Chapter 4: Threads and Concurrency

4.1 Overview

4.1.1 Motivation

Some few examples:

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

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

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.

\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.

\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.

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. 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] ...[N − 1]. On a dual-core system, however, thread A, running on core 0, could sum the elements [0] ...[N⁄2 − 1] while thread B, running on core 1, could sum the elements [N⁄2] ...[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.

4.3 Multithreading Models

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

4.3.2 One-to-One Model

The one-to-one model (Figure 4.8) 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.

4.3.3 Many-to-Many Model

The many-to-many model (Figure 4.9) 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.

Before we proceed with our examples of thread creation, we introduce two general strategies for creating multiple threads: asynchronous threading and synchronous threading. With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another. Because the threads are independent, there is typically little data sharing between them. Asynchronous threading is the strategy used in the multithreaded server illustrated in Figure 4.2 and is also commonly used for designing responsive user interfaces.

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.

4.4.3.1 Java Executor Framework

Java has supported thread creation using the approach we have described thus far since its origins.

4.5 Implicit Threading

This strategy, termed implicit threading, is an increasingly popular trend. In this section, we explore four alternative approaches to designing applications that can take advantage of multicore processors through implicit threading. As we shall see, these strategies generally require application developers to identify tasks—not threads—that can run in parallel. A task is usually written as a function, which the run-time library then maps to a separate thread, typically using the many-to-many model (Section 4.3.3). The advantage of this approach is that developers only need to identify parallel tasks, and the libraries determine the specific details of thread creation and management.

4.5.1 Thread Pools

4.5.2 Fork Join

The strategy for thread creation covered in Section 4.4 is often known as the fork-join model. Recall that with this method, 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

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. Open MP identifies parallel regions as blocks of code that may run in parallel.

4.5.4 Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology developed by Apple for its mac OS 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

4.5.5 Intel Thread Building Blocks

Intel threading building blocks (TBB) is a template library that supports designing parallel applications in C++. As this is a library, it requires no special compiler or language support.

4.6 Threading Issues

4.6.1 The fork() and exec() System Calls

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call. The exec() system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.

4.6.2 Signal Handling

Asignal 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.

4.6.3 Thread Cancellation

Thread cancellation involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.

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