12

POSIX Threads

In Chapter 11, you saw how processes are handled in Linux (and indeed in UNIX). These multi-processing features have long been a feature of UNIX-like operating systems. Sometimes it may be very useful to make a single program do two things at once, or at least to appear to do so, or you might want two or more things to happen at the same time in a closely coupled way but consider the overhead of creating a new process with `fork` too great. For these occasions you can use threads, which allow a single process to multitask.

In this chapter, you look at

- Creating new threads within a process
- Synchronizing data access between threads in a single process
- Modifying the attributes of a thread
- Controlling one thread from another in the same process

What Is a Thread?

Multiple strands of execution in a single program are called *threads*. A more precise definition is that a thread is a sequence of control within a process. All the programs you have seen so far have executed as a single process, although, like many other operating systems, Linux is quite capable of running multiple processes simultaneously. Indeed, all processes have at least one thread of execution. All the processes that you have seen so far in this book have had just one thread of execution.

It's important to be clear about the difference between the `fork` system call and the creation of new threads. When a process executes a `fork` call, a new copy of the process is created with its own variables and its own PID. This new process is scheduled independently, and (in general) executes almost independently of the process that created it. When we create a new thread in a process, in contrast, the new thread of execution gets its own stack (and hence local variables) but shares global variables, file descriptors, signal handlers, and its current directory state with the process that created it.

The concept of threads has been around for some time, but until the IEEE POSIX committee published some standards, they had not been widely available in UNIX-like operating systems, and the implementations that did exist tended to vary between different vendors. With the advent of the POSIX 1003.1c specification, all that changed; threads are not only better standardized, but are also available on most Linux distributions. Now that multi-core processors have also become common even in desktop machines, most machines also have underlying hardware support that allows them to physically execute multiple threads simultaneously. Previously, with single-core CPUs, the simultaneous execution of threads was just a clever, though very efficient, illusion.

Linux first acquired thread support around 1996, with a library often referred to as "LinuxThreads." This was very close to the POSIX standard (indeed, for many purposes the differences are not noticeable) and it was a significant step forward that enabled Linux programmers to use threads for the first time. However, there where slight discrepancies between the Linux implementation and the POSIX standard, most notably with regard to signal handling. These limitations were imposed not so much by the library implementation, but more by the limitations of the underlying support from the Linux kernel.

Various projects looked at how the thread support on Linux might be improved, not just to clear up the slight discrepancies between the POSIX standard and the Linux implementation, but also to improve performance and remove any unnecessary restrictions. Much work centered on how user-level threads should map to kernel-level threads. The two principal projects were New Generation POSIX Threads (NGPT) and Native POSIX Thread Library (NPTL). Both projects had to make changes to the Linux kernel to support the new libraries, and both offered significant performance improvements over the older Linux threads.

In 2002, the NGPT team announced that they did not wish to split the community and would cease adding new features to NGPT, but would continue to work on thread support in Linux, effectively throwing their weight behind the NPTL effort. NPTL became the new standard for threads on Linux, with its first mainstream release in Red Hat Linux 9. You can find some interesting background information on NPTL in a paper titled "The Native POSIX Thread Library for Linux" by Ulrich Drepper and Ingo Molnar, which, at the time of this book's writing, is at `http://people.redhat.com/drepper/nptl-design.pdf`.

Most of the code in this chapter should work with any of the thread libraries, because it is based on the POSIX standard that is common across all the thread libraries. However, you may see some slight differences if you are using an older Linux distribution, particularly if you use `ps` to look at the examples while they are running.

Advantages and Drawbacks of Threads

Creating a new thread has some distinct advantages over creating a new process in certain circumstances. The overhead cost of creating a new thread is significantly less than that of creating a new process (though Linux is particularly efficient at creating new processes compared with many other operating systems).

Following are some advantages of using threads:

- Sometimes it is very useful to make a program appear to do two things at once. The classic example is to perform a real-time word count on a document while still editing the text. One thread can manage the user's input and perform editing. The other, which can see the same document content, can continuously update a word count variable. The first thread (or even a third one) can use this

shared variable to keep the user informed. Another example is a multithreaded database server where an apparent single process serves multiple clients, improving the overall data throughput by servicing some requests while blocking others, waiting for disk activity. For a database server, this apparent multitasking is quite hard to do efficiently in different processes, because the requirements for locking and data consistency cause the different processes to be very tightly coupled. This can be done much more easily with multiple threads than with multiple processes.

- The performance of an application that mixes input, calculation, and output may be improved by running these as three separate threads. While the input or output thread is waiting for a connection, one of the other threads can continue with calculations. A server application processing multiple network connects may also be a natural fit for a multithreaded program.
- Now that multi-cored CPUs are common even in desktop and laptop machines, using multiple threads inside a process can, if the application is suitable, enable a single process to better utilize the hardware resources available.
- In general, switching between threads requires the operating system to do much less work than switching between processes. Thus, multiple threads are much less demanding on resources than multiple processes, and it is more practical to run programs that logically require many threads of execution on single-processor systems. That said, the design difficulties of writing a multithreaded program are significant and should not be taken lightly.

Threads also have drawbacks:

- Writing multithreaded programs requires very careful design. The potential for introducing subtle timing faults, or faults caused by the unintentional sharing of variables in a multithreaded program is considerable. Alan Cox (the well respected Linux guru) has commented that threads are also known as "how to shoot yourself in both feet at once."
- Debugging a multithreaded program is much, much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control.
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine, unless the calculation truly allows multiple parts to be calculated simultaneously and the machine it is executing on has multiple processor cores to support true multiprocessing.

A First Threads Program

There is a whole set of library calls associated with threads, most of whose names start with `pthread`. To use these library calls, you must define the macro `_REENTRANT`, include the file `pthread.h`, and link with the threads library using `-lpthread`.

When the original UNIX and POSIX library routines were designed, it was assumed that there would be only a single thread of execution in any process. An obvious example is `errno`, the variable used for retrieving error information after a call fails. In a multithreaded program there would, by default, be only one `errno` variable shared between all the threads. The variable could easily be updated by a call in one thread before a different thread has been able to retrieve a previous error code. Similar problems exist with functions, such as `fputs`, that normally use a single global area for buffering output.

You need routines known as *re-entrant* routines. Re-entrant code can be called more than once, whether by different threads or by nested invocations in some way, and still function correctly. Thus, the re-entrant section of code usually must use local variables only in such a way that each and every call to the code gets its own unique copy of the data.

In multithreaded programs, you tell the compiler that you need this feature by defining the `_REENTRANT` macro before any `#include` lines in your program. This does three things, and does them so elegantly that usually you don't even need to know what was done:

- Some functions get prototypes for a re-entrant safe equivalent. These are normally the same function name, but with `_r` appended so that, for example, `gethostbyname` is changed to `gethostbyname_r`.
- Some `stdio.h` functions that are normally implemented as macros become proper re-entrant safe functions.
- The variable `errno`, from `errno.h`, is changed to call a function, which can determine the real `errno` value in a multithread safe way.

Including the file `pthread.h` provides you with other definitions and prototypes that you will need in your code, much like `stdio.h` for standard input and output routines. Finally, you need to ensure that you include the appropriate thread header file and link with the appropriate threads library that implements the `pthread` functions. The Try It Out example later in this section offers more detail about compiling your program, but first let's look at the new functions you need for managing threads. `pthread_create` creates a new thread, much as `fork` creates a new process.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);
```

This may look imposing, but it is actually quite easy to use. The first argument is a pointer to `pthread_t`. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables you to refer to the thread. The next argument sets the thread attributes. You do not usually need any special attributes, and you can simply pass `NULL` as this argument. Later in the chapter you will see how to use these attributes. The final two arguments tell the thread the function that it is to start executing and the arguments that are to be passed to this function.

```
void *(*start_routine)(void *)
```

The preceding line simply says that you must pass the address of a function taking a pointer to `void` as a parameter and the function will return a pointer to `void`. Thus, you can pass any type of single argument and return a pointer to any type. Using `fork` causes execution to continue in the same location with a different return code, whereas using a new thread explicitly provides a pointer to a function where the new thread should start executing.

The return value is `0` for success or an error number if anything goes wrong. The manual pages have details of error conditions for this and other functions used in this chapter.

> `pthread_create`, *like most* `pthread_` *functions, is among the few Linux functions that do not follow the convention of using a return value of 1 for errors. Unless you are very sure, it's always safest to double-check the manual before checking the return code.*

When a thread terminates, it calls the `pthread_exit` function, much as a process calls `exit` when it terminates. This function terminates the calling thread, returning a pointer to an object. Never use it to return a pointer to a local variable, because the variable will cease to exist when the thread does so, causing a serious bug. `pthread_exit` is declared as follows:

```
#include <pthread.h>

void pthread_exit(void *retval);
```

`pthread_join` is the thread equivalent of `wait` that processes use to collect child processes. This function is declared as follows:

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

The first parameter is the thread for which to wait, the identifier that `pthread_create` filled in for you. The second argument is a pointer to a pointer that itself points to the return value from the thread. Like `pthread_create`, this function returns zero for success and an error code on failure.

Try It Out A Simple Threaded Program

This program creates a single extra thread, shows that it is sharing variables with the original thread, and gets the new thread to return a result to the original thread. Multithreaded programs don't get much simpler than this! Here is `thread1.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

void *thread_function(void *arg);

char message[] = "Hello World";

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
```

```
    printf("Thread joined, it returned %s\n", (char *)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for the CPU time");
}
```

1. To compile this, first you need to ensure that `_REENTRANT` is defined. On a few systems, you may also need to define `_POSIX_C_SOURCE`, but normally this will not be necessary.
2. Next you must ensure that the appropriate thread library is linked. In the unlikely event that you are using an older Linux distribution where NPTL is not the default thread library, you may want to consider an upgrade, although most code in this chapter is compatible with the older Linux threads implementation. An easy way to check is to look in `/usr/include/pthread.h`. If this file shows a copyright date of 2003 or later, it's almost certainly the NPTL implementation. If the date is earlier, it's probably time to get a more recent Linux installation.
3. Having identified and installed the appropriate files, you can now compile and link your program like this:

```
$ cc -D_REENTRANT -I/usr/include/nptl thread1.c
-o thread1 -L/usr/lib/nptl -lpthread
```

If NPTL is the default on your system (which is quite likely), you almost certainly don't need the –I and –L options, and can use the simpler:

```
$ cc -D_REENTRANT thread1.c -o thread1 -lpthread
```

We will use the simpler version of the compile line throughout this chapter.

4. When you run this program, you should see the following:

```
$ ./thread1
Waiting for thread to finish...
thread_function is running. Argument was Hello World
Thread joined, it returned Thank you for the CPU time
Message is now Bye!
```

It's worth spending a little time on understanding this program, because we will be using it as the basis for most of the examples in this chapter.

How It Works

You declare a prototype for the function that the thread will call when you create it:

```
void *thread_function(void *arg);
```

As required by `pthread_create`, it takes a pointer to `void` as its only argument and returns a pointer to `void`. (We will come to the implementation of `thread_function` in a moment.)

In `main`, you declare some variables and then call `pthread_create` to start running your new thread.

```
pthread_t a_thread;
void *thread_result;

res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
```

You pass the address of a `pthread_t` object that you can use to refer to the thread afterward. You don't want to modify the default thread attributes, so you pass `NULL` as the second parameter. The final two parameters are the function to call and a parameter to pass to it.

If the call succeeds, two threads will now be running. The original thread (`main`) continues and executes the code after `pthread_create`, and a new thread starts executing in the imaginatively named `thread_function`.

The original thread checks that the new thread has started and then calls `pthread_join`.

```
res = pthread_join(a_thread, &thread_result);
```

Here you pass the identifier of the thread that you are waiting to join and a pointer to a result. This function will wait until the other thread terminates before it returns. It then prints the return value from the thread and the contents of a variable, and exits.

The new thread starts executing at the start of `thread_function`, which prints out its arguments, sleeps for a short period, updates global variables, and then exits, returning a string to the main thread. The new thread writes to the same array, `message`, to which the original thread has access. If you had called `fork` rather than `pthread_create`, the array would have been a copy of the original `message`, rather than the same array.

Simultaneous Execution

The next example shows you how to write a program that checks that the execution of two threads occurs simultaneously. (Of course, if you are using a single-processor system, the CPU would be cleverly switched between the threads, rather than having the hardware simultaneously execute both threads using separate processor cores). Because you haven't yet met any of the thread synchronization functions, this will be a very inefficient program that does what is known as a *polling* between the two threads. Again, you will make use of the fact that everything except local function variables are shared between the different threads in a process.

Try It Out — Simultaneous Execution of Two Threads

The program you create in this section, `thread2.c`, is created by slightly modifying `thread1.c`. You add an extra file scope variable to test which thread is running:

The full code files for the examples are provided in the downloads from the book's website.

```
int run_now = 1;
```

Set `run_now` to `1` when the main function is executing and to `2` when your new thread is executing.

In the `main` function, after the creation of the new thread, add the following code:

```
    int print_count1 = 0;

    while(print_count1++ < 20) {
        if (run_now == 1) {
            printf("1");
            run_now = 2;
        }
        else {
            sleep(1);
        }
    }
```

If `run_now` is `1`, print `"1"` and set it to `2`. Otherwise, you sleep briefly and check the value again. You are waiting for the value to change to `1` by checking over and over again. This is called a *busy wait*, although here it is slowed down by sleeping for a second between checks. You'll see a better way to do this later in the chapter.

In `thread_function`, where your new thread is executing, you do much the same but with the values reversed:

```
    int print_count2 = 0;

    while(print_count2++ < 20) {
        if (run_now == 2) {
            printf("2");
            run_now = 1;
        }
        else {
            sleep(1);
        }
    }
```

You remove the parameter passing and return value passing because you are no longer interested in them.

When you run the program, you see the following output. (You may find that it takes a few seconds for the program to produce output, particularly on a single-core CPU machine.)

```
$ cc -D_REENTRANT thread2.c -o thread2 -lpthread
$ ./thread2
12121212121212121212
Waiting for thread to finish...
Thread joined
```

How It Works

Each thread tells the other one to run by setting the `run_now` variable and then waits till the other thread has changed its value before running again. This shows that execution passes between the two threads automatically and again illustrates the point that both threads are sharing the `run_now` variable.

Synchronization

In the previous section, you saw that both threads execute together, but the method of switching between them was clumsy and very inefficient. Fortunately, there is a set functions specifically designed to provide better ways to control the execution of threads and access to critical sections of code.

We look at two basic methods here: *semaphores*, which act as gatekeepers around a piece of code, and *mutexes*, which act as a mutual exclusion (hence the name mutex) device to protect sections of code. These methods are similar; indeed, one can be implemented in terms of the other. However, there are some cases where the semantics of the problem suggest that one is more expressive than the other. For example, controlling access to some shared memory, which only one thread can access at a time, would most naturally involve a mutex. However, controlling access to a set of identical objects as a whole, such as giving one telephone line out of a set of five available lines to a thread, suits a counting semaphore better. Which one you choose depends on personal preference and the most appropriate mechanism for your program.

Synchronization with Semaphores

There are two sets of interface functions for semaphores: One is taken from POSIX Realtime Extensions and used for threads, and the other is known as System V semaphores, which are commonly used for process synchronization. (We discuss the second type in Chapter 14.) The two are not guaranteed to be interchangeable and, although very similar, use different function calls.

Dijkstra, a Dutch computer scientist, first conceived the concept of semaphores. A semaphore is a special type of variable that can be incremented or decremented, but crucial access to the variable is guaranteed to be atomic, even in a multithreaded program. This means that if two (or more) threads in a program attempt to change the value of a semaphore, the system guarantees that all the operations will in fact take place in sequence. With normal variables the result of conflicting operations from different threads within the same program is undefined.

In this section we look at the simplest type of semaphore, a *binary* semaphore that takes only values `0` or `1`. There is also a more general semaphore, a *counting* semaphore that takes a wider range of values. Normally, semaphores are used to protect a piece of code so that only one thread of execution can run it at any one time. For this job a binary semaphore is needed. Occasionally, you want to permit a limited number of threads to execute a given piece of code; for this you would use a counting semaphore. Because counting semaphores are much less common, we won't consider them further here except to say that they are just a logical extension of a binary semaphore and that the actual function calls needed are identical.

The semaphore functions do not start with `pthread_`, as most thread-specific functions do, but with `sem_`. Four basic semaphore functions are used in threads. They are all quite simple.

A semaphore is created with the `sem_init` function, which is declared as follows:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

This function initializes a semaphore object pointed to by `sem`, sets its sharing option (which we discuss more in a moment), and gives it an initial integer value. The `pshared` parameter controls the type of semaphore. If the value of `pshared` is `0`, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are interested only in semaphores that are not shared between processes. At the time of writing, Linux doesn't support this sharing, and passing a nonzero value for `pshared` will cause the call to fail.

The next pair of functions controls the value of the semaphore and is declared as follows:

```
#include <semaphore.h>

int sem_wait(sem_t * sem);

int sem_post(sem_t * sem);
```

These both take a pointer to the semaphore object initialized by a call to `sem_init`.

The `sem_post` function atomically increases the value of the semaphore by 1. *Atomically* here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time. If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2.

The `sem_wait` function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call `sem_wait` on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If `sem_wait` is called on a semaphore with a value of `0`, the function will wait until some other thread has incremented the value so that it is no longer `0`. If two threads are both waiting in `sem_wait` for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic "test and set" ability in a single function is what makes semaphores so valuable.

> *There is another semaphore function,* `sem_trywait`*, that is the nonblocking partner of* `sem_wait`*. We don't discuss it further here; you can find more details in the manual pages.*

The last semaphore function is `sem_destroy`. This function tidies up the semaphore when you have finished with it. It is declared as follows:

```
#include <semaphore.h>

int sem_destroy(sem_t * sem);
```

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If you attempt to destroy a semaphore for which some thread is waiting, you will get an error.

Like most Linux functions, these functions all return `0` on success.

Try It Out A Thread Semaphore

This code, `thread3.c`, is also based on `thread1.c`. Because a lot has changed, we present it in full.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = sem_init(&bin_sem, 0, 0);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Input some text. Enter 'end' to finish\n");
    while(strncmp("end", work_area, 3) != 0) {
        fgets(work_area, WORK_SIZE, stdin);
        sem_post(&bin_sem);
    }
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sem_wait(&bin_sem);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        sem_wait(&bin_sem);
```

```
    }
    pthread_exit(NULL);
}
```

The first important change is the inclusion of `semaphore.h` to provide access to the semaphore functions. Then you declare a semaphore and some variables and initialize the semaphore *before* you create your new thread.

```
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = sem_init(&bin_sem, 0, 0);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }
```

Note that the initial value of the semaphore is set to 0.

In the function `main`, after you have started the new thread, you read some text from the keyboard, load your work area, and then increment the semaphore with `sem_post`.

```
    printf("Input some text. Enter 'end' to finish\n");
    while(strncmp("end", work_area, 3) != 0) {
        fgets(work_area, WORK_SIZE, stdin);
        sem_post(&bin_sem);
    }
```

In the new thread, you wait for the semaphore and then count the characters from the input.

```
    sem_wait(&bin_sem);
    while(strncmp("end", work_area, 3) != 0) {
      printf("You input %d characters\n", strlen(work_area) -1);
      sem_wait(&bin_sem);
    }
```

While the semaphore is set, you are waiting for keyboard input. When you have some input, you release the semaphore, allowing the second thread to count the characters before the first thread reads the keyboard again.

Again both threads share the same `work_area` array. Again, we have omitted some error checking, such as the returns from `sem_wait` to make the code samples more succinct and easier to follow. However, in production code you should always check for error returns unless there is a very good reason to omit this check.

Give the program a run:

```
$ cc -D_REENTRANT thread3.c -o thread3 -lpthread
$ ./thread3
Input some text. Enter 'end' to finish
The Wasp Factory
You input 16 characters
Iain Banks
You input 10 characters
end

Waiting for thread to finish...
Thread joined
```

In threaded programs, timing faults are always hard to find, but the program seems resilient to both quick input of text and more leisurely pauses.

How It Works

When you initialize the semaphore, you set its value to `0`. Thus, when the thread's function starts, the call to `sem_wait` blocks and waits for the semaphore to become nonzero.

In the `main` thread, you wait until you have some text and then increment the semaphore with `sem_post`, which immediately allows the other thread to return from its `sem_wait` and start executing. Once it has counted the characters, it again calls `sem_wait` and is blocked until the `main` thread again calls `sem_post` to increment the semaphore.

It is easy to overlook subtle design errors that result in subtle errors. Let's modify the program slightly to `thread3a.c` to pretend that text input from the keyboard is sometimes replaced with automatically available text. Modify the reading loop in the `main` to this:

```
    printf("Input some text. Enter 'end' to finish\n");
    while(strncmp("end", work_area, 3) != 0) {
      if (strncmp(work_area, "FAST", 4) == 0) {
        sem_post(&bin_sem);
        strcpy(work_area, "Wheeee...");
      } else {
        fgets(work_area, WORK_SIZE, stdin);
      }
      sem_post(&bin_sem);
    }
```

Now if you type **FAST**, the program calls `sem_post` to allow the character counter to run, but immediately updates `work_area` with something different.

```
$ cc -D_REENTRANT thread3a.c -o thread3a -lpthread
$ ./thread3a
Input some text. Enter 'end' to finish
Excession
You input 9 characters
```

```
FAST
You input 7 characters
You input 7 characters
You input 7 characters
end

Waiting for thread to finish...
Thread joined
```

The problem is that the program was relying on text input from the program taking so long that there was time for the other thread to count the words before the main thread was ever ready to give it more words to count. When you tried to give it two different sets of words to count in quick succession (**FAST** from the keyboard and then **Wheeee...** automatically), there was no time for the second thread to execute. However, the semaphore had been incremented more than once, so the counter thread just kept counting the words and decreasing the semaphore until it became zero again.

This shows just how careful you need to be with timing considerations in multithreaded programs. It's possible to fix the program by using an extra semaphore to make the main thread wait until the counter thread has had the chance to finish its counting, but an easier way is to use a *mutex*, which we look at next.

Synchronization with Mutexes

The other way of synchronizing access in multithreaded programs is with *mutexes* (short for *mutual exclusions*), which act by allowing the programmer to "lock" an object so that only one thread can access it. To control access to a critical section of code you lock a mutex before entering the code section and then unlock it when you have finished.

The basic functions required to use mutexes are very similar to those needed for semaphores. They are declared as follows:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr);

int pthread_mutex_lock(pthread_mutex_t *mutex));

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

As usual, `0` is returned for success, and on failure an error code is returned, but `errno` is not set; you must use the return code.

As with semaphores, they all take a pointer to a previously declared object, in this case a `pthread_mutex_t`. The extra attribute parameter `pthread_mutex_init` allows you to provide attributes for the mutex, which control its behavior. The attribute type by default is "fast." This has the slight drawback that, if your program tries to call `pthread_mutex_lock` on a mutex that it has already locked, the program will block. Because the thread that holds the lock is the one that is now blocked, the mutex can never be unlocked and the program is deadlocked. It is possible to alter the attributes of the mutex so that it either checks for this and returns an error or acts recursively and allows multiple locks by the same thread if there are the same number of unlocks afterward.

Setting the attribute of a mutex is beyond the scope of this book, so we will pass `NULL` for the attribute pointer and use the default behavior. You can find more about changing the attributes by reading the manual page for `pthread_mutex_init`.

Try It Out A Thread Mutex

Again, this is a modification of the original `thread1.c` but heavily modified. This time, you will be extra careful about access to your critical variables and use a mutex to ensure they are ever accessed by only one thread at any one time. To keep the example code easy to read, we have omitted some error checking on the returns from mutex lock and unlock. In production code, we would check these return values. Here is the new program, `thread4.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex; /* protects both work_area and time_to_exit */

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_mutex_init(&work_mutex, NULL);
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_lock(&work_mutex);
    printf("Input some text. Enter 'end' to finish\n");
    while(!time_to_exit) {
        fgets(work_area, WORK_SIZE, stdin);
        pthread_mutex_unlock(&work_mutex);
        while(1) {
            pthread_mutex_lock(&work_mutex);
            if (work_area[0] != '\0') {
                pthread_mutex_unlock(&work_mutex);
                sleep(1);
            }
            else {
                break;
            }
```

```
        }
    }
    pthread_mutex_unlock(&work_mutex);
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

```
$ cc -D_REENTRANT thread4.c -o thread4 -lpthread
$ ./thread4
Input some text. Enter 'end' to finish
Whit
You input 4 characters
The Crow Road
You input 13 characters
end

Waiting for thread to finish...
Thread joined
```

How It Works

You start by declaring a mutex, your work area, and this time, an additional variable: `time_to_exit`.

```
pthread_mutex_t work_mutex; /* protects both work_area and time_to_exit */

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;
```

Then initialize the mutex:

```
res = pthread_mutex_init(&work_mutex, NULL);
if (res != 0) {
    perror("Mutex initialization failed");
    exit(EXIT_FAILURE);
}
```

Next start the new thread. Here is the code that executes in the thread function:

```
pthread_mutex_lock(&work_mutex);
while(strncmp("end", work_area, 3) != 0) {
    printf("You input %d characters\n", strlen(work_area) -1);
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while (work_area[0] == '\0' ) {
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
    }
}
time_to_exit = 1;
work_area[0] = '\0';
pthread_mutex_unlock(&work_mutex);
```

First, the new thread tries to lock the mutex. If it's already locked, the call will block until it is released. Once you have access, you check to see whether you are being requested to exit. If you are requested to exit, simply set `time_to_exit`, zap the first character of the work area, and exit.

If you don't want to exit, count the characters and then zap the first character to a null. You use the first character being null as a way of telling the reader program that you have finished the counting. You then unlock the mutex and wait for the main thread to run. Periodically, you attempt to lock the mutex and, when you succeed, check whether the main thread has given you any more work to do. If it hasn't, you unlock the mutex and wait some more. If it has, you count the characters and go around the loop again.

Here is the main thread:

```
pthread_mutex_lock(&work_mutex);
printf("Input some text. Enter 'end' to finish\n");
while(!time_to_exit) {
    fgets(work_area, WORK_SIZE, stdin);
    pthread_mutex_unlock(&work_mutex);
    while(1) {
        pthread_mutex_lock(&work_mutex);
        if (work_area[0] != '\0') {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
        }
        else {
            break;
        }
```

```
        }
    }
    pthread_mutex_unlock(&work_mutex);
```

This is quite similar. You lock the work area so that you can read text into it and then unlock it to allow the other thread access to count the words. Periodically, you relock the mutex, check whether the words have been counted (`work_area[0]` set to a null), and release the mutex if you need to wait longer. As we noted earlier, this kind of polling for an answer is generally not good programming practice, and in the real world you would probably have used a semaphore to avoid this. However, the code served its purpose as an example of using a mutex.

Thread Attributes

When we first looked at threads, we did not discuss the more advanced topic of thread attributes. Now that we have covered the key topic of synchronizing threads, we can come back and look at these more advanced features of threads themselves. There are quite a few attributes of threads that you can control; here we are only going to look at those that you are most likely to need. You can find details of the others in the manual pages.

In all of the previous examples, you had to resynchronize your threads using `pthread_join` before allowing the program to exit. You needed to do this if you wanted to allow one thread to return data to the thread that created it. Sometimes you neither need the second thread to return information to the main thread nor want the main thread to wait for it.

Suppose that you create a second thread to spool a backup copy of a data file that is being edited while the main thread continues to service the user. When the backup has finished, the second thread can just terminate. There is no need for it to rejoin the main thread.

You can create threads that behave like this. They are called *detached threads*, and you create them by modifying the thread attributes or by calling `pthread_detach`. Because we want to demonstrate attributes, we use the former method here.

The most important function that you need is `pthread_attr_init`, which initializes a thread attribute object.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

Once again, `0` is returned for success and an error code is returned on failure.

There is also a destroy function: `pthread_attr_destroy`. Its purpose is to allow clean destruction of the attribute object. Once the object has been destroyed, it cannot be used again until it has been reinitialized.

When you have a thread attribute object initialized, there are many additional functions that you can call to set different attribute behaviors. We list the main ones here (you can find the complete list in the man pages, usually under the `pthread.h` entry), but look closely at only two, `detachedstate` and `schedpolicy`:

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

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

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);

int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param
*param);

int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param
*param);

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);

int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inherit);

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);

int pthread_attr_setstacksize(pthread_attr_t *attr, int scope);

int pthread_attr_getstacksize(const pthread_attr_t *attr, int *scope);
```

As you can see, there are quite a few attributes you can use, but fortunately you will generally get by without ever having to use most of these.

- `detachedstate`: This attribute allows you to avoid the need for threads to rejoin. As with most of these `_set` functions, it takes a pointer to the attribute and a flag to determine the state required. The two possible flag values for `pthread_attr_setdetachstate` are `PTHREAD_CREATE_JOINABLE` and `PTHREAD_CREATE_DETACHED`. By default, the attribute will have the value `PTHREAD_CREATE_JOINABLE` so that you can allow the two threads to join. If the state is set to `PTHREAD_CREATE_DETACHED`, you cannot call `pthread_join` to recover the exit state of another thread.
- `schedpolicy`: This controls how threads are scheduled. The options are `SCHED_OTHER`, `SCHED_RP`, and `SCHED_FIFO`. By default, the attribute is `SCHED_OTHER`. The other two types of scheduling are available only to processes running with superuser permissions, because they both have real-time scheduling but with slightly different behavior. `SCHED_RR` uses a round-robin scheduling scheme, and `SCHED_FIFO` uses a "first in, first out" policy. Discussion of these is beyond the scope of this book.
- `schedparam`: This is a partner to `schedpolicy` and allows control over the scheduling of threads running with schedule policy `SCHED_OTHER`. We take a look at an example of this a bit later in the chapter.

- `inheritsched`: This attribute takes two possible values: `PTHREAD_EXPLICIT_SCHED` and `PTHREAD_INHERIT_SCHED`. By default, the value is `PTHREAD_EXPLICIT_SCHED`, which means scheduling is explicitly set by the attributes. By setting it to `PTHREAD_INHERIT_SCHED`, a new thread will instead use the parameters that its creator thread was using.
- `scope`: This attribute controls how the scheduling of a thread is calculated. Because Linux currently supports only the value `PTHREAD_SCOPE_SYSTEM`, we will not look at this further here.
- `stacksize`: This attribute controls the thread creation stack size, set in bytes. This is part of the "optional" section of the specification and is supported only on implementations where `_POSIX_THREAD_ATTR_STACKSIZE` is defined. Linux implements threads with a large amount of stack by default, so the feature is generally redundant on Linux.

Try It Out — Setting the Detached State Attribute

For the detached thread example, `thread5.c`, you create a thread attribute, set it to be detached, and then create a thread using the attribute. Now when the child thread has finished, it calls `pthread_exit` in the normal way. This time, however, the originating thread no longer waits for the thread that it created to rejoin. In this example, you use a simple `thread_finished` flag to allow the main thread to detect whether the child has finished and to show that the threads are still sharing variables.

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

void *thread_function(void *arg);

char message[] = "Hello World";
int thread_finished = 0;

int main() {
    int res;
    pthread_t a_thread;

    pthread_attr_t thread_attr;

    res = pthread_attr_init(&thread_attr);
    if (res != 0) {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
    if (res != 0) {
        perror("Setting detached attribute failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, &thread_attr,
thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
```

```
    (void)pthread_attr_destroy(&thread_attr);
    while(!thread_finished) {
        printf("Waiting for thread to say it's finished...\n");
        sleep(1);
    }
    printf("Other thread finished, bye!\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(4);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    pthread_exit(NULL);
}
```

There are no surprises in the output:

```
$ ./thread5
Waiting for thread to say it's finished...
thread_function is running. Argument was Hello World
Waiting for thread to say it's finished...
Waiting for thread to say it's finished...
Waiting for thread to say it's finished...
Second thread setting finished flag, and exiting now
Other thread finished, bye!
```

As you can see, setting the detached state allowed the secondary thread to complete independently, without the originating thread needing to wait for it.

How It Works

The two important sections of code are

```
pthread_attr_t thread_attr;

res = pthread_attr_init(&thread_attr);
if (res != 0) {
    perror("Attribute creation failed");
    exit(EXIT_FAILURE);
}
```

which declares a thread attribute and initializes it, and

```
res = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
if (res != 0) {
    perror("Setting detached attribute failed");
    exit(EXIT_FAILURE);
}
```

which sets the attribute values to have the detached state.

The other slight differences are creating the thread, passing the address of the attributes,

```
    res = pthread_create(&a_thread, &thread_attr, thread_function, (void
*)message);
```

and, for completeness, destroying the attributes when you have used them:

```
    pthread_attr_destroy(&thread_attr);
```

Thread Attributes Scheduling

Let's take a look at a second thread attribute you might want to change: scheduling. Changing the scheduling attribute is very similar to setting the detached state, but there are two more functions that you can use to find the available priority levels, `sched_get_priority_max` and `sched_get_priority_min`.

Try It Out — Scheduling

Because this `thread6.c` is very similar to the previous example, we'll just look at the differences.

1. First, you need some additional variables:

```
    int max_priority;
    int min_priority;
    struct sched_param scheduling_value;
```

2. After you have set the detached attribute, you set the scheduling policy:

```
    res = pthread_attr_setschedpolicy(&thread_attr, SCHED_OTHER);
    if (res != 0) {
        perror("Setting scheduling policy failed");
        exit(EXIT_FAILURE);
    }
```

3. Next find the range of priorities that are allowed:

```
    max_priority = sched_get_priority_max(SCHED_OTHER);
    min_priority = sched_get_priority_min(SCHED_OTHER);
```

4. and set one:

```
    scheduling_value.sched_priority = min_priority;
    res = pthread_attr_setschedparam(&thread_attr, &scheduling_value);
    if (res != 0) {
        perror("Setting scheduling priority failed");
        exit(EXIT_FAILURE);
    }
```

When you run it, the output you get is:

```
$ ./thread6
Waiting for thread to say it's finished...
```

```
thread_function is running. Argument was Hello World
Waiting for thread to say it's finished...
Waiting for thread to say it's finished...
Waiting for thread to say it's finished...
Second thread setting finished flag, and exiting now
Other thread finished, bye!
```

How It Works

This is very similar to setting a detached state attribute, except that you set the scheduling policy instead.

Canceling a Thread

Sometimes, you want one thread to be able to ask another thread to terminate, rather like sending it a signal. There is a way to do this with threads, and, in parallel with signal handling, threads get a way of modifying how they behave when they are asked to terminate.

Let's look first at the function to request a thread to terminate:

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

This is pretty straightforward: Given a thread identifier, you can request that it be canceled. On the receiving end of the cancel request, things are slightly more complicated, but not much. A thread can set its cancel state using `pthread_setcancelstate`.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

The first parameter is either `PTHREAD_CANCEL_ENABLE`, which allows it to receive cancel requests, or `PTHREAD_CANCEL_DISABLE`, which causes them to be ignored. The `oldstate` pointer allows the previous state to be retrieved. If you are not interested, you can simply pass `NULL`. If cancel requests are accepted, there is a second level of control the thread can take, the cancel type, which is set with `pthread_setcanceltype`.

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

The type can take one of two values, `PTHREAD_CANCEL_ASYNCHRONOUS`, which causes cancellation requests to be acted upon immediately, and `PTHREAD_CANCEL_DEFERRED`, which makes cancellation requests wait until the thread executes one of these functions: `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_testcancel`, `sem_wait`, or `sigwait`.

We have not covered all of these calls in this chapter, because not all are generally needed. As ever, you can find more details in the manual pages.

> *According to the POSIX standard, other system calls that may block, such as* `read`, `wait`, *and so on, should also be cancellation points. At the time of this writing, support for this in Linux seems to be incomplete. Some experimentation does, however, suggest that some blocked calls, such as* `sleep`, *do allow cancellation to take place. To be on the safe side, you may want to add some* `pthread_testcancel` *calls in code that you expect to be canceled.*

Again, the `oldtype` allows the previous state to be retrieved, or a `NULL` can be passed if you are not interested in knowing the previous state. By default, threads start with the cancellation state `PTHREAD_CANCEL_ENABLE` and the cancellation type `PTHREAD_CANCEL_DEFERRED`.

Try It Out Canceling a Thread

The program `thread7.c` is derived, yet again, from `thread1.c`. This time, the main thread sends a cancel request to the thread that it has created.

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

void *thread_function(void *arg);

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    sleep(3);
    printf("Canceling thread...\n");
    res = pthread_cancel(a_thread);
    if (res != 0) {
        perror("Thread cancelation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

```
void *thread_function(void *arg) {
    int i, res;
    res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (res != 0) {
        perror("Thread pthread_setcancelstate failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    if (res != 0) {
        perror("Thread pthread_setcanceltype failed");
        exit(EXIT_FAILURE);
    }
    printf("thread_function is running\n");
    for(i = 0; i < 10; i++) {
        printf("Thread is still running (%d)...\n", i);
        sleep(1);
    }
    pthread_exit(0);
}
```

When you run this, you get the following output, showing that the thread is canceled:

```
$ ./thread7
thread_function is running
Thread is still running (0)...
Thread is still running (1)...
Thread is still running (2)...
Canceling thread...
Waiting for thread to finish...
$
```

How It Works

After the new thread has been created in the usual way, the main thread sleeps (to allow the new thread some time to get started) and then issues a cancel request:

```
sleep(3);
printf("Cancelling thread...\n");
res = pthread_cancel(a_thread);
if (res != 0) {
    perror("Thread cancelation failed");
    exit(EXIT_FAILURE);
}
```

In the created thread, you first set the cancel state to allow canceling:

```
res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
if (res != 0) {
    perror("Thread pthread_setcancelstate failed");
    exit(EXIT_FAILURE);
}
```

Then you set the cancel type to be deferred:

```
res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
if (res != 0) {
    perror("Thread pthread_setcanceltype failed");
    exit(EXIT_FAILURE);
}
```

Finally, the thread waits around to be canceled:

```
for(i = 0; i < 10; i++) {
    printf("Thread is still running (%d)...\n", i);
    sleep(1);
}
```

Threads in Abundance

Up until now, we have always had the normal thread of execution of a program create just one other thread. However, we don't want you to think that you can create only one extra thread.

Try It Out — Many Threads

For the final example in this chapter, `thread8.c`, we show how to create several threads in the same program and then collect them again in an order different from that in which they were started.

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

#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {
    int res;
    pthread_t a_thread[NUM_THREADS];
    void *thread_result;
    int lots_of_threads;

    for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {
        res = pthread_create(&(a_thread[lots_of_threads]),
NULL, thread_function, (void *)&lots_of_threads);
        if (res != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
        sleep(1);
    }
    printf("Waiting for threads to finish...\n");
```

```
    for(lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0;
lots_of_threads--) {
        res = pthread_join(a_thread[lots_of_threads], &thread_result);
        if (res == 0) {
            printf("Picked up a thread\n");
        }
        else {
            perror("pthread_join failed");
        }
    }
    printf("All done\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int my_number = *(int *)arg;
    int rand_num;

    printf("thread_function is running. Argument was %d\n", my_number);
    rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
    sleep(rand_num);
    printf("Bye from %d\n", my_number);
    pthread_exit(NULL);
}
```

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

```
$ ./thread8
thread_function is running. Argument was 0
thread_function is running. Argument was 1
thread_function is running. Argument was 2
thread_function is running. Argument was 3
thread_function is running. Argument was 4
Bye from 1
thread_function is running. Argument was 5
Waiting for threads to finish...
Bye from 5
Picked up a thread
Bye from 0
Bye from 2
Bye from 3
Bye from 4
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
```

As you can see, you created many threads and allowed them to finish out of sequence. There is a subtle bug in this program that makes itself evident if you remove the call to `sleep` from the loop that starts the threads. We have included it to show you just how careful you need to be when writing programs that use threads. Can you spot it? We explain in the following "How It Works" section.

How It Works

This time you create an array of thread IDs:

```
pthread_t a_thread[NUM_THREADS];
```

and loop around creating several threads:

```
for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {
    res = pthread_create(&(a_thread[lots_of_threads]), NULL,
                         thread_function, (void *)&lots_of_threads);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    sleep(1);
}
```

The threads themselves then wait for a random time before exiting:

```
void *thread_function(void *arg) {
    int my_number = *(int *)arg;
    int rand_num;

    printf("thread_function is running. Argument was %d\n", my_number);
    rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
    sleep(rand_num);
    printf("Bye from %d\n", my_number);
    pthread_exit(NULL);
}
```

While in the main (original) thread, you wait to pick them up, but not in the order in which you created them:

```
    for(lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0; lots_of_threads--)
{
        res = pthread_join(a_thread[lots_of_threads], &thread_result);
        if (res == 0) {
            printf("Picked up a thread\n");
        }
        else {
            perror("pthread_join failed");
        }
    }
```

If you try to run the program with no `sleep`, you might see some strange effects, including some threads being started with the same argument; for example, you might see output similar to this:

```
thread_function is running. Argument was 0
thread_function is running. Argument was 2
thread_function is running. Argument was 2
thread_function is running. Argument was 4
thread_function is running. Argument was 4
thread_function is running. Argument was 5
Waiting for threads to finish...
```

```
Bye from 5
Picked up a thread
Bye from 2
Bye from 0
Bye from 2
Bye from 4
Bye from 4
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
```

Did you spot why this could happen? The threads are being started using a local variable for the argument to the thread function. This variable is updated in the loop. The offending lines are

```
for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {
    res = pthread_create(&(a_thread[lots_of_threads]), NULL,
                         thread_function, (void *)&lots_of_threads);
```

If the main thread runs fast enough, it might alter the argument (`lots_of_threads`) for some of the threads. Behavior like this arises when not enough care is taken with shared variables and multiple execution paths. We did warn you that programming threads required careful attention to design! To correct the problem, you need to pass the value directly like this:

```
res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function, (void
*)lots_of_threads);
```

and of course change `thread_function`:

```
void *thread_function(void *arg) {
    int my_number = (int)arg;
```

This is shown in the program `thread8a.c`, with the changes highlighted:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {

  int res;
  pthread_t a_thread[NUM_THREADS];
  void *thread_result;
  int lots_of_threads;
```

```
    for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {

      res = pthread_create(&(a_thread[lots_of_threads]), NULL,
  thread_function, (void *)lots_of_threads);
      if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
      }
    }

    printf("Waiting for threads to finish...\n");
    for(lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0; lots_of_threads--) {
      res = pthread_join(a_thread[lots_of_threads], &thread_result);
      if (res == 0) {
        printf("Picked up a thread\n");
      } else {
        perror("pthread_join failed");
      }
    }

    printf("All done\n");

    exit(EXIT_SUCCESS);
  }

  void *thread_function(void *arg) {
      int my_number = (int)arg;
      int rand_num;

      printf("thread_function is running. Argument was %d\n", my_number);
      rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
      sleep(rand_num);
      printf("Bye from %d\n", my_number);

      pthread_exit(NULL);
  }
```

Summary

In this chapter, you learned how to create several threads of execution inside a process, where each thread shares file scope variables. You looked at the two ways that threads can control access to critical code and data, using both semaphores and mutexes. Next, you saw how to control the attributes of threads and, in particular, how you could separate them from the main thread so that it no longer had to wait for threads that it had created to complete. After a quick look at how one thread can request another to finish and at how the receiving thread can manage such requests, we presented an example of a program with many simultaneous threads executing.

We haven't had the space to cover every last function call and nuance associated with threads, but you should now have sufficient understanding to start writing your own programs with threads and to investigate the more esoteric aspects of threads by reading the manual pages.