**Chapter 4:**

**Multithreaded Programming**

**Multithreaded Programming**

* Overview
* Multicore Programming
* Multithreading Models
* Thread Libraries
* Implicit Threading
* Threading Issues
* Operating System Examples

**Objectives**

* To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
* To discuss the APIs for the Pthreads, Windows, and Java thread libraries
* To explore several strategies that provide implicit threading
* To examine issues related to multithreaded programming
* To cover operating system support for threads in Windows and Linux

**Motivation**

* Most modern applications are multithreaded
* Threads run within application
* Multiple tasks with the application can be implemented by separate threads
1. Update display
2. Fetch data
3. Spell checking
4. Answer a network request
* Process creation is heavy-weight while thread creation is light-weight
* Can simplify code, increase efficiency
* Kernels are generally multithreaded

**Multithreaded Server Architecture**



**Benefits**

* **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
* **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
* **Economy –** cheaper than process creation, thread switching lower overhead than context switching
* **Scalability –** process can take advantage of multiprocessor architectures

**Multicore Programming**

* **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
	+ **Dividing activities**
	+ **Balance**
	+ **Data splitting**
	+ **Data dependency**
	+ **Testing and debugging**
* ***Parallelism*** implies a system can perform more than one task simultaneously
* ***Concurrency*** supports more than one task making progress
	+ Single processor / core, scheduler providing concurrency
* Types of parallelism
	+ **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
	+ **Task parallelism** – distributing threads across cores, each thread performing unique operation
* As # of threads grows, so does architectural support for threading
	+ CPUs have cores as well as ***hardware threads***
	+ Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

**Concurrency vs. Parallelism**

* **Concurrent execution on single-core system:**



* **Parallelism on a multi-core system:**



**Single and Multithreaded Processes**



**Amdahl’s Law**

* Identifies performance gains from adding additional cores to an application that has both serial and parallel components
* *S* is serial portion
* *N* processing cores
* I.e. if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
* As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

* But does the law take into account contemporary multicore systems?

**User Threads and Kernel Threads**

* **User threads** - management done by user-level threads library
* Three primary thread libraries:
	+ POSIX **Pthreads**
	+ Win32 threads
	+ Java threads
* **Kernel threads** - Supported by the Kernel
* Examples – virtually all general purpose operating systems, including:
	+ Windows
	+ Solaris
	+ Linux
	+ Tru64 UNIX
	+ Mac OS X

**Multithreading Models**

* Many-to-One
* One-to-One
* Many-to-Many

**Many-to-One**

* Many user-level threads mapped to single kernel thread
* One thread blocking causes all to block
* Multiple threads may not run in parallel on muticore system because only one
* may be in kernel at a time
* Few systems currently use this model
* Examples:
	+ **Solaris Green Threads**
	+ **GNU Portable Threads**

**One-to-One**

* Each user-level thread maps to kernel thread
* Creating a user-level thread creates a kernel thread
* More concurrency than many-to-one
* Number of threads per process sometimes
* restricted due to overhead
* Examples
	+ Windows NT/XP/2000
	+ Linux
	+ Solaris 9 and later



**Many-to-Many Model**

* Allows many user level threads to be mapped to many kernel threads
* Allows the operating system to create a sufficient number of kernel threads
* Solaris prior to version 9
* Windows NT/2000 with the *ThreadFiber* package

**Two-level Model**

* Similar to M:M, except that it allows a user thread to be

 **bound** to kernel thread

* Examples
	+ IRIX
	+ HP-UX
	+ Tru64 UNIX
	+ Solaris 8 and earlier

**Thread Libraries**

* **Thread library** provides programmer with API for creating and managing threads
* Two primary ways of implementing
	+ Library entirely in user space
	+ Kernel-level library supported by the OS

**Pthreads**

* May be provided either as user-level or kernel-level
* A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
* ***Specification***, not ***implementation***
* API specifies behavior of the thread library, implementation is up to development of the library
* Common in UNIX operating systems (Solaris, Linux, Mac OS X)

**Java Threads**

* Java threads are managed by the JVM
* Typically implemented using the threads model provided by underlying OS
* Java threads may be created by:
	+ Extending Thread class
	+ Implementing the Runnable interface

**Java Multithreaded Program**

****

**Implicit Threading**

* Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
* Creation and management of threads done by compilers and run-time libraries rather than programmers
* **Three methods explored**
	+ Thread Pools
	+ OpenMP
	+ Grand Central Dispatch
* Other methods include Microsoft Threading Building Blocks (TBB), **java.util.concurrent** package

**Thread Pools**

* Create a number of threads in a pool where they await work
* Advantages:
	1. Usually slightly faster to service a request with an existing thread than create a new thread
	2. Allows the number of threads in the application(s) to be bound to the size of the pool
	3. Separating task to be performed from mechanics of creating task allows different strategies for running task
		+ i.e.Tasks could be scheduled to run periodically
* Windows API supports thread pools:



**OpenMP**

1. Set of compiler directives and an API for C, C++, FORTRAN
2. Provides support for parallel programming in shared-memory environments
3. Identifies **parallel regions** – blocks of code that can run in parallel

**#pragma omp parallel**

Create as many threads as there are cores

**#pragma omp parallel for for(i=0;i<N;i++) {**

 **c[i] = a[i] + b[i];**

**}**

Run for loop in parallel



**Grand Central Dispatch**

1. Apple technology for Mac OS X and iOS operating systems
2. Extensions to C, C++ languages, API, and run-time library
3. Allows identification of parallel sections
4. Manages most of the details of threading
5. Block is in “^{ }” - **ˆ{ printf("I am a block"); }**
6. Blocks placed in dispatch queue
	* Assigned to available thread in thread pool when removed from queue
7. Two types of dispatch queues:
	* serial – blocks removed in FIFO order, queue is per process, called **main queue**
		+ Programmers can create additional serial queues within program
	* concurrent – removed in FIFO order but several may be removed at a time
		+ Three system wide queues with priorities low, default, high



**Threading Issues**

1. Semantics of **fork()** and **exec()** system calls
2. Signal handling
	* Synchronous and asynchronous
3. Thread cancellation of target thread
	* Asynchronous or deferred
4. Thread-local storage
5. Scheduler Activations

**Semantics of fork() and exec()**

* Does **fork()**duplicate only the calling thread or all threads?
	+ Some UNIXes have two versions of fork
* **Exec()** usually works as normal – replace the running process including all threads

**Signal Handling**

* **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
* A **signal handler** is used to process signals
	1. Signal is generated by particular event
	2. Signal is delivered to a process
	3. Signal is handled by one of two signal handlers:
		1. default
		2. user-defined
* Every signal has **default handler** that kernel runs when handling signal
	1. **User-defined signal handler** can override default
	2. For single-threaded, signal delivered to process
* **Where should a signal be delivered for multi-threaded?**
	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

**Thread Cancellation**

* Terminating a thread before it has finished
* Thread to be canceled is **target thread**
* Two general approaches:
	1. **Asynchronous cancellation** terminates the target thread immediately
	2. **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
* Pthread code to create and cancel a thread:



* Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state



* If thread has cancellation disabled, cancellation remains pending until thread enables it
* Default type is deferred
	+ Cancellation only occurs when thread reaches **cancellation point**
		- I.e. **pthread\_testcancel()**
		- Then **cleanup handler** is invoked
* On Linux systems, thread cancellation is handled through signals

**Thread-Local Storage**

* **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
* Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
* Different from local variables
	+ Local variables visible only during single function invocation
	+ TLS visible across function invocations
* Similar to **static** data
	+ TLS is unique to each thread

**Scheduler Activations**

* Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
* ****Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
	+ Appears to be a virtual processor on which process can schedule user thread to run
	+ Each LWP attached to kernel thread
	+ How many LWPs to create?
* Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
* This communication allows an application to maintain the correct number kernel threads

**Operating System Examples**

1. Windows XP Threads
2. Linux Thread

**Windows Threads**

1. Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
2. Implements the one-to-one mapping, kernel-level
3. Each thread contains
	1. A thread id
	2. Register set representing state of processor
	3. Separate user and kernel stacks for when thread runs in user mode or kernel mode
	4. Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
4. The register set, stacks, and private storage area are known as the **context** of the thread
5. The primary data structures of a thread include:
	1. ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
	2. KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
	3. TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

**Windows XP Threads Data Structures**



**Linux Threads**

* Linux refers to them as ***tasks*** rather than ***threads***
* Thread creation is done through **clone()** system call
* **clone() allows a child task to share the address space of the parent task (process)**
	+ Flags control behavior



* **struct task\_struct** points to process data structures (shared or unique)