BBL PREV NEXT

Chapter 12. Thread Control

BBL BBL PREV NEXT PREV NEXT

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.

BBL BBL PREV NEXT PREV NEXT

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.

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

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

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.

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.

Figure 12.2. Examples of thread configuration limits

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

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.

Figure 12.3. POSIX.1 thread attributes

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	•	•	•	•

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.

[View full width]

```
#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.

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.

Figure 12.4. Creating 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);
}
```

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.

[View full width]

```
#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.

[View full width]

```
#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.

[View full width]

```
#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.

BBL BBL PREV NEXT PREV NEXT

12.4. Synchronization Attributes

Just as threads have attributes, so too do their synchronization objects. In this section, we discuss the attributes of mutexes, readerwriter 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.

[View full width]

```
#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.

[View full width]

```
#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.

Figure 12.5. Mutex type behavior

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

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

[View full width]

```
#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.

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.

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.

Figure 12.6. Recursive locking opportunity

[View full size image]

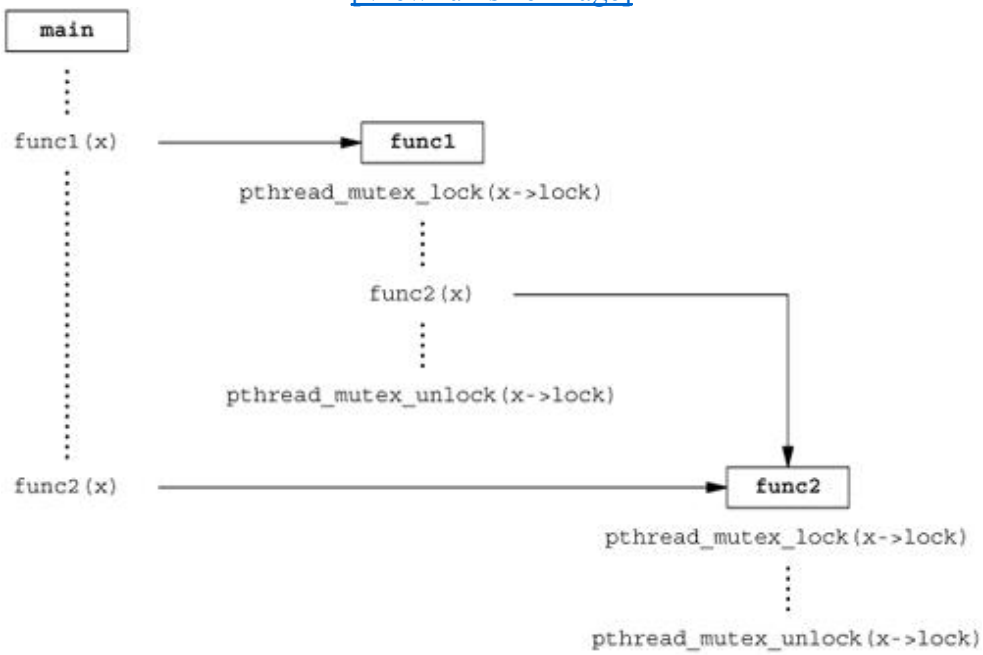
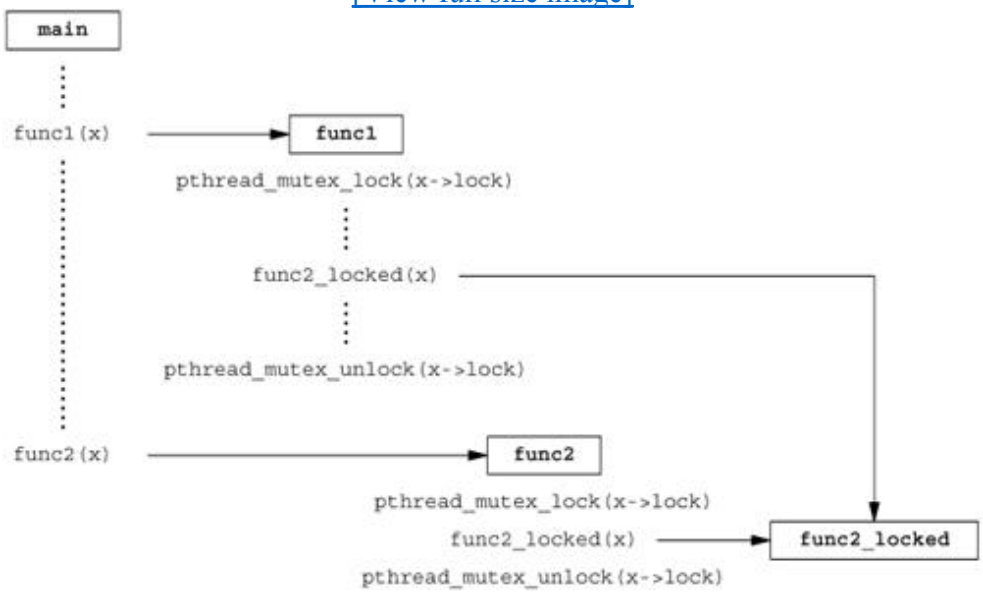


Figure 12.7. Avoiding a recursive locking opportunity

[View full size image]



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.

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.

Figure 12.8. Using a recursive mutex

```
#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);
}
```

ReaderWriter Lock Attributes

Readerwriter 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.

[View full width]

```
#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 readerwriter 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 readerwriter locks.

[View full width]

```
#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 readerwriter 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 readerwriter locks.

[View full width]

```
#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.

[View full width]

```
#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

BBL BBL PREV PREV NEXT NEXT

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.

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

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

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 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.

Figure 12.10. Alternate thread-safe functions

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

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.

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`.

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.

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 readerwriter 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.

Figure 12.11. A nonreentrant version of `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 12.12. A reentrant (thread-safe) version of `getenv`

```
#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);
}
```

BBL BBL PREV NEXT PREV NEXT

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.

[View full width]

```
#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

[View full width]

```
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.

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.

Figure 12.13. A thread-safe, compatible version of `getenv`

```
#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);
}
```

BBL BBL PREV NEXT PREV NEXT

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.

Figure 12.14. Cancellation points defined by POSIX.1

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	

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.

Figure 12.15. Optional cancellation points defined by POSIX.1

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	

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

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.

BBL BBL PREV NEXT PREV NEXT

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.

[View full width]

```
#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`.

[View full width]

```
#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.

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
```

Figure 12.16. Synchronous signal handling

```
#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);
}
```

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.

BBL BBL PREV NEXT PREV NEXT

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, readerwriter 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`.

[View full width]

```
#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.

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.

Figure 12.17. `pthread_atfork` example

```
#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);
}
```

BBL BBL PREV NEXT PREV NEXT

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.

BBL BBL PREV NEXT PREV NEXT

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.

BBL BBL PREV NEXT PREV NEXT

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`?