Chapter 4: Threads

Objectives

Thread of a process: basic unit of CPU utilization and is composed of a:

- thread ID
- program counter: register EIP
- register set
- stack
- and it shared with other threads of the same process, the
  - code segment
  - data segment
  - OS resources: open files, signals, etc
- introduce the notion of a thread
- discuss APIs for the Pthreads, Windows and Java thread libraries
- explore several strategies that provide implicit threading
- examing issues related to multithreaded programming
- to cover operating system support for thread in Windows/Linux

Multicore Programming

Types of parallelism (and concurrencies):

- **Data parallelism**: distributes subsets of the same data across multiple cores, same operation on each
- **Task parallelism**: distributing threads across cores, each thread performing a unique operation

Amdahl's Law

Identifies perf gains from adding additional cores to an application that has both serial and parallel components

- S is serial portion and $(1 - S)$ is parallel portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- as N approaches infinity, speedup approaches $\frac{1}{S}$

User Threads and Kernel Threads

Support for threads either at user level or kernel level

User threads: management done by user-level threads library

- supports thread programming: creating & managing program threads
- three primary thread libraries: POSIX Pthreads, Windows threads, Java threads

Kernel threads: supported by the kernel, managed directly by OS

Multithreading Models

Many-to-One

- many user-level threads mapped to single kernel thread
  - efficiently managed by thread library
- one thread blocking causes all threads to block
  - if the thread makes a blocking system-call
- multiple threads are **unable to run in parallel** on multicore system because only one thread can be in kernel at a time
  - does not benefit from multiple cores

One-to-One

- each user-level thread maps to **one** kernel thread
- **Problem**: creating a user thread requires creating the corresponding kernel thread. K-thead creations burden the performan of an application; an *overhead*
- provides more concurrency that many-to-one model in case a thread has blocked, allows multiple threads to run in parallel on multiple CPU/core systems
- number of k-threads per process sometimes restricted due to creation overhead

Many-to-Many Model

- allows m user-level threads to be mapped to n kernel threads with $n \leq m$
- user can create as any amount of user threads
- allows OS to create a sufficient number of kernel threads

Two-Level Model

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

Thread Libraries

Two primary ways of implementing:

1. thread library is entirely in user space **with no kernel support**
   - codes and data structures for thread library are available to user
   - thread library functions are **not** system-calls
2. kernel-level thread library is supported directly by the OS
   - codes and data structures for thread library are **not available** to the user
   - thread library functions are system-calls to the kernel

POSIX Pthread

A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization. Threads extensions of POSIX may be provided either as user-level or kernel-level

Implicit Threading

Growing in popularity as numbers of threads increase, program correctness is more difficult with explicit threads.
Solution = Implicit Threading: Let compilers and runtine libraries create and manage threads rather that programmers
Three methods explored: **Thread Pools**, **OpenMP**, **Grand Central Dispatch**

Thread Pools

Create a number of threads **at process startup** in a pool where they await work

Advantages:

- slightly faster to service a request with an existing thread than create a new thread. A thread returns to pool once it completes servicing a request; request = independent task needed to be executed
- allows number of threads to be bound to size of the pool
- separating task to be performed from mechanics of creating task allows different strategies for running task

OpenMP

- set of compiler **directives** and an API for C, C++, FORTRAN
- provides support for parallel programming in shared-memory environments
- identifies **parallel regions -** blocks of code that can run in parallel

Grand Central Dispatch

Apple technology for Mac OS X and iOS operating systems

- allows identification of parallel/concurrent sections known as blocks
- **block** specified by `^{ }`
  - block = self-contained **unit of work** identified by programmer
- blocks are placed in a dispatch queu
  - assigned to available thread in thead pool when removed from queue

Two types of queues:

- **serial**: each block removed in FIFO order, one at a time, then assigned to a thread; each process has its own serial queue called **main queue**
  - programmers can create additional serial queues within a program
  - block must complete execution before another block is removed
  - each process has its own serial queue
- **concurrent**: removed in FIFO order but several may by removed at a time
  - multiple blocks can execute in parallel or concurrently
  - three system wide concurrent queues with priorities: low, default, high

Example: Linux Threads

- thread creation is done through `clone()` system-calll
- `clone()` allows a child task to share the address space of the parent task (process)
- `struct task_struct` points to process data structures (shared or unique)

Table 1: Flag control behaviour: determine what/how much to share

flag	**meaning**
`CLONE_FS`	File-system information is shared
`CLONE_VM`	The same memory space is shared
`CLONE_SIGHAND`	Signal handlers are shared
`CLONE_FILES`	The set of open files is shared