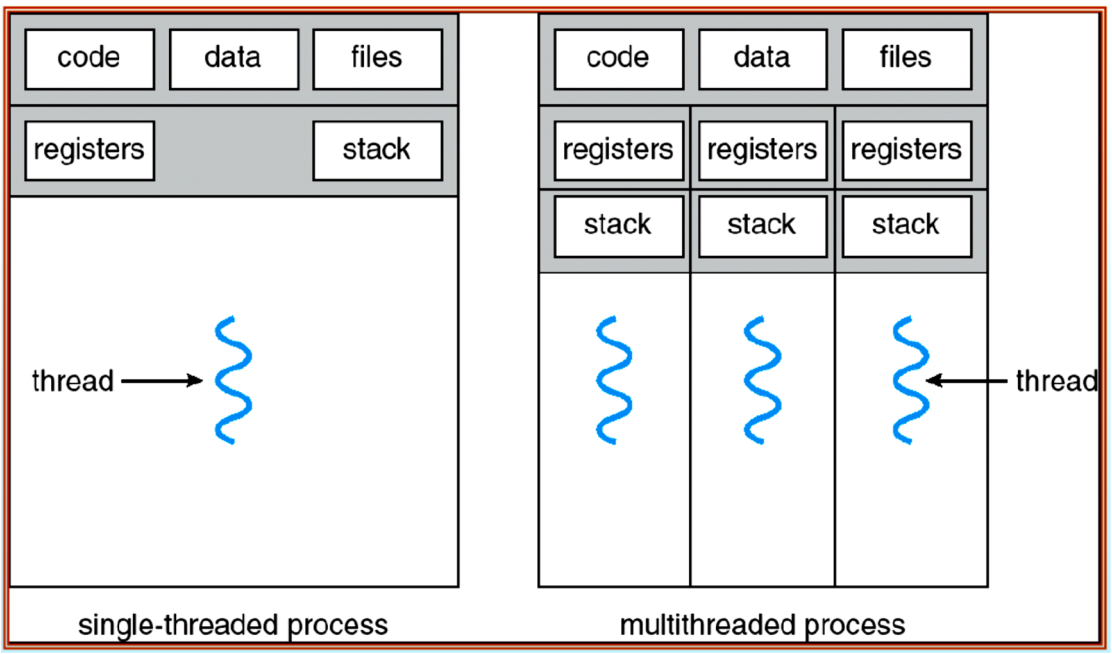
Chapter 6
Threads

This chapter will discuss the following concepts:

6.1 Overview
6.2 Multithreading Models
6.3 Operating System Examples



6.1 Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its **code section**, **data section**, and other operating-system resources, such as **open files** and **signals**. A traditional (or **heavyweight**) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 6.1 illustrates the difference between a traditional **single-threaded** process and a **multithreaded process**.

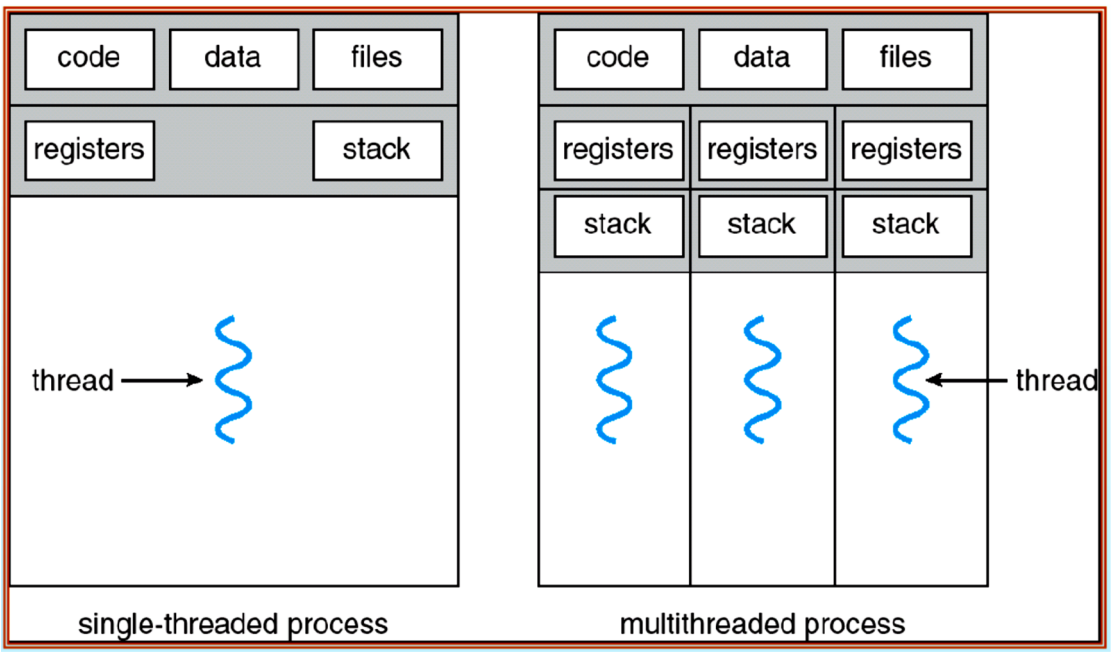


Figure 6.1 Single-threaded and multithreaded processes.

Many software packages that run on modern desktop PCs are multithreaded. An application typically is implemented as a separate process with several threads of control. For example, A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands) of clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time. The amount of time that a client might have to wait for its request to be serviced could be enormous.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is **time consuming and resource intensive**. If the new process will perform the same tasks as the existing process, why incur all that overhead?

It is generally more efficient to use one process that contains multiple threads. This approach would multithread the web-server process. The server would create a separate thread that would listen for client requests; when a request was made, rather than creating another process, the server would create another thread to service the request.

6.2 Multithreading Models

Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all modern operating systems—including **Windows XP**, **Linux**, **Mac OS X**, **Solaris**, and **Tru64 UNIX**—support kernel threads. Ultimately, there must a relationship exist between user

threads and kernel threads. In this section, we look at three common ways of establishing this relationship.

6.2.1 Many-to-One Model

The many-to-one model (Figure 6.2) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

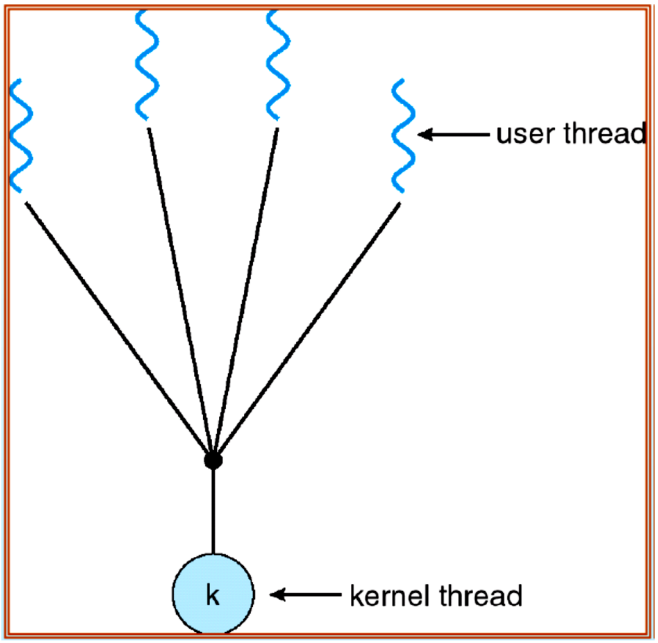


Figure 6.2 Many-to-one model.

6.2.2 One-to-One Model

The one-to-one model (Figure 6.3) maps each user thread to a kernel thread. It provides more **concurrency** than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors.

The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. **Linux**, along with the family of Windows operating systems—including Windows 95, 98, NT, 2000, and XP—implement the one-to-one model.

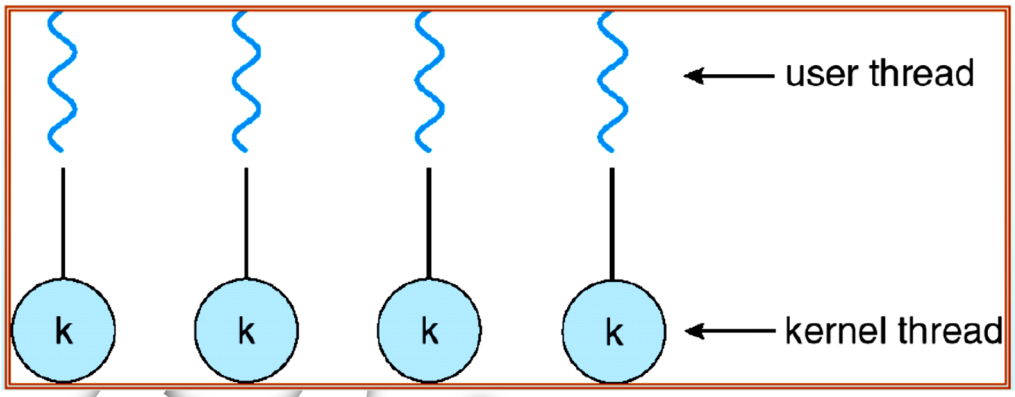


Figure 6.3 One-to-one model

4.2.3 Many-to-Many Model

The many-to-many model (Figure 6.4) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time. The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create).

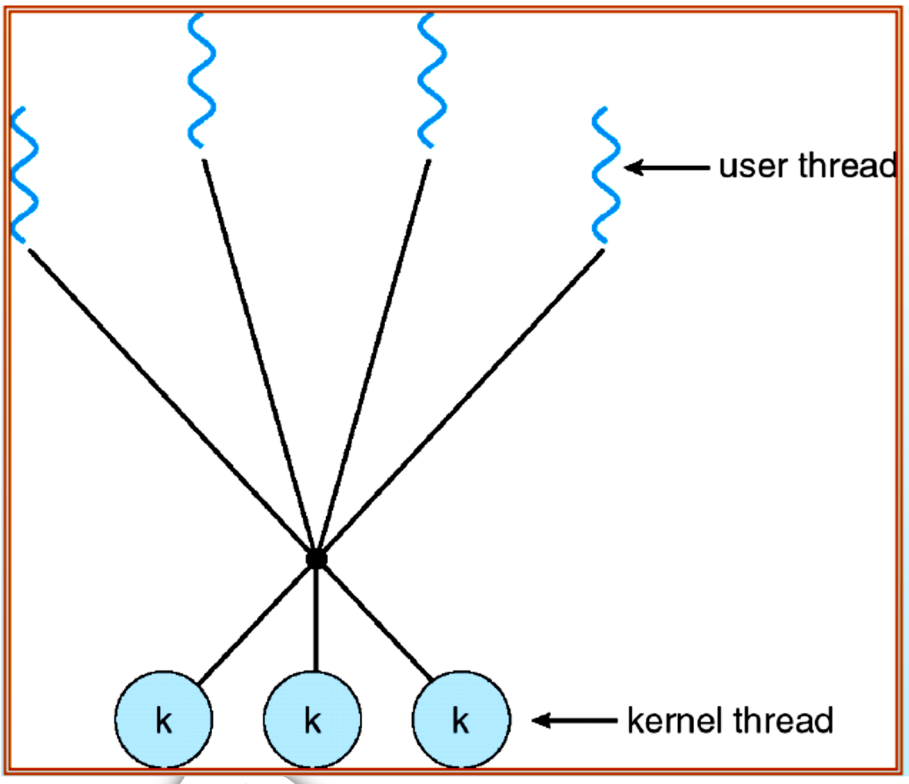


Figure 6.4 Many-to-many model.

6.3 Operating-System Examples

In this section, we explore how threads are implemented in Windows XP and Linux systems.

6.3.1 Windows XP Threads

Windows XP implements the **Win32 API**. The Win32 API is the primary API for the family of Microsoft operating systems (Windows 95, 98, NT, 2000, and XP). Indeed, much of what is mentioned in this section applies to this entire family of operating systems.

A Windows XP application runs as a separate process, and each process may contain one or more threads. Windows XP uses the one-to-one mapping, where each user-level thread maps to an associated kernel thread.

The general components of a thread include:

- **A thread ID** uniquely identifying the thread
- **A register set** representing the status of the processor
- **A user stack**, employed when the thread is running in user mode, and a **kernel stack**, employed when the thread is running in kernel mode
- A **private storage area** used by various run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context of the thread**. The primary data structures of a thread include:

- ETHREAD—executive thread block
- KTHREAD—kernel thread block
- TEB—thread environment block

6.3.2 Linux Threads

Linux provides the **fork ( )** system call with the traditional functionality of duplicating a process. Linux also provides the ability to create threads using the **clone ( )** system call. However, Linux does not distinguish between processes and threads. In fact, Linux generally uses the term **task** rather than process or thread.

End of chapter 6