Multithreaded Programming

4.1 Overview

* A **thread** is a **basic unit of CPU utilization**.
* It **comprises** a thread ID, a program counter, a register set, and a stack.
* It **shares** with other threads belonging to the same process **its code section**, **data section**, and **other operating-system resources,** such as open files and signals.

|  |  |
| --- | --- |
| * traditional **single-threaded** process | * **Multithreaded process**. |
| * It has a single thread of control. | * process has multiple threads of control, it can perform more than one task at a time. |
|  |  |

4.1.1 Motivation

* **Most software** applications that run on modem computers are **multithreaded**.
* An application is implemented as a separate process with several threads of control. EX: web browser.
* In certain situations, a single application may be required to perform several similar tasks.
* A busy web server may have several (perhaps thousands of) clients concurrently accessing it.
* One **solution** is to have the server run as a single process that accepts requests (threads).
* Threads also play a vital role in **remote procedure call** (RPC) systems (Typically, RPC servers are multithreaded.)
* Most operating-system kernels are now multithreaded. Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling.

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.

* useful in designing user interfaces

1. Resource sharing.

Processes can only share resources through techniques such as shared memory and message passing.

1. Economy.

Allocating memory and resources for process creation is costly.

1. Scalability.

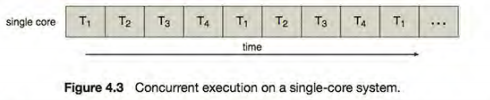
The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

4.2Multicore Programming (multiprocessor systems):

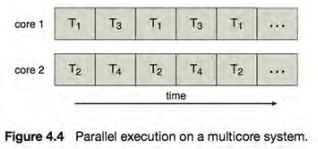
Whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems.

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

* On a system with **a single computing core**, **concurrency** merely **means** that the execution of the threads will be interleaved over time.



* On a system **with multiple cores,** however, **concurrency** means that the threads can run in **parallel**, because the system can assign a separate thread to each core.



The distinction between **parallelism** and **concurrency:**

* A **system is parallel** if it can perform more than one task **simultaneously**.
* In contrast, **a concurrent system** supports **more than one task b**y allowing all the tasks to make progress.
* Thus, it is possible to have concurrency **without** parallelism.

4.2.1Programming Challenges

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

1. **Identifying tasks:** tasks are independent of one another and thus can run in parallel on individual cores.
2. **Balance:** ensure that the tasks perform equal work of equal value.
3. **Data splitting:** data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency:** data must be examined for dependencies between two or more tasks.
5. **Testing and debugging** is more difficult than testing and debugging single-threaded applications.

4.2.2 Types of Parallelism

**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.
* **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 different data.
* In most instances, applications use **a hybrid of these two strategies**.

4.3 Multithreading Models

Support for threads may be provided either at the user level, for **user threads (above the kernel)**, or by the kernel, for **kernel threads**.

Virtually all contemporary operating systems-including Windows, Linux, Mac OS X, and Solaris support kernel threads.

Three common ways of establishing a relationship between kernel and user thread:

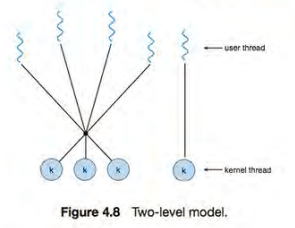
4.3.1 Many-to-One Model

4.3.2 One-to-One Model

4.3.3 Many-to-Many Model

|  |  |  |
| --- | --- | --- |
| The many-to-one model maps many user-level threads to one kernel thread. | The one-to-one model maps each user thread to a kernel thread. | The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. |
| Thread management is done by the thread library in user space, so it is efficient.  \*Multiple threads are **unable** to run in parallel on multicore systems.  Green threads- a thread library available for Solaris used this model.  \*very few systems continue to use the model because of its inability to take advantage of multiple processing cores. | Provides **more** **concurrency** than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.  \***allows** multiple threads to run in parallel on multiprocessors.  \*The only drawback is that creating a user thread requires creating the corresponding kernel thread. | the effect of this design on concurrency:  Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for  Execution. |
|  |  |  |

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**



4.4 Thread libraries

**A thread library** provides the programmer with an API for creating and managing threads.

**Two primary ways of implementing a thread library:**

* The first approach is to provide a library **entirely in user space** with no kernel support.
* The second approach is to implement a kernel-level library supported directly by the operating system.

Three main thread libraries are in use today: POSIX Pthreads (kernel or user), Windows (kernel), and Java.

* Pthreads, extension of the **POSIX** standard.
* The **Java thread API** is generally implemented using a thread library available on the host system.
* This means that on Windows systems, Java threads are typically implemented using the Windows API; UNIX and Linux systems often use Pthreads.

**Two 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.
* Each thread runs **independently** of every other thread, and the parent thread need not know when its child terminates.
* **Little** data sharing between threads.
* **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-the so-called **fork-join strategy.**
* **Significant** data sharing among threads.

4.4.1 Pthreads

4.4.2 Windows Threads

4.4.3 Java Threads look textbook p170-175

4.5 Implicit Threading

One way to address difficulties and better support the design of multithreaded applications is to **transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed implicit threading.**

**Three approaches that can take advantage of multicore processors through implicit threading.**

4.5.1 Thread Pools

**The first issue** 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: No** bound on **the number of threads** concurrently active in the system. Unlimited threads could exhaust system resources.

* One solution to this problem is to use a ***thread pool.***
* ***Thread pool*** is to create a number of threads at process startup and place them into a pool, where they sit and wait for work.
* When a server receives a request, it **awakens** a thread from this pool- if one is available-and **passes** it the request for service.
* Once the thread **completes** its service, it **returns** to the pool and awaits more work.
* If the pool contains **no** available thread, the server **waits** until one becomes free.

**Thread pools offer these benefits:**

1. Servicing a request with an existing thread **is faster than** waiting to create a thread.

2. 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.

3**. 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:** **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**.

The Windows API provides several functions related to thread pools. Using the thread pool API is similar to creating a thread with the **Thread.Create ()**

* One such member in the th.read 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.

**The java. util. concurrent** package in the Java API provides a thread-pool utility as well.

4.5.2 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.
* When OpenMP encounters the directive

**#pragma omp parallel**

* It creates as many threads as there are processing cores in the system.
* OpenMP provides **several additional directives** for running code regions in parallel, including parallelizing loops and allows developer to **choose** among several levels of parallelism
* It also allows developers to identify whether data are shared between threads or are private to a thread.
* OpenMP is available on several open-source and commercial compilers for Linux, Windows, and Mac OS X systems.

4.5.3 Grand Central Dispatch

a technology for Apple's Mac OS X and iOS -is **a combination of extensions** to the C language, an API, and a run-time library that allows application developers to **identify sections of code to run in parallel**.

* GCD identifies extensions to the C and C++ languages known as **blocks**. A block is simply a self-contained unit of work.
* It is specified by a **caret^** inserted in front of a pair of braces { } .
* ~{ printf("I am a block"); }
* GCD **schedules** blocks for run-time execution by placing them on a **dispatch queue**.
* When it removes a block from a queue, it assigns the block to an available thread from the thread pool it manages.

GCD identifies **two types of dispatch queues**: **serial** and **concurrent**.

**1-Blocks placed on a serial queue** are removed in FIFO order. Once a block has been removed from the queue, it must complete execution before another block is removed.

* Each process has its **own serial queue** (known as its **main queue**).
* **Useful** for **ensuring the sequential execution** of several tasks.

**2-Blocks placed on a concurrent queue** are also removed in FIFO order, but several blocks may be removed at a time

* Thus allowing multiple blocks to execute in **parallel**.

**Three system-wide concurrent dispatch queues,** and they are distinguished according to priority**: low, default, and high.**

**Priorities** represent **an approximation of the relative importance of blocks**.

* Quite simply, blocks with a higher priority should be placed on the high-priority dispatch queue.

The following code segment illustrates obtaining the default-priority concurrent queue and submitting a block to the queue using the **dispatch\_async ()** function:

Dispatch\_queue\_t queue = dispatch-get-global-queue

(DISPATCH\_QUEUE.PRIORITYJDEFAULT, 0);

Dispatch\_async (queue, ~ {printf ("I am a block."); } );

* 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.

4.5.4 Other Approaches

Other commercial approaches include parallel and concurrent libraries, such as **Intel's**

**Threading Building Blocks (TBB)** and several products from Microsoft.

4.6 Threading Issues

The semantics of **the fork() and exec()** system calls change in a multithreaded program.

* **fork() system call** is used to **create a separate, duplicate process**.
* 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.
* **exec () will replace the entire process- including all threads**.
* 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. Duplicating **only** the calling thread is appropriate.

* the separate process **does not call exec () after forking:**

Then, the separate process should duplicate all threads.

4.6.2 Signal Handling

* **A signal** is used in UNIX systems to notify a process that a particular event has occurred.
* Received either **synchronously** or **asynchronously**, depending on the source of and the reason for the event being signaled.
* 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
* **Examples of synchronous signal** include illegal memory access and division by 0.
* When a signal is generated by an event external to a running process, that process receives the signal asynchronously**. Examples of such signal**s include terminating a process with specific keystrokes (such as <control> <C>).

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

1. A default signal handler: that the kernel runs when handling that signal.(can be overridden with user handler).

2. A user-defined signal handler: that is called to handle the signal.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process, more complicated in multithreaded programs.

**In general, the following options exist:**

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.

4.6.3 Thread Cancellation

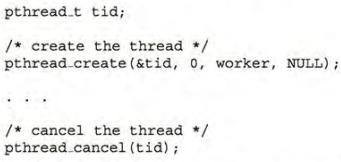
**Thread cancellation** involves terminating a thread before it has completed.

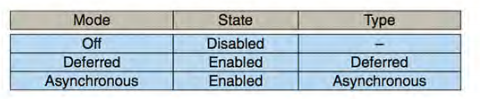
A thread that is to be canceled is a **target thread**.

Cancellation 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 have been allocated to a canceled thread while in the midst of updating data it is sharing with other threads.
* Therefore, **canceling a thread asynchronously** may not free a necessary system-wide resource.
* **With deferred cancellation**, in contrast, cancellation occurs only after the target thread has checked a flag. The thread can perform this check at a point at which it can be canceled safely.
* **In Pthreads**, thread cancellation is **initiated** using the **pthread\_cancel ()**
* The following code illustrates creating- and then canceling thread:
* 
* Pthreads supports three cancellation modes.



The default cancellation type is **deferred** cancellation.

Pthreads allows threads to **disable** or **enable** cancellation. Obviously, a thread **cannot** be canceled if cancellation is disabled. However, cancellation requests remain **pending**, so the thread can later enable cancellation and respond to the request.

4.6.4 Thread-Local Storage

* Threads belonging to a process share the data of the process.
* in some circumstances, each thread might need its own copy of certain data. We will call such **data thread-local storage** (or TLS).
* In some ways, TIS is similar to static data. The difference is that TIS data are unique to each thread.

4.6.5 Scheduler Activations

A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library.

* Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads(known as lightweight process, or LWP)
* To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run.
* Each LWP is attached to a kernel thread
* If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well.
* An application may require any number of LWPs to run efficiently.
* One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**.
* It works as follows: The kernel provides an application with a set of virtual processors (LWPs).
* 2- The application schedule user threads onto an available virtual processor.
* 3- The kernel must inform an application about certain events. This procedure is known as an **upcall** which is handled be **upcall handler**.
* More details look p186.

4. 7 Operating-System Examples

4.7.1 Windows Threads

4.7.2 Linux Threads

Please make sure that you read it from textbook because it couldn’t be summarized.p187-188.