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 behavior.

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 2.11. 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. If the implementation's limit is indeterminate, "no limit" is listed. This doesn't mean that the value is unlimited, however.

> 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 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
PTHREAD_DESTRUCTOR_ITERATIONS	4	4	4	no limit
PTHREAD_KEYS_MAX	256	1,024	512	no limit
PTHREAD_STACK_MIN	2,048	16,384	8,192	8,192
PTHREAD_THREADS_MAX	no limit	no limit	no limit	no limit

Figure 12.2 Examples of thread configuration limits

12.3 Thread Attributes

The pthread interface allows us to fine-tune the behavior of threads and synchronization objects by setting various attributes associated with each object. Generally, the functions for managing these attributes follow the same pattern:

1. Each object is associated with its own type of attribute object (threads with thread attributes, mutexes with mutex attributes, and so on). An attribute object can represent multiple attributes. The attribute object is opaque to applications. This means that applications aren't supposed to know anything about its internal structure, which promotes application portability. Instead, functions are provided to manage the attributes objects.

2. An initialization function exists to set the attributes to their default values.
3. Another function exists to destroy the attributes object. If the initialization function allocated any resources associated with the attributes object, the destroy function frees those resources.
4. Each attribute has a function to get the value of the attribute from the attribute object. Because the function returns 0 on success or an error number on failure, the value is returned to the caller by storing it in the memory location specified by one of the arguments.
5. Each attribute has a function to set the value of the attribute. In this case, the value is passed as an argument, *by value*.

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.

```
#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 code.

The thread attributes defined by POSIX.1 are summarized in Figure 12.3. POSIX.1 defines additional attributes in the Thread Execution Scheduling option, intended to support real-time applications, but we don't discuss them here. In Figure 12.3, we also show which platforms support each thread attribute.

Name	Description	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
detachstate	detached thread attribute	•	•	•	•
guardsize	guard buffer size in bytes at end of thread stack	•	•	•	•
stackaddr	lowest address of thread stack	•	•	•	•
stacksize	minimum 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 might already be running, asynchronous to the execution of this function. When we choose to ignore 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 had 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 supports the XSI option in the Single UNIX Specification. At compile time, you can check whether your system supports each thread stack attribute by using the _POSIX_THREAD_ATTR_STACKADDR and _POSIX_THREAD_ATTR_STACKSIZE symbols. If one of these symbols is defined, then the system supports the corresponding thread stack attribute. Alternatively, you can check for support at runtime, by using the _SC_THREAD_ATTR_STACKADDR and _SC_THREAD_ATTR_STACKSIZE parameters to the sysconf function.

We can manage the stack attributes using the pthread_attr_getstack and pthread_attr_setstack functions.

```
#include <pthread.h>

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

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

Both return: 0 if OK, error number on failure

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 14.8) to allocate space for an alternative 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. Of course, this assumes that the virtual address range used by malloc or mmap is different from the range currently in use for a thread's stack.

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 addresses to lower addresses for a given processor architecture, the *stackaddr* thread attribute will be the end of the stack instead of the beginning.

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. When setting the *stacksize* attribute, the size we choose can't be smaller than `PTHREAD_STACK_MIN`.

The *guardsize* thread attribute controls the size of the memory extent after the end of the thread's stack to protect against stack overflow. Its default value is implementation defined, but a commonly used value is the system page size. 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 intended for use by real-time applications. We will not discuss them here.

Threads have other attributes not represented by the `pthread_attr_t` structure: the cancelability state and the cancelability type. We discuss them in Section 12.7.

12.4 Synchronization Attributes

Just as threads have attributes, so too do their synchronization objects. In Section 11.6.7, we saw how spin locks have one attribute called the *process-shared* attribute. In this section, we discuss the attributes of mutexes, reader–writer locks, condition variables, and barriers.

12.4.1 Mutex Attributes

Mutex attributes are represented by a `pthread_mutexattr_t` structure. Whenever we initialized a mutex in Chapter 11, we accepted the default attributes by using the

PTHREAD_MUTEX_INITIALIZER constant or by calling the pthread_mutex_init function with a null pointer for the argument that points to the mutex attribute structure.

When dealing with nondefault 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. There are three attributes of interest: the *process-shared* attribute, the *robust* 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 it.

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. The pthread library can then

restrict the more expensive implementation to the case in which mutexes are shared among processes.

The *robust* mutex attribute is related to mutexes that are shared among multiple processes. It is meant to address the problem of mutex state recovery when a process terminates while holding a mutex. When this happens, the mutex is left in a locked state and recovery is difficult. Threads blocked on the lock in other processes will block indefinitely.

We can use the `pthread_mutexattr_getrobust` function to get the value of the *robust* mutex attribute. To set the value of the *robust* mutex attribute, we can call the `pthread_mutexattr_setrobust` function.

```
#include <pthread.h>

int pthread_mutexattr_getrobust(const pthread_mutexattr_t *
                                restrict attr,
                                int *restrict robust);

int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
                                int robust);
```

Both return: 0 if OK, error number on failure

There are two possible values for the *robust* attribute. The default is `PTHREAD_MUTEX_STALLED`, which means that no special action is taken when a process terminates while holding a mutex. In this case, use of the mutex can result in undefined behavior, and applications waiting for it to be unlocked are effectively "stalled." The other value is `PTHREAD_MUTEX_ROBUST`. This value will cause a thread blocked in a call to `pthread_mutex_lock` to acquire the lock when another process holding the lock terminates without first unlocking it, but the return value from `pthread_mutex_lock` is `EOWNERDEAD` instead of 0. Applications can use this special return value as an indication that they need to recover whatever state the mutex was protecting, if possible (the details of what state is being protected and how it can be recovered will vary among applications). Note that the `EOWNERDEAD` error return isn't really an error in this case, because the caller will own the lock.

Using robust mutexes changes the way we use `pthread_mutex_lock`, because we now have to check for three return values instead of two: success with no recovery needed, success but recovery needed, and failure. However, if we don't use robust mutexes, then we can continue to check only for success and failure.

> Of the four platforms covered in this text, only Linux 3.2.0 currently supports robust pthread mutexes. Solaris 10 supports robust mutexes only in its Solaris threads library (see the `mutex_init`(3C) Solaris manual page for more information). However, in Solaris 11, robust pthread mutexes are supported.

If the application state can't be recovered, the mutex will be in a permanently unusable state after the thread unlocks the mutex. To prevent this problem, the thread can call the `pthread_mutex_consistent` function to indicate that the state associated with the mutex is consistent before unlocking the mutex.

```
#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t * mutex);
```

Returns: 0 if OK, error number on failure

If a thread unlocks a mutex without first calling `pthread_mutex_consistent`, then other threads that are blocked while trying to acquire the mutex will see error returns of `ENOTRECOVERABLE`. If this happens, the mutex is no longer usable. By calling `pthread_mutex_consistent` beforehand, a thread allows the mutex to behave normally, so it can continue to be used.

The *type* mutex attribute controls the locking characteristics of the mutex. POSIX.1 defines four types:

`PTHREAD_MUTEX_NORMAL` A standard mutex type that doesn't do any special error checking or deadlock detection.

`PTHREAD_MUTEX_ERRORCHECK` A mutex type that provides error checking.

`PTHREAD_MUTEX_RECURSIVE` A mutex type that 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. Thus, if you lock a recursive mutex twice and then unlock it, the mutex remains locked until it is unlocked a second time.

`PTHREAD_MUTEX_DEFAULT` A mutex type providing default characteristics and behavior. Implementations are free to map it to one of the other mutex types. For example, Linux 3.2.0 maps this type to the normal mutex type, whereas FreeBSD 8.0 maps it to the error-checking type.

The behavior of the four types is summarized 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 the `pthread_mutexattr_gettype` function to get the mutex *type* attribute. To change the attribute, we can use the `pthread_mutexattr_settype` function.

```
#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 11.6.6 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 those applications can't be changed.

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 structure. 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

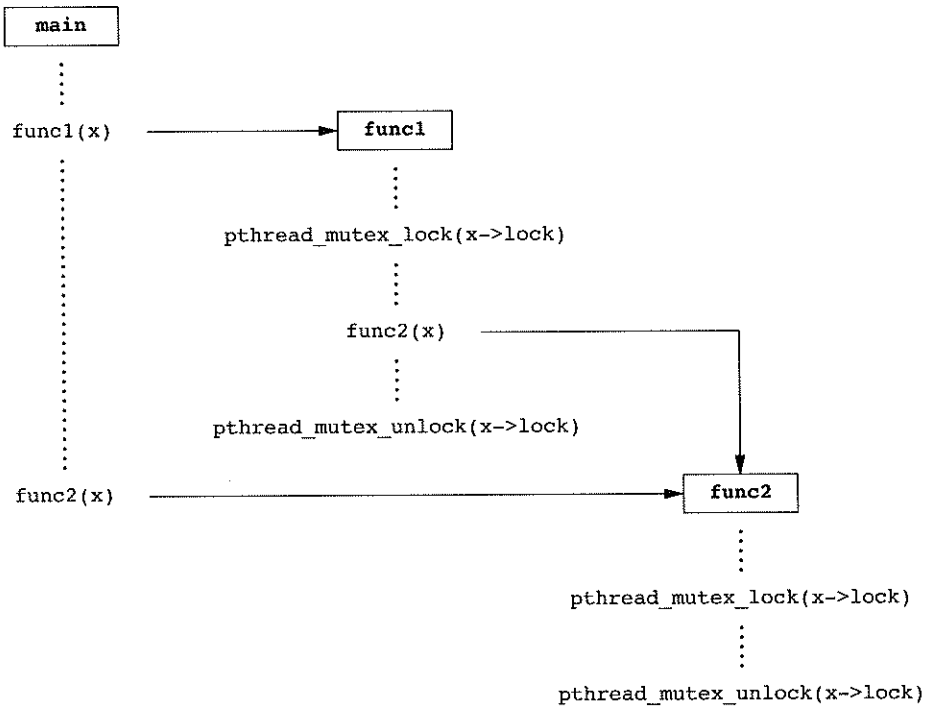

Figure 12.6 Recursive locking opportunity

the mutex before calling `func2` and reacquire it after `func2` returns, but this approach 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.

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 them 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. □

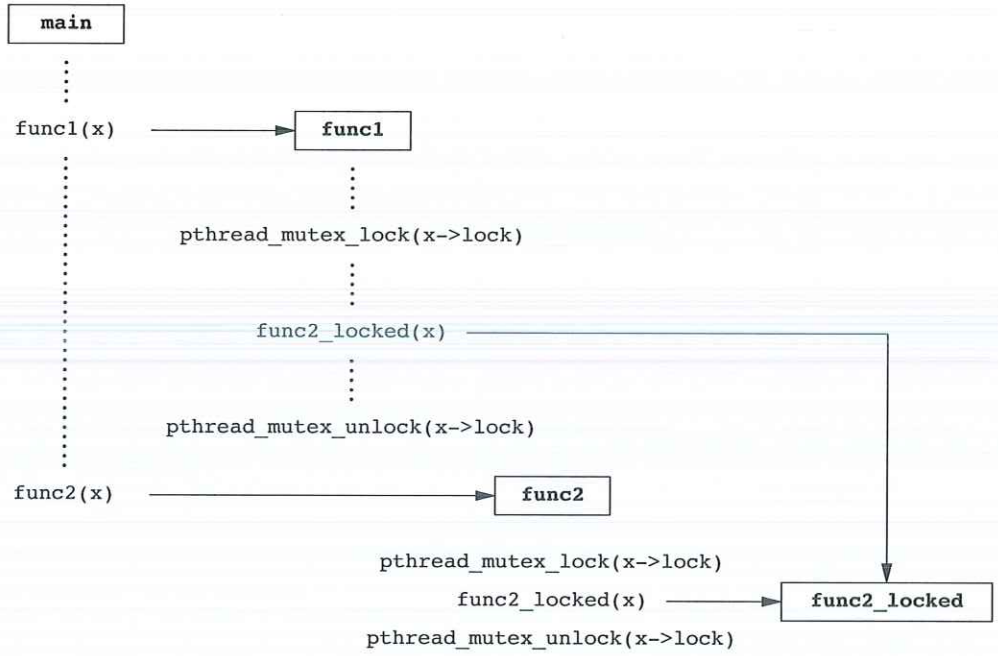

Figure 12.7 Avoiding a recursive locking opportunity

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 */

#if !defined(CLOCK_REALTIME) || defined(BSD)
#define clock_nanosleep(ID, FL, REQ, REM)   nanosleep((REQ), (REM))
#endif

#ifndef CLOCK_REALTIME
#define CLOCK_REALTIME 0
#define USECTONSEC 1000     /* microseconds to nanoseconds */

void
clock_gettime(int id, struct timespec *tsp)
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    tsp->tv_sec = tv.tv_sec;
    tsp->tv_nsec = tv.tv_usec * USECTONSEC;
}
#endif

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

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

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

    clock_gettime(CLOCK_REALTIME, &now);
    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;
            else
                free(tip);
        }
    }

    /*
     * 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");

    /* continue processing ... */

    pthread_mutex_lock(&mutex);

    /*
     * Check the condition under the protection of a lock to
     * make the check and the call to timeout atomic.
     */
    if (condition) {
        /*
```

```
         * Calculate the absolute time when we want to retry.
         */
        clock_gettime(CLOCK_REALTIME, &when);
        when.tv_sec += 10;  /* 10 seconds from now */
        timeout(&when, retry, (void *)((unsigned long)arg));
    }
    pthread_mutex_unlock(&mutex);

    /* continue processing ... */

    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. Because the `func` function argument passed to the `timeout` function will run in the future, 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` or `clock_nanosleep`, both of which allow us to sleep at higher resolution.

> On systems that don't define `CLOCK_REALTIME`, we define `clock_nanosleep` in terms of `nanosleep`. However, FreeBSD 8.0 defines this symbol to support `clock_gettime` and `clock_settime`, but doesn't support `clock_nanosleep` (only Linux 3.2.0 and Solaris 10 currently support `clock_nanosleep`.)
>
> Additionally, on systems that don't define `CLOCK_REALTIME`, we provide our own implementation of `clock_gettime` that calls `gettimeofday` and translates microseconds to nanoseconds.

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. □

12.4.2 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.

12.4.3 Condition Variable Attributes

The Single UNIX Specification currently defines two attributes for condition variables: the *process-shared* attribute and the *clock* attribute. As with the other attribute objects, a pair of functions initialize and deinitialize condition variable attribute objects.

```
#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

The *process-shared* attribute is the same as with the other synchronization attributes. It controls whether condition variables can be used by threads within a single process only or from within multiple processes. To find the current value of the *process-shared* attribute, we use the `pthread_condattr_getpshared` function. To set its value, we use the `pthread_condattr_setpshared` function.

```
#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

The *clock* attribute controls which clock is used when evaluating the timeout argument (*tsptr*) of the `pthread_cond_timedwait` function. The legal values are the

clock IDs listed in Figure 6.8. We can use the pthread_condattr_getclock function to retrieve the clock ID that will be used by the pthread_cond_timedwait function for the condition variable that was initialized with the pthread_condattr_t object. We can change the clock ID with the pthread_condattr_setclock function.

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *
                              restrict attr,
                              clockid_t *restrict clock_id);

int pthread_condattr_setclock(pthread_condattr_t *attr,
                              clockid_t clock_id);
```

Both return: 0 if OK, error number on failure

Curiously, the Single UNIX Specification doesn't define the *clock* attribute for any of the other attribute objects that have a wait function with a timeout.

12.4.4 Barrier Attributes

Barriers have attributes, too. We can use the pthread_barrierattr_init function to initialize a barrier attributes object and the pthread_barrierattr_destroy function to deinitialize a barrier attributes object.

```
#include <pthread.h>

int pthread_barrierattr_init(pthread_barrierattr_t *attr);

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

Both return: 0 if OK, error number on failure

The only barrier attribute currently defined is the *process-shared* attribute, which controls whether a barrier can be used by threads from multiple processes or only from within the process that initialized the barrier. As with the other attribute objects, we have one function to get the attribute (pthread_barrierattr_getpshared) value and one function to set the value (pthread_barrierattr_setpshared).

```
#include <pthread.h>

int pthread_barrierattr_getpshared(const pthread_barrierattr_t *
                                   restrict attr,
                                   int *restrict pshared);

int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                                   int pshared);
```

Both return: 0 if OK, error number on failure

The value of the *process-shared* attribute can be either `PTHREAD_PROCESS_SHARED` (accessible to threads from multiple processes) or `PTHREAD_PROCESS_PRIVATE` (accessible to only threads in the process that initialized the barrier).

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. In both cases, multiple threads of control can potentially call the same function at the same time.

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

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

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. Prior to Version 4 of the Single UNIX Specification, all XSI-conforming implementations were required to support thread-safe functions. With SUSv4, however, thread-safe function support is now required for an implementation to be considered POSIX conforming.

With thread-safe functions, implementations provide alternative, 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. The functions have the same names as their non-thread-safe relatives, but with an `_r` appended at the end of the name, signifying that these versions are reentrant. Many functions are not 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.

`getgrgid_r`	`localtime_r`
`getgrnam_r`	`readdir_r`
`getlogin_r`	`strerror_r`
`getpwnam_r`	`strtok_r`
`getpwuid_r`	`ttyname_r`
`gmtime_r`	

Figure 12.10 Alternative thread-safe functions

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.

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, all standard I/O routines that manipulate `FILE` objects are required to 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 they are 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>

#define MAXSTRINGSZ 4096

static char envbuf[MAXSTRINGSZ];

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] == '=')) {
            strncpy(envbuf, &environ[i][len+1], MAXSTRINGSZ-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 to ensure that the thread_init function is called only once per process, regardless of how many threads might race to call getenv_r at the same time. We'll have more to say about the pthread_once function in Section 12.6.

```
#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.

Even though we can make `getenv_r` thread-safe, that doesn't mean that it is reentrant with respect to signal handlers. If we were to use a nonrecursive mutex, we would run the risk that a thread would deadlock itself if it called `getenv_r` from a signal handler. If the signal handler interrupted the thread while it was executing `getenv_r`, we would already be holding `env_mutex` locked, so another attempt to lock it would 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 to 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 that is 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 by making it more difficult for threads to gain access to thread-specific data from other 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 such 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, either by calling `pthread_exit` or by returning, the destructor is called. Also, if the thread is canceled, the destructor is called, but only after the last cleanup handler returns. But if the thread calls `exit`, `_exit`, `_Exit`, or `abort`, or otherwise exits abnormally, the destructor is not called.

Threads typically 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 all of the keys can 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 to call another function that creates 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 repeats 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 return value 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>

#define MAXSTRINGSZ 4096

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(MAXSTRINGSZ);
        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] == '=')) {
            strncpy(envbuf, &environ[i][len+1], MAXSTRINGSZ-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 whether it has been canceled, and if so, 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	pthread_join	sendto
aio_suspend	msgrcv	pthread_testcancel	sigsuspend
clock_nanosleep	msgsnd	pwrite	sigtimedwait
close	msync	read	sigwait
connect	nanosleep	readv	sigwaitinfo
creat	open	recv	sleep
fcntl	openat	recvfrom	system
fdatasync	pause	recvmsg	tcdrain
fsync	poll	select	wait
lockf	pread	sem_timedwait	waitid
mq_receive	pselect	sem_wait	waitpid
mq_send	pthread_cond_timedwait	send	write
mq_timedreceive	pthread_cond_wait	sendmsg	writev

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.

access	fseeko	getwchar	putwc
catclose	fsetpos	glob	putwchar
catgets	fstat	iconv_close	readdir
catopen	fstatat	iconv_open	readdir_r
chmod	ftell	ioctl	readlink
chown	ftello	link	readlinkat
closedir	futimens	linkat	remove
closelog	fwprintf	lio_listio	rename
ctermid	fwrite	localtime	renameat
dbm_close	fwscanf	localtime_r	rewind
dbm_delete	getaddrinfo	lockf	rewinddir
dbm_fetch	getc	lseek	scandir
dbm_nextkey	getc_unlocked	lstat	scanf
dbm_open	getchar	mkdir	seekdir
dbm_store	getchar_unlocked	mkdirat	semop
dlclose	getcwd	mkdtemp	setgrent
dlopen	getdate	mkfifo	sethostent
dprintf	getdelim	mkfifoat	setnetent
endgrent	getgrent	mknod	setprotoent
endhostent	getgrgid	mknodat	setpwent
endnetent	getgrgid_r	mkstemp	setservent
endprotoent	getgrnam	mktime	setutxent
endpwent	getgrnam_r	nftw	stat
endservent	gethostent	opendir	strerror
endutxent	gethostid	openlog	strerror_r
faccessat	gethostname	pathconf	strftime
fchmod	getline	pclose	symlink
fchmodat	getlogin	perror	symlinkat
fchown	getlogin_r	popen	sync
fchownat	getnameinfo	posix_fadvise	syslog
fclose	getnetbyaddr	posix_fallocate	tmpfile
fcntl	getnetbyname	posix_madvise	ttyname
fflush	getnetent	posix_openpt	ttyname_r
fgetc	getopt	posix_spawn	tzset
fgetpos	getprotobyname	posix_spawnp	ungetc
fgets	getprotobynumber	posix_typed_mem_open	ungetwc
fgetwc	getprotoent	printf	unlink
fgetws	getpwent	psiginfo	unlinkat
fmtmsg	getpwnam	psignal	utimensat
fopen	getpwnam_r	pthread_rwlock_rdlock	utimes
fpathconf	getpwuid	pthread_rwlock_timedrdlock	vdprintf
fprintf	getpwuid_r	pthread_rwlock_timedwrlock	vfprintf
fputc	getservbyname	pthread_rwlock_wrlock	vfwprintf
fputs	getservbyport	putc	vprintf
fputwc	getservent	putc_unlocked	vwprintf
fputws	getutxent	putchar	wcsftime
fread	getutxid	putchar_unlocked	wordexp
freopen	getutxline	puts	wprintf
fscanf	getwc	pututxline	wscanf
fseek			

Figure 12.15 Optional cancellation points defined by POSIX.1

Several of the functions listed in Figure 12.15, such as the ones dealing with message catalogs and wide character sets, are not discussed further in this text.

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
```

The `pthread_setcanceltype` function sets the cancellation type to *type* (either `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`) 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. As a consequence, 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 that signal.

Signals are delivered to a single thread in the process. If the signal is related to a hardware fault, the signal is usually 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 the `sigprocmask` function to block signals from delivery. However, the behavior of `sigprocmask` is undefined in a multithreaded process. Threads have to use the `pthread_sigmask` function 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. Recall that the *set* argument contains the set of signals that the thread will use to modify its signal mask. The *how* argument can take on one of three values: `SIG_BLOCK` to add the set of signals to the thread's signal mask, `SIG_SETMASK` to replace the thread's signal mask with the set of signals, or `SIG_UNBLOCK` to remove the set of signals from the thread's signal mask. If the *oset* argument is not null, the thread's previous signal mask is stored in the `sigset_t` structure to which it points. A thread can get its current signal mask by setting the *set* argument to `NULL` and setting the *oset* argument to the address of a `sigset_t` structure. In this case, the *how* argument is ignored.

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. If the implementation supports queued signals, and multiple instances of a signal are pending, `sigwait` will remove only one instance of the signal; the other instances will remain queued.

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. 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 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 waitloc = 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(&waitloc);
                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(&waitloc, &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 when we 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
```

□

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 same 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 avoid problems with inconsistent state in a multithreaded process, POSIX.1 states that only async-signal safe functions should be called by a child process between the time that fork returns and the time that the child calls one of the exec functions. This limits what the child can do before calling exec, but doesn't address the problem of lock state in the child process.

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 must release all the locks acquired by the *prepare* fork handler.

Note that the locks are not locked once and unlocked twice, as it might 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 ordering 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 of module A's locks.
2. The *prepare* fork handler from module B is called to acquire all of module B's locks.

3. A child process is created.
4. The *child* fork handler from module B is called to release all of module B's locks in the child process.
5. The *child* fork handler from module A is called to release all of 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 of module B's locks in the parent process.
8. The *parent* fork handler from module A is called to release all of 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)
{
    int err;

    printf("preparing locks...\n");
    if ((err = pthread_mutex_lock(&lock1)) != 0)
        err_cont(err, "can't lock lock1 in prepare handler");
    if ((err = pthread_mutex_lock(&lock2)) != 0)
        err_cont(err, "can't lock lock2 in prepare handler");
}

void
parent(void)
{
```

```
    int err;

    printf("parent unlocking locks...\n");
    if ((err = pthread_mutex_unlock(&lock1)) != 0)
        err_cont(err, "can't unlock lock1 in parent handler");
    if ((err = pthread_mutex_unlock(&lock2)) != 0)
        err_cont(err, "can't unlock lock2 in parent handler");
}

void
child(void)
{
    int err;

    printf("child unlocking locks...\n");
    if ((err = pthread_mutex_unlock(&lock1)) != 0)
        err_cont(err, "can't unlock lock1 in child handler");
    if ((err = pthread_mutex_unlock(&lock2)) != 0)
        err_cont(err, "can't unlock lock2 in child handler");
}

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

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

    if ((err = pthread_atfork(prepare, parent, child)) != 0)
        err_exit(err, "can't install fork handlers");
    if ((err = pthread_create(&tid, NULL, thr_fn, 0)) != 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");
    exit(0);
}
```

Figure 12.17 `pthread_atfork` example

In Figure 12.17, 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. □

Although the `pthread_atfork` mechanism is intended to make locking state consistent after a `fork`, it has several drawbacks that make it usable in only limited circumstances:

- There is no good way to reinitialize the state for more complex synchronization objects such as condition variables and barriers.
- Some implementations of error-checking mutexes will generate errors when the child fork handler tries to unlock a mutex that was locked by the parent.
- Recursive mutexes can't be cleaned up in the child fork handler, because there is no way to determine the number of times one has been locked.
- If child processes are allowed to call only async-signal safe functions, then the child fork handler shouldn't even be able to clean up synchronization objects, because none of the functions that are used to manipulate them are async-signal safe. The practical problem is that a synchronization object might be in an intermediate state when one thread calls `fork`, but the synchronization object can't be cleaned up unless it is in a consistent state.
- If an application calls `fork` in a signal handler (which is legal, because `fork` is async-signal safe), then the fork handlers registered by `pthread_atfork` can call only async-signal safe functions, or else the results are undefined.

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 call to `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 alternative model for partitioning concurrent tasks in UNIX systems. They 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 `getenv` function shown 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` or `clock_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`?

12.8 The `timeout` function in Figure 12.8 can be simplified substantially. Explain how.