Chapter 11. Threads

11.1. Introduction

We discussed processes in earlier chapters. We learned about the environment of a UNIX process, the relationships between processes, and ways to control processes. We saw that a limited amount of sharing can occur between related processes.

In this chapter, we'll look inside a process further to see how we can use multiple threads of control (or simply threads) to perform multiple tasks within the environment of a single process. All threads within a single process have access to the same process components, such as file descriptors and memory.

Any time you try to share a single resource among multiple users, you have to deal with consistency. We'll conclude the chapter with a look at the synchronization mechanisms available to prevent multiple threads from viewing inconsistencies in their shared resources.

11.2. Thread Concepts

A typical UNIX process can be thought of as having a single thread of control: each process is doing only one thing at a time. With multiple threads of control, we can design our programs to do more than one thing at a time within a single process, with each thread handling a separate task. This approach can have several benefits.

- We can simplify code that deals with asynchronous events by assigning a separate thread to handle each event type. Each thread can then handle its event using a synchronous programming model. A synchronous programming model is much simpler than an asynchronous one.

- Multiple processes have to use complex mechanisms provided by the operating system to share memory and file descriptors, as we will see in Chapters 15 and 17. Threads, on the other hand, automatically have access to the same memory address space and file descriptors.

- Some problems can be partitioned so that overall program throughput can be improved. A single process that has multiple tasks to perform implicitly serializes those tasks, because there is only one thread of control. With multiple threads of control, the processing of independent tasks can be interleaved by assigning a separate thread per task. Two tasks can be interleaved only if they don't depend on the processing performed by each other.

- Similarly, interactive programs can realize improved response time by using multiple threads to separate the portions of the program that deal with user input and output from the other parts of the program.

Some people associate multithreaded programming with multiprocessor systems. The benefits of a multithreaded programming model can be realized even if your program is running on a uniprocessor. A program can be simplified using threads regardless of the number of processors, because the number of processors doesn't affect the program structure. Furthermore, as long as your program has to block when serializing tasks, you can still see improvements in response time and throughput when running on a uniprocessor, because some threads might be able to run while others are blocked.

A thread consists of the information necessary to represent an execution context within a process. This includes a thread ID that identifies the thread within a process, a set of register values, a stack, a scheduling priority and policy, a signal mask, an `errno` variable (recall Section 1.7), and thread-specific data (Section 12.6). Everything within a process is sharable among the threads in a process, including the text of the executable program, the program's global and heap memory, the stacks, and the file descriptors.

The threads interface we're about to see is from POSIX.1-2001. The threads interface, also known as "pthreads" for "POSIX threads," is an optional feature in POSIX.1-2001. The feature test macro for POSIX threads is `_POSIX_THREADS`. Applications can either use this in an `#ifdef` test to determine at compile time whether threads are supported or call `sysconf` with the `_SC_THREADS` constant to determine at runtime whether threads are supported.

11.3. Thread Identification

Just as every process has a process ID, every thread has a thread ID. Unlike the process ID, which is unique in the system, the thread ID has significance only within the context of the process to which it belongs.

Recall that a process ID, represented by the `pid_t` data type, is a non-negative integer. A thread ID is represented by the `pthread_t` data type. Implementations are allowed to use a structure to represent the `pthread_t` data type, so portable implementations can't treat them as integers. Therefore, a function must be used to compare two thread IDs.

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

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Returns: nonzero if equal, 0 otherwise

> Linux 2.4.22 uses an unsigned long integer for the `pthread_t` data type. Solaris 9 represents the `pthread_t` data type as an unsigned integer. FreeBSD 5.2.1 and Mac OS X 10.3 use a pointer to the `pthread` structure for the `pthread_t` data type.

A consequence of allowing the `pthread_t` data type to be a structure is that there is no portable way to print its value. Sometimes, it is useful to print thread IDs during program debugging, but there is usually no need to do so otherwise. At worst, this results in nonportable debug code, so it is not much of a limitation.

A thread can obtain its own thread ID by calling the `pthread_self` function.

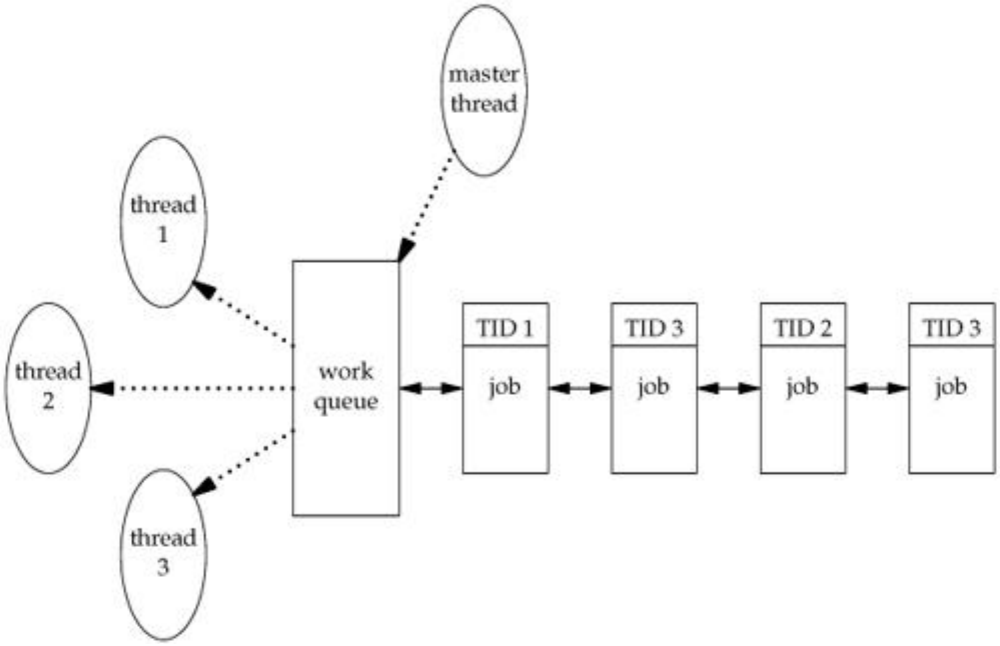
```
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns: the thread ID of the calling thread

This function can be used with `pthread_equal` when a thread needs to identify data structures that are tagged with its thread ID. For example, a master thread might place work assignments on a queue and use the thread ID to control which jobs go to each worker thread. This is illustrated in Figure 11.1. A single master thread places new jobs on a work queue. A pool of three worker threads removes jobs from the queue. Instead of allowing each thread to process whichever job is at the head of the queue, the master thread controls job assignment by placing the ID of the thread that should process the job in each job structure. Each worker thread then removes only jobs that are tagged with its own thread ID.

Figure 11.1. Work queue example



BBL
PREV NEXT
BBL
PREV NEXT

11.4. Thread Creation

The traditional UNIX process model supports only one thread of control per process. Conceptually, this is the same as a threads-based model whereby each process is made up of only one thread. With pthreads, when a program runs, it also starts out as a single process with a single thread of control. As the program runs, its behavior should be indistinguishable from the traditional process, until it creates more threads of control. Additional threads can be created by calling the `pthread_create` function.

[View full width]

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict
➥ attr,
                   void *(*start_rtn)(void), void
➥ *restrict arg);
```

Returns: 0 if OK, error number on failure

The memory location pointed to by tidp is set to the thread ID of the newly created thread when `pthread_create` returns successfully. The attr argument is used to customize various thread attributes. We'll cover thread attributes in Section 12.3, but for now, we'll set this to `NULL` to create a thread with the default attributes.

The newly created thread starts running at the address of the start_rtn function. This function takes a single argument, arg, which is a typeless pointer. If you need to pass more than one argument to the start_rtn function, then you need to store them in a structure and pass the address of the structure in arg.

When a thread is created, there is no guarantee which runs first: the newly created thread or the calling thread. The newly created thread has access to the process address space and inherits the calling thread's floating-point environment and signal mask; however, the set of pending signals for the thread is cleared.

Note that the pthread functions usually return an error code when they fail. They don't set `errno` like the other POSIX functions. The per thread copy of `errno` is provided only for compatibility with existing functions that use it. With threads, it is cleaner to return the error code from the function, thereby restricting the scope of the error to the function that caused it, instead of relying on some global state that is changed as a side effect of the function.

Example

Although there is no portable way to print the thread ID, we can write a small test program that does, to gain some insight into how threads work. The program in Figure 11.2 creates one thread and prints the process and thread IDs of the new thread and the initial thread.

This example has two oddities, necessary to handle races between the main thread and the new thread. (We'll learn better ways to deal with these later in this chapter.) The first is the need to sleep in the main thread. If it doesn't sleep, the main thread might exit, thereby terminating the entire process before the new thread gets a chance to run. This behavior is dependent on the operating system's threads implementation and scheduling algorithms.

The second oddity is that the new thread obtains its thread ID by calling `pthread_self` instead of reading it out of shared memory or receiving it as an argument to its thread-start routine. Recall that `pthread_create` will return the thread ID of the newly created thread through the first parameter (tidp). In our example, the main thread stores this in `ntid`, but the new thread can't safely use it. If the new thread runs before the main thread returns from calling `pthread_create`, then the new thread will see the uninitialized contents of `ntid` instead of the thread ID.

Running the program in Figure 11.2 on Solaris gives us

```
$ ./a.out
main thread: pid 7225 tid 1 (0x1)
new thread:  pid 7225 tid 4 (0x4)
```

As we expect, both threads have the same process ID, but different thread IDs. Running the program in Figure 11.2 on FreeBSD gives us

```
$ ./a.out
main thread: pid 14954 tid 134529024 (0x804c000)
new thread:  pid 14954 tid 134530048 (0x804c400)
```

As we expect, both threads have the same process ID. If we look at the thread IDs as decimal integers, the values look strange, but if we look at them in hexadecimal, they make more sense. As we noted earlier, FreeBSD uses a pointer to the thread data structure for its thread ID.

We would expect Mac OS X to be similar to FreeBSD; however, the thread ID for the main thread is from a different address range than the thread IDs for threads created with `pthread_create`:

```
$ ./a.out
main thread: pid 779 tid 2684396012 (0xa000a1ec)
new thread:  pid 779 tid 25166336 (0x1800200)
```

Running the same program on Linux gives us slightly different results:

```
$ ./a.out
new thread:  pid 6628 tid 1026 (0x402)
main thread: pid 6626 tid 1024 (0x400)
```

The Linux thread IDs look more reasonable, but the process IDs don't match. This is an artifact of the Linux threads implementation, where the `clone` system call is used to implement `pthread_create`. The `clone` system call creates a child process that can share a configurable amount of its parent's execution context, such as file descriptors and memory.

Note also that the output from the main thread appears before the output from the thread we create, except on Linux. This illustrates that we can't make any assumptions about how threads will be scheduled.

Figure 11.2. Printing thread IDs

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

pthread_t ntid;

void
printids(const char *s)
{
    pid_t      pid;
    pthread_t  tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
      (unsigned int)tid, (unsigned int)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int     err;
```

```
    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_quit("can't create thread: %s\n", strerror(err));
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

BBL

BBL

PREV NEXT

PREV NEXT

11.5. Thread Termination

If any thread within a process calls `exit`, `_Exit`, or `_exit`, then the entire process terminates. Similarly, when the default action is to terminate the process, a signal sent to a thread will terminate the entire process (we'll talk more about the interactions between signals and threads in Section 12.8).

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.
2. The thread can be canceled by another thread in the same process.
3. The thread can call `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

The rval_ptr is a typeless pointer, similar to the single argument passed to the start routine. This pointer is available to other threads in the process by calling the `pthread_join` function.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **rval_ptr);
```

Returns: 0 if OK, error number on failure

The calling thread will block until the specified thread calls `pthread_exit`, returns from its start routine, or is canceled. If the thread simply returned from its start routine, rval_ptr will contain the return code. If the thread was canceled, the memory location specified by rval_ptr is set to `PTHREAD_CANCELED`.

By calling `pthread_join`, we automatically place a thread in the detached state (discussed shortly) so that its resources can be recovered. If the thread was already in the detached state, calling `pthread_join` fails, returning `EINVAL`.

If we're not interested in a thread's return value, we can set rval_ptr to `NULL`. In this case, calling `pthread_join` allows us to wait for the specified thread, but does not retrieve the thread's termination status.

Example

Figure 11.3 shows how to fetch the exit code from a thread that has terminated.

Running the program in Figure 11.3 gives us

```
$ ./a.out
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

As we can see, when a thread exits by calling `pthread_exit` or by simply returning from the start routine, the exit status can be obtained by another thread by calling `pthread_join`.

Figure 11.3. Fetching the thread exit status

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

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
```

```
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    printf("thread 1 exit code %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("can't join with thread 2: %s\n", strerror(err));
    printf("thread 2 exit code %d\n", (int)tret);
    exit(0);
}
```

The typeless pointer passed to `pthread_create` and `pthread_exit` can be used to pass more than a single value. The pointer can be used to pass the address of a structure containing more complex information. Be careful that the memory used for the structure is still valid when the caller has completed. If the structure was allocated on the caller's stack, for example, the memory contents might have changed by the time the structure is used. For example, if a thread allocates a structure on its stack and passes a pointer to this structure to `pthread_exit`, then the stack might be destroyed and its memory reused for something else by the time the caller of `pthread_join` tries to use it.

Example

The program in Figure 11.4 shows the problem with using an automatic variable (allocated on the stack) as the argument to `pthread_exit`.

When we run this program on Linux, we get

```
$ ./a.out
thread 1:
  structure at 0x409a2abc
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 32770
parent:
  structure at 0x409a2abc
  foo.a = 0
  foo.b = 32770
  foo.c = 1075430560
  foo.d = 1073937284
```

Of course, the results vary, depending on the memory architecture, the compiler, and the implementation of the threads library. The results on FreeBSD are similar:

```
$ ./a.out
thread 1:
  structure at 0xbfafefc0
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 134534144
```

```
   parent:
     structure at 0xbfafefc0
     foo.a = 0
     foo.b = 134534144
     foo.c = 3
     foo.d = 671642590
```

As we can see, the contents of the structure (allocated on the stack of thread tid1) have changed by the time the main thread can access the structure. Note how the stack of the second thread (tid2) has overwritten the first thread's stack. To solve this problem, we can either use a global structure or allocate the structure using `malloc`.

Figure 11.4. Incorrect use of `pthread_exit` argument

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

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf(s);
    printf("  structure at 0x%x\n", (unsigned)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{

    struct foo  foo = {1, 2, 3, 4};

    printfoo("thread 1:\n", &foo);
    pthread_exit((void *)&foo);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2: ID is %d\n", pthread_self());
    pthread_exit((void *)0);
}
int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    struct foo  *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
```

```
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    sleep(1);
    printf("parent starting second thread\n");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    sleep(1);
    printfoo("parent:\n", fp);
    exit(0);
}
```

One thread can request that another in the same process be canceled by calling the `pthread_cancel` function.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Returns: 0 if OK, error number on failure

In the default circumstances, `pthread_cancel` will cause the thread specified by tid to behave as if it had called `pthread_exit` with an argument of `PTHREAD_CANCELED`. However, a thread can elect to ignore or otherwise control how it is canceled. We will discuss this in detail in Section 12.7. Note that `pthread_cancel` doesn't wait for the thread to terminate. It merely makes the request.

A thread can arrange for functions to be called when it exits, similar to the way that the `atexit` function (Section 7.3) can be used by a process to arrange that functions can be called when the process exits. The functions are known as thread cleanup handlers. More than one cleanup handler can be established for a thread. The handlers are recorded in a stack, which means that they are executed in the reverse order from that with which they were registered.

[View full width]

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *),
    void *arg);

void pthread_cleanup_pop(int execute);
```

The `pthread_cleanup_push` function schedules the cleanup function, rtn, to be called with the single argument, arg, when the thread performs one of the following actions:

- Makes a call to `pthread_exit`
- Responds to a cancellation request
- Makes a call to `pthread_cleanup_pop` with a nonzero execute argument

If the execute argument is set to zero, the cleanup function is not called. In either case, `pthread_cleanup_pop` removes the cleanup handler established by the last call to `pthread_cleanup_push`.

A restriction with these functions is that, because they can be implemented as macros, they must be used in matched pairs within the same scope in a thread. The macro definition of `pthread_cleanup_push` can include a `{` character, in which case the matching `}` character is in the `pthread_cleanup_pop` definition.

Example

Figure 11.5 shows how to use thread cleanup handlers. Although the example is somewhat contrived, it illustrates the mechanics involved. Note that although we never intend to pass a nonzero argument to the thread start-up routines, we still need to match calls to `pthread_cleanup_pop` with the calls to `pthread_cleanup_push`; otherwise, the program might not compile.

Running the program in Figure 11.5 gives us

```
$ ./a.out
thread 1 start
thread 1 push complete
thread 2 start
thread 2 push complete
cleanup: thread 2 second handler
cleanup: thread 2 first handler
thread 1 exit code 1
thread 2 exit code 2
```

From the output, we can see that both threads start properly and exit, but that only the second thread's cleanup handlers are called. Thus, if the thread terminates by returning from its start routine, its cleanup handlers are not called. Also note that the cleanup handlers are called in the reverse order from which they were installed.

Figure 11.5. Thread cleanup handler

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

void
cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

void *
thr_fn1(void *arg)
{
```

```
    printf("thread 1 start\n");
    pthread_cleanup_push(cleanup, "thread 1 first handler");
    pthread_cleanup_push(cleanup, "thread 1 second handler");
    printf("thread 1 push complete\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 start\n");
    pthread_cleanup_push(cleanup, "thread 2 first handler");
    pthread_cleanup_push(cleanup, "thread 2 second handler");
    printf("thread 2 push complete\n");
    if (arg)
        pthread_exit((void *)2);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);
      if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    printf("thread 1 exit code %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("can't join with thread 2: %s\n", strerror(err));
    printf("thread 2 exit code %d\n", (int)tret);
    exit(0);
}
```

By now, you should begin to see similarities between the thread functions and the process functions. Figure 11.6 summarizes the similar functions.

Figure 11.6. Comparison of process and thread primitives

Process primitive	Thread primitive	Description
fork	pthread_create	create a new flow of control

`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cancel_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

By default, a thread's termination status is retained until `pthread_join` is called for that thread. A thread's underlying storage can be reclaimed immediately on termination if that thread has been detached. When a thread is detached, the `pthread_join` function can't be used to wait for its termination status. A call to `pthread_join` for a detached thread will fail, returning `EINVAL`. We can detach a thread by calling `pthread_detach`.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, error number on failure

As we will see in the next chapter, we can create a thread that is already in the detached state by modifying the thread attributes we pass to `pthread_create`.

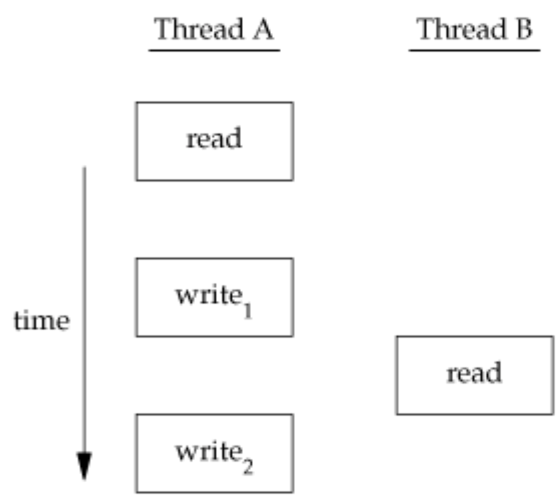
11.6. Thread Synchronization

When multiple threads of control share the same memory, we need to make sure that each thread sees a consistent view of its data. If each thread uses variables that other threads don't read or modify, no consistency problems exist. Similarly, if a variable is read-only, there is no consistency problem with more than one thread reading its value at the same time. However, when one thread can modify a variable that other threads can read or modify, we need to synchronize the threads to ensure that they don't use an invalid value when accessing the variable's memory contents.

When one thread modifies a variable, other threads can potentially see inconsistencies when reading the value of the variable. On processor architectures in which the modification takes more than one memory cycle, this can happen when the memory read is interleaved between the memory write cycles. Of course, this behavior is architecture dependent, but portable programs can't make any assumptions about what type of processor architecture is being used.

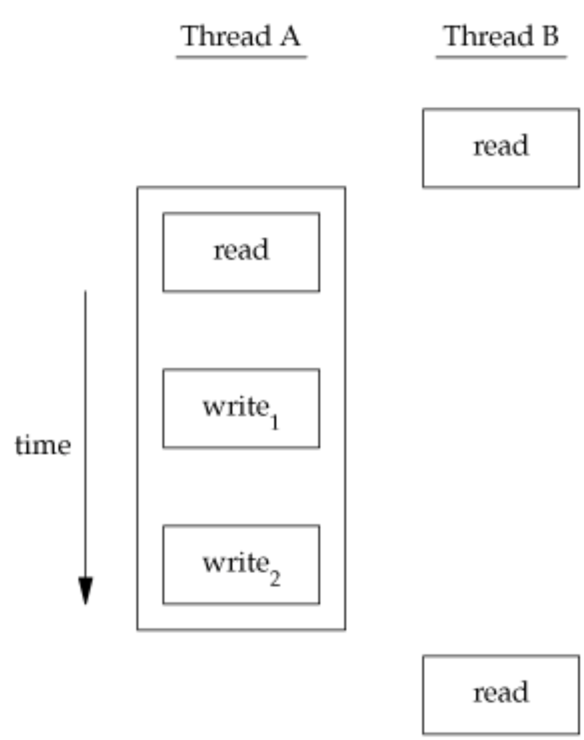
Figure 11.7 shows a hypothetical example of two threads reading and writing the same variable. In this example, thread A reads the variable and then writes a new value to it, but the write operation takes two memory cycles. If thread B reads the same variable between the two write cycles, it will see an inconsistent value.

Figure 11.7. Interleaved memory cycles with two threads



To solve this problem, the threads have to use a lock that will allow only one thread to access the variable at a time. Figure 11.8 shows this synchronization. If it wants to read the variable, thread B acquires a lock. Similarly, when thread A updates the variable, it acquires the same lock. Thus, thread B will be unable to read the variable until thread A releases the lock.

Figure 11.8. Two threads synchronizing memory access

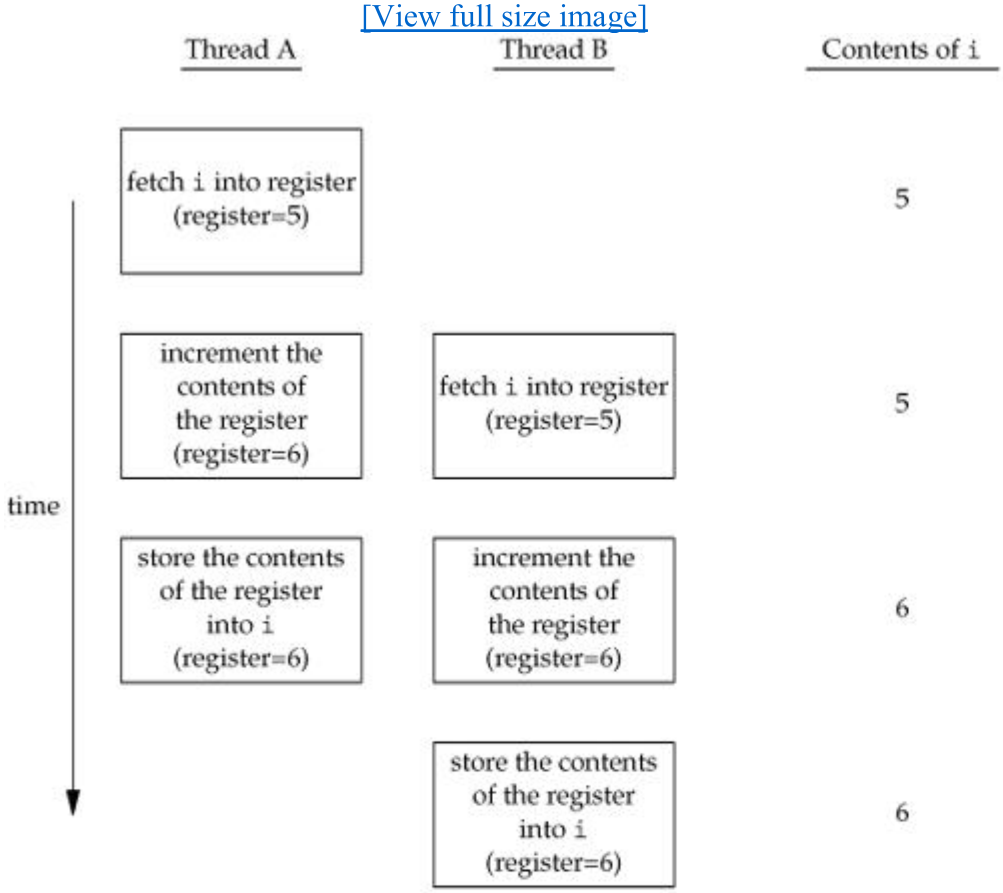


You also need to synchronize two or more threads that might try to modify the same variable at the same time. Consider the case in which you increment a variable (Figure 11.9). The increment operation is usually broken down into three steps.

1. Read the memory location into a register.
2. Increment the value in the register.
3. Write the new value back to the memory location.

Figure 11.9. Two unsynchronized threads incrementing the same variable

[View full size image]



If two threads try to increment the same variable at almost the same time without synchronizing with each other, the results can be inconsistent. You end up with a value that is either one or two greater than before, depending on the value observed when the second thread starts its operation. If the second thread performs step 1 before the first thread performs step 3, the second thread will read the same initial value as the first thread, increment it, and write it back, with no net effect.

If the modification is atomic, then there isn't a race. In the previous example, if the increment takes only one memory cycle, then no race exists. If our data always appears to be sequentially consistent, then we need no additional synchronization. Our operations are sequentially consistent when multiple threads can't observe inconsistencies in our data. In modern computer systems, memory accesses take multiple bus cycles, and multiprocessors generally interleave bus cycles among multiple processors, so we aren't guaranteed that our data is sequentially consistent.

In a sequentially consistent environment, we can explain modifications to our data as a sequential step of operations taken by the running threads. We can say such things as "Thread A incremented the variable, then thread B incremented the variable, so its value is two greater than before" or "Thread B incremented the variable, then thread A incremented the variable, so its value is two greater than before." No possible ordering of the two threads can result in any other value of the variable.

Besides the computer architecture, races can arise from the ways in which our programs use variables, creating places where it is possible to view inconsistencies. For example, we might increment a variable and then make a decision based on its value. The combination of the increment step and the decision-making step aren't atomic, so this opens a window where inconsistencies can arise.

Mutexes

We can protect our data and ensure access by only one thread at a time by using the pthreads mutual-exclusion interfaces. A mutex is basically a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done. While it is set, any other thread that tries to set it will block until we release it. If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock. The others will see that the mutex is still locked and go back to waiting for it to become available again. In this way, only one thread will proceed at a time.

This mutual-exclusion mechanism works only if we design our threads to follow the same data-access rules. The operating system doesn't serialize access to data for us. If we allow one thread to access a shared resource without first acquiring a lock, then inconsistencies can occur even though the rest of our threads do acquire the lock before attempting to access the shared resource.

A mutex variable is represented by the `pthread_mutex_t` data type. Before we can use a mutex variable, we must first initialize it by either setting it to the constant `PTHREAD_MUTEX_INITIALIZER` (for statically-allocated mutexes only) or calling `pthread_mutex_init`. If we allocate the mutex dynamically (by calling `malloc`, for example), then we need to call `pthread_mutex_destroy` before freeing the memory.

[View full width]

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict
➥ mutex,
                       const pthread_mutexattr_t
➥ *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Both return: 0 if OK, error number on failure

To initialize a mutex with the default attributes, we set attr to `NULL`. We will discuss nondefault mutex attributes in Section 12.4.

To lock a mutex, we call `pthread_mutex_lock`. If the mutex is already locked, the calling thread will block until the mutex is unlocked. To unlock a mutex, we call `pthread_mutex_unlock`.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

All return: 0 if OK, error number on failure

If a thread can't afford to block, it can use `pthread_mutex_trylock` to lock the mutex conditionally. If the mutex is unlocked at the time `pthread_mutex_trylock` is called, then `pthread_mutex_trylock` will lock the mutex without blocking and return 0. Otherwise, `pthread_mutex_trylock` will fail, returning `EBUSY` without locking the mutex.

Example

Figure 11.10 illustrates a mutex used to protect a data structure. When more than one thread needs to access a dynamically-allocated object, we can embed a reference count in the object to ensure that we don't free its memory before all threads are done using it.

We lock the mutex before incrementing the reference count, decrementing the reference count, and checking whether the reference count reaches zero. No locking is necessary when we initialize the reference count to 1 in the `foo_alloc` function, because the allocating thread is the only reference to it so far. If we were to place the structure on a list at this point, it could be found by other threads, so we would need to lock it first.

Before using the object, threads are expected to add a reference count to it. When they are done, they must release the reference. When the last reference is released, the object's memory is freed.

Figure 11.10. Using a mutex to protect a data structure

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

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo *fp;
```

```
    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}

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

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Deadlock Avoidance

A thread will deadlock itself if it tries to lock the same mutex twice, but there are less obvious ways to create deadlocks with mutexes. For example, when we use more than one mutex in our programs, a deadlock can occur if we allow one thread to hold a mutex and block while trying to lock a second mutex at the same time that another thread holding the second mutex tries to lock the first mutex. Neither thread can proceed, because each needs a resource that is held by the other, so we have a deadlock.

Deadlocks can be avoided by carefully controlling the order in which mutexes are locked. For example, assume that you have two mutexes, A and B, that you need to lock at the same time. If all threads always lock mutex A before mutex B, no deadlock can occur from the use of the two mutexes (but you can still deadlock on other resources). Similarly, if all threads always lock mutex B before mutex A, no deadlock will occur. You'll have the potential for a deadlock only when one thread attempts to lock the mutexes in the opposite order from another thread.

Sometimes, an application's architecture makes it difficult to apply a lock ordering. If enough locks and data structures are involved that the functions you have available can't be molded to fit a simple hierarchy, then you'll have to try some other approach. In this case, you might be able to release your locks and try again at a later time. You can use the `pthread_mutex_trylock` interface to avoid deadlocking in this case. If you are already holding locks and `pthread_mutex_trylock` is successful, then you can proceed. If it can't acquire the lock, however, you can release the locks you already hold, clean up, and try again later.

Example

In this example, we update Figure 11.10 to show the use of two mutexes. We avoid deadlocks by ensuring that when we need to acquire two mutexes at the same time, we always lock them in the same order. The second mutex protects a hash list that we use to keep track of the `foo` data structures. Thus, the `hashlock` mutex protects both the `fh` hash table and the `f_next` hash link field in the `foo` structure. The `f_lock` mutex in the `foo` structure protects access to the remainder of the `foo` structure's fields.

Comparing Figure 11.11 with Figure 11.10, we see that our allocation function now locks the hash list lock, adds the new structure to a hash bucket, and before unlocking the hash list lock, locks the mutex in the new structure. Since the new structure is placed on a global list, other threads can find it, so we need to block them if they try to access the new structure, until we are done initializing it.

The `foo_find` function locks the hash list lock and searches for the requested structure. If it is found, we increase the reference count and return a pointer to the structure. Note that we honor the lock ordering by locking the hash list lock in `foo_find` before `foo_hold` locks the `foo` structure's `f_lock` mutex.

Now with two locks, the `foo_rele` function is more complicated. If this is the last reference, we need to unlock the structure mutex so that we can acquire the hash list lock, since we'll need to remove the structure from the hash list. Then we reacquire the structure mutex. Because we could have blocked since the last time we held the structure mutex, we need to recheck the condition to see whether we still need to free the structure. If another thread found the structure and added a reference to it while we blocked to honor the lock ordering, we simply need to decrement the reference count, unlock everything, and return.

This locking is complex, so we need to revisit our design. We can simplify things considerably by using the hash list lock to protect the structure reference count, too. The structure mutex can be used to protect everything else in the `foo` structure. Figure 11.12 reflects this change.

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

Figure 11.11. Using two mutexes

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

#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)
struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;
```

```
struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    struct foo     *f_next; /* protected by hashlock */
    int             f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

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

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

    idx = HASH(fp);

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

void
```

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

    pthread_mutex_lock(&fp->f_lock);
    if (fp->f_count == 1) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&fp->f_lock);
        /* need to recheck the condition */
        if (fp->f_count != 1) {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        /* remove from list */
        idx = HASH(fp);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        fp->f_count--;
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Figure 11.12. Simplified locking

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

#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)

struct foo *fh[NHASH];
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count; /* protected by hashlock */
    pthread_mutex_t f_lock;
    struct foo     *f_next; /* protected by hashlock */
    int             f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
```

```
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
    }
    return(fp);

}

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

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

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

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

    pthread_mutex_lock(&hashlock);
    if (--fp->f_count == 0) { /* last reference, remove from list */
        idx = HASH(fp);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
```

```
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&hashlock);
    }
}
```

ReaderWriter Locks

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

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

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

Readerwriter locks are also called sharedexclusive locks. When a readerwriter lock is read-locked, it is said to be locked in shared mode. When it is write-locked, it is said to be locked in exclusive mode.

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

[View full width]

```
#include <pthread.h>

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

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Both return: 0 if OK, error number on failure

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

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

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

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

All return: 0 if OK, error number on failure

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

The Single UNIX Specification also defines conditional versions of the readerwriter locking primitives.

[View full width]

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t
 *rwlock);

int pthread_rwlock_trywrlock(pthread_rwlock_t
 *rwlock);
```

Both return: 0 if OK, error number on failure

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

Example

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

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

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

Figure 11.13. Using readerwriter locks

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

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

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

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

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

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

```
    return(0);
}

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

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

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

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

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

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

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

Condition Variables

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

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

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

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

[View full width]

```
#include <pthread.h>

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

int pthread_cond_destroy(pthread_cond_t *cond);
```

Both return: 0 if OK, error number on failure

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

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

[View full width]

```
#include <pthread.h>

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

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

Both return: 0 if OK, error number on failure

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

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

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

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

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

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

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

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

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

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

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Both return: 0 if OK, error number on failure

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

Example

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

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

Figure 11.14. Using condition variables

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

```
struct msg {
    struct msg *m_next;
    /* ... more stuff here ... */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

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

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

BBL PREV NEXT
BBL PREV NEXT

11.7. Summary

In this chapter, we introduced the concept of threads and discussed the POSIX.1 primitives available to create and destroy them. We also introduced the problem of thread synchronization. We discussed three fundamental synchronization mechanismsmutexes, readerwriter locks, and condition variablesand we saw how to use them to protect shared resources.

BBL PREV NEXT
BBL PREV NEXT

Exercises

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

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

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

11.4 Which sequence of steps is correct?

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

or

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

BBL PREV NEXT