4

Threads

Threads, like processes, are a mechanism to allow a program to do more than one thing at a time. As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute.

Conceptually, a thread exists within a process. Threads are a finer-grained unit of execution than processes. When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially. That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.

We've seen how a program can fork a child process. The child process is initially running its parent's program, with its parent's virtual memory, file descriptors, and so on copied. The child process can modify its memory, close file descriptors, and the like without affecting its parent, and vice versa. When a program creates another thread, though, nothing is copied. The creating and the created thread share the same memory space, file descriptors, and other system resources as the original. If one thread changes the value of a variable, for instance, the other thread subsequently will see the modified value. Similarly, if one thread closes a file descriptor, other threads may not read

from or write to that file descriptor. Because a process and all its threads can be executing only one program at a time, if any thread inside a process calls one of the `exec` functions, all the other threads are ended (the new program may, of course, create new threads).

GNU/Linux implements the POSIX standard thread API (known as *pthreads*). All thread functions and data types are declared in the header file `<pthread.h>`. The pthread functions are not included in the standard C library. Instead, they are in `libpthread`, so you should add `-lpthread` to the command line when you link your program.

4.1 Thread Creation

Each thread in a process is identified by a *thread ID*. When referring to thread IDs in C or C++ programs, use the type `pthread_t`.

Upon creation, each thread executes a *thread function*. This is just an ordinary function and contains the code that the thread should run. When the function returns, the thread exits. On GNU/Linux, thread functions take a single parameter, of type `void*`, and have a `void*` return type. The parameter is the *thread argument*: GNU/Linux passes the value along to the thread without looking at it. Your program can use this parameter to pass data to a new thread. Similarly, your program can use the return value to pass data from an exiting thread back to its creator.

The `pthread_create` function creates a new thread. You provide it with the following:

1. A pointer to a `pthread_t` variable, in which the thread ID of the new thread is stored.
2. A pointer to a *thread attribute* object. This object controls details of how the thread interacts with the rest of the program. If you pass `NULL` as the thread attribute, a thread will be created with the default thread attributes. Thread attributes are discussed in Section 4.1.5, "Thread Attributes."
3. A pointer to the thread function. This is an ordinary function pointer, of this type:

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

4. A thread argument value of type `void*`. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

A call to `pthread_create` returns immediately, and the original thread continues executing the instructions following the call. Meanwhile, the new thread begins executing the thread function. Linux schedules both threads asynchronously, and your program must not rely on the relative order in which instructions are executed in the two threads.

The program in Listing 4.1 creates a thread that prints x's continuously to standard error. After calling `pthread_create`, the main thread prints o's continuously to standard error.

Listing 4.1 **(*thread-create.c*) Create a Thread**

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

/* Prints x's to stderr.  The parameter is unused.  Does not return.  */

void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  pthread_t thread_id;
  /* Create a new thread.  The new thread will run the print_xs
     function.  */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Print o's continuously to stderr.  */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

Compile and link this program using the following code:

```
% cc -o thread-create thread-create.c -lpthread
```

Try running it to see what happens. Notice the unpredictable pattern of x's and o's as Linux alternately schedules the two threads.

Under normal circumstances, a thread exits in one of two ways. One way, as illustrated previously, is by returning from the thread function. The return value from the thread function is taken to be the return value of the thread. Alternately, a thread can exit explicitly by calling `pthread_exit`. This function may be called from within the thread function or from some other function called directly or indirectly by the thread function. The argument to `pthread_exit` is the thread's return value.

4.1.1 Passing Data to Threads

The thread argument provides a convenient method of passing data to threads. Because the type of the argument is `void*`, though, you can't pass a lot of data directly via the argument. Instead, use the thread argument to pass a pointer to some structure or array of data. One commonly used technique is to define a structure for each thread function, which contains the "parameters" that the thread function expects.

Using the thread argument, it's easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data.

The program in Listing 4.2 is similar to the previous example. This one creates two new threads, one to print x's and the other to print o's. Instead of printing infinitely, though, each thread prints a fixed number of characters and then exits by returning from the thread function. The same thread function, `char_print`, is used by both threads, but each is configured differently using `struct char_print_parms`.

Listing 4.2 **(*thread-create2*) Create Two Threads**

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

/* Parameters to print_function.  */

struct char_print_parms
{
  /* The character to print.  */
  char character;
  /* The number of times to print it.  */
  int count;
};

/* Prints a number of characters to stderr, as given by PARAMETERS,
   which is a pointer to a struct char_print_parms.  */

void* char_print (void* parameters)
{
  /* Cast the cookie pointer to the right type.  */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;

  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  pthread_t thread1_id;
```

```
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  /* Create a new thread to print 30,000 'x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20,000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  return 0;
}
```

But wait! The program in Listing 4.2 has a serious bug in it. The main thread (which runs the `main` function) creates the thread parameter structures (`thread1_args` and `thread2_args`) as local variables, and then passes pointers to these structures to the threads it creates. What's to prevent Linux from scheduling the three threads in such a way that `main` finishes executing before either of the other two threads are done? *Nothing!* But if this happens, the memory containing the thread parameter structures will be deallocated while the other two threads are still accessing it.

4.1.2 Joining Threads

One solution is to force `main` to wait until the other two threads are done. What we need is a function similar to `wait` that waits for a thread to finish instead of a process. That function is `pthread_join`, which takes two arguments: the thread ID of the thread to wait for, and a pointer to a `void*` variable that will receive the finished thread's return value. If you don't care about the thread return value, pass `NULL` as the second argument.

Listing 4.3 shows the corrected `main` function for the buggy example in Listing 4.2. In this version, `main` does not exit until both of the threads printing x's and o's have completed, so they are no longer using the argument structures.

Listing 4.3 **Revised *Main* Function for *thread-create2.c***

```
int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;
```

continues

Listing 4.3 **Continued**

```
  /* Create a new thread to print 30,000 x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20,000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  /* Make sure the first thread has finished.  */
  pthread_join (thread1_id, NULL);
  /* Make sure the second thread has finished.  */
  pthread_join (thread2_id, NULL);

  /* Now we can safely return.  */
  return 0;
}
```

The moral of the story: Make sure that any data you pass to a thread by reference is not deallocated, *even by a different thread*, until you're sure that the thread is done with it. This is true both for local variables, which are deallocated when they go out of scope, and for heap-allocated variables, which you deallocate by calling `free` (or using `delete` in C++).

4.1.3 Thread Return Values

If the second argument you pass to `pthread_join` is non-null, the thread's return value will be placed in the location pointed to by that argument. The thread return value, like the thread argument, is of type `void*`. If you want to pass back a single `int` or other small number, you can do this easily by casting the value to `void*` and then casting back to the appropriate type after calling `pthread_join`.[1]

The program in Listing 4.4 computes the *n*th prime number in a separate thread. That thread returns the desired prime number as its thread return value. The main thread, meanwhile, is free to execute other code. Note that the successive division algorithm used in `compute_prime` is quite inefficient; consult a book on numerical algorithims if you need to compute many prime numbers in your programs.

1. Note that this is not portable, and it's up to you to make sure that your value can be cast safely to `void*` and back without losing bits.

Listing 4.4 (*primes.c*) Compute Prime Numbers in a Thread

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

/* Compute successive prime numbers (very inefficiently).  Return the
   Nth prime number, where N is the value pointed to by *ARG.  */

void* compute_prime (void* arg)
{
  int candidate = 2;
  int n = *((int*) arg);

  while (1) {
    int factor;
    int is_prime = 1;

    /* Test primality by successive division.  */
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor == 0) {
        is_prime = 0;
        break;
      }
    /* Is this the prime number we're looking for?  */
    if (is_prime) {
      if (--n == 0)
        /* Return the desired prime number as the thread return value.  */
        return (void*) candidate;
    }
    ++candidate;
  }
  return NULL;
}

int main ()
{
  pthread_t thread;
  int which_prime = 5000;
  int prime;

  /* Start the computing thread, up to the 5,000th prime number.  */
  pthread_create (&thread, NULL, &compute_prime, &which_prime);
  /* Do some other work here...  */
  /* Wait for the prime number thread to complete, and get the result.  */
  pthread_join (thread, (void*) &prime);
  /* Print the largest prime it computed.  */
  printf("The %dth prime number is %d.\n", which_prime, prime);
  return 0;
}
```

4.1.4 More on Thread IDs

Occasionally, it is useful for a sequence of code to determine which thread is executing it. The `pthread_self` function returns the thread ID of the thread in which it is called. This thread ID may be compared with another thread ID using the `pthread_equal` function.

These functions can be useful for determining whether a particular thread ID corresponds to the current thread. For instance, it is an error for a thread to call `pthread_join` to join itself. (In this case, `pthread_join` would return the error code `EDEADLK`.) To check for this beforehand, you might use code like this:

```
if (!pthread_equal (pthread_self (), other_thread))
  pthread_join (other_thread, NULL);
```

4.1.5 Thread Attributes

Thread attributes provide a mechanism for fine-tuning the behavior of individual threads. Recall that `pthread_create` accepts an argument that is a pointer to a thread attribute object. If you pass a null pointer, the default thread attributes are used to configure the new thread. However, you may create and customize a thread attribute object to specify other values for the attributes.

To specify customized thread attributes, you must follow these steps:

1. Create a `pthread_attr_t` object. The easiest way is simply to declare an automatic variable of this type.
2. Call `pthread_attr_init`, passing a pointer to this object. This initializes the attributes to their default values.
3. Modify the attribute object to contain the desired attribute values.
4. Pass a pointer to the attribute object when calling `pthread_create`.
5. Call `pthread_attr_destroy` to release the attribute object. The `pthread_attr_t` variable itself is not deallocated; it may be reinitialized with `pthread_attr_init`.

A single thread attribute object may be used to start several threads. It is not necessary to keep the thread attribute object around after the threads have been created.

For most GNU/Linux application programming tasks, only one thread attribute is typically of interest (the other available attributes are primarily for specialty real-time programming). This attribute is the thread's *detach state*. A thread may be created as a *joinable thread* (the default) or as a *detached thread*. A joinable thread, like a process, is not automatically cleaned up by GNU/Linux when it terminates. Instead, the thread's exit state hangs around in the system (kind of like a zombie process) until another thread calls `pthread_join` to obtain its return value. Only then are its resources released. A detached thread, in contrast, is cleaned up automatically when it terminates. Because a detached thread is immediately cleaned up, another thread may not synchronize on its completion by using `pthread_join` or obtain its return value.

To set the detach state in a thread attribute object, use `pthread_attr_setdetachstate`. The first argument is a pointer to the thread attribute object, and the second is the desired detach state. Because the joinable state is the default, it is necessary to call this only to create detached threads; pass `PTHREAD_CREATE_DETACHED` as the second argument.

The code in Listing 4.5 creates a detached thread by setting the detach state thread attribute for the thread.

Listing 4.5 **(*detached.c*) Skeleton Program That Creates a Detached Thread**

```
#include <pthread.h>

void* thread_function (void* thread_arg)
{
  /* Do work here...  */
}

int main ()
{
  pthread_attr_t attr;
  pthread_t thread;

  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  pthread_create (&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);

  /* Do work here...  */

  /* No need to join the second thread.  */
  return 0;
}
```

Even if a thread is created in a joinable state, it may later be turned into a detached thread. To do this, call `pthread_detach`. Once a thread is detached, it cannot be made joinable again.

4.2 Thread Cancellation

Under normal circumstances, a thread terminates when it exits normally, either by returning from its thread function or by calling `pthread_exit`. However, it is possible for a thread to request that another thread terminate. This is called *canceling* a thread.

To cancel a thread, call `pthread_cancel`, passing the thread ID of the thread to be canceled. A canceled thread may later be joined; in fact, you should join a canceled thread to free up its resources, unless the thread is detached (see Section 4.1.5, "Thread Attributes"). The return value of a canceled thread is the special value given by `PTHREAD_CANCELED`.

Often a thread may be in some code that must be executed in an all-or-nothing fashion. For instance, the thread may allocate some resources, use them, and then deallocate them. If the thread is canceled in the middle of this code, it may not have the opportunity to deallocate the resources, and thus the resources will be leaked. To counter this possibility, it is possible for a thread to control whether and when it can be canceled.

A thread may be in one of three states with regard to thread cancellation.

- The thread may be *asynchronously cancelable*. The thread may be canceled at any point in its execution.
- The thread may be *synchronously cancelable*. The thread may be canceled, but not at just any point in its execution. Instead, cancellation requests are queued, and the thread is canceled only when it reaches specific points in its execution.
- A thread may be *uncancelable*. Attempts to cancel the thread are quietly ignored.

When initially created, a thread is synchronously cancelable.

4.2.1 Synchronous and Asynchronous Threads

An asynchronously cancelable thread may be canceled at any point in its execution. A synchronously cancelable thread, in contrast, may be canceled only at particular places in its execution. These places are called *cancellation points*. The thread will queue a cancellation request until it reaches the next cancellation point.

To make a thread asynchronously cancelable, use `pthread_setcanceltype`. This affects the thread that actually calls the function. The first argument should be `PTHREAD_CANCEL_ASYNCHRONOUS` to make the thread asynchronously cancelable, or `PTHREAD_CANCEL_DEFERRED` to return it to the synchronously cancelable state. The second argument, if not null, is a pointer to a variable that will receive the previous cancellation type for the thread. This call, for example, makes the calling thread asynchronously cancelable.

```
pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

What constitutes a cancellation point, and where should these be placed? The most direct way to create a cancellation point is to call `pthread_testcancel`. This does nothing except process a pending cancellation in a synchronously cancelable thread. You should call `pthread_testcancel` periodically during lengthy computations in a thread function, at points where the thread can be canceled without leaking any resources or producing other ill effects.

Certain other functions are implicitly cancellation points as well. These are listed on the `pthread_cancel` man page. Note that other functions may use these functions internally and thus will indirectly be cancellation points.

4.2.2 Uncancelable Critical Sections

A thread may disable cancellation of itself altogether with the `pthread_setcancelstate` function. Like `pthread_setcanceltype`, this affects the calling thread. The first argument is `PTHREAD_CANCEL_DISABLE` to disable cancellation, or `PTHREAD_CANCEL_ENABLE` to re-enable cancellation. The second argument, if not null, points to a variable that will receive the previous cancellation state. This call, for instance, disables thread cancellation in the calling thread.

```
pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, NULL);
```

Using `pthread_setcancelstate` enables you to implement *critical sections*. A critical section is a sequence of code that must be executed either in its entirety or not at all; in other words, if a thread begins executing the critical section, it must continue until the end of the critical section without being canceled.

For example, suppose that you're writing a routine for a banking program that transfers money from one account to another. To do this, you must add value to the balance in one account and deduct the same value from the balance of another account. If the thread running your routine happened to be canceled at just the wrong time between these two operations, the program would have spuriously increased the bank's total deposits by failing to complete the transaction. To prevent this possibility, place the two operations in a critical section.

You might implement the transfer with a function such as `process_transaction`, shown in Listing 4.6. This function disables thread cancellation to start a critical section before it modifies either account balance.

Listing 4.6 **(*critical-section.c*) Protect a Bank Transaction with a Critical Section**

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

/* An array of balances in accounts, indexed by account number.  */

float* account_balances;

/* Transfer DOLLARS from account FROM_ACCT to account TO_ACCT.  Return
   0 if the transaction succeeded, or 1 if the balance FROM_ACCT is
   too small.  */

int process_transaction (int from_acct, int to_acct, float dollars)
{
  int old_cancel_state;

  /* Check the balance in FROM_ACCT.  */
  if (account_balances[from_acct] < dollars)
    return 1;
```

continues

Listing 4.6 **Continued**

```
  /* Begin critical section.  */
  pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &old_cancel_state);
  /* Move the money.  */
  account_balances[to_acct] += dollars;
  account_balances[from_acct] -= dollars;
  /* End critical section.  */
  pthread_setcancelstate (old_cancel_state, NULL);

  return 0;
}
```

Note that it's important to restore the old cancel state at the end of the critical section rather than setting it unconditionally to `PTHREAD_CANCEL_ENABLE`. This enables you to call the `process_transaction` function safely from within another critical section—in that case, your function will leave the cancel state the same way it found it.

4.2.3 When to Use Thread Cancellation

In general, it's a good idea not to use thread cancellation to end the execution of a thread, except in unusual circumstances. During normal operation, a better strategy is to indicate to the thread that it should exit, and then to wait for the thread to exit on its own in an orderly fashion. We'll discuss techniques for communicating with the thread later in this chapter, and in Chapter 5, "Interprocess Communication."

4.3 Thread-Specific Data

Unlike processes, all threads in a single program share the same address space. This means that if one thread modifies a location in memory (for instance, a global variable), the change is visible to all other threads. This allows multiple threads to operate on the same data without the use interprocess communication mechanisms (which are described in Chapter 5).

Each thread has its own call stack, however. This allows each thread to execute different code and to call and return from subroutines in the usual way. As in a single-threaded program, each invocation of a subroutine in each thread has its own set of local variables, which are stored on the stack for that thread.

Sometimes, however, it is desirable to duplicate a certain variable so that each thread has a separate copy. GNU/Linux supports this by providing each thread with a *thread-specific data* area. The variables stored in this area are duplicated for each thread, and each thread may modify its copy of a variable without affecting other threads. Because all threads share the same memory space, thread-specific data may not be accessed using normal variable references. GNU/Linux provides special functions for setting and retrieving values from the thread-specific data area.

You may create as many thread-specific data items as you want, each of type `void*`. Each item is referenced by a key. To create a new key, and thus a new data item for each thread, use `pthread_key_create`. The first argument is a pointer to a `pthread_key_t` variable. That key value can be used by each thread to access its own copy of the corresponding data item. The second argument to `pthread_key_t` is a cleanup function. If you pass a function pointer here, GNU/Linux automatically calls that function when each thread exits, passing the thread-specific value corresponding to that key. This is particularly handy because the cleanup function is called even if the thread is canceled at some arbitrary point in its execution. If the thread-specific value is null, the thread cleanup function is not called. If you don't need a cleanup function, you may pass null instead of a function pointer.

After you've created a key, each thread can set its thread-specific value corresponding to that key by calling `pthread_setspecific`. The first argument is the key, and the second is the `void*` thread-specific value to store. To retrieve a thread-specific data item, call `pthread_getspecific`, passing the key as its argument.

Suppose, for instance, that your application divides a task among multiple threads. For audit purposes, each thread is to have a separate log file, in which progress messages for that thread's tasks are recorded. The thread-specific data area is a convenient place to store the file pointer for the log file for each individual thread.

Listing 4.7 shows how you might implement this. The `main` function in this sample program creates a key to store the thread-specific file pointer and then stores it in `thread_log_key`. Because this is a global variable, it is shared by all threads. When each thread starts executing its thread function, it opens a log file and stores the file pointer under that key. Later, any of these threads may call `write_to_thread_log` to write a message to the thread-specific log file. That function retrieves the file pointer for the thread's log file from thread-specific data and writes the message.

Listing 4.7 **(*tsd.c*) Per-Thread Log Files Implemented with Thread-Specific Data**

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

/* The key used to associate a log file pointer with each thread.  */
static pthread_key_t thread_log_key;

/* Write MESSAGE to the log file for the current thread.  */

void write_to_thread_log (const char* message)
{
  FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
  fprintf (thread_log, "%s\n", message);
}

/* Close the log file pointer THREAD_LOG.  */

void close_thread_log (void* thread_log)
```

continues

Listing 4.7 **Continued**

```
{
  fclose ((FILE*) thread_log);
}

void* thread_function (void* args)
{
  char thread_log_filename[20];
  FILE* thread_log;

  /* Generate the filename for this thread's log file.  */
  sprintf (thread_log_filename, "thread%d.log", (int) pthread_self ());
  /* Open the log file.  */
  thread_log = fopen (thread_log_filename, "w");
  /* Store the file pointer in thread-specific data under thread_log_key.  */
  pthread_setspecific (thread_log_key, thread_log);

  write_to_thread_log ("Thread starting.");
  /* Do work here... */

  return NULL;
}

int main ()
{
  int i;
  pthread_t threads[5];

  /* Create a key to associate thread log file pointers in
     thread-specific data.  Use close_thread_log to clean up the file
     pointers.  */
  pthread_key_create (&thread_log_key, close_thread_log);
  /* Create threads to do the work.  */
  for (i = 0; i < 5; ++i)
    pthread_create (&(threads[i]), NULL, thread_function, NULL);
  /* Wait for all threads to finish.  */
  for (i = 0; i < 5; ++i)
    pthread_join (threads[i], NULL);
  return 0;
}
```

Observe that `thread_function` does not need to close the log file. That's because when the log file key was created, `close_thread_log` was specified as the cleanup function for that key. Whenever a thread exits, GNU/Linux calls that function, passing the thread-specific value for the thread log key. This function takes care of closing the log file.

4.3.1 Cleanup Handlers

The cleanup functions for thread-specific data keys can be very handy for ensuring that resources are not leaked when a thread exits or is canceled. Sometimes, though, it's useful to be able to specify cleanup functions without creating a new thread-specific data item that's duplicated for each thread. GNU/Linux provides *cleanup handlers* for this purpose.

A cleanup handler is simply a function that should be called when a thread exits. The handler takes a single `void*` parameter, and its argument value is provided when the handler is registered—this makes it easy to use the same handler function to deallocate multiple resource instances.

A cleanup handler is a temporary measure, used to deallocate a resource only if the thread exits or is canceled instead of finishing execution of a particular region of code. Under normal circumstances, when the thread does not exit and is not canceled, the resource should be deallocated explicitly and the cleanup handler should be removed.

To register a cleanup handler, call `pthread_cleanup_push`, passing a pointer to the cleanup function and the value of its `void*` argument. The call to `pthread_cleanup_push` must be balanced by a corresponding call to `pthread_cleanup_pop`, which unregisters the cleanup handler. As a convenience, `pthread_cleanup_pop` takes an `int` flag argument; if the flag is nonzero, the cleanup action is actually performed as it is unregistered.

The program fragment in Listing 4.8 shows how you might use a cleanup handler to make sure that a dynamically allocated buffer is cleaned up if the thread terminates.

Listing 4.8 **(*cleanup.c*) Program Fragment Demonstrating a Thread Cleanup Handler**

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

/* Allocate a temporary buffer.  */

void* allocate_buffer (size_t size)
{
  return malloc (size);
}

/* Deallocate a temporary buffer.  */

void deallocate_buffer (void* buffer)
{
  free (buffer);
}

void do_some_work ()
{
  /* Allocate a temporary buffer.  */
```

continues

Listing 4.8 **Continued**

```
  void* temp_buffer = allocate_buffer (1024);
  /* Register a cleanup handler for this buffer, to deallocate it in
     case the thread exits or is cancelled.  */
  pthread_cleanup_push (deallocate_buffer, temp_buffer);

  /* Do some work here that might call pthread_exit or might be
     cancelled...  */

  /* Unregister the cleanup handler.  Because we pass a nonzero value,
     this actually performs the cleanup by calling
     deallocate_buffer.  */
  pthread_cleanup_pop (1);
}
```

Because the argument to `pthread_cleanup_pop` is nonzero in this case, the cleanup function `deallocate_buffer` is called automatically here and does not need to be called explicitly. In this simple case, we could have used the standard library function `free` directly as our cleanup handler function instead of `deallocate_buffer`.

4.3.2 Thread Cleanup in C++

C++ programmers are accustomed to getting cleanup "for free" by wrapping cleanup actions in object destructors. When the objects go out of scope, either because a block is executed to completion or because an exception is thrown, C++ makes sure that destructors are called for those automatic variables that have them. This provides a handy mechanism to make sure that cleanup code is called no matter how the block is exited.

If a thread calls `pthread_exit`, though, C++ doesn't guarantee that destructors are called for all automatic variables on the thread's stack. A clever way to recover this functionality is to invoke `pthread_exit` at the top level of the thread function by throwing a special exception.

The program in Listing 4.9 demonstrates this. Using this technique, a function indicates its intention to exit the thread by throwing a `ThreadExitException` instead of calling `pthread_exit` directly. Because the exception is caught in the top-level thread function, all local variables on the thread's stack will be destroyed properly as the exception percolates up.

Listing 4.9 **(*cxx-exit.cpp*) Implementing Safe Thread Exit with C++ Exceptions**

```
#include <pthread.h>

class ThreadExitException
{
public:
  /* Create an exception-signaling thread exit with RETURN_VALUE.  */
  ThreadExitException (void* return_value)
    : thread_return_value_ (return_value)
```

```
  {
  }

  /* Actually exit the thread, using the return value provided in the
     constructor.  */
  void* DoThreadExit ()
  {
    pthread_exit (thread_return_value_);
  }

private:
  /* The return value that will be used when exiting the thread.  */
  void* thread_return_value_;
};

void do_some_work ()
{
  while (1) {
    /* Do some useful things here...  */

    if (should_exit_thread_immediately ())
      throw ThreadExitException (/* thread's return value = */ NULL);
  }
}

void* thread_function (void*)
{
  try {
    do_some_work ();
  }
  catch (ThreadExitException ex) {
    /* Some function indicated that we should exit the thread.  */
    ex.DoThreadExit ();
  }
  return NULL;
}
```

4.4 Synchronization and Critical Sections

Programming with threads is very tricky because most threaded programs are concurrent programs. In particular, there's no way to know when the system will schedule one thread to run and when it will run another. One thread might run for a very long time, or the system might switch among threads very quickly. On a system with multiple processors, the system might even schedule multiple threads to run at literally the same time.

Debugging a threaded program is difficult because you cannot always easily reproduce the behavior that caused the problem. You might run the program once and have everything work fine; the next time you run it, it might crash. There's no way to make the system schedule the threads exactly the same way it did before.

The ultimate cause of most bugs involving threads is that the threads are accessing the same data. As mentioned previously, that's one of the powerful aspects of threads, but it can also be dangerous. If one thread is only partway through updating a data structure when another thread accesses the same data structure, chaos is likely to ensue. Often, buggy threaded programs contain a code that will work only if one thread gets scheduled more often—or sooner—than another thread. These bugs are called *race conditions*; the threads are racing one another to change the same data structure.

4.4.1 Race Conditions

Suppose that your program has a series of queued jobs that are processed by several concurrent threads. The queue of jobs is represented by a linked list of `struct job` objects.

After each thread finishes an operation, it checks the queue to see if an additional job is available. If `job_queue` is non-null, the thread removes the head of the linked list and sets `job_queue` to the next job on the list.

The thread function that processes jobs in the queue might look like Listing 4.10.

Listing 4.10 **(*job-queue1.c*) Thread Function to Process Jobs from the Queue**

```
#include <malloc.h>

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (job_queue != NULL) {
    /* Get the next available job.  */
    struct job* next_job = job_queue;
    /* Remove this job from the list.  */
    job_queue = job_queue->next;
    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
```

Now suppose that two threads happen to finish a job at about the same time, but only one job remains in the queue. The first thread checks whether `job_queue` is null; finding that it isn't, the thread enters the loop and stores the pointer to the job object in `next_job`. At this point, Linux happens to interrupt the first thread and schedules the second. The second thread also checks `job_queue` and finding it non-null, also assigns the same job pointer to `next_job`. By unfortunate coincidence, we now have two threads executing the same job.

To make matters worse, one thread will unlink the job object from the queue, leaving `job_queue` containing null. When the other thread evaluates `job_queue->next`, a segmentation fault will result.

This is an example of a race condition. Under "lucky" circumstances, this particular schedule of the two threads may never occur, and the race condition may never exhibit itself. Only under different circumstances, perhaps when running on a heavily loaded system (or on an important customer's new multiprocessor server!) may the bug exhibit itself.

To eliminate race conditions, you need a way to make operations *atomic*. An atomic operation is indivisible and uninterruptible; once the operation starts, it will not be paused or interrupted until it completes, and no other operation will take place meanwhile. In this particular example, you want to check `job_queue`; if it's not empty, remove the first job, all as a single atomic operation.

4.4.2 Mutexes

The solution to the job queue race condition problem is to let only one thread access the queue of jobs at a time. Once a thread starts looking at the queue, no other thread should be able to access it until the first thread has decided whether to process a job and, if so, has removed the job from the list.

Implementing this requires support from the operating system. GNU/Linux provides *mutexes*, short for *MUTual EXclusion locks*. A mutex is a special lock that only one thread may lock at a time. If a thread locks a mutex and then a second thread also tries to lock the same mutex, the second thread is *blocked*, or put on hold. Only when the first thread unlocks the mutex is the second thread *unblocked*—allowed to resume execution. GNU/Linux guarantees that race conditions do not occur among threads attempting to lock a mutex; only one thread will ever get the lock, and all other threads will be blocked.

Think of a mutex as the lock on a lavatory door. Whoever gets there first enters the lavatory and locks the door. If someone else attempts to enter the lavatory while it's occupied, that person will find the door locked and will be forced to wait outside until the occupant emerges.

To create a mutex, create a variable of type `pthread_mutex_t` and pass a pointer to it to `pthread_mutex_init`. The second argument to `pthread_mutex_init` is a pointer to a mutex attribute object, which specifies attributes of the mutex. As with

`pthread_create`, if the attribute pointer is null, default attributes are assumed. The mutex variable should be initialized only once. This code fragment demonstrates the declaration and initialization of a mutex variable.

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

Another simpler way to create a mutex with default attributes is to initialize it with the special value `PTHREAD_MUTEX_INITIALIZER`. No additional call to `pthread_mutex_init` is necessary. This is particularly convenient for global variables (and, in C++, static data members). The previous code fragment could equivalently have been written like this:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

A thread may attempt to lock a mutex by calling `pthread_mutex_lock` on it. If the mutex was unlocked, it becomes locked and the function returns immediately. If the mutex was locked by another thread, `pthread_mutex_lock` blocks execution and returns only eventually when the mutex is unlocked by the other thread. More than one thread may be blocked on a locked mutex at one time. When the mutex is unlocked, only one of the blocked threads (chosen unpredictably) is unblocked and allowed to lock the mutex; the other threads stay blocked.

A call to `pthread_mutex_unlock` unlocks a mutex. This function should always be called from the same thread that locked the mutex.

Listing 4.11 shows another version of the job queue example. Now the queue is protected by a mutex. Before accessing the queue (either for read or write), each thread locks a mutex first. Only when the entire sequence of checking the queue and removing a job is complete is the mutex unlocked. This prevents the race condition previously described.

Listing 4.11 **(*job-queue2.c*) Job Queue Thread Function, Protected by a Mutex**

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

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Now it's safe to check if the queue is empty.  */
    if (job_queue == NULL)
      next_job = NULL;
    else {
      /* Get the next available job.  */
      next_job = job_queue;
      /* Remove this job from the list.  */
      job_queue = job_queue->next;
    }
    /* Unlock the mutex on the job queue because we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);

    /* Was the queue empty?  If so, end the thread.  */
    if (next_job == NULL)
      break;

    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
```

All accesses to `job_queue`, the shared data pointer, come between the call to `pthread_mutex_lock` and the call to `pthread_mutex_unlock`. A job object, stored in `next_job`, is accessed outside this region only after that object has been removed from the queue and is therefore inaccessible to other threads.

Note that if the queue is empty (that is, `job_queue` is null), we don't break out of the loop immediately because this would leave the mutex permanently locked and would prevent any other thread from accessing the job queue ever again. Instead, we remember this fact by setting `next_job` to null and breaking out only after unlocking the mutex.

Use of the mutex to lock `job_queue` is not automatic; it's up to you to add code to lock the mutex before accessing that variable and then to unlock it afterward. For example, a function to add a job to the job queue might look like this:

```
void enqueue_job (struct job* new_job)
{
  pthread_mutex_lock (&job_queue_mutex);
```

```
  new_job->next = job_queue;
  job_queue = new_job;
  pthread_mutex_unlock (&job_queue_mutex);
}
```

4.4.3 Mutex Deadlocks

Mutexes provide a mechanism for allowing one thread to block the execution of another. This opens up the possibility of a new class of bugs, called *deadlocks*. A deadlock occurs when one or more threads are stuck waiting for something that never will occur.

A simple type of deadlock may occur when the same thread attempts to lock a mutex twice in a row. The behavior in this case depends on what kind of mutex is being used. Three kinds of mutexes exist:

- Locking a *fast mutex* (the default kind) will cause a deadlock to occur. An attempt to lock the mutex blocks until the mutex is unlocked. But because the thread that locked the mutex is blocked on the same mutex, the lock cannot ever be released.
- Locking a *recursive mutex* does not cause a deadlock. A recursive mutex may safely be locked many times by the same thread. The mutex remembers how many times `pthread_mutex_lock` was called on it by the thread that holds the lock; that thread must make the same number of calls to `pthread_mutex_unlock` before the mutex is actually unlocked and another thread is allowed to lock it.
- GNU/Linux will detect and flag a double lock on an *error-checking mutex* that would otherwise cause a deadlock. The second consecutive call to `pthread_mutex_lock` returns the failure code `EDEADLK`.

By default, a GNU/Linux mutex is of the fast kind. To create a mutex of one of the other two kinds, first create a mutex attribute object by declaring a `pthread_mutexattr_t` variable and calling `pthread_mutexattr_init` on a pointer to it. Then set the mutex kind by calling `pthread_mutexattr_setkind_np`; the first argument is a pointer to the mutex attribute object, and the second is `PTHREAD_MUTEX_RECURSIVE_NP` for a recursive mutex, or `PTHREAD_MUTEX_ERRORCHECK_NP` for an error-checking mutex. Pass a pointer to this attribute object to `pthread_mutex_init` to create a mutex of this kind, and then destroy the attribute object with `pthread_mutexattr_destroy`.

This code sequence illustrates creation of an error-checking mutex, for instance:

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

pthread_mutexattr_init (&attr);
pthread_mutexattr_setkind_np (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
pthread_mutexattr_destroy (&attr);
```

As suggested by the "np" suffix, the recursive and error-checking mutex kinds are specific to GNU/Linux and are not portable. Therefore, it is generally not advised to use them in programs. (Error-checking mutexes can be useful when debugging, though.)

4.4.4 Nonblocking Mutex Tests

Occasionally, it is useful to test whether a mutex is locked without actually blocking on it. For instance, a thread may need to lock a mutex but may have other work to do instead of blocking if the mutex is already locked. Because `pthread_mutex_lock` will not return until the mutex becomes unlocked, some other function is necessary.

GNU/Linux provides `pthread_mutex_trylock` for this purpose. If you call `pthread_mutex_trylock` on an unlocked mutex, you will lock the mutex as if you had called `pthread_mutex_lock`, and `pthread_mutex_trylock` will return zero. However, if the mutex is already locked by another thread, `pthread_mutex_trylock` will not block. Instead, it will return immediately with the error code `EBUSY`. The mutex lock held by the other thread is not affected. You may try again later to lock the mutex.

4.4.5 Semaphores for Threads

In the preceding example, in which several threads process jobs from a queue, the main thread function of the threads carries out the next job until no jobs are left and then exits the thread. This scheme works if all the jobs are queued in advance or if new jobs are queued at least as quickly as the threads process them. However, if the threads work too quickly, the queue of jobs will empty and the threads will exit. If new jobs are later enqueued, no threads may remain to process them. What we might like instead is a mechanism for blocking the threads when the queue empties until new jobs become available.

A *semaphore* provides a convenient method for doing this. A semaphore is a counter that can be used to synchronize multiple threads. As with a mutex, GNU/Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition.

Each semaphore has a counter value, which is a non-negative integer. A semaphore supports two basic operations:

- A *wait* operation decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread). When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns.
- A *post* operation increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero).

Note that GNU/Linux provides two slightly different semaphore implementations. The one we describe here is the POSIX standard semaphore implementation. Use these semaphores when communicating among threads The other implementation, used for communication among processes, is described in Section 5.2, "Process Semaphores." If you use semaphores, include `<semaphore.h>`.

A semaphore is represented by a `sem_t` variable. Before using it, you must initialize it using the `sem_init` function, passing a pointer to the `sem_t` variable. The second parameter should be zero,[2] and the third parameter is the semaphore's initial value. If you no longer need a semaphore, it's good to deallocate it with `sem_destroy`.

To wait on a semaphore, use `sem_wait`. To post to a semaphore, use `sem_post`. A nonblocking wait function, `sem_trywait`, is also provided. It's similar to `pthread_mutex_trylock`—if the wait would have blocked because the semaphore's value was zero, the function returns immediately, with error value `EAGAIN`, instead of blocking.

GNU/Linux also provides a function to retrieve the current value of a semaphore, `sem_getvalue`, which places the value in the `int` variable pointed to by its second argument. You should not use the semaphore value you get from this function to make a decision whether to post to or wait on the semaphore, though. To do this could lead to a race condition: Another thread could change the semaphore's value between the call to `sem_getvalue` and the call to another semaphore function. Use the atomic post and wait functions instead.

Returning to our job queue example, we can use a semaphore to count the number of jobs waiting in the queue. Listing 4.12 controls the queue with a semaphore. The function `enqueue_job` adds a new job to the queue.

Listing 4.12 (*job-queue3.c*) **Job Queue Controlled by a Semaphore**

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

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

2. A nonzero value would indicate a semaphore that can be shared across processes, which is not supported by GNU/Linux for this type of semaphore.

```
/* A semaphore counting the number of jobs in the queue.  */
sem_t job_queue_count;

/* Perform one-time initialization of the job queue.  */

void initialize_job_queue ()
{
  /* The queue is initially empty.  */
  job_queue = NULL;
  /* Initialize the semaphore which counts jobs in the queue.  Its
     initial value should be zero.  */
  sem_init (&job_queue_count, 0, 0);
}

/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Wait on the job queue semaphore.  If its value is positive,
       indicating that the queue is not empty, decrement the count by
       1.  If the queue is empty, block until a new job is enqueued.  */
    sem_wait (&job_queue_count);

    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Because of the semaphore, we know the queue is not empty.  Get
       the next available job.  */
    next_job = job_queue;
    /* Remove this job from the list.  */
    job_queue = job_queue->next;
    /* Unlock the mutex on the job queue because we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);

    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}

/* Add a new job to the front of the job queue.  */

void enqueue_job (/* Pass job-specific data here...  */)
{
  struct job* new_job;
```

continues

Listing 4.12 **Continued**

```
  /* Allocate a new job object.  */
  new_job = (struct job*) malloc (sizeof (struct job));
  /* Set the other fields of the job struct here...  */

  /* Lock the mutex on the job queue before accessing it.  */
  pthread_mutex_lock (&job_queue_mutex);
  /* Place the new job at the head of the queue.  */
  new_job->next = job_queue;
  job_queue = new_job;

  /* Post to the semaphore to indicate that another job is available.  If
     threads are blocked, waiting on the semaphore, one will become
     unblocked so it can process the job.  */
  sem_post (&job_queue_count);

  /* Unlock the job queue mutex.  */
  pthread_mutex_unlock (&job_queue_mutex);
}
```

Before taking a job from the front of the queue, each thread will first wait on the semaphore. If the semaphore's value is zero, indicating that the queue is empty, the thread will simply block until the semaphore's value becomes positive, indicating that a job has been added to the queue.

The `enqueue_job` function adds a job to the queue. Just like `thread_function`, it needs to lock the queue mutex before modifying the queue. After adding a job to the queue, it posts to the semaphore, indicating that a new job is available. In the version shown in Listing 4.12, the threads that process the jobs never exit; if no jobs are available for a while, all the threads simply block in `sem_wait`.

4.4.6 Condition Variables

We've shown how to use a mutex to protect a variable against simultaneous access by two threads and how to use semaphores to implement a shared counter. A *condition variable* is a third synchronization device that GNU/Linux provides; with it, you can implement more complex conditions under which threads execute.

Suppose that you write a thread function that executes a loop infinitely, performing some work on each iteration. The thread loop, however, needs to be controlled by a flag: The loop runs only when the flag is set; when the flag is not set, the loop pauses.

Listing 4.13 shows how you might implement this by spinning in a loop. During each iteration of the loop, the thread function checks that the flag is set. Because the flag is accessed by multiple threads, it is protected by a mutex. This implementation may be correct, but it is not efficient. The thread function will spend lots of CPU

whenever the flag is not set, checking and rechecking the flag, each time locking and unlocking the mutex. What you really want is a way to put the thread to sleep when the flag is not set, until some circumstance changes that might cause the flag to become set.

Listing 4.13 **(*spin-condvar.c*) A Simple Condition Variable Implementation**

```
#include <pthread.h>

int thread_flag;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
  pthread_mutex_init (&thread_flag_mutex, NULL);
  thread_flag = 0;
}

/* Calls do_work repeatedly while the thread flag is set; otherwise
   spins.  */

void* thread_function (void* thread_arg)
{
  while (1) {
    int flag_is_set;

    /* Protect the flag with a mutex lock.  */
    pthread_mutex_lock (&thread_flag_mutex);
    flag_is_set = thread_flag;
    pthread_mutex_unlock (&thread_flag_mutex);

    if (flag_is_set)
      do_work ();
    /* Else don't do anything.  Just loop again.  */
  }
  return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Protect the flag with a mutex lock.  */
  pthread_mutex_lock (&thread_flag_mutex);
  thread_flag = flag_value;
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

A condition variable enables you to implement a condition under which a thread executes and, inversely, the condition under which the thread is blocked. As long as every thread that potentially changes the sense of the condition uses the condition variable properly, Linux guarantees that threads blocked on the condition will be unblocked when the condition changes.

As with a semaphore, a thread may *wait* on a condition variable. If thread A waits on a condition variable, it is blocked until some other thread, thread B, signals the same condition variable. Unlike a semaphore, a condition variable has no counter or memory; thread A must wait on the condition variable *before* thread B signals it. If thread B signals the condition variable before thread A waits on it, the signal is lost, and thread A blocks until some other thread signals the condition variable again.

This is how you would use a condition variable to make the previous sample more efficient:

- The loop in `thread_function` checks the flag. If the flag is not set, the thread waits on the condition variable.
- The `set_thread_flag` function signals the condition variable after changing the flag value. That way, if `thread_function` is blocked on the condition variable, it will be unblocked and will check the condition again.

There's one problem with this: There's a race condition between checking the flag value and signaling or waiting on the condition variable. Suppose that `thread_function` checked the flag and found that it was not set. At that moment, the Linux scheduler paused that thread and resumed the main one. By some coincidence, the main thread is in `set_thread_flag`. It sets the flag and then signals the condition variable. Because no thread is waiting on the condition variable at the time (remember that `thread_function` was paused before it could wait on the condition variable), the signal is lost. Now, when Linux reschedules the other thread, it starts waiting on the condition variable and may end up blocked forever.

To solve this problem, we need a way to lock the flag and the condition variable together with a single mutex. Fortunately, GNU/Linux provides exactly this mechanism. Each condition variable must be used in conjunction with a mutex, to prevent this sort of race condition. Using this scheme, the thread function follows these steps:

1. The loop in `thread_function` locks the mutex and reads the flag value.
2. If the flag is set, it unlocks the mutex and executes the work function.
3. If the flag is not set, it atomically unlocks the mutex and waits on the condition variable.

The critical feature here is in step 3, in which GNU/Linux allows you to unlock the mutex and wait on the condition variable atomically, without the possibility of another thread intervening. This eliminates the possibility that another thread may change the flag value and signal the condition variable in between `thread_function`'s test of the flag value and wait on the condition variable.

A condition variable is represented by an instance of `pthread_cond_t`. Remember that each condition variable should be accompanied by a mutex. These are the functions that manipulate condition variables:

- `pthread_cond_init` initializes a condition variable. The first argument is a pointer to a `pthread_cond_t` instance. The second argument, a pointer to a condition variable attribute object, is ignored under GNU/Linux.

  The mutex must be initialized separately, as described in Section 4.4.2, "Mutexes."

- `pthread_cond_signal` signals a condition variable. A single thread that is blocked on the condition variable will be unblocked. If no other thread is blocked on the condition variable, the signal is ignored. The argument is a pointer to the `pthread_cond_t` instance.

  A similar call, `pthread_cond_broadcast`, unblocks *all* threads that are blocked on the condition variable, instead of just one.

- `pthread_cond_wait` blocks the calling thread until the condition variable is signaled. The argument is a pointer to the `pthread_cond_t` instance. The second argument is a pointer to the `pthread_mutex_t` mutex instance.

  When `pthread_cond_wait` is called, the mutex must already be locked by the calling thread. That function atomically unlocks the mutex and blocks on the condition variable. When the condition variable is signaled and the calling thread unblocks, `pthread_cond_wait` automatically reacquires a lock on the mutex.

Whenever your program performs an action that may change the sense of the condition you're protecting with the condition variable, it should perform these steps. (In our example, the condition is the state of the thread flag, so these steps must be taken whenever the flag is changed.)

1. Lock the mutex accompanying the condition variable.
2. Take the action that may change the sense of the condition (in our example, set the flag).
3. Signal or broadcast the condition variable, depending on the desired behavior.
4. Unlock the mutex accompanying the condition variable.

Listing 4.14 shows the previous example again, now using a condition variable to protect the thread flag. Note that in `thread_function`, a lock on the mutex is held before checking the value of `thread_flag`. That lock is automatically released by `pthread_cond_wait` before blocking and is automatically reacquired afterward. Also note that `set_thread_flag` locks the mutex before setting the value of `thread_flag` and signaling the mutex.

Listing 4.14 (*condvar.c*) Control a Thread Using a Condition Variable

```
#include <pthread.h>

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
  /* Initialize the mutex and condition variable.  */
  pthread_mutex_init (&thread_flag_mutex, NULL);
  pthread_cond_init (&thread_flag_cv, NULL);
  /* Initialize the flag value.  */
  thread_flag = 0;
}

/* Calls do_work repeatedly while the thread flag is set; blocks if
   the flag is clear.  */

void* thread_function (void* thread_arg)
{
  /* Loop infinitely.  */
  while (1) {
    /* Lock the mutex before accessing the flag value.  */
    pthread_mutex_lock (&thread_flag_mutex);
    while (!thread_flag)
      /* The flag is clear.  Wait for a signal on the condition
         variable, indicating that the flag value has changed.  When the
         signal arrives and this thread unblocks, loop and check the
         flag again.  */
      pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
    /* When we've gotten here, we know the flag must be set.  Unlock
       the mutex.  */
    pthread_mutex_unlock (&thread_flag_mutex);
    /* Do some work.  */
    do_work ();
  }
  return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Lock the mutex before accessing the flag value.  */
  pthread_mutex_lock (&thread_flag_mutex);
  /* Set the flag value, and then signal in case thread_function is
     blocked, waiting for the flag to become set.  However,
     thread_function can't actually check the flag until the mutex is
     unlocked.  */
```

```
  thread_flag = flag_value;
  pthread_cond_signal (&thread_flag_cv);
  /* Unlock the mutex.  */
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

The condition protected by a condition variable can be arbitrarily complex. However, before performing any operation that may change the sense of the condition, a mutex lock should be required, and the condition variable should be signaled afterward.

A condition variable may also be used without a condition, simply as a mechanism for blocking a thread until another thread "wakes it up." A semaphore may also be used for that purpose. The principal difference is that a semaphore "remembers" the wake-up call even if no thread was blocked on it at the time, while a condition variable discards the wake-up call unless some thread is actually blocked on it at the time. Also, a semaphore delivers only a single wake-up per post; with `pthread_cond_broadcast`, an arbitrary and unknown number of blocked threads may be awoken at the same time.

4.4.7 Deadlocks with Two or More Threads

Deadlocks can occur when two (or more) threads are each blocked, waiting for a condition to occur that only the other one can cause. For instance, if thread A is blocked on a condition variable waiting for thread B to signal it, and thread B is blocked on a condition variable waiting for thread A to signal it, a deadlock has occurred because neither thread will ever signal the other. You should take care to avoid the possibility of such situations because they are quite difficult to detect.

One common error that can cause a deadlock involves a problem in which more than one thread is trying to lock the same set of objects. For example, consider a program in which two different threads, running two different thread functions, need to lock the same two mutexes. Suppose that thread A locks mutex 1 and then mutex 2, and thread B happens to lock mutex 2 before mutex 1. In a sufficiently unfortunate scheduling scenario, Linux may schedule thread A long enough to lock mutex 1, and then schedule thread B, which promptly locks mutex 2. Now neither thread can progress because each is blocked on a mutex that the other thread holds locked.

This is an example of a more general deadlock problem, which can involve not only synchronization objects such as mutexes, but also other resources, such as locks on files or devices. The problem occurs when multiple threads try to lock the same set of resources in different orders. The solution is to make sure that all threads that lock more than one resource lock them in the same order.

4.5 GNU/Linux Thread Implementation

The implementation of POSIX threads on GNU/Linux differs from the thread implementation on many other UNIX-like systems in an important way: on GNU/Linux, threads are implemented as processes. Whenever you call `pthread_create` to create a new thread, Linux creates a new process that runs that thread. However, this process is not the same as a process you would create with `fork`; in particular, it shares the same address space and resources as the original process rather than receiving copies.

The program `thread-pid` shown in Listing 4.15 demonstrates this. The program creates a thread; both the original thread and the new one call the `getpid` function and print their respective process IDs and then spin infinitely.

Listing 4.15 **(*thread-pid*) Print Process IDs for Threads**

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

void* thread_function (void* arg)
{
  fprintf (stderr, "child thread pid is %d\n", (int) getpid ());
  /* Spin forever.  */
  while (1);
  return NULL;
}

int main ()
{
  pthread_t thread;
  fprintf (stderr, "main thread pid is %d\n", (int) getpid ());
  pthread_create (&thread, NULL, &thread_function, NULL);
  /* Spin forever.  */
  while (1);
  return 0;
}
```

Run the program in the background, and then invoke `ps x` to display your running processes. Don't forget to kill the `thread-pid` program afterward—it consumes lots of CPU doing nothing. Here's what the output might look like:

```
% cc thread-pid.c -o thread-pid -lpthread
% ./thread-pid &
[1] 14608
main thread pid is 14608
child thread pid is 14610
% ps x
  PID TTY      STAT   TIME COMMAND
14042 pts/9    S      0:00 bash
14608 pts/9    R      0:01 ./thread-pid
```

```
14609 pts/9    S      0:00 ./thread-pid
14610 pts/9    R      0:01 ./thread-pid
14611 pts/9    R      0:00 ps x
% kill 14608
[1]+  Terminated              ./thread-pid
```

Job Control Notification in the Shell

The lines starting with `[1]` are from the shell. When you run a program in the background, the shell assigns a job number to it—in this case, 1—and prints out the program's pid. If a background job terminates, the shell reports that fact the next time you invoke a command.

Notice that there are three processes running the `thread-pid` program. The first of these, with pid 14608, is the main thread in the program; the third, with pid 14610, is the thread we created to execute `thread_function`.

How about the second thread, with pid 14609? This is the "manager thread," which is part of the internal implementation of GNU/Linux threads. The manager thread is created the first time a program calls `pthread_create` to create a new thread.

4.5.1 Signal Handling

Suppose that a multithreaded program receives a signal. In which thread is the signal handler invoked? The behavior of the interaction between signals and threads varies from one UNIX-like system to another. In GNU/Linux, the behavior is dictated by the fact that threads are implemented as processes.

Because each thread is a separate process, and because a signal is delivered to a particular process, there is no ambiguity about which thread receives the signal. Typically, signals sent from outside the program are sent to the process corresponding to the main thread of the program. For instance, if a program forks and the child process execs a multithreaded program, the parent process will hold the process id of the main thread of the child process's program and will use that process id to send signals to its child. This is generally a good convention to follow yourself when sending signals to a multithreaded program.

Note that this aspect of GNU/Linux's implementation of pthreads is at variance with the POSIX thread standard. Do not rely on this behavior in programs that are meant to be portable.

Within a multithreaded program, it is possible for one thread to send a signal specifically to another thread. Use the `pthread_kill` function to do this. Its first parameter is a thread ID, and its second parameter is a signal number.

4.5.2 The *clone* System Call

Although GNU/Linux threads created in the same program are implemented as separate processes, they share their virtual memory space and other resources. A child process created with `fork`, however, gets copies of these items. How is the former type of process created?

The Linux `clone` system call is a generalized form of `fork` and `pthread_create` that allows the caller to specify which resources are shared between the calling process and the newly created process. Also, `clone` requires you to specify the memory region for the execution stack that the new process will use. Although we mention `clone` here to satisfy the reader's curiosity, that system call should not ordinarily be used in programs. Use `fork` to create new processes or `pthread_create` to create threads.

4.6 Processes Vs. Threads

For some programs that benefit from concurrency, the decision whether to use processes or threads can be difficult. Here are some guidelines to help you decide which concurrency model best suits your program:

- All threads in a program must run the same executable. A child process, on the other hand, may run a different executable by calling an `exec` function.
- An errant thread can harm other threads in the same process because threads share the same virtual memory space and other resources. For instance, a wild memory write through an uninitialized pointer in one thread can corrupt memory visible to another thread.

  An errant process, on the other hand, cannot do so because each process has a copy of the program's memory space.
- Copying memory for a new process adds an additional performance overhead relative to creating a new thread. However, the copy is performed only when the memory is changed, so the penalty is minimal if the child process only reads memory.
- Threads should be used for programs that need fine-grained parallelism. For example, if a problem can be broken into multiple, nearly identical tasks, threads may be a good choice. Processes should be used for programs that need coarser parallelism.
- Sharing data among threads is trivial because threads share the same memory. (However, great care must be taken to avoid race conditions, as described previously.) Sharing data among processes requires the use of IPC mechanisms, as described in Chapter 5. This can be more cumbersome but makes multiple processes less likely to suffer from concurrency bugs.