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.)
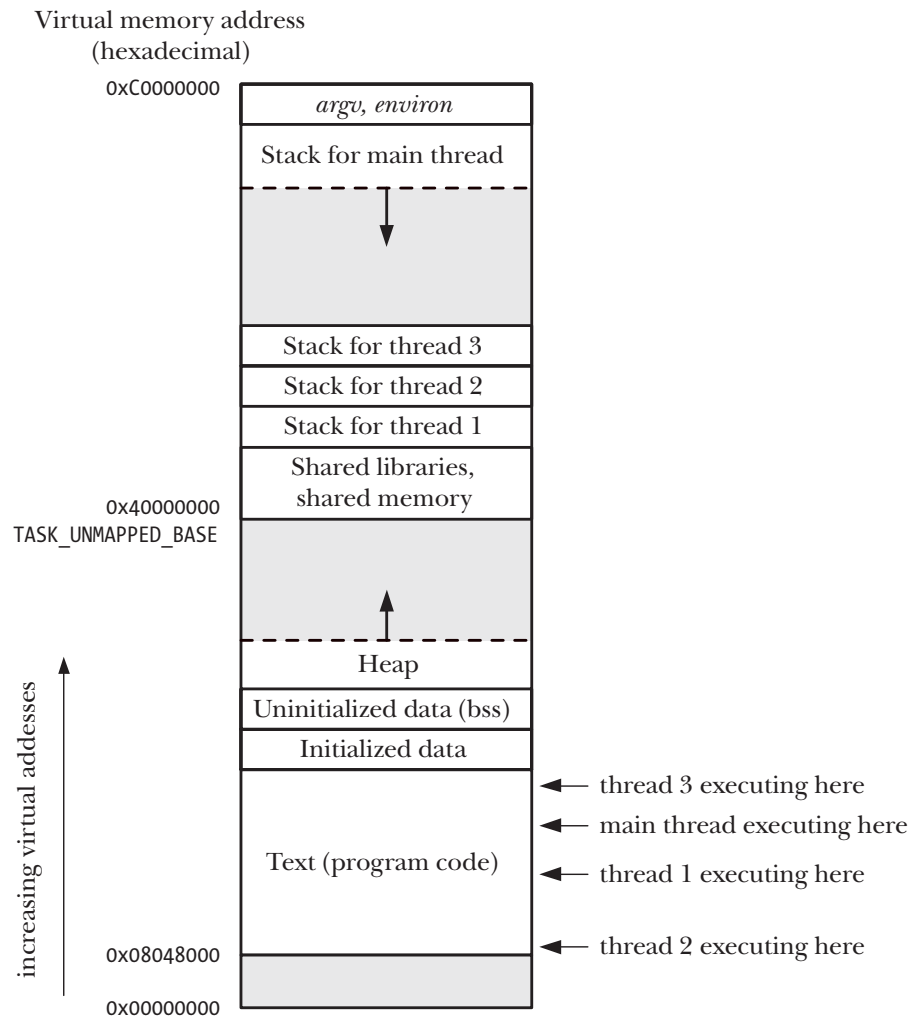
Virtual memory address
(hexadecimal)

0xC0000000	*argv, environ*
	Stack for main thread
	Stack for thread 3
	Stack for thread 2
	Stack for thread 1
	Shared libraries, shared memory
0x40000000 TASK_UNMAPPED_BASE	
	Heap
	Uninitialized data (bss)
	Initialized data
	Text (program code)
0x08048000	
0x00000000	

increasing virtual addesses

⟵ thread 3 executing here
⟵ main thread executing here
⟵ thread 1 executing here
⟵ thread 2 executing here

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 fol-
lowing 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);
}
```