Thread Scheduling and Synchronization

- Thread models and issues in scheduling.
- After this lecture you will know enough to understand the options and intricacies of most thread APIs.

Thread Models

- We can view the kernel as having its own threads
  - kernel threads or LWPs (lightweight processes)
  - a LWP can be viewed as “virtual CPUs” to which the scheduler of a threads library schedules user-level threads.
- Three dominant models for thread libraries, each with its own trade-offs
  - many threads on one LWP (many-to-one)
  - one thread per LWP (one-to-one)
  - many threads on many LWPs (many-to-many)

Many-to-One Model

- In this model, the library maps all threads to a single lightweight process
- Advantages:
  - totally portable
  - easy to do with few systems dependencies
- Disadvantages:
  - cannot take advantage of parallelism
  - may have to block for synchronous I/O
    - there is a clever technique for avoiding it
- Mainly used in language systems, portable libraries

One-to-One Model

- In this model, the library maps each thread to a different lightweight process
- Advantages:
  - can exploit parallelism, blocking system calls
- Disadvantages:
  - thread creation involves LWP creation
  - each thread takes up kernel resources
    - limiting the number of total threads
- Used in LinuxThreads and other systems where LWP creation is not too expensive
Many-to-Many Model

- In this model, the library has two kinds of threads: **bound** and **unbound**
  - bound threads are mapped each to a single lightweight process
  - unbound threads may be mapped to the same LWP

- Probably the best of both worlds
- Used in the Solaris implementation of Pthreads (and several other Unix implementations)

Contestation Scope

- *Contestation scope* is the POSIX term for describing bound and unbound threads
  - Bound thread is said to have **system contention scope**
    - it contends with all threads in system
  - Unbound thread has **process contention scope**
    - it contends with threads in same process
  - In Pthreads, scope is set at thread creation by a parameter in the attribute block:
    PTHREAD_SCOPE_SYSTEM (system), and PTHREAD_SCOPE_PROCESS (process)

Process Scope Context Switching

Four ways to cause a running thread to context switch:

- **Synchronization**
  - most common: thread goes to sleep on mutex or condition variable

- **Pre-emption**
  - running thread does something that causes high-priority thread to become runnable
  - cannot be implemented entirely in user-space except in a one-to-one model

- **Yielding**
  - thread may explicitly yield to another thread of same priority

- **Time-slicing**
  - threads of same priority may be context switched periodically

Process Scope Context Switching

- Time slicing and pre-emption cannot be done completely in user space
  - at very least a signal needs to be sent and/or handled

- Question: What happens when library context switches threads?
In Practical Terms

- What should you do in your programs?
  - ideally you can choose the number of LWPs you need
  - few actual libraries support this
  - if need many thousands of threads, process scope is the only option for the bulk of them
  - if program is CPU bound, need at least one LWP per CPU on your machine
  - if program performs blocking system calls, need one LWP per simultaneous blocking call

- Summary: for most practical purposes, use bound threads
  - it is more of a sure bet with current implementations
  - after all, Solaris has cheap LWP switching, LinuxThreads are one-to-one, Windows NT makes it hard to use unbound threads (fibers)

- you can set the initial thread stack size

Synchronization Building Blocks

- Most synchronization on symmetric multiprocessors is based on an atomic test and set instruction in hardware
  - we need to do a load and store atomically

- Example:
  
  try_again:
  ldstub address -> register
  compare register, 0
  branch_equal got_it
  call go_to_sleep
  jump try_again
  got_it:
  return

  ldstub: load and store unsigned byte (SPARC)

- Other kinds of atomic primitives at the hardware level may be even more powerful
  - e.g., Load Locked and Store Conditional on the Alpha

Cross-Process Synchronization Variables

- A synchronization variable can be used to synchronize between multiple processes

- How it is done in Pthreads:
  - the variable needs to be placed in shared memory (can also be stored persistently in a file!)
  - both processes must know about the variable
  - exactly one of the processes must initialize the variable to be cross-process

- Cross-process synchronization is slower (?)