**THREADS AND CONCURRENCY**

# 4.1 Overview

## 4.1.1 Motivation

* 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.

Graphical user interface, diagram

Description automatically generated

## 4.1.2 Benifits

* **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
* **Resource sharing:** Processes can share resources only through techniques such as shared memory and message passing
* **Economy:** Allocating memory and resources for process creation is costly
* **Scalability:** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores

# 4.2 Multicore programming

## 4.2.1 Programming challenges

* **Identifying tasks:** This involves examining applications to find areas that can be divided into separate, concurrent tasks
* **Balance:** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.
* **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
* **Testing and debugging:** When a program is running in parallel on multiple cores, many different execution paths are possible

Diagram

Description automatically generated

## 4.2.2 Type 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

A picture containing application

Description automatically generated

# 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

Diagram

Description automatically generated

* **Green threads:** a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model

## 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

Scatter chart

Description automatically generated

## 4.3.2 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

Diagram, schematic

Description automatically generated

# 4.4 Thread libraries

**Thread library** provides the programmer with an API for creating and managing threads. There are 2 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
* **The second approach** is to implement a kernel-level library supported directly by the operating system.

There are 2 general **strategies** for **creating multiple threads**:

* **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
* **Synchronous threading**: occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes

## 4.4.1 Pthreads

* **Pthread** 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
* In situations that require waiting for multiple threads to complete, the WaitForMultipleObjects() function is used. This function is passed 4 parameters:

+) The number of objects to wait for

+) A pointer to the array of objects

+) A flag indicating whether all objects have been signaled

+) A timeout duration (or INFINITE)

## 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
* There are 2 techniques for explicitly creating threads in a Java program. One approach is to create a new class that is derived from the Thread class and to override its run() method

Invoking the start() method for the new Thread object does two things:

* It allocates memory and initializes a new thread in the JVM.
* It calls the run() method, making the thread eligible to be run by the JVM

Data sharing between threads belonging to the same process occurs easily in Windows and Pthreads, since shared data are simply declared globally. As a pure object-oriented language, Java has no such notion of global data.

* It is quite easy to notice at first that this model of thread creation appears more complicated than simply creating a thread and joining on its termination

# 4.5 Implicit threading

* With the continued growth of multicore processing, applications containing hundreds—or even thousands—of threads are looming on the horizon
* One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries.

## 4.5.1 Thread pools

* **Thread pools** offer these benefits:

+) Servicing a request with an existing thread is often faster than waiting to create a thread.

+) Athread 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.

* The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests
* A pointer to PoolFunction() is passed to one of the functions in the thread pool API, and a thread from the pool executes this function. One such member in the thread pool API is the QueueUserWorkItem() function, which is passed three parameters:

+) LPTHREAD START ROUTINE Function—a pointer to the function that is to run as a separate thread

+) PVOID Param—the parameter passed to Function

+) ULONG Flags—flags indicating how the thread pool is to create and manage execution of the thread

## 4.5.2 Fork join

Diagram

Description automatically generated

* Java introduced a fork-join library in Version 1.7 of the API that is designed to be used with recursive divide-and-conquer algorithms such as Quicksort and Mergesort library, separate tasks are forked during the divide step and assigned smaller subsets of the original problem. Algorithms must be designed so that these separate tasks can execute concurrently
* Notice that SumTask in Figure 4.18 extends RecursiveTask. The Java forkjoin strategy is organized around the abstract base class ForkJoinTask, and the RecursiveTask and RecursiveAction classes extend this class. The fundamental difference between these two classes is that RecursiveTask returns a result (via the return value specified in compute()), and RecursiveAction does not return a result

Diagram

Description automatically generated

* What is interesting in Java’s fork-join model is the management of tasks wherein the library constructs a pool of worker threads and balances the load of tasks among the available workers
* In some situations, there are thousands of tasks, yet only a handful of threads performing the work (for example, a separate thread for each CPU). Additionally, each thread in a ForkJoinPool maintains a queue of tasks that it has forked, and if a thread’s queue is empty, it can steal a task from another thread’s queue using a ***work stealing*** algorithm, thus balancing the workload of tasks among all threads

## 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. OpenMP identifies parallel regions as blocks of code that may run in parallel.

Diagram

Description automatically generated

* It creates as many threads as there are processing cores in the system. Thus, for a dual-core system, two threads are created; for a quad-core system, four are created; and so forth. All the threads then simultaneously execute the parallel region. As each thread exits the parallel region, it is terminated.

# 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.
* Which of the two versions of fork() to use depends on the application. If **exec()** is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process.

## 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:

+) A signal is **generated** by the occurrence of a particular event.

+) The signal is delivered to a process.

+) Once delivered, the signal must be handled

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

+) A default signal handler

+) A user-defined signal handler

## 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. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further
* Cancellation of a target thread may occur in 2 different scenarios:

+) **Asynchronous cancellation**. One thread immediately terminates the target thread

+) **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 have 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.