33

THREADS: FURTHER DETAILS

This chapter provides further details on various aspects of POSIX threads. We discuss the interaction of threads with aspects of the traditional UNIX API—in particular, signals and the process control primitives (*fork()*, *exec()*, and *_exit()*). We also provide an overview of the two POSIX threads implementations available on Linux—LinuxThreads and NPTL—and note where each of these implementations deviates from the SUSv3 specification of Pthreads.

33.1 Thread Stacks

Each thread has its own stack whose size is fixed when the thread is created. On Linux/x86-32, for all threads other than the main thread, the default size of the per-thread stack is 2 MB. (On some 64-bit architectures, the default size is higher; for example, it is 32 MB on IA-64.) The main thread has a much larger space for stack growth (refer to Figure 29-1, on page 618).

Occasionally, it is useful to change the size of a thread's stack. The *pthread_attr_setstacksize()* function sets a thread attribute (Section 29.8) that determines the size of the stack in threads created using the thread attributes object. The related *pthread_attr_setstack()* function can be used to control both the size and the location of the stack, but setting the location of a stack can decrease application portability. The manual pages provide details of these functions.

One reason to change the size of per-thread stacks is to allow for larger stacks for threads that allocate large automatic variables or make nested function calls of

great depth (perhaps because of recursion). Alternatively, an application may want to reduce the size of per-thread stacks to allow for a greater number of threads within a process. For example, on x86-32, where the user-accessible virtual address space is 3 GB, the default stack size of 2 MB means that we can create a maximum of around 1500 threads. (The precise maximum depends on how much virtual memory is consumed by the text and data segments, shared libraries, and so on.) The minimum stack that can be employed on a particular architecture can be determined by calling *sysconf(_SC_THREAD_STACK_MIN)*. For the NPTL implementation on Linux/x86-32, this call returns the value 16,384.

> Under the NPTL threading implementation, if the stack size resource limit (`RLIMIT_STACK`) is set to anything other than *unlimited*, then it is used as the default stack size when creating new threads. This limit must be set *before* the program is executed, typically by using the *ulimit −s* shell built-in command (*limit stacksize* in the C shell) before executing the program. It is not sufficient to use *setrlimit()* within the main program to set the limit, because NPTL makes its determination of the default stack size during the run-time initialization that occurs before *main()* is invoked.

33.2 Threads and Signals

The UNIX signal model was designed with the UNIX process model in mind, and predated the arrival of Pthreads by a couple of decades. As a result, there are some significant conflicts between the signal and thread models. These conflicts arose primarily from the need to maintain the traditional signal semantics for single-threaded processes (i.e., the signal semantics of traditional programs should not be changed by Pthreads), while at the same time developing a signal model that would be usable within a multithreaded process.

The differences between the signal and thread models mean that combining signals and threads is complex, and should be avoided whenever possible. Nevertheless, sometimes we must deal with signals in a threaded program. In this section, we discuss the interactions between threads and signals, and describe various functions that are useful in threaded programs that deal with signals.

33.2.1 How the UNIX Signal Model Maps to Threads

To understand how UNIX signals map to the Pthreads model, we need to know which aspects of the signal model are process-wide (i.e., are shared by all of the threads in the process) as opposed to those aspects that are specific to individual threads within the process. The following list summarizes the key points:

- Signal actions are process-wide. If any unhandled signal whose default action is *stop* or *terminate* is delivered to any thread in a process, then all of the threads in the process are stopped or terminated.
- Signal dispositions are process-wide; all threads in a process share the same disposition for each signal. If one thread uses *sigaction()* to establish a handler for, say, `SIGINT`, then that handler may be invoked from any thread to which the `SIGINT` is delivered. Similarly, if one thread sets the disposition of a signal to *ignore*, then that signal is ignored by all threads.

- A signal may be directed to either the process as a whole or to a specific thread. A signal is thread-directed if:
  - it is generated as the direct result of the execution of a specific hardware instruction within the context of the thread (i.e., the hardware exceptions described in Section 22.4: `SIGBUS`, `SIGFPE`, `SIGILL`, and `SIGSEGV`);
  - it is a `SIGPIPE` signal generated when the thread tried to write to a broken pipe; or
  - it is sent using *pthread_kill()* or *pthread_sigqueue()*, which are functions (described in Section 33.2.3) that allow one thread to send a signal to another thread within the same process.

  All signals generated by other mechanisms are process-directed. Examples are signals sent from another process using *kill()* or *sigqueue()*; signals such as `SIGINT` and `SIGTSTP`, generated when the user types one of the terminal special characters that generate a signal; and signals generated for software events such as the resizing of a terminal window (`SIGWINCH`) or the expiration of a timer (e.g., `SIGALRM`).
- When a signal is delivered to a multithreaded process that has established a signal handler, the kernel arbitrarily selects one thread in the process to which to deliver the signal and invokes the handler in that thread. This behavior is consistent with maintaining the traditional signal semantics. It would not make sense for a process to perform the signal handling actions multiple times in response to a single signal.
- The signal mask is per-thread. (There is no notion of a process-wide signal mask that governs all threads in a multithreaded process.) Threads can independently block or unblock different signals using *pthread_sigmask()*, a new function defined by the Pthreads API. By manipulating the per-thread signal masks, an application can control which thread(s) may handle a signal that is directed to the whole process.
- The kernel maintains a record of the signals that are pending for the process as a whole, as well as a record of the signals that are pending for each thread. A call to *sigpending()* returns the union of the set of signals that are pending for the process and those that are pending for the calling thread. In a newly created thread, the per-thread set of pending signals is initially empty. A thread-directed signal can be delivered only to the target thread. If the thread is blocking the signal, it will remain pending until the thread unblocks the signal (or terminates).
- If a signal handler interrupts a call to *pthread_mutex_lock()*, then the call is always automatically restarted. If a signal handler interrupts a call to *pthread_cond_wait()*, then the call either is restarted automatically (this is what Linux does) or returns 0, indicating a spurious wake-up (in which case a well-designed application will recheck the corresponding predicate and restart the call, as described in Section 30.2.3). SUSv3 requires these two functions to behave as described here.
- The alternate signal stack is per-thread (refer to the description of *sigaltstack()* in Section 21.3). A newly created thread doesn't inherit the alternate signal stack from its creator.

More precisely, SUSv3 specifies that there is a separate alternate signal stack for each kernel scheduling entity (KSE). On a system with a 1:1 threading implementation, as on Linux, there is one KSE per thread (see Section 33.4).

33.2.2 Manipulating the Thread Signal Mask

When a new thread is created, it inherits a copy of the signal mask of the thread that created it. A thread can use *pthread_sigmask()* to change its signal mask, to retrieve the existing mask, or both.

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
                                        Returns 0 on success, or a positive error number on error
```

Other than the fact that it operates on the thread signal mask, the use of *pthread_sigmask()* is the same as the use of *sigprocmask()* (Section 20.10).

SUSv3 notes that the use of *sigprocmask()* within a multithreaded program is unspecified. We can't portably employ *sigprocmask()* in a multithreaded program. In practice, *sigprocmask()* and *pthread_sigmask()* are identical on many implementations, including Linux.

33.2.3 Sending a Signal to a Thread

The *pthread_kill()* function sends the signal *sig* to another thread in the same process. The target thread is identified by the argument *thread*.

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
                                        Returns 0 on success, or a positive error number on error
```

Because a thread ID is guaranteed to be unique only within a process (see Section 29.5), we can't use *pthread_kill()* to send a signal to a thread in another process.

The *pthread_kill()* function is implemented using the Linux-specific *tgkill(tgid, tid, sig)* system call, which sends the signal *sig* to the thread identified by *tid* (a kernel thread ID of the type returned by *gettid()*) within the thread group identified by *tgid*. See the *tgkill(2)* manual page for further details.

The Linux-specific *pthread_sigqueue()* function combines the functionality of *pthread_kill()* and *sigqueue()* (Section 22.8.1): it sends a signal with accompanying data to another thread in the same process.

```
#define _GNU_SOURCE
#include <signal.h>

int pthread_sigqueue(pthread_t thread, int sig, const union sigval value);
```

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

As with *pthread_kill()*, *sig* specifies the signal to be sent, and *thread* identifies the target thread. The *value* argument specifies the data to accompany the signal, and is used in the same way as the equivalent argument of *sigqueue()*.

> The *pthread_sigqueue()* function was added to *glibc* in version 2.11 and requires support from the kernel. This support is provided by the *rt_tgsigqueueinfo()* system call, which was added in Linux 2.6.31.

33.2.4 Dealing with Asynchronous Signals Sanely

In Chapters 20 to 22, we discussed various factors—such as reentrancy issues, the need to restart interrupted system calls, and avoiding race conditions—that can make it complex to deal with asynchronously generated signals via signal handlers. Furthermore, none of the functions in the Pthreads API is among the set of async-signal-safe functions that we can safely call from within a signal handler (Section 21.1.2). For these reasons, multithreaded programs that must deal with asynchronously generated signals generally should not use a signal handler as the mechanism to receive notification of signal delivery. Instead, the preferred approach is the following:

- All threads block all of the asynchronous signals that the process might receive. The simplest way to do this is to block the signals in the main thread before any other threads are created. Each subsequently created thread will inherit a copy of the main thread's signal mask.
- Create a single dedicated thread that accepts incoming signals using *sigwaitinfo()*, *sigtimedwait()*, or *sigwait()*. We described *sigwaitinfo()* and *sigtimedwait()* in Section 22.10. We describe *sigwait()* below.

The advantage of this approach is that asynchronously generated signals are received synchronously. As it accepts incoming signals, the dedicated thread can safely modify shared variables (under mutex control) and call non-async-signal-safe functions. It can also signal condition variables, and employ other thread and process communication and synchronization mechanisms.

The *sigwait()* function waits for the delivery of one of the signals in the signal set pointed to by *set*, accepts that signal, and returns it in *sig*.

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);
```

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

The operation of *sigwait()* is the same as *sigwaitinfo()*, except that:

- instead of returning a *siginfo_t* structure describing the signal, *sigwait()* returns just the signal number; and
- the return value is consistent with other thread-related functions (rather than the 0 or −1 returned by traditional UNIX system calls).

If multiple threads are waiting for the same signal with *sigwait()*, only one of the threads will actually accept the signal when it arrives. Which of the threads this will be is indeterminate.

33.3 Threads and Process Control

Like the signals mechanism, *exec()*, *fork()*, and *exit()* predate the Pthreads API. In the following paragraphs, we note some details concerning the use of these system calls in threaded programs.

Threads and *exec()*

When any thread calls one of the *exec()* functions, the calling program is completely replaced. All threads, except the one that called *exec()*, vanish immediately. None of the threads executes destructors for thread-specific data or calls cleanup handlers. All of the (process-private) mutexes and condition variables belonging to the process also disappear. After an *exec()*, the thread ID of the remaining thread is unspecified.

Threads and *fork()*

When a multithreaded process calls *fork()*, only the calling thread is replicated in the child process. (The ID of the thread in the child is the same as the ID of the thread that called *fork()* in the parent.) All of the other threads vanish in the child; no thread-specific data destructors or cleanup handlers are executed for those threads. This can lead to various problems:

- Although only the calling thread is replicated in the child, the states of global variables, as well as all Pthreads objects such as mutexes and condition variables, are preserved in the child. (This is so because these Pthreads objects are allocated within the parent's memory, and the child gets a duplicate of that memory.) This can lead to tricky scenarios. For example, suppose that another thread had locked a mutex at the time of the *fork()* and is part-way through updating a global data structure. In this case, the thread in the child would not be able to unlock the mutex (since it is not the mutex owner) and would block if it tried to acquire the mutex. Furthermore, the child's copy of the global data structure is probably in an inconsistent state, because the thread that was updating it vanished part-way through the update.
- Since destructors for thread-specific data and cleanup handlers are not called, a *fork()* in a multithreaded program can cause memory leaks in the child. Furthermore, the thread-specific data items created by other threads are likely to be inaccessible to the thread in the new child, since it doesn't have pointers referring to these items.

Because of these problems, the usual recommendation is that the only use of *fork()* in a multithreaded process should be one that is followed by an immediate *exec()*. The *exec()* causes all of the Pthreads objects in the child process to disappear as the new program overwrites the memory of the process.

For programs that must use a *fork()* that is not followed by an *exec()*, the Pthreads API provides a mechanism for defining *fork handlers*. Fork handlers are established using a *pthread_atfork()* call of the following form:

```
pthread_atfork(prepare_func, parent_func, child_func);
```

Each *pthread_atfork()* call adds *prepare_func* to a list of functions that will be automatically executed (in reverse order of registration) before the new child process is created when *fork()* is called. Similarly, *parent_func* and *child_func* are added to a list functions that will be called automatically (in order of registration), in, respectively, the parent and child process, just before *fork()* returns.

Fork handlers are sometimes useful for library code that makes use of threads. In the absence of fork handlers, there would be no way for the library to deal with applications that naively make use of the library and call *fork()*, unaware that the library has created some threads.

The child produced by *fork()* inherits fork handlers from the thread that called *fork()*. During an *exec()*, fork handlers are not preserved (they can't be, since the code of the handlers is overwritten during the *exec()*).

Further details on fork handlers, and examples of their use, can be found in [Butenhof, 1996].

> On Linux, fork handlers are not called if a program using the NPTL threading library calls *vfork()*. However, in a program using LinuxThreads, fork handlers are called in this case.

Threads and *exit()*

If any thread calls *exit()* or, equivalently, the main thread does a `return`, all threads immediately vanish; no thread-specific data destructors or cleanup handlers are executed.

33.4 Thread Implementation Models

In this section, we go into some theory, briefly considering three different models for implementing a threading API. This provides useful background for Section 33.5, where we consider the Linux threading implementations. The differences between these implementation models hinge on how threads are mapped onto *kernel scheduling entities* (KSEs), which are the units to which the kernel allocates the CPU and other system resources. (In traditional UNIX implementations that predate threads, the term *kernel scheduling entity* is synonymous with the term *process*.)

Many-to-one (M:1) implementations (user-level threads)

In M:1 threading implementations, all of the details of thread creation, scheduling, and synchronization (mutex locking, waiting on condition variables, and so on) are

handled entirely within the process by a user-space threading library. The kernel knows nothing about the existence of multiple threads within the process.

M:1 implementations have a few advantages. The greatest advantage is that many threading operations—for example, creating and terminating a thread, context switching between threads, and mutex and condition variable operations—are fast, since a switch to kernel mode is not required. Furthermore, since kernel support for the threading library is not required, an M:1 implementation can be relatively easily ported from one system to another.

However, M:1 implementations suffer from some serious disadvantages:

- When a thread makes a system call such as *read()*, control passes from the user-space threading library to the kernel. This means that if the *read()* call blocks, then all threads in the process are blocked.
- The kernel can't schedule the threads of a process. Since the kernel is unaware of the existence of multiple threads within the process, it can't schedule the separate threads to different processors on multiprocessor hardware. Nor is it possible to meaningfully assign a thread in one process a higher priority than a thread in another process, since the scheduling of the threads is handled entirely within the process.

One-to-one (1:1) implementations (kernel-level threads)

In a 1:1 threading implementation, each thread maps onto a separate KSE. The kernel handles each thread's scheduling separately. Thread synchronization operations are implemented using system calls into the kernel.

1:1 implementations eliminate the disadvantages suffered by M:1 implementations. A blocking system call does not cause all of the threads in a process to block, and the kernel can schedule the threads of a process onto different CPUs on multiprocessor hardware.

However, operations such as thread creation, context switching, and synchronization are slower on a 1:1 implementations, since a switch into kernel mode is required. Furthermore, the overhead required to maintain a separate KSE for each of the threads in an application that contains a large number of threads may place a significant load on the kernel scheduler, degrading overall system performance.

Despite these disadvantages, a 1:1 implementation is usually preferred over an M:1 implementation. Both of the Linux threading implementations—LinuxThreads and NPTL—employ the 1:1 model.

> During the development of NPTL, significant effort went into rewriting the kernel scheduler and devising a threading implementation that would allow the efficient execution of multithreaded processes containing many thousands of threads. Subsequent testing showed that this goal was achieved.

Many-to-many (M:N) implementations (two-level model)

M:N implementations aim to combine the advantages of the 1:1 and M:1 models, while eliminating their disadvantages.

In the M:N model, each process can have multiple associated KSEs, and several threads may map to each KSE. This design permits the kernel to distribute the threads of an application across multiple CPUs, while eliminating the possible scaling problems associated with applications that employ large numbers of threads.

The most significant disadvantage of the M:N model is complexity. The task of thread scheduling is shared between the kernel and the user-space threading library, which must cooperate and communicate information with one another. Managing signals according to the requirements of SUSv3 is also complex under an M:N implementation.

> An M:N implementation was initially considered for the NPTL threading implementation, but rejected as requiring changes to the kernel that were too wide ranging and perhaps unnecessary, given the ability of the Linux scheduler to scale well, even when dealing with large numbers of KSEs.

33.5 Linux Implementations of POSIX Threads

Linux has two main implementations of the Pthreads API:

- *LinuxThreads*: This is the original Linux threading implementation, developed by Xavier Leroy.
- *NPTL (Native POSIX Threads Library)*: This is the modern Linux threading implementation, developed by Ulrich Drepper and Ingo Molnar as a successor to LinuxThreads. NPTL provides performance that is superior to LinuxThreads, and it adheres more closely to the SUSv3 specification for Pthreads. Support for NPTL required changes to the kernel, and these changes appeared in Linux 2.6.

  > For a while, it appeared that the successor to LinuxThreads would be another implementation, called Next Generation POSIX Threads (NGPT), a threading implementation developed at IBM. NGPT employed an M:N design and performed significantly better than LinuxThreads. However, the NPTL developers decided to pursue a new implementation. This approach was justified—the 1:1-design NPTL was shown to perform better than NGPT. Following the release of NPTL, development of NGPT was discontinued.

In the following sections, we consider further details of these two implementations, and note the points where they deviate from the SUSv3 requirements for Pthreads.

At this point, it is worth emphasizing that the LinuxThreads implementation is now obsolete; it is not supported in *glibc* 2.4 and later. All new thread library development occurs only in NPTL.

33.5.1 LinuxThreads

For many years, LinuxThreads was the main threading implementation on Linux, and it was sufficient for implementing a variety of threaded applications. The essentials of the LinuxThreads implementation are as follows:

- Threads are created using a *clone()* call that specifies the following flags:

  ```
  CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
  ```

  This means that LinuxThreads threads share virtual memory, file descriptors, file system–related information (umask, root directory, and current working directory), and signal dispositions. However, threads don't share process IDs and parent process IDs.

- In addition to the threads created by the application, LinuxThreads creates an additional "manager" thread that handles thread creation and termination.
- The implementation uses signals for its internal operation. With kernels that support realtime signals (Linux 2.2 and later), the first three realtime signals are used. With older kernels, `SIGUSR1` and `SIGUSR2` are used. Applications can't use these signals. (The use of signals results in high latency for various thread synchronization operations.)

LinuxThreads deviations from specified behavior

LinuxThreads doesn't conform to the SUSv3 specification for Pthreads on a number of points. (The LinuxThreads implementation was constrained by the kernel features available at the time that it was developed; it was as conformant as practicable within those constraints.) The following list summarizes the nonconformances:

- Calls to *getpid()* return a different value in each of the threads of a process. Calls to *getppid()* reflect the fact that every thread other than the main thread is created by the process's manager thread (i.e., *getppid()* returns the process ID of the manager thread). Calls to *getppid()* in the other threads should return the same value as a call to *getppid()* in the main thread.
- If one thread creates a child using *fork()*, then any other thread should be able to obtain the termination status of that child using *wait()* (or similar). However, this is not so; only the thread that created the child process can *wait()* for it.
- If a thread calls *exec()*, then, as required by SUSv3, all other threads are terminated. However, if the *exec()* is done from any thread other than the main thread, then the resulting process will have the same process ID as the calling thread—that is, a process ID that is different from the main thread's process ID. According to SUSv3, the process ID should be the same as that of the main thread.
- Threads don't share credentials (user and group IDs). When a multithreaded process is executing a set-user-ID program, this can lead to scenarios in which one thread can't send a signal to another thread using *pthread_kill()*, because the credentials of the two threads have been changed in such a way that the sending thread no longer has permission to signal the target thread (refer to Figure 20-2, on page 403). Furthermore, since the LinuxThreads implementation uses signals internally, various Pthreads operations can fail or hang if a thread changes its credentials.
- Various aspects of the SUSv3 specification for the interaction between threads and signals are not honored:
  - A signal that is sent to a process using *kill()* or *sigqueue()* should be delivered to, and handled by, an arbitrary thread in the target process that is not blocking the signal. However, since LinuxThreads threads have different process IDs, a signal can be targeted only at a specific thread. If that thread is blocking the signal, it remains pending, even if there are other threads that are not blocking the signal.

- LinuxThreads doesn't support the notion of signals that are pending for a process as whole; only per-thread pending signals are supported.
- If a signal is directed at a process group that contains a multithreaded application, then the signal will be handled by all threads in the application (i.e., all threads that have established a signal handler), rather than by a single (arbitrary) thread. Such a signal may, for example, be generated by typing one of the terminal characters that generates a job-control signal for the foreground process group.
- The alternate signal stack settings (established by *sigaltstack()*) are per-thread. However, because a new thread wrongly inherits its alternate signal stack settings from the caller of *pthread_create()*, the two threads share an alternate signal stack. SUSv3 requires that a new thread should start with no alternate signal stack defined. The consequence of this LinuxThreads nonconformance is that if two threads happen to simultaneously handle different signals on their shared alternate signal stacks at the same time, chaos is likely to result (e.g., a program crash). This problem may be very hard to reproduce and debug, since its occurrence depends on the probably rare event that the two signals are handled at the same time.

  In a program using LinuxThreads, a new thread could make a call to *sigaltstack()* to ensure that it uses a different alternate signal stack from the thread that created it (or no stack at all). However, portable programs (and library functions that create threads) won't know to do this, since it is not a requirement on other implementations. Furthermore, even if we employ this technique, there is still a possible race condition: the new thread could receive and handle a signal on the alternate stack before it has a chance to call *sigaltstack()*.

- Threads don't share a common session ID and process group ID. The *setsid()* and *setpgid()* system calls can't be used to change the session or process group membership of a multithreaded process.
- Record locks established using *fcntl()* are not shared. Overlapping lock requests of the same type are not merged.
- Threads don't share resource limits. SUSv3 specifies that resource limits are process-wide attributes.
- The CPU time returned by *times()* and the resource usage information returned by *getrusage()* are per-thread. These system calls should return process-wide totals.
- Some versions of *ps(1)* show all of the threads in a process (including the manager thread) as separate items with distinct process IDs.
- Threads don't share nice value set by *setpriority()*.
- Interval timers created using *setitimer()* are not shared between the threads.
- Threads don't share System V semaphore undo (*semadj*) values.

Other problems with LinuxThreads

In addition to the above deviations from SUSv3, the LinuxThreads implementation has the following problems:

- If the manager thread is killed, then the remaining threads must be manually cleaned up.
- A core dump of a multithreaded program may not include all of the threads of the process (or even the one that triggered the core dump).
- The nonstandard *ioctl()* `TIOCNOTTY` operation can remove the process's association with a controlling terminal only when called from the main thread.

33.5.2 NPTL

NPTL was designed to address most of the shortcomings of LinuxThreads. In particular:

- NPTL provides much closer conformance to the SUSv3 specification for Pthreads.
- Applications that employ large numbers of threads scale much better under NPTL than under LinuxThreads.

> NPTL allows an application to create large numbers of threads. The NPTL implementers were able to run test programs that created 100,000 threads. With LinuxThreads, the practical limit on the number of threads is a few thousand. (Admittedly, very few applications need such large numbers of threads.)

Work on implementing NPTL began in 2002 and progressed over the next year or so. In parallel, various changes were made within the Linux kernel to accommodate the requirements of NPTL. The changes that appeared in the Linux 2.6 kernel to support NPTL included the following:

- refinements to the implementation of thread groups (Section 28.2.1);
- the addition of futexes as a synchronization mechanism (futexes are a generic mechanism that was designed not just for NPTL);
- the addition of new system calls (*get_thread_area()* and *set_thread_area()*) to support thread-local storage;
- support for threaded core dumps and debugging of multithreaded processes;
- modifications to support management of signals in a manner consistent with the Pthreads model;
- the addition of a new *exit_group()* system call to terminate all of the threads in a process (starting with *glibc* 2.3, *_exit()*—and thus also the *exit()* library function—is aliased as a wrapper that invokes *exit_group()*, while a call to *pthread_exit()* invokes the true *_exit()* system call in the kernel, which terminates just the calling thread);
- a rewrite of the kernel scheduler to allow efficient scheduling of very large numbers (i.e., thousands) of KSEs;

- improved performance for the kernel's process termination code; and
- extensions to the *clone()* system call (Section 28.2).

The essentials of the NPTL implementation are as follows:

- Threads are created using a *clone()* call that specifies the following flags:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND |
CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID |
CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
```

  NPTL threads share all of the information that LinuxThreads threads share, and more. The `CLONE_THREAD` flag means that a thread is placed in the same thread group as its creator and shares the same process ID and parent process ID. The `CLONE_SYSVSEM` flag means that a thread shares System V semaphore undo values with its creator.

  When we use *ps(1)* to list a multithreaded process running under NPTL, we see just a single line of output. To see information about the threads within a process, we can use the *ps −L* option.

- The implementation makes internal use of the first two realtime signals. Applications can't use these signals.

  One of these signals is used to implement thread cancellation. The other signal is used as part of a technique that ensures that all of the threads in a process have the same user and group IDs. This technique is required because, at the kernel level, threads have distinct user and group credentials. Therefore, the NPTL implementation does some work in the wrapper function for each system call that changes user and group IDs (*setuid()*, *setresuid()*, and so on, and their group analogs) that causes the IDs to be changed in all of the threads of the process.

- Unlike LinuxThreads, NPTL doesn't use manager threads.

NPTL standards conformance

These changes mean that NPTL achieves much closer SUSv3 conformance than LinuxThreads. At the time of writing, the following nonconformance remains:

- Threads don't share a nice value.

There are some additional NPTL nonconformances in earlier 2.6.*x* kernels:

- In kernels before 2.6.16, the alternate signal stack was per-thread, but a new thread wrongly inherited alternate signal stack settings (established by *sigaltstack()*) from the caller of *pthread_create()*, with the consequence that the two threads shared an alternate signal stack.
- In kernels before 2.6.16, only a thread group leader (i.e., the main thread) could start a new session by calling *setsid()*.
- In kernels before 2.6.16, only a thread group leader could use *setpgid()* to make the host process a process group leader.

- In kernels prior to 2.6.12, interval timers created using *setitimer()* were not shared between the threads of a process.
- In kernels prior to 2.6.10, resource limit settings were not shared between the threads of a process.
- In kernels prior to 2.6.9, the CPU time returned by *times()* and the resource usage information returned by *getrusage()* were per-thread.

NPTL was designed to be ABI-compatible with LinuxThreads. This means that programs that were linked against a GNU C library providing LinuxThreads don't need to be relinked in order to use NPTL. However, some behaviors may change when the program is run with NPTL, primarily because NPTL adheres more closely to the SUSv3 specification for Pthreads.

33.5.3 Which Threading Implementation?

Some Linux distributions ship with a GNU C library that provides both LinuxThreads and NPTL, with the default being determined by the dynamic linker according to the underlying kernel on which the system is running. (These distributions are by now historical because, since version 2.4, *glibc* no longer provides LinuxThreads.) Therefore, we may sometimes need to answer the following questions:

- Which threading implementation is available in a particular Linux distribution?
- On a Linux distribution that provides both LinuxThreads and NPTL, which implementation is used by default, and how can we explicitly select the implementation that is used by a program?

Discovering the threading implementation

We can use a few techniques to discover the threading implementation that is available on a particular system, or to discover the default implementation that will be employed when a program is run on a system that provides both threading implementations.

On a system providing *glibc* version 2.3.2 or later, we can use the following command to discover which threading implementation the system provides, or, if it provides both implementation, then which one is used by default:

```
$ getconf GNU_LIBPTHREAD_VERSION
```

On a system where NPTL is the only or the default implementation, this will display a string such as the following:

```
NPTL 2.3.4
```

> Since *glibc* 2.3.2, a program can obtain similar information by using *confstr(3)* to retrieve the value of the *glibc*-specific `_CS_GNU_LIBPTHREAD_VERSION` configuration variable.

On systems with older GNU C libraries, we must do a little more work. First, the following command can be used to show the pathname of the GNU C library that is

used when we run a program (here, we use the example of the standard *ls* program, which resides at `/bin/ls`):

```
$ ldd /bin/ls | grep libc.so
        libc.so.6 => /lib/tls/libc.so.6 (0x40050000)
```

> We say a little more about the *ldd* (list dynamic dependencies) program in Section 41.5.

The pathname of the GNU C library is shown after the `=>`. If we execute this pathname as a command, then *glibc* displays a range of information about itself. We can *grep* though this information to select the line that displays the threading implementation:

```
$ /lib/tls/libc.so.6 | egrep -i 'threads|nptl'
        Native POSIX Threads Library by Ulrich Drepper et al
```

We include *nptl* in the *egrep* regular expression because some *glibc* releases containing NPTL instead display a string like this:

```
        NPTL 0.61 by Ulrich Drepper
```

Since the *glibc* pathname may vary from one Linux distribution to another, we can employ shell command substitution to produce a command line that will display the threading implementation in use on any Linux system, as follows:

```
$ $(ldd /bin/ls | grep libc.so | awk '{print $3}') | egrep -i 'threads|nptl'
        Native POSIX Threads Library by Ulrich Drepper et al
```

Selecting the threading implementation used by a program

On a Linux system that provides both NPTL and LinuxThreads, it is sometimes useful to be able to explicitly control which threading implementation is used. The most common example of this requirement is when we have an older program that depends on some (probably nonstandard) behavior of LinuxThreads, so that we want to force the program to run with that threading implementation, instead of the default NPTL.

For this purpose, we can employ a special environment variable understood by the dynamic linker: `LD_ASSUME_KERNEL`. As its name suggests, this environment variable tells the dynamic linker to operate as though it is running on top of a particular Linux kernel version. By specifying a kernel version that doesn't provide support for NPTL (e.g., `2.2.5`), we can ensure that LinuxThreads is used. Thus, we could run a multithreaded program with LinuxThreads using the following command:

```
$ LD_ASSUME_KERNEL=2.2.5 ./prog
```

When we combine this environment variable setting with the command that we described earlier to show the threading implementation that is used, we see something like the following:

```
$ export LD_ASSUME_KERNEL=2.2.5
$ $(ldd /bin/ls | grep libc.so | awk '{print $3}') | egrep -i 'threads|nptl'
        linuxthreads-0.10 by Xavier Leroy
```

The range of kernel version numbers that can be specified in LD_ASSUME_KERNEL is subject to some limits. In several common distributions that supply both NPTL and LinuxThreads, specifying the version number as 2.2.5 is sufficient to ensure the use of LinuxThreads. For a fuller description of the use of this environment variable, see *http://people.redhat.com/drepper/assumekernel.html*.

33.6 Advanced Features of the Pthreads API

Some advanced features of the Pthreads API include the following:

- *Realtime scheduling*: We can set realtime scheduling policies and priorities for threads. This is similar to the process realtime scheduling system calls described in Section 35.3.
- *Process shared mutexes and condition variables*: SUSv3 specifies an option to allow mutexes and condition variables to be shared between processes (rather than just among the threads of a single process). In this case, the condition variable or mutex must be located in a region of memory shared between the processes. NPTL supports this feature.
- *Advanced thread-synchronization primitives*: These facilities include barriers, read-write locks, and spin locks.

Further details on all of these features can be found in [Butenhof, 1996].

33.7 Summary

Threads don't mix well with signals; multithreaded application designs should avoid the use of signals whenever possible. If a multithreaded application must deal with asynchronous signals, usually the cleanest way to do so is to block signals in all threads, and have a single dedicated thread that accepts incoming signals using *sigwait()* (or similar). This thread can then safely perform tasks such as modifying shared variables (under mutex control) and calling non-async-signal-safe functions.

Two threading implementations are commonly available on Linux: LinuxThreads and NPTL. LinuxThreads has been available on Linux for many years, but there are a number of points where it doesn't conform to the requirements of SUSv3 and it is now obsolete. The more recent NPTL implementation provides closer SUSv3 conformance and superior performance, and is the implementation provided in modern Linux distributions.

Further information

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

The author of LinuxThreads documented the implementation in a web page that can be found at *http://pauillac.inria.fr/~xleroy/linuxthreads/*. The NPTL implementation is described by its implementers in a (now somewhat out-of-date) paper that is available online at *http://people.redhat.com/drepper/nptl-design.pdf*.

33.8 Exercises

33-1. Write a program to demonstrate that different threads in the same process can have different sets of pending signals, as returned by *sigpending()*. You can do this by using *pthread_kill()* to send different signals to two different threads that have blocked these signals, and then have each of the threads call *sigpending()* and display information about pending signals. (You may find the functions in Listing 20-4 useful.)

33-2. Suppose that a thread creates a child using *fork()*. When the child terminates, is it guaranteed that the resulting `SIGCHLD` signal will be delivered to the thread that called *fork()* (as opposed to some other thread in the process)?