Chapter 4: Threads and Concurrency

*4.1 Overview*

*4.1.1 Motivation*

Examples of multithreaded applications:

* An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.
* A web browser might have one thread display images or text while another thread retrieves data from the network.
* 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.

*4.1.2 Benefits*

* Responsiveness
* Resource sharing
* Economy
* Scalability

*4.2 Multicore Programming*

*4.2.1 Programming Challenges*

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. 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.
* 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.
* 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.
* 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.

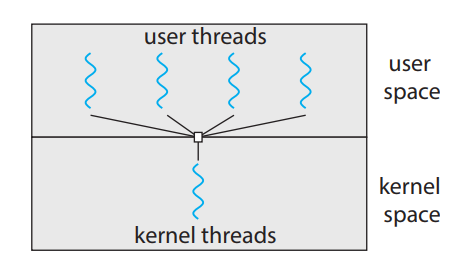
*4.2.2 Types of Parallelism*

* 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 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.

*4.3 Multithreading Models*

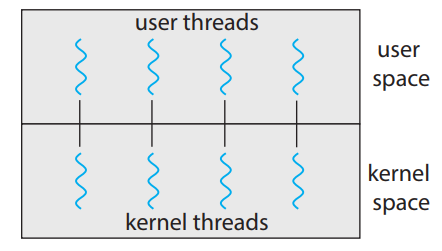
*4.3.1 Many-to-One Model*

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. Because one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

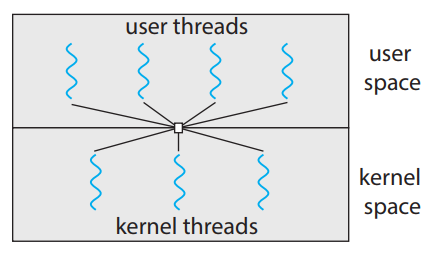


*4.3.2 One-to-One Model*

The 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.



*4.3.3 Many-to-Many Model*

The many-to-many model 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 system with eight processing cores than a system with four cores).

*4.4. Thread Libraries*

A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library.

* The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.
* The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

*4.4.1 Pthreads*

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a specification for thread behavior, not an implementation.

*4.4.2 Windows Threads*

The technique for creating threads using the Windows thread library is similar to the Pthreads technique in several ways.

*4.4.3 Java Threads*

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM. Java threads are available on any system that provides a JVM including Windows, Linux, and macOS. The Java thread API is available for Android applications as well.

*4.5 Implicit Threading*

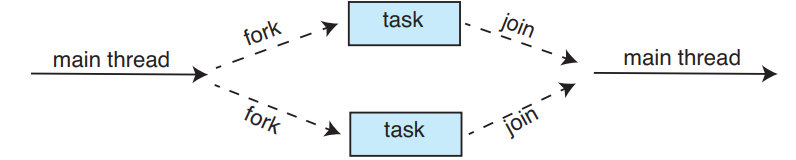
With the continued growth of multicore processing, applications containing hundreds—or even thousands—of threads are looming on the horizon. Designing such applications is not a trivial undertaking: programmers must address not only the challenges outlined but also additional difficulties.

*4.5.1 Thread Pools*

* Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems. The first issue concerns the amount of time required to create the thread, together with the fact that the thread will be discarded once it has completed its work. The second issue is more troublesome. If we allow each concurrent request to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a thread pool.\
* Thread pools offer these benefits:
* Servicing a request with an existing thread is often faster than waiting to create a thread.
* A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
* Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.

*4.5.2 Fork Join*

Fork join model is that the main parent thread creates (forks) one or more child threads and then waits for the children to terminate and join with it, at which point it can retrieve and combine their results. This synchronous model is often characterized as explicit thread creation, but it is also an excellent candidate for implicit threading. In the latter situation, threads are not constructed directly during the fork stage; rather, parallel tasks are designated.



*4.5.3 OpenMP*

Is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies parallel regions as blocks of code that may run in parallel.

Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run- time library to execute the region in parallel.

OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops.

OpenMP divides the work contained in the for loop among the threads it has created in response to the directive.

In addition to providing directives for parallelization, OpenMP allows developers to choose among several levels of parallelism.

OpenMP is available on several open-source and commercial compilers for Linux, Windows, and macOS systems.

*4.5.4 Grand Central Dispatch*

Grand Central Dispatch (GCD) is a technology developed by Apple for its macOS and iOS operating systems. It is a combination of a run-time library, an API, and language extensions that allow developers to identify sections of code (tasks) to run in parallel. Like OpenMP, GCD manages most of the details of threading.

GCD schedules tasks for run-time execution by placing them on a dispatch queue. When it removes a task from a queue, it assigns the task to an available thread from a pool of threads that it manages. GCD identifies two types of dispatch queues: serial and concurrent.

Tasks placed on a serial queue are removed in FIFO order. Once a task has been removed from the queue, it must complete execution before another task is removed. Each process has its own serial queue (known as its main queue).

Tasks placed on a concurrent queue are also removed in FIFO order, but several tasks may be removed at a time, thus allowing multiple tasks to execute in parallel.

Several system-wide concurrent queues are divided into four primary quality-of-service classes:

* *QOS\_CLASS\_USER\_INTERACTIVE*: The user-interactive class represents tasks that interact with the user, such as the user interface and event handling, to ensure a responsive user interface. Completing a task belonging to this class should require only a small amount of work.
* *QOS\_CLASS\_USER\_INITIATED*: The user-initiated class is similar to the user-interactive class in that tasks are associated with a responsive user interface; however, user-initiated tasks may require longer processing times. Opening a file or a URL is a user-initiated task, for example. Tasks belonging to this class must be completed for the user to continue inter-acting with the system, but they do not need to be serviced as quickly as tasks in the user-interactive queue.
* *QOS\_CLASS\_UTILITY*: The utility class represents tasks that require a longer time to complete but do not demand immediate results. This class includes work such as importing data.
* *QOS\_CLASS\_BACKGROUND:* Tasks belonging to the background class are not visible to the user and are not time sensitive. Examples include indexing a mailbox system and performing backups.

Internally, GCD’s thread pool is composed of POSIX threads. GCD actively manages the pool, allowing the number of threads to grow and shrink according to application demand and system capacity

*4.5.5 Intel Thread Building Blocks*

Intel threading building blocks (TBB) is a template library that supports designing parallel applications in C++. Developers specify tasks that can run in parallel, and the TBB task scheduler maps these tasks onto underlying threads.

TBB provides a rich set of features, including templates for parallel loop structures, atomic operations, and mutual exclusion locking.

TBB also provides concurrent data structures, including a hash map, queue, and vector, which can serve as equivalent thread-safe versions of the C++ standard template library data structures.

The TBB library will divide the loop iterations into separate “chunks” and create a number of tasks that operate on those chunks. TBB will also create a number of threads and assign tasks to available threads

*4.6 Threading Issues*

*4.6.1 The fork() and exec() System Calls*

* fork() system call is used to create a separate, duplicate process
* exec() system call, the program specified in the parameter to exec() will replace the entire process - including all threads.

*4.6.2 Signal Handling*

A signal is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

Synchronous signals are delivered to the same process that performed the operation that caused the signal.

When a signal is generated by an event external to a running process, that process receives the signal asynchronously.

A signal may be handled by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a default signal handler that the kernel runs when handling that signal. This default action can be overridden by a user-define signal handler that is called to handle the signal.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, there are multiple options for delivering signals in multithreaded programs:

1. Deliver the signal to the thread to which the signal applies
2. Deliver the signal to every thread in the process
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process

The method for delivering a signal depends on the type of signal generated.

*4.6.3 Thread Cancellation*

Thread cancellation involves terminating a thread before it has completed. A thread that is to be canceled is often referred to as the target thread. Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation. One thread immediately terminates the target thread.
2. Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion

The difficulty with cancellation occurs in situations where resources ha ve been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.

*4.6.4 Thread-Local Storage*

Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data thread-local storage (or TLS)

*4.6.5 Scheduler Activations*

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a lightweight process, or LWP.

Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well.

