29

THREADS: INTRODUCTION

In this and the next few chapters, we describe POSIX threads, often known as *Pthreads*. We won't attempt to cover the entire Pthreads API, since it is rather large. Various sources of further information about threads are listed at the end of this chapter.

These chapters mainly describe the standard behavior specified for the Pthreads API. In Section 33.5, we discuss those points where the two main Linux threading implementations—LinuxThreads and Native POSIX Threads Library (NPTL)—deviate from the standard.

In this chapter, we provide an overview of the operation of threads, and then look at how threads are created and how they terminate. We conclude with a discussion of some factors that may influence the choice of a multithreaded approach versus a multiprocess approach when designing an application.

29.1 Overview

Like processes, threads are a mechanism that permits an application to perform multiple tasks concurrently. A single process can contain multiple threads, as illustrated in Figure 29-1. All of these threads are independently executing the same program, and they all share the same global memory, including the initialized data, uninitialized data, and heap segments. (A traditional UNIX process is simply a special case of a multithreaded processes; it is a process that contains just one thread.)

We have simplified things somewhat in Figure 29-1. In particular, the location of the per-thread stacks may be intermingled with shared libraries and shared memory regions, depending on the order in which threads are created, shared libraries loaded, and shared memory regions attached. Furthermore, the location of the per-thread stacks can vary depending on the Linux distribution.

The threads in a process can execute concurrently. On a multiprocessor system, multiple threads can execute parallel. If one thread is blocked on I/O, other threads are still eligible to execute. (Although it sometimes useful to create a separate thread purely for the purpose of performing I/O, it is often preferable to employ one of the alternative I/O models that we describe in Chapter 63.)

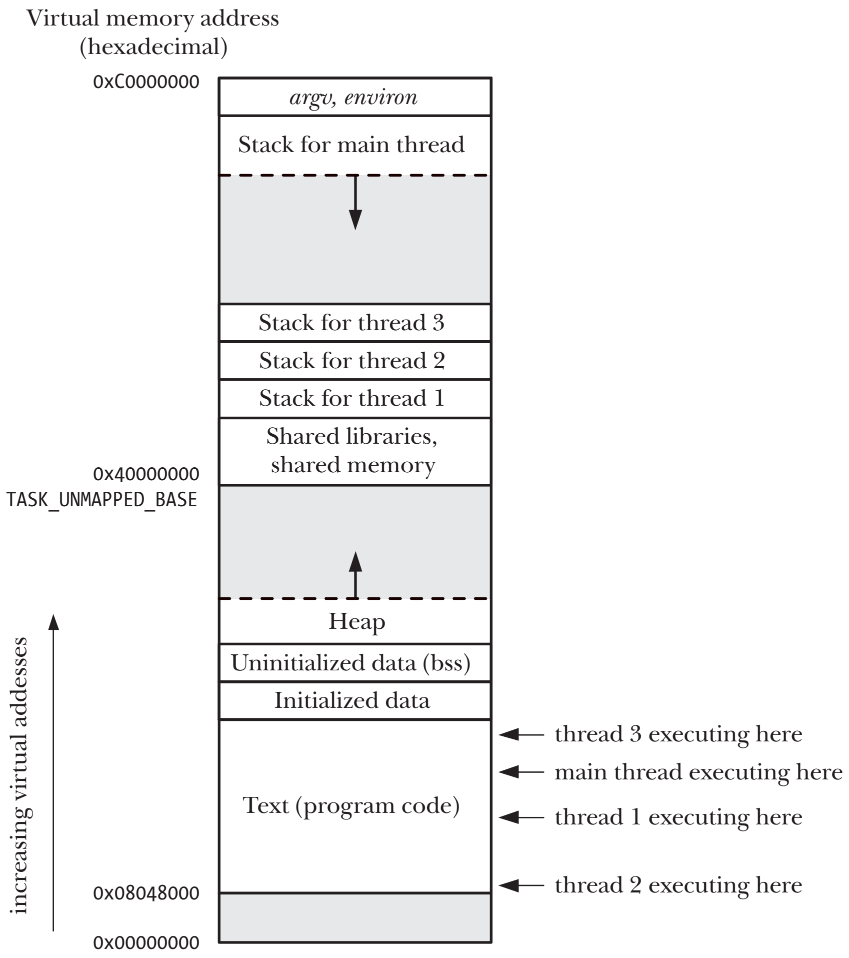

Figure 29-1: Four threads executing in a process (Linux/x86-32)

Threads offer advantages over processes in certain applications. Consider the traditional UNIX approach to achieving concurrency by creating multiple processes. An example of this is a network server design in which a parent process accepts incoming connections from clients, and then uses *fork()* to create a separate child process to handle communication with each client (refer to Section 60.3). Such a design makes it possible to serve multiple clients simultaneously. While this approach works well for many scenarios, it does have the following limitations in some applications:

- It is difficult to share information between processes. Since the parent and child don't share memory (other than the read-only text segment), we must use some form of interprocess communication in order to exchange information between processes.
- Process creation with *fork()* is relatively expensive. Even with the copy-on-write technique described in Section 24.2.2, the need to duplicate various process attributes such as page tables and file descriptor tables means that a *fork()* call is still time-consuming.

Threads address both of these problems:

- Sharing information between threads is easy and fast. It is just a matter of copying data into shared (global or heap) variables. However, in order to avoid the problems that can occur when multiple threads try to update the same information, we must employ the synchronization techniques described in Chapter 30.
- Thread creation is faster than process creation—typically, ten times faster or better. (On Linux, threads are implemented using the *clone()* system call, and Table 28-3, on page 610, shows the differences in speed between *fork()* and *clone()*.) Thread creation is faster because many of the attributes that must be duplicated in a child created by *fork()* are instead shared between threads. In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables.

Besides global memory, threads also share a number of other attributes (i.e., these attributes are global to a process, rather than specific to a thread). These attributes include the following:

- process ID and parent process ID;
- process group ID and session ID;
- controlling terminal;
- process credentials (user and group IDs);
- open file descriptors;
- record locks created using *fcntl()*;
- signal dispositions;
- file system–related information: umask, current working directory, and root directory;
- interval timers (*setitimer()*) and POSIX timers (*timer_create()*);
- System V semaphore undo (*semadj*) values (Section 47.8);
- resource limits;
- CPU time consumed (as returned by *times()*);
- resources consumed (as returned by *getrusage()*); and
- nice value (set by *setpriority()* and *nice()*).

Among the attributes that are distinct for each thread are the following:

- thread ID (Section 29.5);
- signal mask;
- thread-specific data (Section 31.3);
- alternate signal stack (*sigaltstack()*);
- the *errno* variable;
- floating-point environment (see *fenv(3)*);
- realtime scheduling policy and priority (Sections 35.2 and 35.3);
- CPU affinity (Linux-specific, described in Section 35.4);
- capabilities (Linux-specific, described in Chapter 39); and
- stack (local variables and function call linkage information).

> As can be seen from Figure 29-1, all of the per-thread stacks reside within the same virtual address space. This means that, given a suitable pointer, it is possible for threads to share data on each other's stacks. This is occasionally useful, but it requires careful programming to handle the dependency that results from the fact that a local variable remains valid only for the lifetime of the stack frame in which it resides. (If a function returns, the memory region used by its stack frame may be reused by a later function call. If the thread terminates, a new thread may reuse the memory region used for the terminated thread's stack.) Failing to correctly handle this dependency can create bugs that are hard to track down.

29.2 Background Details of the Pthreads API

In the late 1980s and early 1990s, several different threading APIs existed. In 1995, POSIX.1c standardized the POSIX threads API, and this standard was later incorporated into SUSv3.

Several concepts apply to the Pthreads API as a whole, and we briefly introduce these before looking in detail at the API.

Pthreads data types

The Pthreads API defines a number of data types, some of which are listed in Table 29-1. We describe most of these data types in the following pages.

Table 29-1: Pthreads data types

Data type	Description
pthread_t	Thread identifier
pthread_mutex_t	Mutex
pthread_mutexattr_t	Mutex attributes object
pthread_cond_t	Condition variable
pthread_condattr_t	Condition variable attributes object
pthread_key_t	Key for thread-specific data
pthread_once_t	One-time initialization control context
pthread_attr_t	Thread attributes object

SUSv3 doesn't specify how these data types should be represented, and portable programs should treat them as opaque data. By this, we mean that a program should avoid any reliance on knowledge of the structure or contents of a variable of one of these types. In particular, we can't compare variables of these types using the C == operator.

Threads and *errno*

In the traditional UNIX API, *errno* is a global integer variable. However, this doesn't suffice for threaded programs. If a thread made a function call that returned an error in a global *errno* variable, then this would confuse other threads that might also be making function calls and checking *errno*. In other words, race conditions would result. Therefore, in threaded programs, each thread has its own *errno* value. On Linux, a thread-specific *errno* is achieved in a similar manner to most other UNIX implementations: *errno* is defined as a macro that expands into a function call returning a modifiable lvalue that is distinct for each thread. (Since the lvalue is modifiable, it is still possible to write assignment statements of the form *errno = value* in threaded programs.)

To summarize, the *errno* mechanism has been adapted for threads in a manner that leaves error reporting unchanged from the traditional UNIX API.

> The original POSIX.1 standard followed K&R C usage in allowing a program to declare *errno* as *extern int errno*. SUSv3 doesn't permit this usage (the change actually occurred in 1995 in POSIX.1c). Nowadays, a program is required to declare *errno* by including `<errno.h>`, which enables the implementation of a per-thread *errno*.

Return value from Pthreads functions

The traditional method of returning status from system calls and some library functions is to return 0 on success and –1 on error, with *errno* being set to indicate the error. The functions in the Pthreads API do things differently. All Pthreads functions return 0 on success or a positive value on failure. The failure value is one of the same values that can be placed in *errno* by traditional UNIX system calls.

Because each reference to *errno* in a threaded program carries the overhead of a function call, our example programs don't directly assign the return value of a Pthreads function to *errno*. Instead, we use an intermediate variable and employ our *errExitEN()* diagnostic function (Section 3.5.2), like so:

```
pthread_t *thread;
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```

Compiling Pthreads programs

On Linux, programs that use the Pthreads API must be compiled with the *cc –pthread* option. The effects of this option include the following:

- The `_REENTRANT` preprocessor macro is defined. This causes the declarations of a few reentrant functions to be exposed.
- The program is linked with the *libpthread* library (the equivalent of *–lpthread*).

> The precise options for compiling a multithreaded program vary across implementations (and compilers). Some other implementations (e.g., Tru64) also use *cc –pthread*; Solaris and HP-UX use *cc –mt*.

29.3 Thread Creation

When a program is started, the resulting process consists of a single thread, called the *initial* or *main* thread. In this section, we look at how to create additional threads.

The *pthread_create()* function creates a new thread.

```
#include <pthread.h>

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

Returns 0 on success, or a positive error number on error

The new thread commences execution by calling the function identified by *start* with the argument *arg* (i.e., *start(arg)*). The thread that calls *pthread_create()* continues execution with the next statement that follows the call. (This behavior is the same as the *glibc* wrapper function for the *clone()* system call described in Section 28.2.)

The *arg* argument is declared as *void **, meaning that we can pass a pointer to any type of object to the *start* function. Typically, *arg* points to a global or heap variable, but it can also be specified as `NULL`. If we need to pass multiple arguments to *start*, then *arg* can be specified as a pointer to a structure containing the arguments as separate fields. With judicious casting, we can even specify *arg* as an *int*.

> Strictly speaking, the C standards don't define the results of casting *int* to *void ** and vice versa. However, most C compilers permit these operations, and they produce the desired result; that is, *int j == (int) ((void *) j)*.

The return value of *start* is likewise of type *void **, and it can be employed in the same way as the *arg* argument. We'll see how this value is used when we describe the *pthread_join()* function below.

> Caution is required when using a cast integer as the return value of a thread's start function. The reason for this is that `PTHREAD_CANCELED`, the value returned when a thread is canceled (see Chapter 32), is usually some implementation-defined integer value cast to *void **. If a thread's start function returns the same integer value, then, to another thread that is doing a *pthread_join()*, it will

wrongly appear that the thread was canceled. In an application that employs thread cancellation and chooses to return cast integer values from a thread's start functions, we must ensure that a normally terminating thread does not return an integer whose value matches `PTHREAD_CANCELED` on that Pthreads implementation. A portable application would need to ensure that normally terminating threads don't return integer values that match `PTHREAD_CANCELED` on any of the implementations on which the application is to run.

The *thread* argument points to a buffer of type *pthread_t* into which the unique identifier for this thread is copied before *pthread_create()* returns. This identifier can be used in later Pthreads calls to refer to the thread.

SUSv3 explicitly notes that the implementation need not initialize the buffer pointed to by *thread* before the new thread starts executing; that is, the new thread may start running before *pthread_create()* returns to its caller. If the new thread needs to obtain its own ID, then it must do so using *pthread_self()* (described in Section 29.5).

The *attr* argument is a pointer to a *pthread_attr_t* object that specifies various attributes for the new thread. We say some more about these attributes in Section 29.8. If *attr* is specified as `NULL`, then the thread is created with various default attributes, and this is what we'll do in most of the example programs in this book.

After a call to *pthread_create()*, a program has no guarantees about which thread will next be scheduled to use the CPU (on a multiprocessor system, both threads may simultaneously execute on different CPUs). Programs that implicitly rely on a particular order of scheduling are open to the same sorts of race conditions that we described in Section 24.4. If we need to enforce a particular order of execution, we must use one of the synchronization techniques described in Chapter 30.

29.4 Thread Termination

The execution of a thread terminates in one of the following ways:

- The thread's start function performs a `return` specifying a return value for the thread.
- The thread calls *pthread_exit()* (described below).
- The thread is canceled using *pthread_cancel()* (described in Section 32.1).
- Any of the threads calls *exit()*, or the main thread performs a `return` (in the *main()* function), which causes all threads in the process to terminate immediately.

The *pthread_exit()* function terminates the calling thread, and specifies a return value that can be obtained in another thread by calling *pthread_join()*.

```
include <pthread.h>

void pthread_exit(void *retval);
```

Calling *pthread_exit()* is equivalent to performing a `return` in the thread's start function, with the difference that *pthread_exit()* can be called from any function that has been called by the thread's start function.

The *retval* argument specifies the return value for the thread. The value pointed to by *retval* should not be located on the thread's stack, since the contents of that stack become undefined on thread termination. (For example, that region of the process's virtual memory might be immediately reused by the stack for a new thread.) The same statement applies to the value given to a `return` statement in the thread's start function.

If the main thread calls *pthread_exit()* instead of calling *exit()* or performing a `return`, then the other threads continue to execute.

29.5 Thread IDs

Each thread within a process is uniquely identified by a thread ID. This ID is returned to the caller of *pthread_create()*, and a thread can obtain its own ID using *pthread_self()*.

```
include <pthread.h>

pthread_t pthread_self(void);
```

Returns the thread ID of the calling thread

Thread IDs are useful within applications for the following reasons:

- Various Pthreads functions use thread IDs to identify the thread on which they are to act. Examples of such functions include *pthread_join()*, *pthread_detach()*, *pthread_cancel()*, and *pthread_kill()*, all of which we describe in this and the following chapters.
- In some applications, it can be useful to tag dynamic data structures with the ID of a particular thread. This can serve to identify the thread that created or "owns" a data structure, or can be used by one thread to identify a specific thread that should subsequently do something with that data structure.

The *pthread_equal()* function allows us check whether two thread IDs are the same.

```
include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Returns nonzero value if *t1* and *t2* are equal, otherwise 0

For example, to check if the ID of the calling thread matches a thread ID saved in the variable *tid*, we could write the following:

```
if (pthread_equal(tid, pthread_self())
    printf("tid matches self\n");
```

The *pthread_equal()* function is needed because the *pthread_t* data type must be treated as opaque data. On Linux, *pthread_t* happens to be defined as an *unsigned long*, but on other implementations, it could be a pointer or a structure.

> In NPTL, *pthread_t* is actually a pointer that has been cast to *unsigned long*.

SUSv3 doesn't require *pthread_t* to be implemented as a scalar type; it could be a structure. Therefore, we can't portably use code such as the following to display a thread ID (though it does work on many implementations, including Linux, and is sometimes useful for debugging purposes):

```
pthread_t thr;

printf("Thread ID = %ld\n", (long) thr);        /* WRONG! */
```

In the Linux threading implementations, thread IDs are unique across processes. However, this is not necessarily the case on other implementations, and SUSv3 explicitly notes that an application can't portably use a thread ID to identify a thread in another process. SUSv3 also notes that an implementation is permitted to reuse a thread ID after a terminated thread has been joined with *pthread_join()* or after a detached thread has terminated. (We explain *pthread_join()* in the next section, and detached threads in Section 29.7.)

> POSIX thread IDs are not the same as the thread IDs returned by the Linux-specific *gettid()* system call. POSIX thread IDs are assigned and maintained by the threading implementation. The thread ID returned by *gettid()* is a number (similar to a process ID) that is assigned by the kernel. Although each POSIX thread has a unique kernel thread ID in the Linux NPTL threading implementation, an application generally doesn't need to know about the kernel IDs (and won't be portable if it depends on knowing them).

29.6 Joining with a Terminated Thread

The *pthread_join()* function waits for the thread identified by *thread* to terminate. (If that thread has already terminated, *pthread_join()* returns immediately.) This operation is termed *joining*.

```
include <pthread.h>

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

Returns 0 on success, or a positive error number on error

If *retval* is a non-`NULL` pointer, then it receives a copy of the terminated thread's return value—that is, the value that was specified when the thread performed a `return` or called *pthread_exit()*.

Calling *pthread_join()* for a thread ID that has been previously joined can lead to unpredictable behavior; for example, it might instead join with a thread created later that happened to reuse the same thread ID.

If a thread is not detached (see Section 29.7), then we must join with it using *pthread_join()*. If we fail to do this, then, when the thread terminates, it produces the thread equivalent of a zombie process (Section 26.2). Aside from wasting system resources, if enough thread zombies accumulate, we won't be able to create additional threads.

The task that *pthread_join()* performs for threads is similar to that performed by *waitpid()* for processes. However, there are some notable differences:

- Threads are peers. Any thread in a process can use *pthread_join()* to join with any other thread in the process. For example, if thread A creates thread B, which creates thread C, then it is possible for thread A to join with thread C, or vice versa. This differs from the hierarchical relationship between processes. When a parent process creates a child using *fork()*, it is the only process that can *wait()* on that child. There is no such relationship between the thread that calls *pthread_create()* and the resulting new thread.
- There is no way of saying "join with any thread" (for processes, we can do this using the call *waitpid(–1, &status, options)*); nor is there a way to do a nonblocking join (analogous to the *waitpid()* `WNOHANG` flag). There are ways to achieve similar functionality using condition variables; we show an example in Section 30.2.4.

> The limitation that *pthread_join()* can join only with a specific thread ID is intentional. The idea is that a program should join only with the threads that it "knows" about. The problem with a "join with any thread" operation stems from the fact that there is no hierarchy of threads, so such an operation could indeed join with *any* thread, including one that was privately created by a library function. (The condition-variable technique that we show in Section 30.2.4 allows a thread to join only with any other thread that it knows about.) As a consequence, the library would no longer be able to join with that thread in order to obtain its status, and it would erroneously try to join with a thread ID that had already been joined. In other words, a "join with any thread" operation is incompatible with modular program design.

Example program

The program in Listing 29-1 creates another thread and then joins with it.

Listing 29-1: A simple program using Pthreads

threads/simple_thread.c

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

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;

    printf("%s", s);

    return (void *) strlen(s);
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

threads/simple_thread.c

When we run the program in Listing 29-1, we see the following:

```
$ ./simple_thread
Message from main()
Hello world
Thread returned 12
```

Depending on how the two threads were scheduled, the order of the first two lines of output might be reversed.

29.7 Detaching a Thread

By default, a thread is *joinable*, meaning that when it terminates, another thread can obtain its return status using *pthread_join()*. Sometimes, we don't care about the thread's return status; we simply want the system to automatically clean up and remove the thread when it terminates. In this case, we can mark the thread as *detached*, by making a call to *pthread_detach()* specifying the thread's identifier in *thread*.

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

As an example of the use of *pthread_detach()*, a thread can detach itself using the following call:

```
pthread_detach(pthread_self());
```

Once a thread has been detached, it is no longer possible to use *pthread_join()* to obtain its return status, and the thread can't be made joinable again.

Detaching a thread doesn't make it immune to a call to *exit()* in another thread or a `return` in the main thread. In such an event, all threads in the process are immediately terminated, regardless of whether they are joinable or detached. To put things another way, *pthread_detach()* simply controls what happens after a thread terminates, not how or when it terminates.

29.8 Thread Attributes

We mentioned earlier that the *pthread_create() attr* argument, whose type is *pthread_attr_t*, can be used to specify the attributes used in the creation of a new thread. We won't go into the details of these attributes (for those details, see the references listed at the end of this chapter) or show the prototypes of the various Pthreads functions that can be used to manipulate a *pthread_attr_t* object. We'll just mention that these attributes include information such as the location and size of the thread's stack, the thread's scheduling policy and priority (akin to the process realtime scheduling policies and priorities described in Sections 35.2 and 35.3), and whether the thread is joinable or detached.

As an example of the use of thread attributes, the code shown in Listing 29-2 creates a new thread that is made detached at the time of thread creation (rather than subsequently, using *pthread_detach()*). This code first initializes a thread attributes structure with default values, sets the attribute required to create a detached thread, and then creates a new thread using the thread attributes structure. Once the thread has been created, the attributes object is no longer needed, and so is destroyed.

Listing 29-2: Creating a thread with the detached attribute

from **threads/detached_attrib.c**

```
pthread_t thr;
pthread_attr_t attr;
int s;

s = pthread_attr_init(&attr);            /* Assigns default values */
if (s != 0)
    errExitEN(s, "pthread_attr_init");

s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
if (s != 0)
    errExitEN(s, "pthread_attr_setdetachstate");

s = pthread_create(&thr, &attr, threadFunc, (void *) 1);
if (s != 0)
    errExitEN(s, "pthread_create");

s = pthread_attr_destroy(&attr);         /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_attr_destroy");
```

from **threads/detached_attrib.c**

29.9 Threads Versus Processes

In this section, we briefly consider some of the factors that might influence our choice of whether to implement an application as a group of threads or as a group of processes. We begin by considering the advantages of a multithreaded approach:

- Sharing data between threads is easy. By contrast, sharing data between processes requires more work (e.g., creating a shared memory segment or using a pipe).
- Thread creation is faster than process creation; context-switch time may be lower for threads than for processes.

Using threads can have some disadvantages compared to using processes:

- When programming with threads, we need to ensure that the functions we call are thread-safe or are called in a thread-safe manner. (We describe the concept of thread safety in Section 31.1.) Multiprocess applications don't need to be concerned with this.
- A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes. By contrast, processes are more isolated from one another.
- Each thread is competing for use of the finite virtual address space of the host process. In particular, each thread's stack and thread-specific data (or thread-local storage) consumes a part of the process virtual address space, which is consequently unavailable for other threads. Although the available virtual address space is large (e.g., typically 3 GB on x86-32), this factor may be a significant limitation for processes employing large numbers of threads or threads that require large amounts of memory. By contrast, separate processes can each employ the full range of available virtual memory (subject to the limitations of RAM and swap space).

The following are some other points that may influence our choice of threads versus processes:

- Dealing with signals in a multithreaded application requires careful design. (As a general principle, it is usually desirable to avoid the use of signals in multithreaded programs.) We say more about threads and signals in Section 33.2.
- In a multithreaded application, all threads must be running the same program (although perhaps in different functions). In a multiprocess application, different processes can run different programs.
- Aside from data, threads also share certain other information (e.g., file descriptors, signal dispositions, current working directory, and user and group IDs). This may be an advantage or a disadvantage, depending on the application.

29.10 Summary

In a multithreaded process, multiple threads are concurrently executing the same program. All of the threads share the same global and heap variables, but each thread has a private stack for local variables. The threads in a process also share a number of other attributes, including process ID, open file descriptors, signal dispositions, current working directory, and resource limits.

The key difference between threads and processes is the easier sharing of information that threads provide, and this is the main reason that some application designs map better onto a multithread design than onto a multiprocess design. Threads can also provide better performance for some operations (e.g., thread creation is faster than process creation), but this factor is usually secondary in influencing the choice of threads versus processes.

Threads are created using *pthread_create()*. Each thread can then independently terminate using *pthread_exit()*. (If any thread calls *exit()*, then all threads immediately terminate.) Unless a thread has been marked as detached (e.g., via a call to *pthread_detach()*), it must be joined by another thread using *pthread_join()*, which returns the termination status of the joined thread.

Further information

[Butenhof, 1996] provides an exposition of Pthreads that is both readable and thorough. [Robbins & Robbins, 2003] also provides good coverage of Pthreads. [Tanenbaum, 2007] provides a more theoretical introduction to thread concepts, covering topics such as mutexes, critical regions, conditional variables, and deadlock detection and avoidance. [Vahalia, 1996] provides background on the implementation of threads.

29.11 Exercises

29-1. What possible outcomes might there be if a thread executes the following code:

```
pthread_join(pthread_self(), NULL);
```

Write a program to see what actually happens on Linux. If we have a variable, *tid*, containing a thread ID, how can a thread prevent itself from making a call, *pthread_join(tid, NULL)*, that is equivalent to the above statement?

29-2. Aside from the absence of error checking and various variable and structure declarations, what is the problem with the following program?

```
static void *
threadFunc(void *arg)
{
    struct someStruct *pbuf = (struct someStruct *) arg;

    /* Do some work with structure pointed to by 'pbuf' */
}

int
main(int argc, char *argv[])
{
    struct someStruct buf;

    pthread_create(&thr, NULL, threadFunc, (void *) &buf);
    pthread_exit(NULL);
}
```

32

THREADS: THREAD CANCELLATION

Typically, multiple threads execute in parallel, with each thread performing its task until it decides to terminate by calling *pthread_exit()* or returning from the thread's start function.

Sometimes, it can be useful to *cancel* a thread; that is, to send it a request asking it to terminate now. This could be useful, for example, if a group of threads is performing a calculation, and one thread detects an error condition that requires the other threads to terminate. Alternatively, a GUI-driven application may provide a cancel button to allow the user to terminate a task that is being performed by a thread in the background; in this case, the main thread (controlling the GUI) needs to tell the background thread to terminate.

In this chapter, we describe the POSIX threads cancellation mechanism.

32.1 Canceling a Thread

The *pthread_cancel()* function sends a cancellation request to the specified *thread*.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

Having made the cancellation request, *pthread_cancel()* returns immediately; that is, it doesn't wait for the target thread to terminate.

Precisely what happens to the target thread, and when it happens, depends on that thread's cancellation state and type, as described in the next section.

32.2 Cancellation State and Type

The *pthread_setcancelstate()* and *pthread_setcanceltype()* functions set flags that allow a thread to control how it responds to a cancellation request.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

Both return 0 on success, or a positive error number on error

The *pthread_setcancelstate()* function sets the calling thread's cancelability state to the value given in *state*. This argument has one of the following values:

`PTHREAD_CANCEL_DISABLE`
: The thread is not cancelable. If a cancellation request is received, it remains pending until cancelability is enabled.

`PTHREAD_CANCEL_ENABLE`
: The thread is cancelable. This is the default cancelability state in newly created threads.

The thread's previous cancelability state is returned in the location pointed to by *oldstate*.

> If we are not interested in the previous cancelability state, Linux allows *oldstate* to be specified as `NULL`. This is the case on many other implementations as well; however, SUSv3 doesn't specify this feature, so portable applications can't rely on it. We should always specify a non-`NULL` value for *oldstate*.

Temporarily disabling cancellation (`PTHREAD_CANCEL_DISABLE`) is useful if a thread is executing a section of code where *all* of the steps must be completed.

If a thread is cancelable (`PTHREAD_CANCEL_ENABLE`), then the treatment of a cancellation request is determined by the thread's cancelability type, which is specified by the *type* argument in a call to *pthread_setcanceltype()*. This argument has one of the following values:

`PTHREAD_CANCEL_ASYNCHRONOUS`
: The thread may be canceled at any time (perhaps, but not necessarily, immediately). Asynchronous cancelability is rarely useful, and we defer discussion of it until Section 32.6.

`PTHREAD_CANCEL_DEFERRED`
: The cancellation remains pending until a cancellation point (see the next section) is reached. This is the default cancelability type in newly created threads. We say more about deferred cancelability in the following sections.

The thread's previous cancelability type is returned in the location pointed to by *oldtype*.

> As with the *pthread_setcancelstate() oldstate* argument, many implementations, including Linux, allow *oldtype* to be specified as `NULL` if we are not interested in the previous cancelability type. Again, SUSv3 doesn't specify this feature, and portable applications can't rely on it We should always specify a non-`NULL` value for *oldtype*.

When a thread calls *fork()*, the child inherits the calling thread's cancelability type and state. When a thread calls *exec()*, the cancelability type and state of the main thread of the new program are reset to `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED`, respectively.

32.3 Cancellation Points

When cancelability is enabled and deferred, a cancellation request is acted upon only when a thread next reaches a *cancellation point*. A cancellation point is a call to one of a set of functions defined by the implementation.

SUSv3 specifies that the functions shown in Table 32-1 *must* be cancellation points if they are provided by an implementation. Most of these are functions that are capable of blocking the thread for an indefinite period of time.

Table 32-1: Functions required to be cancellation points by SUSv3

accept()	*nanosleep()*	*sem_timedwait()*
aio_suspend()	*open()*	*sem_wait()*
clock_nanosleep()	*pause()*	*send()*
close()	*poll()*	*sendmsg()*
connect()	*pread()*	*sendto()*
creat()	*pselect()*	*sigpause()*
fcntl(F_SETLKW)	*pthread_cond_timedwait()*	*sigsuspend()*
fsync()	*pthread_cond_wait()*	*sigtimedwait()*
fdatasync()	*pthread_join()*	*sigwait()*
getmsg()	*pthread_testcancel()*	*sigwaitinfo()*
getpmsg()	*putmsg()*	*sleep()*
lockf(F_LOCK)	*putpmsg()*	*system()*
mq_receive()	*pwrite()*	*tcdrain()*
mq_send()	*read()*	*usleep()*
mq_timedreceive()	*readv()*	*wait()*
mq_timedsend()	*recv()*	*waitid()*
msgrcv()	*recvfrom()*	*waitpid()*
msgsnd()	*recvmsg()*	*write()*
msync()	*select()*	*writev()*

In addition to the functions in Table 32-1, SUSv3 specifies a larger group of functions that an implementation *may* define as cancellation points. These include the *stdio* functions, the *dlopen* API, the *syslog* API, *nftw()*, *popen()*, *semop()*, *unlink()*, and various functions that retrieve information from system files such as the *utmp* file. A portable program must correctly handle the possibility that a thread may be canceled when calling these functions.

SUSv3 specifies that aside from the two lists of functions that must and may be cancellation points, none of the other functions in the standard may act as cancellation points (i.e., a portable program doesn't need to handle the possibility that calling these other functions could precipitate thread cancellation).

SUSv4 adds *openat()* to the list of functions that must be cancellation points, and removes *sigpause()* (it moves to the list of functions that *may* be cancellation points) and *usleep()* (which is dropped from the standard).

> An implementation is free to mark additional functions that are not specified in the standard as cancellation points. Any function that might block (perhaps because it might access a file) is a likely candidate to be a cancellation point. Within *glibc*, many nonstandard functions are marked as cancellation points for this reason.

Upon receiving a cancellation request, a thread whose cancelability is enabled and deferred terminates when it next reaches a cancellation point. If the thread was not detached, then some other thread in the process must join with it, in order to prevent it from becoming a zombie thread. When a canceled thread is joined, the value returned in the second argument to *pthread_join()* is a special thread return value: `PTHREAD_CANCELED`.

Example program

Listing 32-1 shows a simple example of the use of *pthread_cancel()*. The main program creates a thread that executes an infinite loop, sleeping for a second and printing the value of a loop counter. (This thread will terminate only if it is sent a cancellation request or if the process exits.) Meanwhile, the main program sleeps for 3 seconds, and then sends a cancellation request to the thread that it created. When we run this program, we see the following:

```
$ ./t_pthread_cancel
New thread started
Loop 1
Loop 2
Loop 3
Thread was canceled
```

Listing 32-1: Canceling a thread with *pthread_cancel()*

threads/thread_cancel.c

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

static void *
threadFunc(void *arg)
{
    int j;
    printf("New thread started\n");     /* May be a cancellation point */
    for (j = 1; ; j++) {
        printf("Loop %d\n", j);         /* May be a cancellation point */
        sleep(1);                       /* A cancellation point */
    }

    /* NOTREACHED */
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t thr;
    int s;
    void *res;

    s = pthread_create(&thr, NULL, threadFunc, NULL);
    if (s != 0)
        errExitEN(s, "pthread_create");

    sleep(3);                           /* Allow new thread to run a while */

    s = pthread_cancel(thr);
    if (s != 0)
        errExitEN(s, "pthread_cancel");

    s = pthread_join(thr, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    if (res == PTHREAD_CANCELED)
        printf("Thread was canceled\n");
    else
        printf("Thread was not canceled (should not happen!)\n");

    exit(EXIT_SUCCESS);
}
```
threads/thread_cancel.c

32.4 Testing for Thread Cancellation

In Listing 32-1, the thread created by *main()* accepted the cancellation request because it executed a function that was a cancellation point (*sleep()* is a cancellation point; *printf()* may be one). However, suppose a thread executes a loop that contains no cancellation points (e.g., a compute-bound loop). In this case, the thread would never honor the cancellation request.

The purpose of *pthread_testcancel()* is simply to be a cancellation point. If a cancellation is pending when this function is called, then the calling thread is terminated.

```
#include <pthread.h>

void pthread_testcancel(void);
```

A thread that is executing code that does not otherwise include cancellation points can periodically call *pthread_testcancel()* to ensure that it responds in a timely fashion to a cancellation request sent by another thread.

32.5 Cleanup Handlers

If a thread with a pending cancellation were simply terminated when it reached a cancellation point, then shared variables and Pthreads objects (e.g., mutexes) might be left in an inconsistent state, perhaps causing the remaining threads in the process to produce incorrect results, deadlock, or crash. To get around this problem, a thread can establish one or more *cleanup handlers*—functions that are automatically executed if the thread is canceled. A cleanup handler can perform tasks such as modifying the values of global variables and unlocking mutexes before the thread is terminated.

Each thread can have a stack of cleanup handlers. When a thread is canceled, the cleanup handlers are executed working down from the top of the stack; that is, the most recently established handler is called first, then the next most recently established, and so on. When all of the cleanup handlers have been executed, the thread terminates.

The *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions respectively add and remove handlers on the calling thread's stack of cleanup handlers.

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void*), void *arg);
void pthread_cleanup_pop(int execute);
```

The *pthread_cleanup_push()* function adds the function whose address is specified in *routine* to the top of the calling thread's stack of cleanup handlers. The *routine* argument is a pointer to a function that has the following form:

```
void
routine(void *arg)
{
    /* Code to perform cleanup */
}
```

The *arg* value given to *pthread_cleanup_push()* is passed as the argument of the cleanup handler when it is invoked. This argument is typed as *void **, but, using judicious casting, other data types can be passed in this argument.

Typically, a cleanup action is needed only if a thread is canceled during the execution of a particular section of code. If the thread reaches the end of that section without being canceled, then the cleanup action is no longer required. Thus, each call to *pthread_cleanup_push()* has an accompanying call to *pthread_cleanup_pop()*. This function removes the topmost function from the stack of cleanup handlers. If the *execute* argument is nonzero, the handler is also executed. This is convenient if we want to perform the cleanup action even if the thread was not canceled.

Although we have described *pthread_cleanup_push()* and *pthread_cleanup_pop()* as functions, SUSv3 permits them to be implemented as macros that expand to statement sequences that include an opening (`{`) and closing (`}`) brace, respectively. Not all UNIX implementations do things this way, but Linux and many others do. This means that each use of *pthread_cleanup_push()* must be paired with exactly one corresponding *pthread_cleanup_pop()* in the same lexical block. (On implementations that do things this way, variables declared between the *pthread_cleanup_push()* and *pthread_cleanup_pop()* will be limited to that lexical scope.) For example, it is not correct to write code such as the following:

```
pthread_cleanup_push(func, arg);
...
if (cond) {
    pthread_cleanup_pop(0);
}
```

As a coding convenience, any cleanup handlers that have not been popped are also executed automatically if a thread terminates by calling *pthread_exit()* (but not if it does a simple `return`).

Example program

The program in Listing 32-2 provides a simple example of the use of a cleanup handler. The main program creates a thread ⑧ whose first actions are to allocate a block of memory ③ whose location is stored in *buf*, and then lock the mutex *mtx* ④. Since the thread may be canceled, it uses *pthread_cleanup_push()* ⑤ to install a cleanup handler that is called with the address stored in *buf*. If it is invoked, the cleanup handler deallocates the freed memory ① and unlocks the mutex ②.

The thread then enters a loop waiting for the condition variable *cond* to be signaled ⑥. This loop will terminate in one of two ways, depending on whether the program is supplied with a command-line argument:

- If no command-line argument is supplied, the thread is canceled by *main()* ⑨. In this case, cancellation will occur at the call to *pthread_cond_wait()* ⑥, which is one of the cancellation points shown in Table 32-1. As part of cancellation, the cleanup handler established using *pthread_cleanup_push()* is invoked automatically.
- If a command-line argument is supplied, the condition variable is signaled ⑩ after the associated global variable, *glob*, is first set to a nonzero value. In this case, the thread falls through to execute *pthread_cleanup_pop()* ⑦, which, given a nonzero argument, also causes the cleanup handler to be invoked.

The main program joins with the terminated thread ⑪, and reports whether the thread was canceled or terminated normally.

Listing 32-2: Using cleanup handlers

threads/thread_cleanup.c

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

   static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
   static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
   static int glob = 0;                    /* Predicate variable */

   static void     /* Free memory pointed to by 'arg' and unlock mutex */
   cleanupHandler(void *arg)
   {
       int s;

       printf("cleanup: freeing block at %p\n", arg);
①      free(arg);

       printf("cleanup: unlocking mutex\n");
②      s = pthread_mutex_unlock(&mtx);
       if (s != 0)
           errExitEN(s, "pthread_mutex_unlock");
   }

   static void *
   threadFunc(void *arg)
   {
       int s;
       void *buf = NULL;                   /* Buffer allocated by thread */

③      buf = malloc(0x10000);              /* Not a cancellation point */
       printf("thread:  allocated memory at %p\n", buf);

④      s = pthread_mutex_lock(&mtx);       /* Not a cancellation point */
       if (s != 0)
           errExitEN(s, "pthread_mutex_lock");

⑤      pthread_cleanup_push(cleanupHandler, buf);

       while (glob == 0) {
⑥          s = pthread_cond_wait(&cond, &mtx);    /* A cancellation point */
           if (s != 0)
               errExitEN(s, "pthread_cond_wait");
       }

       printf("thread:  condition wait loop completed\n");
⑦      pthread_cleanup_pop(1);             /* Executes cleanup handler */
       return NULL;
   }

   int
   main(int argc, char *argv[])
   {
       pthread_t thr;
       void *res;
       int s;
⑧       s = pthread_create(&thr, NULL, threadFunc, NULL);
        if (s != 0)
            errExitEN(s, "pthread_create");

        sleep(2);                   /* Give thread a chance to get started */

        if (argc == 1) {            /* Cancel thread */
            printf("main:    about to cancel thread\n");
⑨           s = pthread_cancel(thr);
            if (s != 0)
                errExitEN(s, "pthread_cancel");

        } else {                    /* Signal condition variable */
            printf("main:    about to signal condition variable\n");
            glob = 1;
⑩           s = pthread_cond_signal(&cond);
            if (s != 0)
                errExitEN(s, "pthread_cond_signal");
        }

⑪       s = pthread_join(thr, &res);
        if (s != 0)
            errExitEN(s, "pthread_join");
        if (res == PTHREAD_CANCELED)
            printf("main:    thread was canceled\n");
        else
            printf("main:    thread terminated normally\n");

        exit(EXIT_SUCCESS);
    }
```
threads/thread_cleanup.c

If we invoke the program in Listing 32-2 without any command-line arguments, then *main()* calls *pthread_cancel()*, the cleanup handler is invoked automatically, and we see the following:

```
$ ./thread_cleanup
thread:  allocated memory at 0x804b050
main:    about to cancel thread
cleanup: freeing block at 0x804b050
cleanup: unlocking mutex
main:    thread was canceled
```

If we invoke the program with a command-line argument, then *main()* sets *glob* to 1 and signals the condition variable, the cleanup handler is invoked by *pthread_cleanup_pop()*, and we see the following:

```
$ ./thread_cleanup s
thread:  allocated memory at 0x804b050
main:    about to signal condition variable
thread:  condition wait loop completed
cleanup: freeing block at 0x804b050
cleanup: unlocking mutex
main:    thread terminated normally
```

32.6 Asynchronous Cancelability

When a thread is made asynchronously cancelable (cancelability type `PTHREAD_CANCEL_ASYNCHRONOUS`), it may be canceled at any time (i.e., at any machine-language instruction); delivery of a cancellation is not held off until the thread next reaches a cancellation point.

The problem with asynchronous cancellation is that, although cleanup handlers are still invoked, the handlers have no way of determining the state of a thread. In the program in Listing 32-2, which employs the deferred cancelability type, the thread can be canceled only when it executes the call to *pthread_cond_wait()*, which is the only cancellation point. By this time, we know that *buf* has been initialized to point to a block of allocated memory and that the mutex *mtx* has been locked. However, with asynchronous cancelability, the thread could be canceled at any point; for example, before the *malloc()* call, between the *malloc()* call and locking the mutex, or after locking the mutex. The cleanup handler has no way of knowing where cancellation has occurred, or precisely which cleanup steps are required. Furthermore, the thread might even be canceled *during* the *malloc()* call, after which chaos is likely to result (Section 7.1.3).

As a general principle, an asynchronously cancelable thread can't allocate any resources or acquire any mutexes, semaphores, or locks. This precludes the use of a wide range of library functions, including most of the Pthreads functions. (SUSv3 makes exceptions for *pthread_cancel()*, *pthread_setcancelstate()*, and *pthread_setcanceltype()*, which are explicitly required to be *async-cancel-safe*; that is, an implementation must make them safe to call from a thread that is asynchronously cancelable.) In other words, there are few circumstances where asynchronous cancellation is useful. One such circumstance is canceling a thread that is in a compute-bound loop.

32.7 Summary

The *pthread_cancel()* function allows one thread to send another thread a cancellation request, which is a request that the target thread should terminate.

How the target thread reacts to this request is determined by its cancelability state and type. If the cancelability state is currently set to disabled, the request will remain pending until the cancelability state is set to enabled. If cancelability is enabled, the cancelability type determines when the target thread reacts to the request. If the type is deferred, the cancellation occurs when the thread next calls one of a number of functions specified as cancellation points by SUSv3. If the type is asynchronous, cancellation may occur at any time (this is rarely useful).

A thread can establish a stack of cleanup handlers, which are programmer-defined functions that are invoked automatically to perform cleanups (e.g., restoring the states of shared variables, or unlocking mutexes) if the thread is canceled.

Further information

Refer to the sources of further information listed in Section 29.10.