A thread is a basic unit of CPU utilization; it comprises a thread ID, a PC, a register set and a stack. It shares with other threads belonging to the same process its code and data sections, and other OS resources, such as open files and signals.

The benefits of multithreaded programming can be broken down into 4 major categories:

- 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.
- Resource sharing. It allows an application to have several different threads of activity within the same address space.
- Economy. Allocating memory and resources for process creation is costly. Because threads share the resources of process to which they belong, it is more economical to create and context-switch threads.
- Scalability. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.

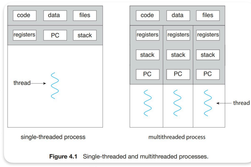

Figure 4.1 Single-threaded and multithreaded processes.

Notice the distinction between concurrency and parallelism. 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.

Figure 4.3 Concurrent execution on a single-core system.

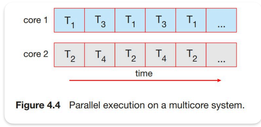

Figure 4.4 Parallel execution on a multicore system.

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

- 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.
- Balance. Programmers must 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 task. Using a separate execution core to run that task may not be worth the cost.
- Data splitting. The data accessed and manipulated by the tasks must be divided to run on separate cores.
- Data dependency. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
- Testing and debugging. Such concurrent programms is inherently more difficult than testing and debugging single-threaded applications.

In general, there are 4 types of parallelism: data 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.

Task parallelism involves distributing tasks across multiple cores, each of them is performing a unique operation.

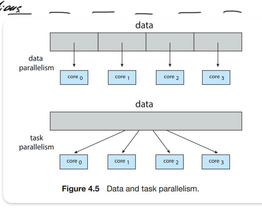

Figure 4.5 Data and task parallelism.

Fundamentally, then, data parallelism involves the distribution of data across multiple cores, and task parallelism involves the distribution of tasks across multiple cores.

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 are supported and managed directly by the OS. Virtually all contemporary OS support kernel threads.

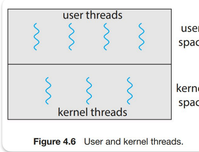

Figure 4.6 User and kernel threads.

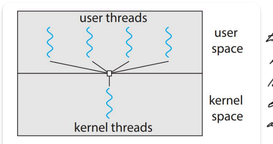

Figure 4.7 Many-to-one model.

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient. However, the entire process will block if a thread makes a blocking system call. Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. *Green threads* uses the many-to-one model.

One-to-one model 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, and a large number of kernel threads may burden the performance of a system. Linux and Windows implement the one-to-one model.

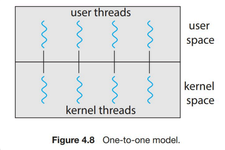

Figure 4.8 One-to-one model.

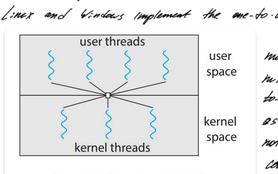

Figure 4.9 Many-to-many model.

The many to many model multiplexes many user-level threads to a smaller or equal number of kernel threads. Whereas the many-to-one model allows the developer to create as many user threads as she wishes, it does not result in parallelism, because the kernel can schedule only one kernel thread per time. The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application. The many-to-many model suffers neither of these shortcomings: developers can create as many user threads as necessary, and corresponding kernel threads can run in parallel on a multiprocessor.

One variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the *two-level model*.

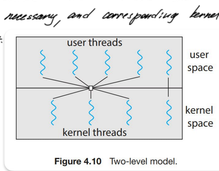

Figure 4.10 Two-level model.