

4.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness.** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.
2. **Resource sharing.** Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy.** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

4.2 Multicore Programming

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A later, yet similar, trend in system design is to place multiple computing cores on a single processing chip where each core appears as a separate CPU to the operating system (Section 1.3.2). We refer to such systems as **multicore**, and multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency

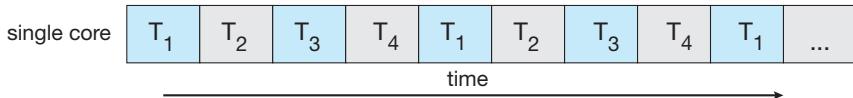


Figure 4.3 Concurrent execution on a single-core system.

means that some threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

Notice the distinction between *concurrency* and *parallelism* in this discussion. A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism. Before the advent of multiprocessor and multicore architectures, most computer systems had only a single processor, and CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

4.2.1 Programming Challenges

The trend toward multicore systems continues to place pressure on system designers and application programmers to make better use of the multiple computing cores. Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution shown in Figure 4.4. For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded.

In general, five areas present challenges in programming for multicore systems:

1. **Identifying tasks.** This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
2. **Balance.** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.

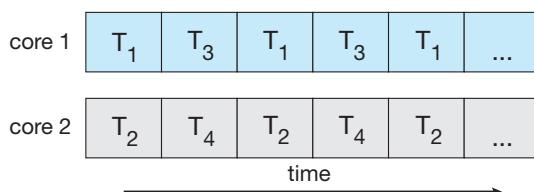


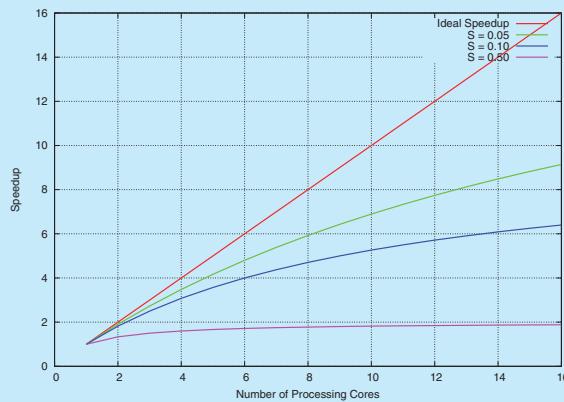
Figure 4.4 Parallel execution on a multicore system.

AMDAHL'S LAW

Amdahl's Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components. If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follows:

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

As an example, assume we have an application that is 75 percent parallel and 25 percent serial. If we run this application on a system with two processing cores, we can get a speedup of 1.6 times. If we add two additional cores (for a total of four), the speedup is 2.28 times. Below is a graph illustrating Amdahl's Law in several different scenarios.



One interesting fact about Amdahl's Law is that as N approaches infinity, the speedup converges to $1/S$. For example, if 50 percent of an application is performed serially, the maximum speedup is 2.0 times, regardless of the number of processing cores we add. This is the fundamental principle behind Amdahl's Law: the serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.

3. **Data splitting.** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency.** The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency. We examine such strategies in Chapter 6.

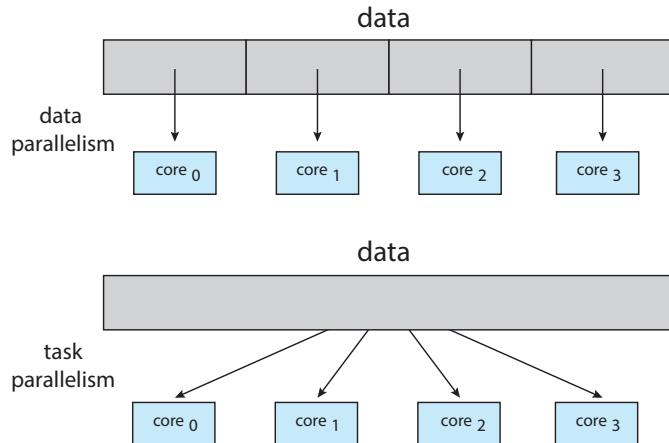


Figure 4.5 Data and task parallelism.

5. **Testing and debugging.** When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

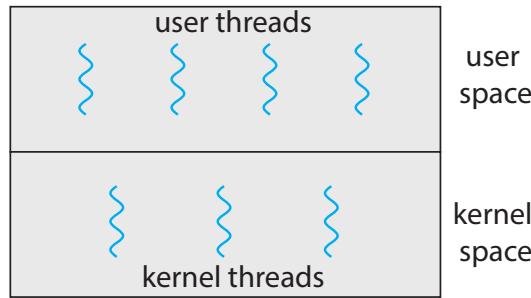
Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future. (Similarly, many computer science educators believe that software development must be taught with increased emphasis on parallel programming.)

4.2.2 Types of Parallelism

In general, there are two types of parallelism: data parallelism and task parallelism. **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Consider, for example, summing the contents of an array of size N . On a single-core system, one thread would simply sum the elements $[0] \dots [N - 1]$. On a dual-core system, however, thread A , running on core 0, could sum the elements $[0] \dots [N/2 - 1]$ while thread B , running on core 1, could sum the elements $[N/2] \dots [N - 1]$. The two threads would be running in parallel on separate computing cores.

Task parallelism involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data. Consider again our example above. In contrast to that situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

Fundamentally, then, data parallelism involves the distribution of data across multiple cores, and task parallelism involves the distribution of tasks across multiple cores, as shown in Figure 4.5. However, data and task paral-

**Figure 4.6** User and kernel threads.

lism are not mutually exclusive, and an application may in fact use a hybrid of these two strategies.

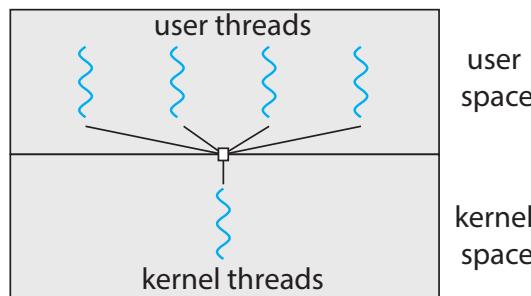
4.3 Multithreading Models

Our discussion so far has treated threads in a generic sense. However, 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 contemporary operating systems—including Windows, Linux, and macOS—support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads, as illustrated in Figure 4.6. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to-many model.

4.3.1 Many-to-One Model

The many-to-one model (Figure 4.7) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient (we discuss thread libraries in Section 4.4). However, the entire process will block if a thread makes a blocking system call. Also, because only

**Figure 4.7** Many-to-one model.