE` objects in a thread-safe way. You can use `flockfile` and `ftrylockfile` to obtain a lock associated with a given `FILE` object. This lock is recursive: you can acquire it again, while you already hold it, without deadlocking. Although the exact implementation of the lock is unspecified, it is required that all standard I/O routines that manipulate `FILE` objects behave as if they call `flockfile` and `funlockfile` internally.

```
#include <stdio.h>

int ftrylockfile(FILE *fp);

                        Returns: 0 if OK, nonzero if lock can't be acquired

void flockfile(FILE *fp);

void funlockfile(FILE *fp);
```

Although the standard I/O routines might be implemented to be thread-safe from the perspective of their own internal data structures, it is still useful to expose the locking to applications. This allows applications to compose multiple calls to standard I/O functions into atomic sequences. Of course, when dealing with multiple `FILE` objects, you need to beware of potential deadlocks and to order your locks carefully.

If the standard I/O routines acquire their own locks, then we can run into serious performance degradation when doing character-at-a-time I/O. In this situation, we end up acquiring and releasing a lock for every character read or written. To avoid this overhead, unlocked versions of the character-based standard I/O routines are available.

```
#include <stdio.h>

int getchar_unlocked(void);

int getc_unlocked(FILE *fp);

              Both return: the next character if OK, EOF on end of file or error

int putchar_unlocked(int c);

int putc_unlocked(int c, FILE *fp);

                                          Both return: c if OK, EOF on error
```

These four functions should not be called unless surrounded by calls to `flockfile` (or `ftrylockfile`) and `funlockfile`. Otherwise, unpredictable results can occur (i.e., the types of problems that result from unsynchronized access to data by multiple threads of control).

Once you lock the `FILE` object, you can make multiple calls to these functions before releasing the lock. This amortizes the locking overhead across the amount of data read or written.

Example

Figure 12.11 shows a possible implementation of `getenv` (Section 7.9). This version is not reentrant. If two threads call it at the same time, they will see inconsistent results, because the string returned is stored in a single static buffer that is shared by all threads calling `getenv`.

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

static char envbuf[ARG_MAX];

extern char **environ;

char *
getenv(const char *name)
{
    int i, len;

    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            strcpy(envbuf, &environ[i][len+1]);
            return(envbuf);
        }
    }
    return(NULL);
}
```

Figure 12.11 A nonreentrant version of getenv

We show a reentrant version of getenv in Figure 12.12. This version is called getenv_r. It uses the pthread_once function (described in Section 12.6) to ensure that the thread_init function is called only once per process.

```
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>

extern char **environ;

pthread_mutex_t env_mutex;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;

static void
thread_init(void)
{
    pthread_mutexattr_t attr;

    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&env_mutex, &attr);
    pthread_mutexattr_destroy(&attr);
}
```

```
int
getenv_r(const char *name, char *buf, int buflen)
{
    int i, len, olen;

    pthread_once(&init_done, thread_init);
    len = strlen(name);
    pthread_mutex_lock(&env_mutex);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            olen = strlen(&environ[i][len+1]);
            if (olen >= buflen) {
                pthread_mutex_unlock(&env_mutex);
                return(ENOSPC);
            }
            strcpy(buf, &environ[i][len+1]);
            pthread_mutex_unlock(&env_mutex);
            return(0);
        }
    }
    pthread_mutex_unlock(&env_mutex);
    return(ENOENT);
}
```

Figure 12.12 A reentrant (thread-safe) version of getenv

To make getenv_r reentrant, we changed the interface so that the caller must provide its own buffer. Thus, each thread can use a different buffer to avoid interfering with the others. Note, however, that this is not enough to make getenv_r thread-safe. To make getenv_r thread-safe, we need to protect against changes to the environment while we are searching for the requested string. We can use a mutex to serialize access to the environment list by getenv_r and putenv.

We could have used a reader–writer lock to allow multiple concurrent calls to getenv_r, but the added concurrency probably wouldn't improve the performance of our program by very much, for two reasons. First, the environment list usually isn't very long, so we won't hold the mutex for too long while we scan the list. Second, calls to getenv and putenv are infrequent, so if we improve their performance, we won't affect the overall performance of the program very much.

If we make getenv_r thread-safe, that doesn't mean that it is reentrant with respect to signal handlers. If we use a nonrecursive mutex, we run the risk that a thread will deadlock itself if it calls getenv_r from a signal handler. If the signal handler interrupts the thread while it is executing getenv_r, we will already be holding env_mutex locked, so another attempt to lock it will block, causing the thread to deadlock. Thus, we must use a recursive mutex to prevent other threads from changing the data structures while we look at them, and also prevent deadlocks from signal handlers. The problem is that the pthread functions are not guaranteed to be async-signal safe, so we can't use them to make another function async-signal safe. □

12.6 Thread-Specific Data

Thread-specific data, also known as thread-private data, is a mechanism for storing and finding data associated with a particular thread. The reason we call the data thread-specific, or thread-private, is that we'd like each thread to access its own separate copy of the data, without worrying about synchronizing access with other threads.

Many people went to a lot of trouble designing a threads model that promotes sharing process data and attributes. So why would anyone want to promote interfaces that prevent sharing in this model? There are two reasons.

First, sometimes we need to maintain data on a per thread basis. Since there is no guarantee that thread IDs are small, sequential integers, we can't simply allocate an array of per thread data and use the thread ID as the index. Even if we could depend on small, sequential thread IDs, we'd like a little extra protection so that one thread can't mess with another's data.

The second reason for thread-private data is to provide a mechanism for adapting process-based interfaces to a multithreaded environment. An obvious example of this is `errno`. Recall the discussion of `errno` in Section 1.7. Older interfaces (before the advent of threads) defined `errno` as an integer accessible globally within the context of a process. System calls and library routines set `errno` as a side effect of failing. To make it possible for threads to use these same system calls and library routines, `errno` is redefined as thread-private data. Thus, one thread making a call that sets `errno` doesn't affect the value of `errno` for the other threads in the process.

Recall that all threads in a process have access to the entire address space of the process. Other than using registers, there is no way for one thread to prevent another from accessing its data. This is true even for thread-specific data. Even though the underlying implementation doesn't prevent access, the functions provided to manage thread-specific data promote data separation among threads.

Before allocating thread-specific data, we need to create a *key* to associate with the data. The key will be used to gain access to the thread-specific data. We use `pthread_key_create` to create a key.

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *keyp,
                       void (*destructor)(void *));
```

Returns: 0 if OK, error number on failure

The key created is stored in the memory location pointed to by *keyp*. The same key can be used by all threads in the process, but each thread will associate a different thread-specific data address with the key. When the key is created, the data address for each thread is set to a null value.

In addition to creating a key, `pthread_key_create` associates an optional destructor function with the key. When the thread exits, if the data address has been set to a non-null value, the destructor function is called with the data address as the only argument. If *destructor* is null, then no destructor function is associated with the key. When the thread exits normally, by calling `pthread_exit` or by returning, the destructor is called. But if the thread calls exit, _exit, _Exit, or abort, or otherwise exits abnormally, the destructor is not called.

Threads usually use malloc to allocate memory for their thread-specific data. The destructor function usually frees the memory that was allocated. If the thread exited without freeing the memory, then the memory would be lost: leaked by the process.

A thread can allocate multiple keys for thread-specific data. Each key can have a destructor associated with it. There can be a different destructor function for each key, or they can all use the same function. Each operating system implementation can place a limit on the number of keys a process can allocate (recall PTHREAD_KEYS_MAX from Figure 12.1).

When a thread exits, the destructors for its thread-specific data are called in an implementation-defined order. It is possible for the destructor function to call another function that might create new thread-specific data and associate it with the key. After all destructors are called, the system will check whether any non-null thread-specific values were associated with the keys and, if so, call the destructors again. This process will repeat until either all keys for the thread have null thread-specific data values or a maximum of PTHREAD_DESTRUCTOR_ITERATIONS (Figure 12.1) attempts have been made.

We can break the association of a key with the thread-specific data values for all threads by calling pthread_key_delete.

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t *key);
```

Returns: 0 if OK, error number on failure

Note that calling pthread_key_delete will not invoke the destructor function associated with the key. To free any memory associated with the key's thread-specific data values, we need to take additional steps in the application.

We need to ensure that a key we allocate doesn't change because of a race during initialization. Code like the following can result in two threads both calling pthread_key_create:

```
void destructor(void *);

pthread_key_t key;
int init_done = 0;

int
threadfunc(void *arg)
{
    if (!init_done) {
        init_done = 1;
        err = pthread_key_create(&key, destructor);
    }
    ...
}
```

Depending on how the system schedules threads, some threads might see one key value, whereas other threads might see a different value. The way to solve this race is to use `pthread_once`.

```
#include <pthread.h>

pthread_once_t initflag = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *initflag, void (*initfn)(void));

                               Returns: 0 if OK, error number on failure
```

The *initflag* must be a nonlocal variable (i.e., global or static) and initialized to `PTHREAD_ONCE_INIT`.

If each thread calls `pthread_once`, the system guarantees that the initialization routine, *initfn*, will be called only once, on the first call to `pthread_once`. The proper way to create a key without a race is as follows:

```
void destructor(void *);

pthread_key_t key;
pthread_once_t init_done = PTHREAD_ONCE_INIT;

void
thread_init(void)
{
     err = pthread_key_create(&key, destructor);
}

int
threadfunc(void *arg)
{
     pthread_once(&init_done, thread_init);
     ...
}
```

Once a key is created, we can associate thread-specific data with the key by calling `pthread_setspecific`. We can obtain the address of the thread-specific data with `pthread_getspecific`.

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);

                Returns: thread-specific data value or NULL if no value
                                     has been associated with the key

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

                               Returns: 0 if OK, error number on failure
```

If no thread-specific data has been associated with a key, pthread_getspecific will return a null pointer. We can use this to determine whether we need to call pthread_setspecific.

Example

In Figure 12.11, we showed a hypothetical implementation of getenv. We came up with a new interface to provide the same functionality, but in a thread-safe way (Figure 12.12). But what would happen if we couldn't modify our application programs to use the new interface? In that case, we could use thread-specific data to maintain a per thread copy of the data buffer used to hold the return string. This is shown in Figure 12.13.

```
#include <limits.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>

static pthread_key_t key;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_mutex_t env_mutex = PTHREAD_MUTEX_INITIALIZER;

extern char **environ;

static void
thread_init(void)
{
    pthread_key_create(&key, free);
}

char *
getenv(const char *name)
{
    int     i, len;
    char    *envbuf;

    pthread_once(&init_done, thread_init);
    pthread_mutex_lock(&env_mutex);
    envbuf = (char *)pthread_getspecific(key);
    if (envbuf == NULL) {
        envbuf = malloc(ARG_MAX);
        if (envbuf == NULL) {
            pthread_mutex_unlock(&env_mutex);
            return(NULL);
        }
        pthread_setspecific(key, envbuf);
    }
    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
            (environ[i][len] == '=')) {
              strcpy(envbuf, &environ[i][len+1]);
              pthread_mutex_unlock(&env_mutex);
              return(envbuf);
          }
      }
      pthread_mutex_unlock(&env_mutex);
      return(NULL);
}
```

Figure 12.13 A thread-safe, compatible version of getenv

We use pthread_once to ensure that only one key is created for the thread-specific data we will use. If pthread_getspecific returns a null pointer, we need to allocate the memory buffer and associate it with the key. Otherwise, we use the memory buffer returned by pthread_getspecific. For the destructor function, we use free to free the memory previously allocated by malloc. The destructor function will be called with the value of the thread-specific data only if the value is non-null.

Note that although this version of getenv is thread-safe, it is not async-signal safe. Even if we made the mutex recursive, we could not make it reentrant with respect to signal handlers, because it calls malloc, which itself is not async-signal safe. □

12.7 Cancel Options

Two thread attributes that are not included in the pthread_attr_t structure are the *cancelability state* and the *cancelability type*. These attributes affect the behavior of a thread in response to a call to pthread_cancel (Section 11.5).

The *cancelability state* attribute can be either PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE. A thread can change its *cancelability state* by calling pthread_setcancelstate.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);

                              Returns: 0 if OK, error number on failure
```

In one atomic operation, pthread_setcancelstate sets the current *cancelability state* to *state* and stores the previous *cancelability state* in the memory location pointed to by *oldstate*.

Recall from Section 11.5 that a call to pthread_cancel doesn't wait for a thread to terminate. In the default case, a thread will continue to execute after a cancellation request is made, until the thread reaches a *cancellation point*. A cancellation point is a place where the thread checks to see whether it has been canceled, and then acts on the request. POSIX.1 guarantees that cancellation points will occur when a thread calls any of the functions listed in Figure 12.14.

accept	mq_timedsend	putpmsg	sigsuspend
aio_suspend	msgrcv	pwrite	sigtimedwait
clock_nanosleep	msgsnd	read	sigwait
close	msync	readv	sigwaitinfo
connect	nanosleep	recv	sleep
creat	open	recvfrom	system
fcntl2	pause	recvmsg	tcdrain
fsync	poll	select	usleep
getmsg	pread	sem_timedwait	wait
getpmsg	pthread_cond_timedwait	sem_wait	waitid
lockf	pthread_cond_wait	send	waitpid
mq_receive	pthread_join	sendmsg	write
mq_send	pthread_testcancel	sendto	writev
mq_timedreceive	putmsg	sigpause	

Figure 12.14 Cancellation points defined by POSIX.1

A thread starts with a default *cancelability state* of PTHREAD_CANCEL_ENABLE. When the state is set to PTHREAD_CANCEL_DISABLE, a call to pthread_cancel will not kill the thread. Instead, the cancellation request remains pending for the thread. When the state is enabled again, the thread will act on any pending cancellation requests at the next cancellation point.

In addition to the functions listed in Figure 12.14, POSIX.1 specifies the functions listed in Figure 12.15 as optional cancellation points.

Note that several of the functions listed in Figure 12.15 are not discussed further in this text. Many are optional in the Single UNIX Specification.

If your application doesn't call one of the functions in Figure 12.14 or Figure 12.15 for a long period of time (if it is compute-bound, for example), then you can call pthread_testcancel to add your own cancellation points to the program.

```
#include <pthread.h>

void pthread_testcancel(void);
```

When you call pthread_testcancel, if a cancellation request is pending and if cancellation has not been disabled, the thread will be canceled. When cancellation is disabled, however, calls to pthread_testcancel have no effect.

The default cancellation type we have been describing is known as *deferred cancellation*. After a call to pthread_cancel, the actual cancellation doesn't occur until the thread hits a cancellation point. We can change the cancellation type by calling pthread_setcanceltype.

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);

                                Returns: 0 if OK, error number on failure
```

`catclose`	`ftell`	`getwc`	`printf`
`catgets`	`ftello`	`getwchar`	`putc`
`catopen`	`ftw`	`getwd`	`putc_unlocked`
`closedir`	`fwprintf`	`glob`	`putchar`
`closelog`	`fwrite`	`iconv_close`	`putchar_unlocked`
`ctermid`	`fwscanf`	`iconv_open`	`puts`
`dbm_close`	`getc`	`ioctl`	`pututxline`
`dbm_delete`	`getc_unlocked`	`lseek`	`putwc`
`dbm_fetch`	`getchar`	`mkstemp`	`putwchar`
`dbm_nextkey`	`getchar_unlocked`	`nftw`	`readdir`
`dbm_open`	`getcwd`	`opendir`	`readdir_r`
`dbm_store`	`getdate`	`openlog`	`remove`
`dlclose`	`getgrent`	`pclose`	`rename`
`dlopen`	`getgrgid`	`perror`	`rewind`
`endgrent`	`getgrgid_r`	`popen`	`rewinddir`
`endhostent`	`getgrnam`	`posix_fadvise`	`scanf`
`endnetent`	`getgrnam_r`	`posix_fallocate`	`seekdir`
`endprotoent`	`gethostbyaddr`	`posix_madvise`	`semop`
`endpwent`	`gethostbyname`	`posix_spawn`	`setgrent`
`endservent`	`gethostent`	`posix_spawnp`	`sethostent`
`endutxent`	`gethostname`	`posix_trace_clear`	`setnetent`
`fclose`	`getlogin`	`posix_trace_close`	`setprotoent`
`fcntl`	`getlogin_r`	`posix_trace_create`	`setpwent`
`fflush`	`getnetbyaddr`	`posix_trace_create_withlog`	`setservent`
`fgetc`	`getnetbyname`	`posix_trace_eventtypelist_getnext_id`	`setutxent`
`fgetpos`	`getnetent`	`posix_trace_eventtypelist_rewind`	`strerror`
`fgets`	`getprotobyname`	`posix_trace_flush`	`syslog`
`fgetwc`	`getprotobynumber`	`posix_trace_get_attr`	`tmpfile`
`fgetws`	`getprotoent`	`posix_trace_get_filter`	`tmpnam`
`fopen`	`getpwent`	`posix_trace_get_status`	`ttyname`
`fprintf`	`getpwnam`	`posix_trace_getnext_event`	`ttyname_r`
`fputc`	`getpwnam_r`	`posix_trace_open`	`ungetc`
`fputs`	`getpwuid`	`posix_trace_rewind`	`ungetwc`
`fputwc`	`getpwuid_r`	`posix_trace_set_filter`	`unlink`
`fputws`	`gets`	`posix_trace_shutdown`	`vfprintf`
`fread`	`getservbyname`	`posix_trace_timedgetnext_event`	`vfwprintf`
`freopen`	`getservbyport`	`posix_typed_mem_open`	`vprintf`
`fscanf`	`getservent`	`pthread_rwlock_rdlock`	`vwprintf`
`fseek`	`getutxent`	`pthread_rwlock_timedrdlock`	`wprintf`
`fseeko`	`getutxid`	`pthread_rwlock_timedwrlock`	`wscanf`
`fsetpos`	`getutxline`	`pthread_rwlock_wrlock`	

Figure 12.15 Optional cancellation points defined by POSIX.1

The *type* parameter can be either `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`. The `pthread_setcanceltype` function sets the cancellation type to *type* and returns the previous type in the integer pointed to by *oldtype*.

Asynchronous cancellation differs from deferred cancellation in that the thread can be canceled at any time. The thread doesn't necessarily need to hit a cancellation point for it to be canceled.

12.8 Threads and Signals

Dealing with signals can be complicated even with a process-based paradigm. Introducing threads into the picture makes things even more complicated.

Each thread has its own signal mask, but the signal disposition is shared by all threads in the process. This means that individual threads can block signals, but when a thread modifies the action associated with a given signal, all threads share the action. Thus, if one thread chooses to ignore a given signal, another thread can undo that choice by restoring the default disposition or installing a signal handler for the signal.

Signals are delivered to a single thread in the process. If the signal is related to a hardware fault or expiring timer, the signal is sent to the thread whose action caused the event. Other signals, on the other hand, are delivered to an arbitrary thread.

In Section 10.12, we discussed how processes can use `sigprocmask` to block signals from delivery. The behavior of `sigprocmask` is undefined in a multithreaded process. Threads have to use `pthread_sigmask` instead.

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

Returns: 0 if OK, error number on failure

The `pthread_sigmask` function is identical to `sigprocmask`, except that `pthread_sigmask` works with threads and returns an error code on failure instead of setting `errno` and returning −1.

A thread can wait for one or more signals to occur by calling `sigwait`.

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict signop);
```

Returns: 0 if OK, error number on failure

The *set* argument specifies the set of signals for which the thread is waiting. On return, the integer to which *signop* points will contain the number of the signal that was delivered.

If one of the signals specified in the set is pending at the time `sigwait` is called, then `sigwait` will return without blocking. Before returning, `sigwait` removes the signal from the set of signals pending for the process. To avoid erroneous behavior, a thread must block the signals it is waiting for before calling `sigwait`. The `sigwait` function will atomically unblock the signals and wait until one is delivered. Before returning, `sigwait` will restore the thread's signal mask. If the signals are not blocked at the time that `sigwait` is called, then a timing window is opened up where one of the signals can be delivered to the thread before it completes its call to `sigwait`.

The advantage to using `sigwait` is that it can simplify signal handling by allowing us to treat asynchronously-generated signals in a synchronous manner. We can prevent the signals from interrupting the threads by adding them to each thread's signal mask. Then we can dedicate specific threads to handling the signals. These dedicated threads can make function calls without having to worry about which functions are safe to call from a signal handler, because they are being called from normal thread context, not from a traditional signal handler interrupting a normal thread's execution.

If multiple threads are blocked in calls to `sigwait` for the same signal, only one of the threads will return from `sigwait` when the signal is delivered. If a signal is being caught (the process has established a signal handler by using `sigaction`, for example) and a thread is waiting for the same signal in a call to `sigwait`, it is left up to the implementation to decide which way to deliver the signal. In this case, the implementation could either allow `sigwait` to return or invoke the signal handler, but not both.

To send a signal to a process, we call `kill` (Section 10.9). To send a signal to a thread, we call `pthread_kill`.

```
#include <signal.h>

int pthread_kill(pthread_t thread, int signo);

                              Returns: 0 if OK, error number on failure
```

We can pass a *signo* value of 0 to check for existence of the thread. If the default action for a signal is to terminate the process, then sending the signal to a thread will still kill the entire process.

Note that alarm timers are a process resource, and all threads share the same set of alarms. Thus, it is not possible for multiple threads in a process to use alarm timers without interfering (or cooperating) with one another (this is the subject of Exercise 12.6).

Example

Recall that in Figure 10.23, we waited for the signal handler to set a flag indicating that the main program should exit. The only threads of control that could run were the main thread and the signal handler, so blocking the signals was sufficient to avoid missing a change to the flag. With threads, we need to use a mutex to protect the flag, as we show in the program in Figure 12.16.

```
#include "apue.h"
#include <pthread.h>

int         quitflag;   /* set nonzero by thread */
sigset_t    mask;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wait = PTHREAD_COND_INITIALIZER;

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            err_exit(err, "sigwait failed");
        switch (signo) {
        case SIGINT:
            printf("\ninterrupt\n");
            break;

        case SIGQUIT:
            pthread_mutex_lock(&lock);
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&wait);
            return(0);

        default:
            printf("unexpected signal %d\n", signo);
            exit(1);
        }
    }
}

int
main(void)
{
    int         err;
    sigset_t    oldmask;
    pthread_t   tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "SIG_BLOCK error");

    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&wait, &lock);
    pthread_mutex_unlock(&lock);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;

    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    exit(0);
}
```

Figure 12.16 Synchronous signal handling

Instead of relying on a signal handler that interrupts the main thread of control, we dedicate a separate thread of control to handle the signals. We change the value of quitflag under the protection of a mutex so that the main thread of control can't miss the wake-up call made when we call pthread_cond_signal. We use the same mutex in the main thread of control to check the value of the flag, and atomically release the mutex and wait for the condition.

Note that we block SIGINT and SIGQUIT in the beginning of the main thread. When we create the thread to handle signals, the thread inherits the current signal mask. Since sigwait will unblock the signals, only one thread is available to receive signals. This enables us to code the main thread without having to worry about interrupts from these signals.

If we run this program, we get output similar to that from Figure 10.23:

```
$ ./a.out
^?                    type the interrupt character
interrupt
^?                    type the interrupt character again
interrupt
^?                    and again
interrupt
^\ $                  now terminate with quit character
```

□

Linux implements threads as separate processes, sharing resources using clone(2). Because of this, the behavior of threads on Linux differs from that on other implementations when it comes to signals. In the POSIX.1 thread model, asynchronous signals are sent to a process, and then an individual thread within the process is selected to receive the signal, based on which threads are not currently blocking the signal. On Linux, an asynchronous signal is sent to a particular thread, and since each thread executes as a separate process, the system is unable to select a thread that isn't currently blocking the signal. The result is that the thread may not notice the signal. Thus, programs like the one in Figure 12.16 work when the signal is generated from the terminal driver, which signals the process group, but when you try to send a signal to the process using kill, it doesn't work as expected on Linux.

12.9 Threads and fork

When a thread calls fork, a copy of the entire process address space is made for the child. Recall the discussion of copy-on-write in Section 8.3. The child is an entirely different process from the parent, and as long as neither one makes changes to its memory contents, copies of the memory pages can be shared between parent and child.

By inheriting a copy of the address space, the child also inherits the state of every mutex, reader–writer lock, and condition variable from the parent process. If the parent consists of more than one thread, the child will need to clean up the lock state if it isn't going to call exec immediately after fork returns.

Inside the child process, only one thread exists. It is made from a copy of the thread that called fork in the parent. If the threads in the parent process hold any locks, the locks will also be held in the child process. The problem is that the child process doesn't contain copies of the threads holding the locks, so there is no way for the child to know which locks are held and need to be unlocked.

This problem can be avoided if the child calls one of the exec functions directly after returning from fork. In this case, the old address space is discarded, so the lock state doesn't matter. This is not always possible, however, so if the child needs to continue processing, we need to use a different strategy.

To clean up the lock state, we can establish *fork handlers* by calling the function pthread_atfork.

```
#include <pthread.h>

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

                                Returns: 0 if OK, error number on failure
```

With pthread_atfork, we can install up to three functions to help clean up the locks. The *prepare* fork handler is called in the parent before fork creates the child process. This fork handler's job is to acquire all locks defined by the parent. The *parent* fork handler is called in the context of the parent after fork has created the child process, but before fork has returned. This fork handler's job is to unlock all the locks acquired by the *prepare* fork handler. The *child* fork handler is called in the context of the child process before returning from fork. Like the *parent* fork handler, the *child* fork handler too must release all the locks acquired by the *prepare* fork handler.

Note that the locks are not locked once and unlocked twice, as it may appear. When the child address space is created, it gets a copy of all locks that the parent defined. Because the *prepare* fork handler acquired all the locks, the memory in the parent and the memory in the child start out with identical contents. When the parent and the child unlock their "copy" of the locks, new memory is allocated for the child, and the memory contents from the parent are copied to the child's memory (copy-on-write), so we are left with a situation that looks as if the parent locked all its copies of the locks and the child locked all its copies of the locks. The parent and the child end up unlocking duplicate locks stored in different memory locations, as if the following sequence of events occurred.

1. The parent acquired all its locks.
2. The child acquired all its locks.
3. The parent released its locks.
4. The child released its locks.

We can call pthread_atfork multiple times to install more than one set of fork handlers. If we don't have a need to use one of the handlers, we can pass a null pointer for the particular handler argument, and it will have no effect. When multiple fork handlers are used, the order in which the handlers are called differs. The *parent* and *child* fork handlers are called in the order in which they were registered, whereas the *prepare* fork handlers are called in the opposite order from which they were registered. This allows multiple modules to register their own fork handlers and still honor the locking hierarchy.

For example, assume that module A calls functions from module B and that each module has its own set of locks. If the locking hierarchy is A before B, module B must install its fork handlers before module A. When the parent calls `fork`, the following steps are taken, assuming that the child process runs before the parent.

1. The *prepare* fork handler from module A is called to acquire all module A's locks.
2. The *prepare* fork handler from module B is called to acquire all module B's locks.
3. A child process is created.
4. The *child* fork handler from module B is called to release all module B's locks in the child process.
5. The *child* fork handler from module A is called to release all module A's locks in the child process.
6. The `fork` function returns to the child.
7. The *parent* fork handler from module B is called to release all module B's locks in the parent process.
8. The *parent* fork handler from module A is called to release all module A's locks in the parent process.
9. The `fork` function returns to the parent.

If the fork handlers serve to clean up the lock state, what cleans up the state of condition variables? On some implementations, condition variables might not need any cleaning up. However, an implementation that uses a lock as part of the implementation of condition variables will require cleaning up. The problem is that no interface exists to allow us to do this. If the lock is embedded in the condition variable data structure, then we can't use condition variables after calling `fork`, because there is no portable way to clean up its state. On the other hand, if an implementation uses a global lock to protect all condition variable data structures in a process, then the implementation itself can clean up the lock in the `fork` library routine. Application programs shouldn't rely on implementation details like this, however.

Example

The program in Figure 12.17 illustrates the use of `pthread_atfork` and fork handlers.

```
#include "apue.h"
#include <pthread.h>

pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void
prepare(void)
{
    printf("preparing locks...\n");
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
}
```

```
void
parent(void)
{
    printf("parent unlocking locks...\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
}

void
child(void)
{
    printf("child unlocking locks...\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
}

void *
thr_fn(void *arg)
{
    printf("thread started...\n");
    pause();
    return(0);
}

int
main(void)
{
    int         err;
    pid_t       pid;
    pthread_t   tid;

#if defined(BSD) || defined(MACOS)
    printf("pthread_atfork is unsupported\n");
#else
    if ((err = pthread_atfork(prepare, parent, child)) != 0)
        err_exit(err, "can't install fork handlers");
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");
    sleep(2);
    printf("parent about to fork...\n");
    if ((pid = fork()) < 0)
        err_quit("fork failed");
    else if (pid == 0)  /* child */
        printf("child returned from fork\n");
    else        /* parent */
        printf("parent returned from fork\n");
#endif
    exit(0);
}
```

Figure 12.17 pthread_atfork example

We define two mutexes, `lock1` and `lock2`. The *prepare* fork handler acquires them both, the *child* fork handler releases them in the context of the child process, and the *parent* fork handler releases them in the context of the parent process.

When we run this program, we get the following output:

```
$ ./a.out
thread started...
parent about to fork...
preparing locks...
child unlocking locks...
child returned from fork
parent unlocking locks...
parent returned from fork
```

As we can see, the *prepare* fork handler runs after `fork` is called, the *child* fork handler runs before `fork` returns in the child, and the *parent* fork handler runs before `fork` returns in the parent. □

12.10 Threads and I/O

We introduced the `pread` and `pwrite` functions in Section 3.11. These functions are helpful in a multithreaded environment, because all threads in a process share the same file descriptors.

Consider two threads reading from or writing to the same file descriptor at the same time.

```
       Thread A                        Thread B
lseek(fd, 300, SEEK_SET);       lseek(fd, 700, SEEK_SET);
read(fd, buf1, 100);            read(fd, buf2, 100);
```

If thread A executes the `lseek` and then thread B calls `lseek` before thread A calls `read`, then both threads will end up reading the same record. Clearly, this isn't what was intended.

To solve this problem, we can use `pread` to make the setting of the offset and the reading of the data one atomic operation.

```
       Thread A                        Thread B
pread(fd, buf1, 100, 300);      pread(fd, buf2, 100, 700);
```

Using `pread`, we can ensure that thread A reads the record at offset 300, whereas thread B reads the record at offset 700. We can use `pwrite` to solve the problem of concurrent threads writing to the same file.

12.11 Summary

Threads provide an alternate model for partitioning concurrent tasks in UNIX systems. Threads promote sharing among separate threads of control, but present unique synchronization problems. In this chapter, we looked at how we can fine-tune our threads and their synchronization primitives. We discussed reentrancy with threads. We also looked at how threads interact with some of the process-oriented system calls.

Exercises

12.1 Run the program in Figure 12.17 on a Linux system, but redirect the output into a file. Explain the results.

12.2 Implement `putenv_r`, a reentrant version of `putenv`. Make sure that your implementation is async-signal safe as well as thread-safe.

12.3 Can you make the program in Figure 12.13 async-signal safe by blocking signals at the beginning of the function and restoring the previous signal mask before returning? Explain.

12.4 Write a program to exercise the version of `getenv` from Figure 12.13. Compile and run the program on FreeBSD. What happens? Explain.

12.5 Given that you can create multiple threads to perform different tasks within a program, explain why you might still need to use `fork`.

12.6 Reimplement the program in Figure 10.29 to make it thread-safe without using `nanosleep`.

12.7 After calling `fork`, could we safely reinitialize a condition variable in the child process by first destroying the condition variable with `pthread_cond_destroy` and then initializing it with `pthread_cond_init`?

13

Daemon Processes

13.1 Introduction

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down. Because they don't have a controlling terminal, we say that they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

In this chapter, we look at the process structure of daemons and how to write a daemon. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

> For a discussion of the historical background of the term *daemon* as it applies to computer systems, see Raymond [1996].

13.2 Daemon Characteristics

Let's look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions that we described in Chapter 9. The ps(1) command prints the status of various processes in the system. There are a multitude of options—consult your system's manual for all the details. We'll execute

```
ps -axj
```

under BSD-based systems to see the information we need for this discussion. The `-a` option shows the status of processes owned by others, and `-x` shows processes that don't have a controlling terminal. The `-j` option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID.

Under System V–based systems, a similar command is `ps -efjc`. (In an attempt to improve security, some UNIX systems don't allow us to use `ps` to look at any processes other than our own.) The output from `ps` looks like

```
 PPID   PID  PGID   SID TTY TPGID UID   COMMAND
    0     1     0     0 ?      -1   0   init
    1     2     1     1 ?      -1   0   [keventd]
    1     3     1     1 ?      -1   0   [kapmd]
    0     5     1     1 ?      -1   0   [kswapd]
    0     6     1     1 ?      -1   0   [bdflush]
    0     7     1     1 ?      -1   0   [kupdated]
    1  1009  1009  1009 ?      -1  32   portmap
    1  1048  1048  1048 ?      -1   0   syslogd -m 0
    1  1335  1335  1335 ?      -1   0   xinetd -pidfile /var/run/xinetd.pid
    1  1403     1     1 ?      -1   0   [nfsd]
    1  1405     1     1 ?      -1   0   [lockd]
 1405  1406     1     1 ?      -1   0   [rpciod]
    1  1853  1853  1853 ?      -1   0   crond
    1  2182  2182  2182 ?      -1   0   /usr/sbin/cupsd
```

We have removed a few columns that don't interest us, such as the accumulated CPU time. The column headings, in order, are the parent process ID, process ID, process group ID, session ID, terminal name, terminal process group ID (the foreground process group associated with the controlling terminal), user ID, and command string.

The system that this `ps` command was run on (Linux) supports the notion of a session ID, which we mentioned with the `setsid` function in Section 9.5. The session ID is simply the process ID of the session leader. A BSD-based system, however, will print the address of the `session` structure corresponding to the process group that the process belongs to (Section 9.11).

The system processes you see will depend on the operating system implementation. Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure. (An exception to this is `init`, since it is a user-level command started by the kernel at boot time.) Kernel processes are special and generally exist for the entire lifetime of the system. They run with superuser privileges and have no controlling terminal and no command line.

Process 1 is usually `init`, as we described in Section 8.2. It is a system daemon responsible for, among other things, starting system services specific to various run levels. These services are usually implemented with the help of their own daemons.

On Linux, the `keventd` daemon provides process context for running scheduled functions in the kernel. The `kapmd` daemon provides support for the advanced power management features available with various computer systems. The `kswapd` daemon is also known as the pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time, so the pages can be reclaimed.

The Linux kernel flushes cached data to disk using two additional daemons: `bdflush` and `kupdated`. The `bdflush` daemon flushes dirty buffers from the buffer cache back to disk when available memory reaches a low-water mark. The `kupdated` daemon flushes dirty pages back to disk at regular intervals to decrease data loss in the event of a system failure.

The portmapper daemon, `portmap`, provides the service of mapping RPC (Remote Procedure Call) program numbers to network port numbers. The `syslogd` daemon is available to any program to log system messages for an operator. The messages may be printed on a console device and also written to a file. (We describe the `syslog` facility in Section 13.4.)

We talked about the `inetd` daemon (`xinetd`) in Section 9.3. It listens on the system's network interfaces for incoming requests for various network servers. The `nfsd`, `lockd`, and `rpciod` daemons provide support for the Network File System (NFS).

The `cron` daemon (`crond`) executes commands at specified dates and times. Numerous system administration tasks are handled by having programs executed regularly by `cron`. The `cupsd` daemon is a print spooler; it handles print requests on the system.

Note that most of the daemons run with superuser privilege (a user ID of 0). None of the daemons has a controlling terminal: the terminal name is set to a question mark, and the terminal foreground process group is −1. The kernel daemons are started without a controlling terminal. The lack of a controlling terminal in the user-level daemons is probably the result of the daemons having called `setsid`. All the user-level daemons are process group leaders and session leaders and are the only processes in their process group and session. Finally, note that the parent of most of these daemons is the `init` process.

13.3 Coding Rules

Some basic rules to coding a daemon prevent unwanted interactions from happening. We state these rules and then show a function, `daemonize`, that implements them.

1. The first thing to do is call `umask` to set the file mode creation mask to 0. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions. For example, if it specifically creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts.
2. Call `fork` and have the parent `exit`. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to `setsid` that is done next.
3. Call `setsid` to create a new session. The three steps listed in Section 9.5 occur. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.

Under System V-based systems, some people recommend calling `fork` again at this point and having the parent terminate. The second child continues as the daemon. This guarantees that the daemon is not a session leader, which prevents it from acquiring a controlling terminal under the System V rules (Section 9.6). Alternatively, to avoid acquiring a controlling terminal, be sure to specify `O_NOCTTY` whenever opening a terminal device.

4. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.

   Alternatively, some daemons might change the current working directory to some specific location, where they will do all their work. For example, line printer spooling daemons often change to their spool directory.

5. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). We can use our `open_max` function (Figure 2.16) or the `getrlimit` function (Section 7.11) to determine the highest descriptor and close all descriptors up to that value.

6. Some daemons open file descriptors 0, 1, and 2 to `/dev/null` so that any library routines that try to read from standard input or write to standard output or standard error will have no effect. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

Example

Figure 13.1 shows a function that can be called from a program that wants to initialize itself as a daemon.

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int                 i, fd0, fd1, fd2;
    pid_t               pid;
    struct rlimit       rl;
    struct sigaction    sa;
```

```
/*
 * Clear file creation mask.
 */
umask(0);

/*
 * Get maximum number of file descriptors.
 */
if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
    err_quit("%s: can't get file limit", cmd);

/*
 * Become a session leader to lose controlling TTY.
 */
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
else if (pid != 0) /* parent */
    exit(0);
setsid();

/*
 * Ensure future opens won't allocate controlling TTYs.
 */
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
    err_quit("%s: can't ignore SIGHUP");
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
else if (pid != 0) /* parent */
    exit(0);

/*
 * Change the current working directory to the root so
 * we won't prevent file systems from being unmounted.
 */
if (chdir("/") < 0)
    err_quit("%s: can't change directory to /");

/*
 * Close all open file descriptors.
 */
if (rl.rlim_max == RLIM_INFINITY)
    rl.rlim_max = 1024;
for (i = 0; i < rl.rlim_max; i++)
    close(i);

/*
 * Attach file descriptors 0, 1, and 2 to /dev/null.
 */
```

```
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
    fd2 = dup(0);

    /*
     * Initialize the log file.
     */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
          fd0, fd1, fd2);
        exit(1);
    }
}
```

Figure 13.1 Initialize a daemon process

If the daemonize function is called from a main program that then goes to sleep, we can check the status of the daemon with the ps command:

```
$ ./a.out
$ ps -axj
 PPID   PID  PGID   SID TTY TPGID UID   COMMAND
    1  3346  3345  3345 ?      -1 501   ./a.out
$ ps -axj | grep 3345
    1  3346  3345  3345 ?      -1 501   ./a.out
```

We can also use ps to verify that no active process exists with ID 3345. This means that our daemon is in an orphaned process group (Section 9.10) and is not a session leader and thus has no chance of allocating a controlling terminal. This is a result of performing the second fork in the daemonize function. We can see that our daemon has been initialized correctly. □

13.4 Error Logging

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system. We also don't want each daemon writing its own error messages into a separate file. It would be a headache for anyone administering the system to keep up with which daemon writes to which log file and to check these files on a regular basis. A central daemon error-logging facility is required.

The BSD syslog facility was developed at Berkeley and used widely in 4.2BSD. Most systems derived from BSD support syslog.

Until SVR4, System V never had a central daemon logging facility.

The syslog function is included as an XSI extension in the Single UNIX Specification.

The BSD syslog facility has been widely used since 4.2BSD. Most daemons use this facility. Figure 13.2 illustrates its structure.

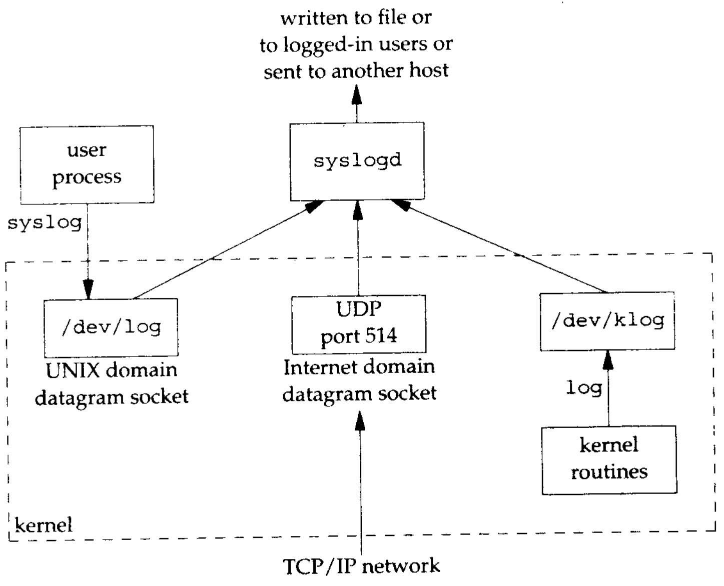

Figure 13.2 The BSD syslog facility

There are three ways to generate log messages:

1. Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device. We won't describe this function any further, since we're not interested in writing kernel routines.
2. Most user processes (daemons) call the syslog(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

Refer to Stevens, Fenner, and Rudoff [2004] for details on UNIX domain sockets and UDP sockets.

Normally, the syslogd daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file.

Our interface to this facility is through the `syslog` function.

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);

void syslog(int priority, const char *format, ...);

void closelog(void);

int setlogmask(int maskpri);

                        Returns: previous log priority mask value
```

Calling `openlog` is optional. If it's not called, the first time `syslog` is called, `openlog` is called automatically. Calling `closelog` is also optional—it just closes the descriptor that was being used to communicate with the `syslogd` daemon.

Calling `openlog` lets us specify an *ident* that is added to each log message. This is normally the name of the program (`cron`, `inetd`, etc.). The *option* argument is a bitmask specifying various options. Figure 13.3 describes the available options, including a bullet in the XSI column if the option is included in the `openlog` definition in the Single UNIX Specification.

option	XSI	Description
LOG_CONS	•	If the log message can't be sent to `syslogd` via the UNIX domain datagram, the message is written to the console instead.
LOG_NDELAY	•	Open the UNIX domain datagram socket to the `syslogd` daemon immediately; don't wait until the first message is logged. Normally, the socket is not opened until the first message is logged.
LOG_NOWAIT	•	Do not wait for child processes that might have been created in the process of logging the message. This prevents conflicts with applications that catch `SIGCHLD`, since the application might have retrieved the child's status by the time that `syslog` calls `wait`.
LOG_ODELAY	•	Delay the open of the connection to the `syslogd` daemon until the first message is logged.
LOG_PERROR		Write the log message to standard error in addition to sending it to `syslogd`. (Unavailable on Solaris.)
LOG_PID	•	Log the process ID with each message. This is intended for daemons that `fork` a child process to handle different requests (as compared to daemons, such as `syslogd`, that never call `fork`).

Figure 13.3 The *option* argument for `openlog`

The *facility* argument for `openlog` is taken from Figure 13.4. Note that the Single UNIX Specification defines only a subset of the facility codes typically available on a given platform. The reason for the *facility* argument is to let the configuration file specify that messages from different facilities are to be handled differently. If we don't call `openlog`, or if we call it with a *facility* of 0, we can still specify the facility as part of the *priority* argument to `syslog`.

facility	XSI	Description
LOG_AUTH		authorization programs: login, su, getty, ...
LOG_AUTHPRIV		same as LOG_AUTH, but logged to file with restricted permissions
LOG_CRON		cron and at
LOG_DAEMON		system daemons: inetd, routed, ...
LOG_FTP		the FTP daemon (ftpd)
LOG_KERN		messages generated by the kernel
LOG_LOCAL0	•	reserved for local use
LOG_LOCAL1	•	reserved for local use
LOG_LOCAL2	•	reserved for local use
LOG_LOCAL3	•	reserved for local use
LOG_LOCAL4	•	reserved for local use
LOG_LOCAL5	•	reserved for local use
LOG_LOCAL6	•	reserved for local use
LOG_LOCAL7	•	reserved for local use
LOG_LPR		line printer system: lpd, lpc, ...
LOG_MAIL		the mail system
LOG_NEWS		the Usenet network news system
LOG_SYSLOG		the syslogd daemon itself
LOG_USER	•	messages from other user processes (default)
LOG_UUCP		the UUCP system

Figure 13.4 The *facility* argument for openlog

level	Description
LOG_EMERG	emergency (system is unusable) (highest priority)
LOG_ALERT	condition that must be fixed immediately
LOG_CRIT	critical condition (e.g., hard device error)
LOG_ERR	error condition
LOG_WARNING	warning condition
LOG_NOTICE	normal, but significant condition
LOG_INFO	informational message
LOG_DEBUG	debug message (lowest priority)

Figure 13.5 The syslog *levels* (ordered)

We call syslog to generate a log message. The *priority* argument is a combination of the *facility* shown in Figure 13.4 and a *level*, shown in Figure 13.5. These *levels* are ordered by priority, from highest to lowest.

The *format* argument and any remaining arguments are passed to the vsprintf function for formatting. Any occurrence of the two characters %m in the *format* are first replaced with the error message string (strerror) corresponding to the value of errno.

The setlogmask function can be used to set the log priority mask for the process. This function returns the previous mask. When the log priority mask is set, messages are not logged unless their priority is set in the log priority mask. Note that attempts to set the log priority mask to 0 will have no effect.

The `logger(1)` program is also provided by many systems as a way to send log messages to the `syslog` facility. Some implementations allow optional arguments to this program, specifying the *facility*, *level*, and *ident*, although the Single UNIX Specification doesn't define any options. The `logger` command is intended for a shell script running noninteractively that needs to generate log messages.

Example

In a (hypothetical) line printer spooler daemon, you might encounter the sequence

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

The first call sets the *ident* string to the program name, specifies that the process ID should always be printed, and sets the default *facility* to the line printer system. The call to `syslog` specifies an error condition and a message string. If we had not called `openlog`, the second call could have been

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Here, we specify the *priority* argument as a combination of a *level* and a *facility*. □

In addition to `syslog`, many platforms provide a variant that handles variable argument lists.

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

All four platforms described in this book provide `vsyslog`, but it is not included in the Single UNIX Specification.

Most `syslogd` implementations will queue messages for a short time. If a duplicate message arrives during this time, the `syslog` daemon will not write it to the log. Instead, the daemon will print out a message similar to "last message repeated *N* times."

13.5 Single-Instance Daemons

Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation. The daemon might need exclusive access to a device, for example. In the case of the `cron` daemon, if multiple instances were running, each copy might try to start a single scheduled operation, resulting in duplicate operations and probably an error.

If the daemon needs to access a device, the device driver will sometimes prevent multiple opens of the corresponding device node in `/dev`. This restricts us to one copy of the daemon running at a time. If no such device is available, however, we need to do the work ourselves.

The file- and record-locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running. (We discuss file and record locking in Section 14.3.) If each daemon creates a file and places a write lock on the entire file, only one such write lock will be allowed to be created. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running.

File and record locking provides a convenient mutual-exclusion mechanism. If the daemon obtains a write-lock on an entire file, the lock will be removed automatically if the daemon exits. This simplifies recovery, removing the need for us to clean up from the previous instance of the daemon.

Example

The function shown in Figure 13.6 illustrates the use of file and record locking to ensure that only one copy of a daemon is running.

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);

int
already_running(void)
{
    int     fd;
    char    buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "can't open %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

Figure 13.6 Ensure that only one copy of a daemon is running

Each copy of the daemon will try to create a file and write its process ID in it. This will allow administrators to identify the process easily. If the file is already locked, the `lockfile` function will fail with `errno` set to `EACCES` or `EAGAIN`, so we return 1, indicating that the daemon is already running. Otherwise, we truncate the file, write our process ID to it, and return 0.

We need to truncate the file, because the previous instance of the daemon might have had a process ID larger than ours, with a larger string length. For example, if the previous instance of the daemon was process ID 12345, and the new instance is process ID 9999, when we write the process ID to the file, we will be left with 99995 in the file. Truncating the file prevents data from the previous daemon appearing as if it applies to the current daemon. □

13.6 Daemon Conventions

Several common conventions are followed by daemons in the UNIX System.

- If the daemon uses a lock file, the file is usually stored in `/var/run`. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually *name*`.pid`, where *name* is the name of the daemon or the service. For example, the name of the `cron` daemon's lock file is `/var/run/crond.pid`.
- If the daemon supports configuration options, they are usually stored in `/etc`. The configuration file is named *name*`.conf`, where *name* is the name of the daemon or the name of the service. For example, the configuration for the `syslogd` daemon is `/etc/syslog.conf`.
- Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (`/etc/rc*` or `/etc/init.d/*`). If the daemon should be restarted automatically when it exits, we can arrange for `init` to restart it if we include a `respawn` entry for it in `/etc/inittab`.
- If a daemon has a configuration file, the daemon reads it when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch `SIGHUP` and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive `SIGHUP`. Thus, they can safely reuse it.

Example

The program shown in Figure 13.7 shows one way a daemon can reread its configuration file. The program uses sigwait and multiple threads, as discussed in Section 12.8.

```
#include "apue.h"
#include <pthread.h>
#include <syslog.h>

sigset_t    mask;

extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            syslog(LOG_ERR, "sigwait failed");
            exit(1);
        }

        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Re-reading configuration file");
            reread();
            break;

        case SIGTERM:
            syslog(LOG_INFO, "got SIGTERM; exiting");
            exit(0);

        default:
            syslog(LOG_INFO, "unexpected signal %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
```

```
{
    int                 err;
    pthread_t           tid;
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Restore SIGHUP default and block all signals.
     */
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't restore SIGHUP default");
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "SIG_BLOCK error");

    /*
     * Create a thread to handle SIGHUP and SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

Figure 13.7 Daemon rereading configuration files

We call daemonize from Figure 13.1 to initialize the daemon. When it returns, we call already_running from Figure 13.6 to ensure that only one copy of the daemon is running. At this point, SIGHUP is still ignored, so we need to reset the disposition to the default behavior; otherwise, the thread calling sigwait may never see the signal.

We block all signals, as is recommended for multithreaded programs, and create a thread to handle signals. The thread's only job is to wait for SIGHUP and SIGTERM. When it receives SIGHUP, the thread calls reread to reread its configuration file. When it receives SIGTERM, the thread logs a message and exits.

Recall from Figure 10.1 that the default action for SIGHUP and SIGTERM is to terminate the process. Because we block these signals, the daemon will not die when one of them is sent to the process. Instead, the thread calling sigwait will return with an indication that the signal has been received. □

Example

As noted in Section 12.8, Linux threads behave differently with respect to signals. Because of this, identifying the proper process to signal in Figure 13.7 will be difficult. In addition, we aren't guaranteed that the daemon will react as we expect, because of the implementation differences.

The program in Figure 13.8 shows how a daemon can catch SIGHUP and reread its configuration file without using multiple threads.

```
#include "apue.h"
#include <syslog.h>
#include <errno.h>

extern int lockfile(int);
extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void
sigterm(int signo)
{
    syslog(LOG_INFO, "got SIGTERM; exiting");
    exit(0);
}

void
sighup(int signo)
{
    syslog(LOG_INFO, "Re-reading configuration file");
    reread();
}
```

```
int
main(int argc, char *argv[])
{
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Handle signals of interest.
     */
    sa.sa_handler = sigterm;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGHUP);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) < 0) {
        syslog(LOG_ERR, "can't catch SIGTERM: %s", strerror(errno));
        exit(1);
    }
    sa.sa_handler = sighup;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0) {
        syslog(LOG_ERR, "can't catch SIGHUP: %s", strerror(errno));
        exit(1);
    }

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

Figure 13.8 Alternate implementation of daemon rereading configuration files

After initializing the daemon, we install signal handlers for SIGHUP and SIGTERM. We can either place the reread logic in the signal handler or just set a flag in the handler and have the main thread of the daemon do all the work instead. □

13.7 Client–Server Model

A common use for a daemon process is as a server process. Indeed, in Figure 13.2, we can call the syslogd process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.

In general, a *server* is a process that waits for a *client* to contact it, requesting some type of service. In Figure 13.2, the service being provided by the syslogd server is the logging of an error message.

In Figure 13.2, the communication between the client and the server is one-way. The client sends its service request to the server; the server sends nothing back to the client. In the upcoming chapters, we'll see numerous examples of two-way communication between a client and a server. The client sends a request to the server, and the server sends a reply back to the client.

13.8 Summary

Daemon processes are running all the time on most UNIX systems. Initializing our own process to run as a daemon takes some care and an understanding of the process relationships that we described in Chapter 9. In this chapter, we developed a function that can be called by a daemon process to initialize itself correctly.

We also discussed the ways a daemon can log error messages, since a daemon normally doesn't have a controlling terminal. We discussed several conventions that daemons follow on most UNIX systems and showed examples of how to implement some of these conventions.

Exercises

13.1 As we might guess from Figure 13.2, when the syslog facility is initialized, either by calling openlog directly or on the first call to syslog, the special device file for the UNIX domain datagram socket, /dev/log, has to be opened. What happens if the user process (the daemon) calls chroot before calling openlog?

13.2 List all the daemons active on your system, and identify the function of each one.

13.3 Write a program that calls the daemonize function in Figure 13.1. After calling this function, call getlogin (Section 8.15) to see whether the process has a login name now that it has become a daemon. Print the results to a file.

14

Advanced I/O

14.1 Introduction

This chapter covers numerous topics and functions that we lump under the term *advanced I/O*: nonblocking I/O, record locking, System V STREAMS, I/O multiplexing (the `select` and `poll` functions), the `readv` and `writev` functions, and memory-mapped I/O (`mmap`). We need to cover these topics before describing interprocess communication in Chapter 15, Chapter 17, and many of the examples in later chapters.

14.2 Nonblocking I/O

In Section 10.5, we said that the system calls are divided into two categories: the "slow" ones and all the others. The slow system calls are those that can block forever. They include

- Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
- Writes that can block the caller forever if the data can't be accepted immediately by these same file types (no room in the pipe, network flow control, etc.)
- Opens that block until some condition occurs on certain file types (such as an open of a terminal device that waits until an attached modem answers the phone, or an open of a FIFO for writing-only when no other process has the FIFO open for reading)
- Reads and writes of files that have mandatory record locking enabled

- Certain ioctl operations
- Some of the interprocess communication functions (Chapter 15)

We also said that system calls related to disk I/O are not considered slow, even though the read or write of a disk file can block the caller temporarily.

Nonblocking I/O lets us issue an I/O operation, such as an open, read, or write, and not have it block forever. If the operation cannot be completed, the call returns immediately with an error noting that the operation would have blocked.

There are two ways to specify nonblocking I/O for a given descriptor.

1. If we call open to get the descriptor, we can specify the O_NONBLOCK flag (Section 3.3).
2. For a descriptor that is already open, we call fcntl to turn on the O_NONBLOCK file status flag (Section 3.14). Figure 3.11 shows a function that we can call to turn on any of the file status flags for a descriptor.

Earlier versions of System V used the flag O_NDELAY to specify nonblocking mode. These versions of System V returned a value of 0 from the read function if there wasn't any data to be read. Since this use of a return value of 0 overlapped with the normal UNIX System convention of 0 meaning the end of file, POSIX.1 chose to provide a nonblocking flag with a different name and different semantics. Indeed, with these older versions of System V, when we get a return of 0 from read, we don't know whether the call would have blocked or whether the end of file was encountered. We'll see that POSIX.1 requires that read return −1 with errno set to EAGAIN if there is no data to read from a nonblocking descriptor. Some platforms derived from System V support both the older O_NDELAY and the POSIX.1 O_NONBLOCK, but in this text, we'll use only the POSIX.1 feature. The older O_NDELAY is for backward compatibility and should not be used in new applications.

4.3BSD provided the FNDELAY flag for fcntl, and its semantics were slightly different. Instead of affecting only the file status flags for the descriptor, the flags for either the terminal device or the socket were also changed to be nonblocking, affecting all users of the terminal or socket, not only the users sharing the same file table entry (4.3BSD nonblocking I/O worked only on terminals and sockets). Also, 4.3BSD returned EWOULDBLOCK if an operation on a nonblocking descriptor could not complete without blocking. Today, BSD-based systems provide the POSIX.1 O_NONBLOCK flag and define EWOULDBLOCK to be the same as EAGAIN. These systems provide nonblocking semantics consistent with other POSIX-compatible systems: changes in file status flags affect all users of the same file table entry, but are independent of accesses to the same device through other file table entries. (Refer to Figures 3.6 and 3.8.)

Example

Let's look at an example of nonblocking I/O. The program in Figure 14.1 reads up to 500,000 bytes from the standard input and attempts to write it to the standard output. The standard output is first set nonblocking. The output is in a loop, with the results of each write being printed on the standard error. The function clr_fl is similar to the function set_fl that we showed in Figure 3.11. This new function simply clears one or more of the flag bits.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>

char    buf[500000];

int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */

    exit(0);
}
```

Figure 14.1 Large nonblocking `write`

If the standard output is a regular file, we expect the `write` to be executed once:

```
$ ls -l /etc/termcap                                print file size
-rw-r--r--  1 root     702559 Feb 23  2002 /etc/termcap
$ ./a.out < /etc/termcap > temp.file                try a regular file first
read 500000 bytes
nwrite = 500000, errno = 0                          a single write
$ ls -l temp.file                                   verify size of output file
-rw-rw-r--  1 sar      500000 Jul  8 04:19 temp.file
```

But if the standard output is a terminal, we expect the `write` to return a partial count sometimes and an error at other times. This is what we see:

```
$ ./a.out < /etc/termcap 2>stderr.out          output to terminal
                                               lots of output to terminal ...
$ cat stderr.out
read 500000 bytes
nwrite = 216041, errno = 0
nwrite = -1, errno = 11                        1,497 of these errors
. . .
nwrite = 16015, errno = 0
nwrite = -1, errno = 11                        1,856 of these errors
. . .
nwrite = 32081, errno = 0
nwrite = -1, errno = 11                        1,654 of these errors
. . .
nwrite = 48002, errno = 0
nwrite = -1, errno = 11                        1,460 of these errors
. . .
                                               and so on ...
nwrite = 7949, errno = 0
```

On this system, the errno of 11 is EAGAIN. The amount of data accepted by the terminal driver varies from system to system. The results will also vary depending on how you are logged in to the system: on the system console, on a hardwired terminal, on network connection using a pseudo terminal. If you are running a windowing system on your terminal, you are also going through a pseudo-terminal device. □

In this example, the program issues thousands of write calls, even though only between 10 and 20 are needed to output the data. The rest just return an error. This type of loop, called *polling*, is a waste of CPU time on a multiuser system. In Section 14.5, we'll see that I/O multiplexing with a nonblocking descriptor is a more efficient way to do this.

Sometimes, we can avoid using nonblocking I/O by designing our applications to use multiple threads (see Chapter 11). We can allow individual threads to block in I/O calls if we can continue to make progress in other threads. This can sometimes simplify the design, as we shall see in Chapter 21; sometimes, however, the overhead of synchronization can add more complexity than is saved from using threads.

14.3 Record Locking

What happens when two people edit the same file at the same time? In most UNIX systems, the final state of the file corresponds to the last process that wrote the file. In some applications, however, such as a database system, a process needs to be certain that it alone is writing to a file. To provide this capability for processes that need it, commercial UNIX systems provide record locking. (In Chapter 20, we develop a database library that uses record locking.)

Record locking is the term normally used to describe the ability of a process to prevent other processes from modifying a region of a file while the first process is reading or modifying that portion of the file. Under the UNIX System, the adjective "record" is a misnomer, since the UNIX kernel does not have a notion of records in a file. A better term is *byte-range locking*, since it is a range of a file (possibly the entire file) that is locked.

History

One of the criticisms of early UNIX systems was that they couldn't be used to run database systems, because there was no support for locking portions of files. As UNIX systems found their way into business computing environments, various groups added support record locking (differently, of course).

Early Berkeley releases supported only the `flock` function. This function locks only entire files, not regions of a file.

Record locking was added to System V Release 3 through the `fcntl` function. The `lockf` function was built on top of this, providing a simplified interface. These functions allowed callers to lock arbitrary byte ranges in a file, from the entire file down to a single byte within the file.

POSIX.1 chose to standardize on the `fcntl` approach. Figure 14.2 shows the forms of record locking provided by various systems. Note that the Single UNIX Specification includes `lockf` in the XSI extension.

System	Advisory	Mandatory	`fcntl`	`lockf`	`flock`
SUS	•		•	XSI	
FreeBSD 5.2.1	•		•	•	•
Linux 2.4.22	•	•	•	•	•
Mac OS X 10.3	•		•	•	•
Solaris 9	•	•	•	•	•

Figure 14.2 Forms of record locking supported by various UNIX systems

We describe the difference between advisory locking and mandatory locking later in this section. In this text, we describe only the POSIX.1 `fcntl` locking.

Record locking was originally added to Version 7 in 1980 by John Bass. The system call entry into the kernel was a function named `locking`. This function provided mandatory record locking and propagated through many versions of System III. Xenix systems picked up this function, and some Intel-based System V derivatives, such as OpenServer 5, still support it in a Xenix-compatibility library.

`fcntl` Record Locking

Let's repeat the prototype for the `fcntl` function from Section 3.14.

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );

                  Returns: depends on cmd if OK (see following), -1 on error
```

For record locking, *cmd* is F_GETLK, F_SETLK, or F_SETLKW. The third argument (which we'll call *flockptr*) is a pointer to an flock structure.

```
struct flock {
  short  l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
  off_t  l_start;  /* offset in bytes, relative to l_whence */
  short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_len;    /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;    /* returned with F_GETLK */
};
```

This structure describes

- The type of lock desired: F_RDLCK (a shared read lock), F_WRLCK (an exclusive write lock), or F_UNLCK (unlocking a region)
- The starting byte offset of the region being locked or unlocked (l_start and l_whence)
- The size of the region in bytes (l_len)
- The ID (l_pid) of the process holding the lock that can block the current process (returned by F_GETLK only)

There are numerous rules about the specification of the region to be locked or unlocked.

- The two elements that specify the starting offset of the region are similar to the last two arguments of the lseek function (Section 3.6). Indeed, the l_whence member is specified as SEEK_SET, SEEK_CUR, or SEEK_END.
- Locks can start and extend beyond the current end of file, but cannot start or extend before the beginning of the file.
- If l_len is 0, it means that the lock extends to the largest possible offset of the file. This allows us to lock a region starting anywhere in the file, up through and including any data that is appended to the file. (We don't have to try to guess how many bytes might be appended to the file.)
- To lock the entire file, we set l_start and l_whence to point to the beginning of the file and specify a length (l_len) of 0. (There are several ways to specify the beginning of the file, but most applications specify l_start as 0 and l_whence as SEEK_SET.)

We mentioned two types of locks: a shared read lock (l_type of F_RDLCK) and an exclusive write lock (F_WRLCK). The basic rule is that any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte. Furthermore, if there are one or more read locks on a byte, there can't be any write locks on that byte; if there is an exclusive write lock on a byte, there can't be any read locks on that byte. We show this compatibility rule in Figure 14.3.

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

Figure 14.3 Compatibility between different lock types

The compatibility rule applies to lock requests made from different processes, not to multiple lock requests made by a single process. If a process has an existing lock on a range of a file, a subsequent attempt to place a lock on the same range by the same process will replace the existing lock with the new one. Thus, if a process has a write lock on bytes 16–32 of a file and then tries to place a read lock on bytes 16–32, the request will succeed (assuming that we're not racing with any other processes trying to lock the same portion of the file), and the write lock will be replaced by a read lock.

To obtain a read lock, the descriptor must be open for reading; to obtain a write lock, the descriptor must be open for writing.

We can now describe the three commands for the `fcntl` function.

`F_GETLK` Determine whether the lock described by *flockptr* is blocked by some other lock. If a lock exists that would prevent ours from being created, the information on that existing lock overwrites the information pointed to by *flockptr*. If no lock exists that would prevent ours from being created, the structure pointed to by *flockptr* is left unchanged except for the `l_type` member, which is set to `F_UNLCK`.

`F_SETLK` Set the lock described by *flockptr*. If we are trying to obtain a read lock (`l_type` of `F_RDLCK`) or a write lock (`l_type` of `F_WRLCK`) and the compatibility rule prevents the system from giving us the lock (Figure 14.3), `fcntl` returns immediately with `errno` set to either `EACCES` or `EAGAIN`.

Although POSIX allows an implementation to return either error code, all four implementations described in this text return `EAGAIN` if the locking request cannot be satisfied.

This command is also used to clear the lock described by *flockptr* (`l_type` of `F_UNLCK`).

`F_SETLKW` This command is a blocking version of `F_SETLK`. (The `W` in the command name means *wait*.) If the requested read lock or write lock cannot be granted because another process currently has some part of the requested region locked, the calling process is put to sleep. The process wakes up either when the lock becomes available or when interrupted by a signal.

Be aware that testing for a lock with `F_GETLK` and then trying to obtain that lock with `F_SETLK` or `F_SETLKW` is not an atomic operation. We have no guarantee that, between the two `fcntl` calls, some other process won't come in and obtain the same lock. If we don't want to block while waiting for a lock to become available to us, we must handle the possible error returns from `F_SETLK`.

Note that POSIX.1 doesn't specify what happens when one process read-locks a range of a file, a second process blocks while trying to get a write lock on the same range, and a third processes then attempts to get another read lock on the range. If the third process is allowed to place a read lock on the range just because the range is already read-locked, then the implementation might starve processes with pending write locks. This means that as additional requests to read lock the same range arrive, the time that the process with the pending write-lock request has to wait is extended. If the read-lock requests arrive quickly enough without a lull in the arrival rate, then the writer could wait for a long time.

When setting or releasing a lock on a file, the system combines or splits adjacent areas as required. For example, if we lock bytes 100 through 199 and then unlock byte 150, the kernel still maintains the locks on bytes 100 through 149 and bytes 151 through 199. Figure 14.4 illustrates the byte-range locks in this situation.

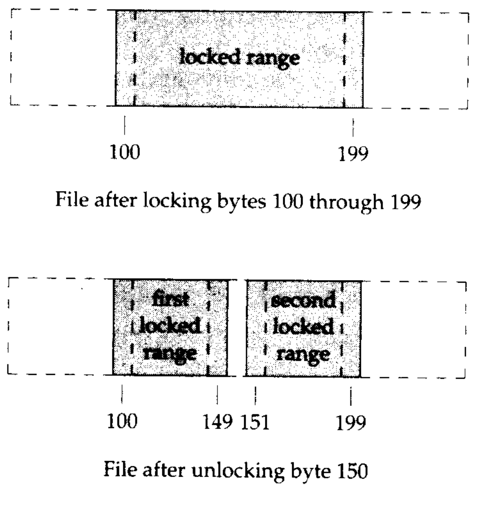
locked range
100
199
File after locking bytes 100 through 199
first locked range
second locked range
100
149
151
199
File after unlocking byte 150

Figure 14.4 File byte-range lock diagram

If we were to lock byte 150, the system would coalesce the adjacent locked regions into a single region from byte 100 through 199. The resulting picture would be the first diagram in Figure 14.4, the same as when we started.

Example—Requesting and Releasing a Lock

To save ourselves from having to allocate an `flock` structure and fill in all the elements each time, the function `lock_reg` in Figure 14.5 handles all these details.

```
#include "apue.h"
#include <fcntl.h>

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;      /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;   /* byte offset, relative to l_whence */
    lock.l_whence = whence;  /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;        /* #bytes (0 means to EOF) */

    return(fcntl(fd, cmd, &lock));
}
```

Figure 14.5 Function to lock or unlock a region of a file

Since most locking calls are to lock or unlock a region (the command `F_GETLK` is rarely used), we normally use one of the following five macros, which are defined in `apue.h` (Appendix B).

```
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

We have purposely defined the first three arguments to these macros in the same order as the `lseek` function. □

Example—Testing for a Lock

Figure 14.6 defines the function `lock_test` that we'll use to test for a lock.

```
#include "apue.h"
#include <fcntl.h>

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;
    lock.l_type = type;       /* F_RDLCK or F_WRLCK */
    lock.l_start = offset;    /* byte offset, relative to l_whence */
    lock.l_whence = whence;   /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;         /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);        /* false, region isn't locked by another proc */
    return(lock.l_pid);   /* true, return pid of lock owner */
}
```

Figure 14.6 Function to test for a locking condition

If a lock exists that would block the request specified by the arguments, this function returns the process ID of the process holding the lock. Otherwise, the function returns 0 (false). We normally call this function from the following two macros (defined in apue.h):

```
#define is_read_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)
```

Note that the lock_test function can't be used by a process to see whether it is currently holding a portion of a file locked. The definition of the F_GETLK command states that the information returned applies to an existing lock that would prevent us from creating our own lock. Since the F_SETLK and F_SETLKW commands always replace a process's existing lock if it exists, we can never block on our own lock; thus, the F_GETLK command will never report our own lock. □

Example—Deadlock

Deadlock occurs when two processes are each waiting for a resource that the other has locked. The potential for deadlock exists if a process that controls a locked region is put to sleep when it tries to lock another region that is controlled by a different process.

Figure 14.7 shows an example of deadlock. The child locks byte 0 and the parent locks byte 1. Then each tries to lock the other's already locked byte. We use the parent–child synchronization routines from Section 8.9 (TELL_xxx and WAIT_xxx) so that each process can wait for the other to obtain its lock. Running the program in Figure 14.7 gives us

```
$ ./a.out
parent: got the lock, byte 1
child: got the lock, byte 0
child: writew_lock error: Resource deadlock avoided
parent: got the lock, byte 0
```

```
#include "apue.h"
#include <fcntl.h>

static void
lockabyte(const char *name, int fd, off_t offset)
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);
    printf("%s: got the lock, byte %ld\n", name, offset);
}

int
main(void)
{
    int     fd;
    pid_t   pid;

    /*
     * Create a file and write two bytes to it.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");

    TELL_WAIT();
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);
    } else {                        /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
```

Figure 14.7 Example of deadlock detection

When a deadlock is detected, the kernel has to choose one process to receive the error return. In this example, the child was chosen, but this is an implementation detail. On some systems, the child always receives the error. On other systems, the parent always gets the error. On some systems, you might even see the errors split between the child and the parent as multiple lock attempts are made. □

Implied Inheritance and Release of Locks

Three rules govern the automatic inheritance and release of record locks.

1. Locks are associated with a process and a file. This has two implications. The first is obvious: when a process terminates, all its locks are released. The second is far from obvious: whenever a descriptor is closed, any locks on the file referenced by that descriptor for that process are released. This means that if we do

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = dup(fd1);
   close(fd2);
   ```

   after the `close(fd2)`, the lock that was obtained on `fd1` is released. The same thing would happen if we replaced the dup with open, as in

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = open(pathname, ...)
   close(fd2);
   ```

   to open the same file on another descriptor.

2. Locks are never inherited by the child across a `fork`. This means that if a process obtains a lock and then calls `fork`, the child is considered another process with regard to the lock that was obtained by the parent. The child has to call `fcntl` to obtain its own locks on any descriptors that were inherited across the `fork`. This makes sense because locks are meant to prevent multiple processes from writing to the same file at the same time. If the child inherited locks across a `fork`, both the parent and the child could write to the same file at the same time.

3. Locks are inherited by a new program across an `exec`. Note, however, that if the close-on-exec flag is set for a file descriptor, all locks for the underlying file are released when the descriptor is closed as part of an `exec`.

FreeBSD Implementation

Let's take a brief look at the data structures used in the FreeBSD implementation. This should help clarify rule 1, that locks are associated with a process and a file.

Consider a process that executes the following statements (ignoring error returns):

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);      /* parent write locks byte 0 */
if ((pid = fork()) > 0) {             /* parent */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
    read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
}
pause();
```

Figure 14.8 shows the resulting data structures after both the parent and the child have paused.

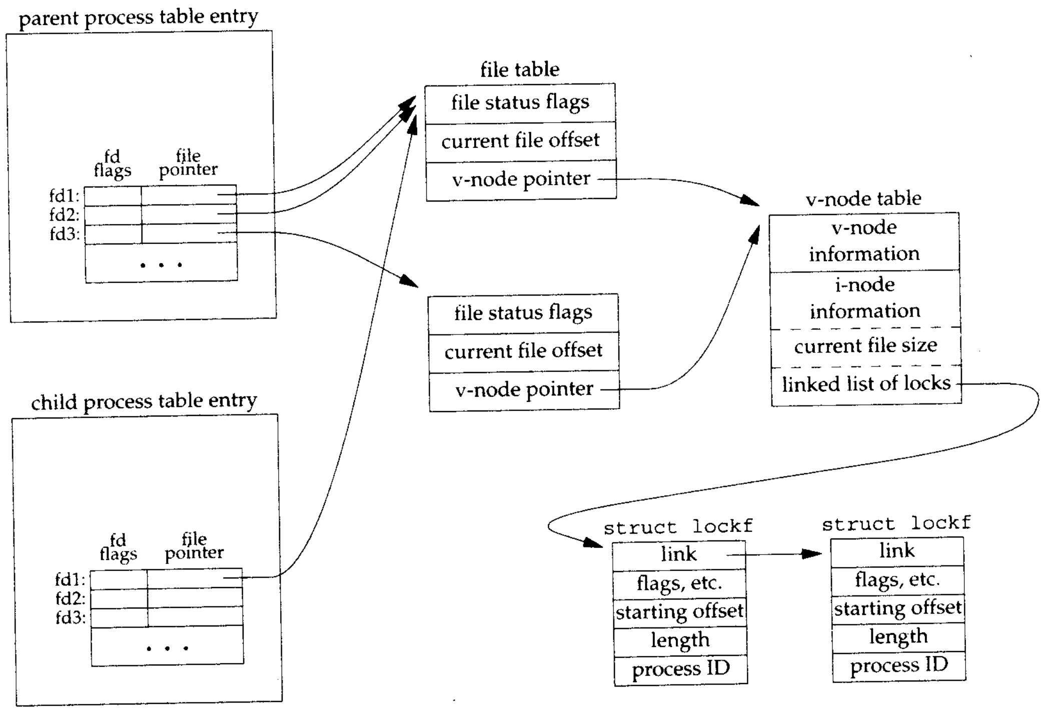

Figure 14.8 The FreeBSD data structures for record locking

We've shown the data structures that result from the `open`, `fork`, and `dup` earlier (Figures 3.8 and 8.2). What is new are the `lockf` structures that are linked together from the i-node structure. Note that each `lockf` structure describes one locked region (defined by an offset and length) for a given process. We show two of these structures: one for the parent's call to `write_lock` and one for the child's call to `read_lock`. Each structure contains the corresponding process ID.

In the parent, closing any one of `fd1`, `fd2`, or `fd3` causes the parent's lock to be released. When any one of these three file descriptors is closed, the kernel goes through the linked list of locks for the corresponding i-node and releases the locks held by the calling process. The kernel can't tell (and doesn't care) which descriptor of the three was used by the parent to obtain the lock.

Example

In the program in Figure 13.6, we saw how a daemon can use a lock on a file to ensure that only one copy of the daemon is running. Figure 14.9 shows the implementation of the `lockfile` function used by the daemon to place a write lock on a file.

```
#include <unistd.h>
#include <fcntl.h>

int
lockfile(int fd)
{
    struct flock fl;

    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return(fcntl(fd, F_SETLK, &fl));
}
```

Figure 14.9 Place a write lock on an entire file

Alternatively, we could define the `lockfile` function in terms of the `write_lock` function:

```
#define lockfile(fd) write_lock((fd), 0, SEEK_SET, 0)
```

□

Locks at End of File

Use caution when locking or unlocking relative to the end of file. Most implementations convert an `l_whence` value of `SEEK_CUR` or `SEEK_END` into an absolute file offset, using `l_start` and the file's current position or current length. Often, however, we need to specify a lock relative to the file's current position or current length, because we can't call `lseek` to obtain the current file offset, since we don't have a lock on the file. (There's a chance that another process could change the file's length between the call to `lseek` and the lock call.)

Consider the following sequence of steps:

```
writew_lock(fd, 0, SEEK_END, 0);
write(fd, buf, 1);
un_lock(fd, 0, SEEK_END);
write(fd, buf, 1);
```

This sequence of code might not do what you expect. It obtains a write lock from the current end of the file onward, covering any future data we might append to the file. Assuming that we are at end of file when we perform the first `write`, that will extend the file by one byte, and that byte will be locked. The unlock that follows has the effect of removing the locks for future writes that append data to the file, but it leaves a lock on the last byte in the file. When the second write occurs, the end of file is extended by one byte, but this byte is not locked. The state of the file locks for this sequence of steps is shown in Figure 14.10

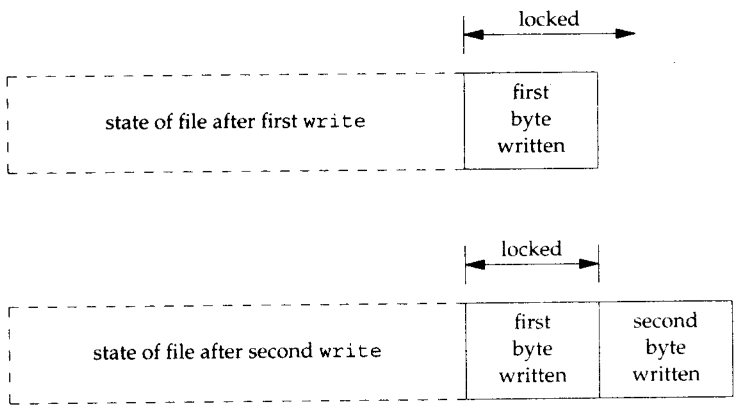

Figure 14.10 File range lock diagram

When a portion of a file is locked, the kernel converts the offset specified into an absolute file offset. In addition to specifying an absolute file offset (SEEK_SET), fcntl allows us to specify this offset relative to a point in the file: current (SEEK_CUR) or end of file (SEEK_END). The kernel needs to remember the locks independent of the current file offset or end of file, because the current offset and end of file can change, and changes to these attributes shouldn't affect the state of existing locks.

If we intended to remove the lock covering the byte we wrote in the first write, we could have specified the length as −1. Negative-length values represent the bytes before the specified offset.

Advisory versus Mandatory Locking

Consider a library of database access routines. If all the functions in the library handle record locking in a consistent way, then we say that any set of processes using these functions to access a database are *cooperating processes*. It is feasible for these database access functions to use advisory locking if they are the only ones being used to access the database. But advisory locking doesn't prevent some other process that has write permission for the database file from writing whatever it wants to the database file. This rogue process would be an uncooperating process, since it's not using the accepted method (the library of database functions) to access the database.

Mandatory locking causes the kernel to check every open, read, and write to verify that the calling process isn't violating a lock on the file being accessed. Mandatory locking is sometimes called *enforcement-mode locking*.

We saw in Figure 14.2 that Linux 2.4.22 and Solaris 9 provide mandatory record locking, but FreeBSD 5.2.1 and Mac OS X 10.3 do not. Mandatory record locking is not part of the Single UNIX Specification. On Linux, if you want mandatory locking, you need to enable it on a per file system basis by using the -o mand option to the mount command.

Mandatory locking is enabled for a particular file by turning on the set-group-ID bit and turning off the group-execute bit. (Recall Figure 4.12.) Since the set-group-ID bit makes no sense when the group-execute bit is off, the designers of SVR3 chose this way to specify that the locking for a file is to be mandatory locking and not advisory locking.

What happens to a process that tries to `read` or `write` a file that has mandatory locking enabled and the specified part of the file is currently read-locked or write-locked by another process? The answer depends on the type of operation (`read` or `write`), the type of lock held by the other process (read lock or write lock), and whether the descriptor for the `read` or `write` is nonblocking. Figure 14.11 shows the eight possibilities.

Type of existing lock on region held by other process	Blocking descriptor, tries to		Nonblocking descriptor, tries to	
	`read`	`write`	`read`	`write`
read lock	OK	blocks	OK	`EAGAIN`
write lock	blocks	blocks	`EAGAIN`	`EAGAIN`

Figure 14.11 Effect of mandatory locking on `read`s and `write`s by other processes

In addition to the `read` and `write` functions in Figure 14.11, the `open` function can also be affected by mandatory record locks held by another process. Normally, `open` succeeds, even if the file being opened has outstanding mandatory record locks. The next `read` or `write` follows the rules listed in Figure 14.11. But if the file being opened has outstanding mandatory record locks (either read locks or write locks), and if the flags in the call to `open` specify either `O_TRUNC` or `O_CREAT`, then `open` returns an error of `EAGAIN` immediately, regardless of whether `O_NONBLOCK` is specified.

Only Solaris treats the `O_CREAT` flag as an error case. Linux allows the `O_CREAT` flag to be specified when opening a file with an outstanding mandatory lock. Generating the `open` error for `O_TRUNC` makes sense, because the file cannot be truncated if it is read-locked or write-locked by another process. Generating the error for `O_CREAT`, however, makes little sense; this flag says to create the file only if it doesn't already exist, but it has to exist to be record-locked by another process.

This handling of locking conflicts with `open` can lead to surprising results. While developing the exercises in this section, a test program was run that opened a file (whose mode specified mandatory locking), established a read lock on an entire file, and then went to sleep for a while. (Recall from Figure 14.11 that a read lock should prevent writing to the file by other processes.) During this sleep period, the following behavior was seen in other typical UNIX System programs.

- The same file could be edited with the `ed` editor, and the results written back to disk! The mandatory record locking had no effect at all. Using the system call trace feature provided by some versions of the UNIX System, it was seen that `ed` wrote the new contents to a temporary file, removed the original file, and then renamed the temporary file to be the original file. The mandatory record locking has no effect on the `unlink` function, which allowed this to happen.

Under Solaris, the system call trace of a process is obtained by the truss(1) command. FreeBSD and Mac OS X use the ktrace(1) and kdump(1) commands. Linux provides the strace(1) command for tracing the system calls made by a process.

- The vi editor was never able to edit the file. It could read the file's contents, but whenever we tried to write new data to the file, EAGAIN was returned. If we tried to append new data to the file, the write blocked. This behavior from vi is what we expect.
- Using the Korn shell's > and >> operators to overwrite or append to the file resulted in the error "cannot create."
- Using the same two operators with the Bourne shell resulted in an error for >, but the >> operator just blocked until the mandatory lock was removed, and then proceeded. (The difference in the handling of the append operator is because the Korn shell opens the file with O_CREAT and O_APPEND, and we mentioned earlier that specifying O_CREAT generates an error. The Bourne shell, however, doesn't specify O_CREAT if the file already exists, so the open succeeds but the next write blocks.)

Results will vary, depending on the version of the operating system you are using. The bottom line with this exercise is to be wary of mandatory record locking. As seen with the ed example, it can be circumvented.

Mandatory record locking can also be used by a malicious user to hold a read lock on a file that is publicly readable. This can prevent anyone from writing to the file. (Of course, the file has to have mandatory record locking enabled for this to occur, which may require the user be able to change the permission bits of the file.) Consider a database file that is world readable and has mandatory record locking enabled. If a malicious user were to hold a read lock on the entire file, the file could not be written to by other processes.

Example

The program in Figure 14.12 determines whether mandatory locking is supported by a system.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int             fd;
    pid_t           pid;
    char            buf[5];
    struct stat     statbuf;
    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
        err_sys("open error");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("write error");

    /* turn on set-group-ID and turn off group-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("fstat error");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("fchmod error");

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {    /* parent */
        /* write lock entire file */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("write_lock error");

        TELL_CHILD(pid);

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
    } else {                 /* child */
        WAIT_PARENT();       /* wait for parent to set lock */

        set_fl(fd, O_NONBLOCK);

        /* first let's see what error we get if region is locked */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1)    /* no wait */
            err_sys("child: read_lock succeeded");
        printf("read_lock of already-locked region returns %d\n",
          errno);

        /* now try to read the mandatory locked file */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("lseek error");
        if (read(fd, buf, 2) < 0)
            err_ret("read failed (mandatory locking works)");
        else
            printf("read OK (no mandatory locking), buf = %2.2s\n",
              buf);
    }
    exit(0);
}
```

Figure 14.12 Determine whether mandatory locking is supported

This program creates a file and enables mandatory locking for the file. The program then splits into parent and child, with the parent obtaining a write lock on the entire file. The child first sets its descriptor nonblocking and then attempts to obtain a read lock on the file, expecting to get an error. This lets us see whether the system returns EACCES or EAGAIN. Next, the child rewinds the file and tries to read from the file. If mandatory locking is provided, the read should return EACCES or EAGAIN (since the descriptor is nonblocking). Otherwise, the read returns the data that it read. Running this program under Solaris 9 (which supports mandatory locking) gives us

```
$ ./a.out temp.lock
read_lock of already-locked region returns 11
read failed (mandatory locking works): Resource temporarily unavailable
```

If we look at either the system's headers or the intro(2) manual page, we see that an errno of 11 corresponds to EAGAIN. Under FreeBSD 5.2.1, we get

```
$ ./a.out temp.lock
read_lock of already-locked region returns 35
read OK (no mandatory locking), buf = ab
```

Here, an errno of 35 corresponds to EAGAIN. Mandatory locking is not supported. □

Example

Let's return to the first question of this section: what happens when two people edit the same file at the same time? The normal UNIX System text editors do not use record locking, so the answer is still that the final result of the file corresponds to the last process that wrote the file.

Some versions of the vi editor use advisory record locking. Even if we were using one of these versions of vi, it still doesn't prevent users from running another editor that doesn't use advisory record locking.

If the system provides mandatory record locking, we could modify our favorite editor to use it (if we have the sources). Not having the source code to the editor, we might try the following. We write our own program that is a front end to vi. This program immediately calls fork, and the parent just waits for the child to complete. The child opens the file specified on the command line, enables mandatory locking, obtains a write lock on the entire file, and then executes vi. While vi is running, the file is write-locked, so other users can't modify it. When vi terminates, the parent's wait returns, and our front end terminates.

A small front-end program of this type can be written, but it doesn't work. The problem is that it is common for most editors to read their input file and then close it. A lock is released on a file whenever a descriptor that references that file is closed. This means that when the editor closes the file after reading its contents, the lock is gone. There is no way to prevent this in the front-end program. □

We'll use record locking in Chapter 20 in our database library to provide concurrent access to multiple processes. We'll also provide some timing measurements to see what effect record locking has on a process.

14.4 STREAMS

The STREAMS mechanism is provided by System V as a general way to interface communication drivers into the kernel. We need to discuss STREAMS to understand the terminal interface in System V, the use of the `poll` function for I/O multiplexing (Section 14.5.2), and the implementation of STREAMS-based pipes and named pipes (Sections 17.2 and 17.2.1).

Be careful not to confuse this usage of the word *stream* with our previous usage of it in the standard I/O library (Section 5.2). The streams mechanism was developed by Dennis Ritchie [Ritchie 1984] as a way of cleaning up the traditional character I/O system (c-lists) and to accommodate networking protocols. The streams mechanism was later added to SVR3, after enhancing it a bit and capitalizing the name. Complete support for STREAMS (i.e., a STREAMS-based terminal I/O system) was provided with SVR4. The SVR4 implementation is described in [AT&T 1990d]. Rago [1993] discusses both user-level STREAMS programming and kernel-level STREAMS programming.

STREAMS is an optional feature in the Single UNIX Specification (included as the XSI STREAMS Option Group). Of the four platforms discussed in this text, only Solaris provides native support for STREAMS. A STREAMS subsystem is available for Linux, but you need to add it yourself. It is not usually included by default.

A stream provides a full-duplex path between a user process and a device driver. There is no need for a stream to talk to a hardware device; a stream can also be used with a pseudo-device driver. Figure 14.13 shows the basic picture for what is called a simple stream.

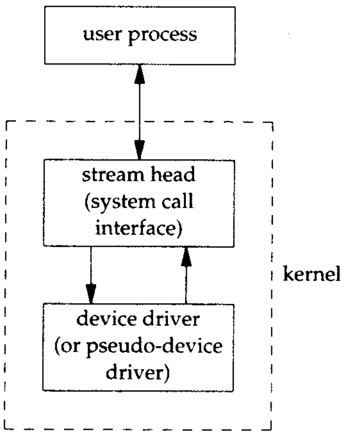

Figure 14.13 A simple stream

Beneath the stream head, we can push processing modules onto the stream. This is done using an `ioctl` command. Figure 14.14 shows a stream with a single processing module. We also show the connection between these boxes with two arrows to stress the full-duplex nature of streams and to emphasize that the processing in one direction is separate from the processing in the other direction.

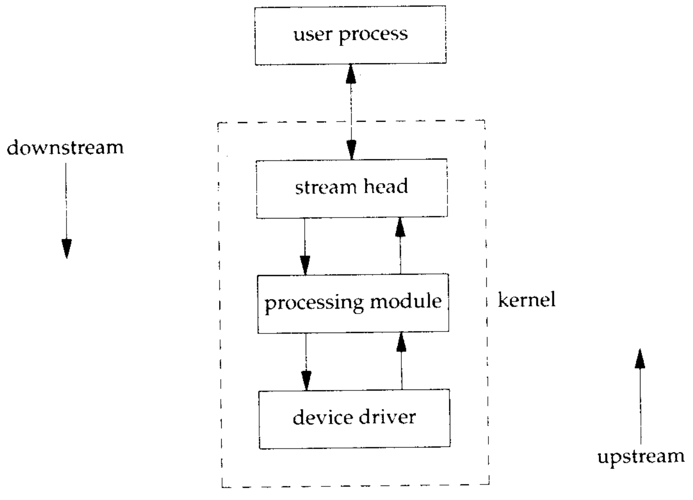

Figure 14.14 A stream with a processing module

Any number of processing modules can be pushed onto a stream. We use the term *push*, because each new module goes beneath the stream head, pushing any previously pushed modules down. (This is similar to a last-in, first-out stack.) In Figure 14.14, we have labeled the downstream and upstream sides of the stream. Data that we write to a stream head is sent downstream. Data read by the device driver is sent upstream.

STREAMS modules are similar to device drivers in that they execute as part of the kernel, and they are normally link edited into the kernel when the kernel is built. If the system supports dynamically-loadable kernel modules (as do Linux and Solaris), then we can take a STREAMS module that has not been link edited into the kernel and try to push it onto a stream; however, there is no guarantee that arbitrary combinations of modules and drivers will work properly together.

We access a stream with the functions from Chapter 3: `open`, `close`, `read`, `write`, and `ioctl`. Additionally, three new functions were added to the SVR3 kernel to support STREAMS (`getmsg`, `putmsg`, and `poll`), and another two (`getpmsg` and `putpmsg`) were added with SVR4 to handle messages with different priority bands within a stream. We describe these five new functions later in this section.

The *pathname* that we open for a stream normally lives beneath the `/dev` directory. Simply looking at the device name using `ls -l`, we can't tell whether the device is a STREAMS device. All STREAMS devices are character special files.

Although some STREAMS documentation implies that we can write processing modules and push them willy-nilly onto a stream, the writing of these modules requires the same skills and care as writing a device driver. Generally, only specialized applications or functions push and pop STREAMS modules.

Before STREAMS, terminals were handled with the existing c-list mechanism. (Section 10.3.1 of Bach [1986] and Section 10.6 of McKusick et al. [1996] describe c-lists in SVR2 and 4.4BSD, respectively.) Adding other character-based devices to the kernel usually involved writing a device driver and putting everything into the driver. Access to the new device was typically through the raw device, meaning that every user read or write ended up directly in the device driver. The STREAMS mechanism cleans up this way of interacting, allowing the data to flow between the stream head and the driver in STREAMS messages and allowing any number of intermediate processing modules to operate on the data.

STREAMS Messages

All input and output under STREAMS is based on messages. The stream head and the user process exchange messages using read, write, ioctl, getmsg, getpmsg, putmsg, and putpmsg. Messages are also passed up and down a stream between the stream head, the processing modules, and the device driver.

Between the user process and the stream head, a message consists of a message type, optional control information, and optional data. We show in Figure 14.15 how the various message types are generated by the arguments to write, putmsg, and putpmsg. The control information and data are specified by strbuf structures:

```
struct strbuf
  int  maxlen;   /* size of buffer */
  int  len;      /* number of bytes currently in buffer */
  char *buf;     /* pointer to buffer */
};
```

When we send a message with putmsg or putpmsg, len specifies the number of bytes of data in the buffer. When we receive a message with getmsg or getpmsg, maxlen specifies the size of the buffer (so the kernel won't overflow the buffer), and len is set by the kernel to the amount of data stored in the buffer. We'll see that a zero-length message is OK and that a len of –1 can specify that there is no control or data.

Why do we need to pass both control information and data? Providing both allows us to implement service interfaces between a user process and a stream. Olander, McGrath, and Israel [1986] describe the original implementation of service interfaces in System V. Chapter 5 of AT&T [1990d] describes service interfaces in detail, along with a simple example. Probably the best-known service interface, described in Chapter 4 of Rago [1993], is the System V Transport Layer Interface (TLI), which provides an interface to the networking system.

Another example of control information is sending a connectionless network message (a datagram). To send the message, we need to specify the contents of the message (the data) and the destination address for the message (the control information). If we couldn't send control and data together, some ad hoc scheme would be required. For example, we could specify the address using an ioctl, followed by a write of the data. Another technique would be to require that the address occupy the first *N* bytes of the data that is written using write. Separating the control information from the data, and providing functions that handle both (putmsg and getmsg) is a cleaner way to handle this.

There are about 25 different types of messages, but only a few of these are used between the user process and the stream head. The rest are passed up and down a stream within the kernel. (These message types are of interest to people writing

STREAMS processing modules, but can safely be ignored by people writing user-level code.) We'll encounter only three of these message types with the functions we use (read, write, getmsg, getpmsg, putmsg, and putpmsg):

- M_DATA (user data for I/O)
- M_PROTO (protocol control information)
- M_PCPROTO (high-priority protocol control information)

Every message on a stream has a queueing priority:

- High-priority messages (highest priority)
- Priority band messages
- Ordinary messages (lowest priority)

Ordinary messages are simply priority band messages with a band of 0. Priority band messages have a band of 1–255, with a higher band specifying a higher priority. High-priority messages are special in that only one is queued by the stream head at a time. Additional high-priority messages are discarded when one is already on the stream head's read queue.

Each STREAMS module has two input queues. One receives messages from the module above (messages moving downstream from the stream head toward the driver), and one receives messages from the module below (messages moving upstream from the driver toward the stream head). The messages on an input queue are arranged by priority. We show in Figure 14.15 how the arguments to write, putmsg, and putpmsg cause these various priority messages to be generated.

There are other types of messages that we don't consider. For example, if the stream head receives an M_SIG message from below, it generates a signal. This is how a terminal line discipline module sends the terminal-generated signals to the foreground process group associated with a controlling terminal.

putmsg and putpmsg Functions

A STREAMS message (control information or data, or both) is written to a stream using either putmsg or putpmsg. The difference in these two functions is that the latter allows us to specify a priority band for the message.

```
#include <stropts.h>

int putmsg(int filedes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flag);

int putpmsg(int filedes, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flag);

                                    Both return: 0 if OK, –1 on error
```

We can also write to a stream, which is equivalent to a putmsg without any control information and with a *flag* of 0.

These two functions can generate the three different priorities of messages: ordinary, priority band, and high priority. Figure 14.15 details the combinations of the arguments to these two functions that generate the various types of messages.

Function	Control?	Data?	*band*	*flag*	Message type generated
`write`	N/A	yes	N/A	N/A	`M_DATA` (ordinary)
`putmsg`	no	no	N/A	0	no message sent, returns 0
`putmsg`	no	yes	N/A	0	`M_DATA` (ordinary)
`putmsg`	yes	yes or no	N/A	0	`M_PROTO` (ordinary)
`putmsg`	yes	yes or no	N/A	`RS_HIPRI`	`M_PCPROTO` (high-priority)
`putmsg`	no	yes or no	N/A	`RS_HIPRI`	error, `EINVAL`
`putpmsg`	yes or no	yes or no	0–255	0	error, `EINVAL`
`putpmsg`	no	no	0–255	`MSG_BAND`	no message sent, returns 0
`putpmsg`	no	yes	0	`MSG_BAND`	`M_DATA` (ordinary)
`putpmsg`	no	yes	1–255	`MSG_BAND`	`M_DATA` (priority band)
`putpmsg`	yes	yes or no	0	`MSG_BAND`	`M_PROTO` (ordinary)
`putpmsg`	yes	yes or no	1–255	`MSG_BAND`	`M_PROTO` (priority band)
`putpmsg`	yes	yes or no	0	`MSG_HIPRI`	`M_PCPROTO` (high-priority)
`putpmsg`	no	yes or no	0	`MSG_HIPRI`	error, `EINVAL`
`putpmsg`	yes or no	yes or no	nonzero	`MSG_HIPRI`	error, `EINVAL`

Figure 14.15 Type of STREAMS message generated for `write`, `putmsg`, and `putpmsg`

The notation "N/A" means *not applicable*. In this figure, a "no" for the control portion of the message corresponds to either a null *ctlptr* argument or *ctlptr->len* being −1. A "yes" for the control portion corresponds to *ctlptr* being non-null and *ctlptr->len* being greater than or equal to 0. The data portion of the message is handled equivalently (using *dataptr* instead of *ctlptr*).

STREAMS `ioctl` Operations

In Section 3.15, we said that the `ioctl` function is the catchall for anything that can't be done with the other I/O functions. The STREAMS system continues this tradition.

Between Linux and Solaris, there are almost 40 different operations that can be performed on a stream using `ioctl`. Most of these operations are documented in the `streamio`(7) manual page. The header `<stropts.h>` must be included in C code that uses any of these operations. The second argument for `ioctl`, *request*, specifies which of the operations to perform. All the *request*s begin with `I_`. The third argument depends on the *request*. Sometimes, the third argument is an integer value; sometimes, it's a pointer to an integer or a structure.

Example—`isastream` Function

We sometimes need to determine if a descriptor refers to a stream or not. This is similar to calling the `isatty` function to determine if a descriptor refers to a terminal device (Section 18.9). Linux and Solaris provide the `isastream` function.

```
#include <stropts.h>

int isastream(int filedes);

                         Returns: 1 (true) if STREAMS device, 0 (false) otherwise
```

Like `isatty`, this is usually a trivial function that merely tries an `ioctl` that is valid only on a STREAMS device. Figure 14.16 shows one possible implementation of this function. We use the `I_CANPUT` `ioctl` command, which checks if the band specified by the third argument (0 in the example) is writable. If the `ioctl` succeeds, the stream is not changed.

```
#include    <stropts.h>
#include    <unistd.h>

int
isastream(int fd)
{
    return(ioctl(fd, I_CANPUT, 0) != -1);
}
```

Figure 14.16 Check if descriptor is a STREAMS device

We can use the program in Figure 14.17 to test this function.

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int     i, fd;

    for (i = 1; i < argc; i++) {
        if ((fd = open(argv[i], O_RDONLY)) < 0) {
            err_ret("%s: can't open", argv[i]);
            continue;
        }

        if (isastream(fd) == 0)
            err_ret("%s: not a stream", argv[i]);
        else
            err_msg("%s: streams device", argv[i]);
    }

    exit(0);
}
```

Figure 14.17 Test the `isastream` function

Running this program on Solaris 9 shows the various errors returned by the `ioctl` function:

```
$ ./a.out /dev/tty /dev/fb /dev/null /etc/motd
/dev/tty: streams device
/dev/fb: not a stream: Invalid argument
/dev/null: not a stream: No such device or address
/etc/motd: not a stream: Inappropriate ioctl for device
```

Note that `/dev/tty` is a STREAMS device, as we expect under Solaris. The character special file `/dev/fb` is not a STREAMS device, but it supports other `ioctl` requests. These devices return EINVAL when the `ioctl` request is unknown. The character special file `/dev/null` does not support any `ioctl` operations, so the error ENODEV is returned. Finally, `/etc/motd` is a regular file, not a character special file, so the classic error ENOTTY is returned. We never receive the error we might expect: ENOSTR ("Device is not a stream").

> The message for ENOTTY used to be "Not a typewriter," a historical artifact because the UNIX kernel returns ENOTTY whenever an `ioctl` is attempted on a descriptor that doesn't refer to a character special device. This message has been updated on Solaris to "Inappropriate ioctl for device."

□

Example

If the `ioctl` *request* is `I_LIST`, the system returns the names of all the modules on the stream—the ones that have been pushed onto the stream, including the topmost driver. (We say topmost because in the case of a multiplexing driver, there may be more than one driver. Chapter 12 of Rago [1993] discusses multiplexing drivers in detail.) The third argument must be a pointer to a `str_list` structure:

```
struct str_list {
  int               sl_nmods;    /* number of entries in array */
  struct str_mlist  *sl_modlist; /* ptr to first element of array */
};
```

We have to set `sl_modlist` to point to the first element of an array of `str_mlist` structures and set `sl_nmods` to the number of entries in the array:

```
struct str_mlist {
  char  l_name[FMNAMESZ+1];  /* null terminated module name */
};
```

The constant FMNAMESZ is defined in the header `<sys/conf.h>` and is often 8. The extra byte in `l_name` is for the terminating null byte.

If the third argument to the `ioctl` is 0, the count of the number of modules is returned (as the value of `ioctl`) instead of the module names. We'll use this to determine the number of modules and then allocate the required number of `str_mlist` structures.

Figure 14.18 illustrates the `I_LIST` operation. Since the returned list of names doesn't differentiate between the modules and the driver, when we print the module names, we know that the final entry in the list is the driver at the bottom of the stream.

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>
#include <sys/conf.h>

int
main(int argc, char *argv[])
{
    int                 fd, i, nmods;
    struct str_list     list;

    if (argc != 2)
        err_quit("usage: %s <pathname>", argv[0]);

    if ((fd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if (isastream(fd) == 0)
        err_quit("%s is not a stream", argv[1]);

    /*
     * Fetch number of modules.
     */
    if ((nmods = ioctl(fd, I_LIST, (void *) 0)) < 0)
        err_sys("I_LIST error for nmods");
    printf("#modules = %d\n", nmods);

    /*
     * Allocate storage for all the module names.
     */
    list.sl_modlist = calloc(nmods, sizeof(struct str_mlist));
    if (list.sl_modlist == NULL)
        err_sys("calloc error");
    list.sl_nmods = nmods;

    /*
     * Fetch the module names.
     */
    if (ioctl(fd, I_LIST, &list) < 0)
        err_sys("I_LIST error for list");

    /*
     * Print the names.
     */
    for (i = 1; i <= nmods; i++)
        printf("  %s: %s\n", (i == nmods) ? "driver" : "module",
          list.sl_modlist++->l_name);

    exit(0);
}
```

Figure 14.18 List the names of the modules on a stream

If we run the program in Figure 14.18 from both a network login and a console login, to see which STREAMS modules are pushed onto the controlling terminal, we get the following:

```
$ who
sar         console       May  1 18:27
sar         pts/7         Jul 12 06:53
$ ./a.out /dev/console
#modules = 5
  module: redirmod
  module: ttcompat
  module: ldterm
  module: ptem
  driver: pts
$ ./a.out /dev/pts/7
#modules = 4
  module: ttcompat
  module: ldterm
  module: ptem
  driver: pts
```

The modules are the same in both cases, except that the console has an extra module on top that helps with virtual console redirection. On this computer, a windowing system was running on the console, so `/dev/console` actually refers to a pseudo terminal instead of to a hardwired device. We'll return to the pseudo terminal case in Chapter 19. □

`write` to STREAMS Devices

In Figure 14.15 we said that a `write` to a STREAMS device generates an `M_DATA` message. Although this is generally true, there are some additional details to consider. First, with a stream, the topmost processing module specifies the minimum and maximum packet sizes that can be sent downstream. (We are unable to query the module for these values.) If we `write` more than the maximum, the stream head normally breaks the data into packets of the maximum size, with one final packet that can be smaller than the maximum.

The next thing to consider is what happens if we `write` zero bytes to a stream. Unless the stream refers to a pipe or FIFO, a zero-length message is sent downstream. With a pipe or FIFO, the default is to ignore the zero-length `write`, for compatibility with previous versions. We can change this default for pipes and FIFOs using an `ioctl` to set the write mode for the stream.

Write Mode

Two `ioctl` commands fetch and set the write mode for a stream. Setting *request* to `I_GWROPT` requires that the third argument be a pointer to an integer, and the current write mode for the stream is returned in that integer. If *request* is `I_SWROPT`, the third argument is an integer whose value becomes the new write mode for the stream. As with the file descriptor flags and the file status flags (Section 3.14), we should always fetch the current write mode value and modify it rather than set the write mode to some absolute value (possibly turning off some other bits that were enabled).

Currently, only two write mode values are defined.

`SNDZERO`	A zero-length `write` to a pipe or FIFO will cause a zero-length message to be sent downstream. By default, this zero-length `write` sends no message.
`SNDPIPE`	Causes `SIGPIPE` to be sent to the calling process that calls either `write` or `putmsg` after an error has occurred on a stream.

A stream also has a read mode, and we'll look at it after describing the `getmsg` and `getpmsg` functions.

`getmsg` and `getpmsg` Functions

STREAMS messages are read from a stream head using `read`, `getmsg`, or `getpmsg`.

```
#include <stropts.h>

int getmsg(int filedes, struct strbuf *restrict ctlptr,
           struct strbuf *restrict dataptr, int *restrict flagptr);

int getpmsg(int filedes, struct strbuf *restrict ctlptr,
            struct strbuf *restrict dataptr, int *restrict bandptr,
            int *restrict flagptr);

                          Both return: non-negative value if OK, −1 on error
```

Note that *flagptr* and *bandptr* are pointers to integers. The integer pointed to by these two pointers must be set before the call to specify the type of message desired, and the integer is also set on return to the type of message that was read.

If the integer pointed to by *flagptr* is 0, `getmsg` returns the next message on the stream head's read queue. If the next message is a high-priority message, the integer pointed to by *flagptr* is set to `RS_HIPRI` on return. If we want to receive only high-priority messages, we must set the integer pointed to by *flagptr* to `RS_HIPRI` before calling `getmsg`.

A different set of constants is used by `getpmsg`. We can set the integer pointed to by *flagptr* to `MSG_HIPRI` to receive only high-priority messages. We can set the integer to `MSG_BAND` and then set the integer pointed to by *bandptr* to a nonzero priority value to receive only messages from that band, or higher (including high-priority messages). If we only want to receive the first available message, we can set the integer pointed to by *flagptr* to `MSG_ANY`; on return, the integer will be overwritten with either `MSG_HIPRI` or `MSG_BAND`, depending on the type of message received. If the message we retrieved was not a high-priority message, the integer pointed to by *bandptr* will contain the message's priority band.

If *ctlptr* is null or *ctlptr->maxlen* is −1, the control portion of the message will remain on the stream head's read queue, and we will not process it. Similarly, if *dataptr* is null or *dataptr->maxlen* is −1, the data portion of the message is not processed and remains on the stream head's read queue. Otherwise, we will retrieve as much control and data portions of the message as our buffers will hold, and any remainder will be left on the head of the queue for the next call.

If the call to `getmsg` or `getpmsg` retrieves a message, the return value is 0. If part of the control portion of the message is left on the stream head read queue, the constant `MORECTL` is returned. Similarly, if part of the data portion of the message is left on the queue, the constant `MOREDATA` is returned. If both control and data are left, the return value is (`MORECTL | MOREDATA`).

Read Mode

We also need to consider what happens if we `read` from a STREAMS device. There are two potential problems.

1. What happens to the record boundaries associated with the messages on a stream?
2. What happens if we call `read` and the next message on the stream has control information?

The default handling for condition 1 is called byte-stream mode. In this mode, a `read` takes data from the stream until the requested number of bytes has been read or until there is no more data. The message boundaries associated with the STREAMS messages are ignored in this mode. The default handling for condition 2 causes the `read` to return an error if there is a control message at the front of the queue. We can change either of these defaults.

Using `ioctl`, if we set *request* to `I_GRDOPT`, the third argument is a pointer to an integer, and the current read mode for the stream is returned in that integer. A *request* of `I_SRDOPT` takes the integer value of the third argument and sets the read mode to that value. The read mode is specified by one of the following three constants:

`RNORM`	Normal, byte-stream mode (the default), as described previously.
`RMSGN`	Message-nondiscard mode. A `read` takes data from a stream until the requested number of bytes have been read or until a message boundary is encountered. If the `read` uses a partial message, the rest of the data in the message is left on the stream for a subsequent `read`.
`RMSGD`	Message-discard mode. This is like the nondiscard mode, but if a partial message is used, the remainder of the message is discarded.

Three additional constants can be specified in the read mode to set the behavior of `read` when it encounters messages containing protocol control information on a stream:

`RPROTNORM`	Protocol-normal mode: `read` returns an error of `EBADMSG`. This is the default.
`RPROTDAT`	Protocol-data mode: `read` returns the control portion as data.
`RPROTDIS`	Protocol-discard mode: `read` discards the control information but returns any data in the message.

Only one of the message read modes and one of the protocol read modes can be set at a time. The default read mode is (RNORM | RPROTNORM).

Example

The program in Figure 14.19 is the same as the one in Figure 3.4, but recoded to use getmsg instead of read.

```
#include "apue.h"
#include <stropts.h>

#define BUFFSIZE    4096

int
main(void)
{
    int             n, flag;
    char            ctlbuf[BUFFSIZE], datbuf[BUFFSIZE];
    struct strbuf   ctl, dat;

    ctl.buf = ctlbuf;
    ctl.maxlen = BUFFSIZE;
    dat.buf = datbuf;
    dat.maxlen = BUFFSIZE;
    for ( ; ; ) {
        flag = 0;       /* return any message */
        if ((n = getmsg(STDIN_FILENO, &ctl, &dat, &flag)) < 0)
            err_sys("getmsg error");
        fprintf(stderr, "flag = %d, ctl.len = %d, dat.len = %d\n",
          flag, ctl.len, dat.len);
        if (dat.len == 0)
            exit(0);
        else if (dat.len > 0)
            if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
                err_sys("write error");
    }
}
```

Figure 14.19 Copy standard input to standard output using getmsg

If we run this program under Solaris, where both pipes and terminals are implemented using STREAMS, we get the following output:

```
$ echo hello, world | ./a.out             requires STREAMS-based pipes
flag = 0, ctl.len = -1, dat.len = 13
hello, world
flag = 0, ctl.len = 0, dat.len = 0        indicates a STREAMS hangup
$ ./a.out                                 requires STREAMS-based terminals
this is line 1
flag = 0, ctl.len = -1, dat.len = 15
```

```
this is line 1
and line 2
flag = 0, ctl.len = -1, dat.len = 11
and line 2
^D                                        type the terminal EOF character
flag = 0, ctl.len = -1, dat.len = 0       tty end of file is not the same as a hangup
$ ./a.out < /etc/motd
getmsg error: Not a stream device
```

When the pipe is closed (when `echo` terminates), it appears to the program in Figure 14.19 as a STREAMS hangup, with both the control length and the data length set to 0. (We discuss pipes in Section 15.2.) With a terminal, however, typing the end-of-file character causes only the data length to be returned as 0. This terminal end of file is not the same as a STREAMS hangup. As expected, when we redirect standard input to be a non-STREAMS device, `getmsg` returns an error. □

14.5 I/O Multiplexing

When we read from one descriptor and write to another, we can use blocking I/O in a loop, such as

```
        while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
```

We see this form of blocking I/O over and over again. What if we have to read from two descriptors? In this case, we can't do a blocking `read` on either descriptor, as data may appear on one descriptor while we're blocked in a `read` on the other. A different technique is required to handle this case.

Let's look at the structure of the `telnet(1)` command. In this program, we read from the terminal (standard input) and write to a network connection, and we read from the network connection and write to the terminal (standard output). At the other end of the network connection, the `telnetd` daemon reads what we typed and presents it to a shell as if we were logged in to the remote machine. The `telnetd` daemon sends any output generated by the commands we type back to us through the `telnet` command, to be displayed on our terminal. Figure 14.20 shows a picture of this.

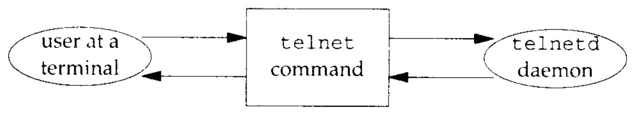

Figure 14.20 Overview of `telnet` program

The `telnet` process has two inputs and two outputs. We can't do a blocking `read` on either of the inputs, as we never know which input will have data for us.

One way to handle this particular problem is to divide the process in two pieces (using fork), with each half handling one direction of data. We show this in Figure 14.21. (The cu(1) command provided with System V's uucp communication package was structured like this.)

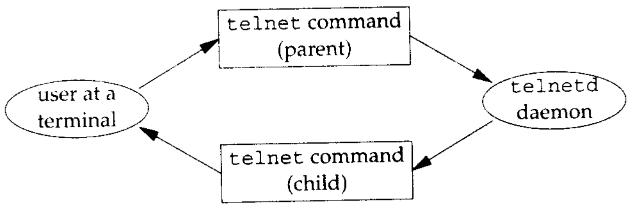

Figure 14.21 The telnet program using two processes

If we use two processes, we can let each process do a blocking read. But this leads to a problem when the operation terminates. If an end of file is received by the child (the network connection is disconnected by the telnetd daemon), then the child terminates, and the parent is notified by the SIGCHLD signal. But if the parent terminates (the user enters an end of file at the terminal), then the parent has to tell the child to stop. We can use a signal for this (SIGUSR1, for example), but it does complicate the program somewhat.

Instead of two processes, we could use two threads in a single process. This avoids the termination complexity, but requires that we deal with synchronization between the threads, which could add more complexity than it saves.

We could use nonblocking I/O in a single process by setting both descriptors nonblocking and issuing a read on the first descriptor. If data is present, we read it and process it. If there is no data to read, the call returns immediately. We then do the same thing with the second descriptor. After this, we wait for some amount of time (a few seconds, perhaps) and then try to read from the first descriptor again. This type of loop is called *polling*. The problem is that it wastes CPU time. Most of the time, there won't be data to read, so we waste time performing the read system calls. We also have to guess how long to wait each time around the loop. Although it works on any system that supports nonblocking I/O, polling should be avoided on a multitasking system.

Another technique is called *asynchronous I/O*. To do this, we tell the kernel to notify us with a signal when a descriptor is ready for I/O. There are two problems with this. First, not all systems support this feature (it is an optional facility in the Single UNIX Specification). System V provides the SIGPOLL signal for this technique, but this signal works only if the descriptor refers to a STREAMS device. BSD has a similar signal, SIGIO, but it has similar limitations: it works only on descriptors that refer to terminal devices or networks. The second problem with this technique is that there is only one of these signals per process (SIGPOLL or SIGIO). If we enable this signal for two descriptors (in the example we've been talking about, reading from two descriptors), the occurrence of the signal doesn't tell us which descriptor is ready. To determine which descriptor is ready, we still need to set each nonblocking and try them in sequence. We describe asynchronous I/O briefly in Section 14.6.

A better technique is to use *I/O multiplexing*. To do this, we build a list of the descriptors that we are interested in (usually more than one descriptor) and call a function that doesn't return until one of the descriptors is ready for I/O. On return from the function, we are told which descriptors are ready for I/O.

Three functions—`poll`, `pselect`, and `select`—allow us to perform I/O multiplexing. Figure 14.22 summarizes which platforms support them. Note that `select` is defined by the base POSIX.1 standard, but `poll` is an XSI extension to the base.

System	`poll`	`pselect`	`select`	`<sys/select.h>`
SUS	XSI	•	•	•
FreeBSD 5.2.1	•	•	•	
Linux 2.4.22	•	•	•	•
Mac OS X 10.3	•	•	•	
Solaris 9	•		•	•

Figure 14.22 I/O multiplexing supported by various UNIX systems

POSIX specifies that `<sys/select>` be included to pull the information for `select` into your program. Historically, however, we have had to include three other header files, and some of the implementations haven't yet caught up to the standard. Check the `select` manual page to see what your system supports. Older systems require that you include `<sys/types.h>`, `<sys/time.h>`, and `<unistd.h>`.

I/O multiplexing was provided with the `select` function in 4.2BSD. This function has always worked with any descriptor, although its main use has been for terminal I/O and network I/O. SVR3 added the `poll` function when the STREAMS mechanism was added. Initially, however, `poll` worked only with STREAMS devices. In SVR4, support was added to allow `poll` to work on any descriptor.

14.5.1 `select` and `pselect` Functions

The `select` function lets us do I/O multiplexing under all POSIX-compatible platforms. The arguments we pass to `select` tell the kernel

- Which descriptors we're interested in.
- What conditions we're interested in for each descriptor. (Do we want to read from a given descriptor? Do we want to write to a given descriptor? Are we interested in an exception condition for a given descriptor?)
- How long we want to wait. (We can wait forever, wait a fixed amount of time, or not wait at all.)

On the return from `select`, the kernel tells us

- The total count of the number of descriptors that are ready
- Which descriptors are ready for each of the three conditions (read, write, or exception condition)

With this return information, we can call the appropriate I/O function (usually `read` or `write`) and know that the function won't block.

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

int select(int maxfdp1, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict exceptfds,
           struct timeval *restrict tvptr);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

Let's look at the last argument first. This specifies how long we want to wait:

```
struct timeval {
  long tv_sec;    /* seconds */
  long tv_usec;   /* and microseconds */
};
```

There are three conditions.

tvptr == NULL

Wait forever. This infinite wait can be interrupted if we catch a signal. Return is made when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `select` returns –1 with `errno` set to `EINTR`.

tvptr–>tv_sec == 0 && *tvptr–>tv_usec* == 0

Don't wait at all. All the specified descriptors are tested, and return is made immediately. This is a way to poll the system to find out the status of multiple descriptors, without blocking in the `select` function.

tvptr–>tv_sec != 0 || *tvptr–>tv_usec* != 0

Wait the specified number of seconds and microseconds. Return is made when one of the specified descriptors is ready or when the timeout value expires. If the timeout expires before any of the descriptors is ready, the return value is 0. (If the system doesn't provide microsecond resolution, the *tvptr–>tv_usec* value is rounded up to the nearest supported value.) As with the first condition, this wait can also be interrupted by a caught signal.

POSIX.1 allows an implementation to modify the `timeval` structure, so after `select` returns, you can't rely on the structure containing the same values it did before calling `select`. FreeBSD 5.2.1, Mac OS X 10.3, and Solaris 9 all leave the structure unchanged, but Linux 2.4.22 will update it with the time remaining if `select` returns before the timeout value expires.

The middle three arguments—*readfds*, *writefds*, and *exceptfds*—are pointers to *descriptor sets*. These three sets specify which descriptors we're interested in and for which conditions (readable, writable, or an exception condition). A descriptor set is stored in an `fd_set` data type. This data type is chosen by the implementation so that it can hold one bit for each possible descriptor. We can consider it to be just a big array of bits, as shown in Figure 14.23.

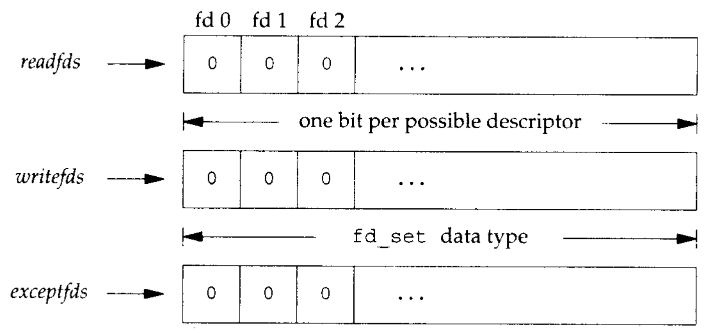

Figure 14.23 Specifying the read, write, and exception descriptors for `select`

The only thing we can do with the `fd_set` data type is allocate a variable of this type, assign a variable of this type to another variable of the same type, or use one of the following four functions on a variable of this type.

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

int FD_ISSET(int fd, fd_set *fdset);

                         Returns: nonzero if fd is in set, 0 otherwise

void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

These interfaces can be implemented as either macros or functions. An `fd_set` is set to all zero bits by calling `FD_ZERO`. To turn on a single bit in a set, we use `FD_SET`. We can clear a single bit by calling `FD_CLR`. Finally, we can test whether a given bit is turned on in the set with `FD_ISSET`.

After declaring a descriptor set, we must zero the set using `FD_ZERO`. We then set bits in the set for each descriptor that we're interested in, as in

```
fd_set   rset;
int      fd;

FD_ZERO(&rset);
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
```

On return from `select`, we can test whether a given bit in the set is still on using `FD_ISSET`:

```
if (FD_ISSET(fd, &rset)) {
    ...
}
```