Threads

- We have previously defined a *process* as a "program in execution" — but we didn't say anything specifically about what's going on inside that program
- Multiple tasks are desirable *within* a single program: Web servers, Web browsers, database clients, etc.
- These tasks are typically called *threads*
- Official textbook definition: a *thread* is a basic unit of *CPU utilization* — consisting of an identifier, program counter, register set, and a stack

- Typical nomenclature: a process is viewed as *heavyweight* — it contains elements shared by all threads (code/text, data, possibly heap)
- Threads in the process are viewed as *lightweight*
- Is a thread an operating system (kernel) entity?
  - It is certainly a user-level entity — otherwise you wouldn't be able to program them!
  - *Many-to-one:* Multiple user threads are bound to a single underlying kernel thread
  - *One-to-one:* Each user thread has its own kernel thread
  - *Many-to-many:* User threads are *multiplexed* across the same or fewer kernel threads

Thread Libraries

- Not surprisingly, you need an API in order to write a thread-savvy program
- There are actually quite a few — a sampling can be found here: *http://www.gnu.org/software/pth/related.html*
- An API may be implemented as:
  - ◇ Completely user-level — OS doesn't know or care
  - ◇ Part of the OS — OS participates in thread activities
- Note how the API is distinct from its implementation: Pthreads and Java have been implemented both ways, while Win32 and Mach threads are kernel-based

Thread API Abstractions

Regardless of the specifics, thread libraries share many common characteristics, since ultimately they try to deliver the same functionality:

- Thread objects — some entity that represents the thread itself; examples include an ID (Pthreads, Mach), reference (Win32), or object (Java)
- Execution entry points — where does the thread start running? May be a function reference (if subroutines are first-class); Java uses interfaces (Runnable, Callable)
- Thread operations — allocation, synchronization, etc.

Threading Issues

A number of issues arise in multithreaded programs:

- Process spawning — When a child process is created as a copy of the parent (e.g., *fork()*), does the child also duplicate that parent's threads?
- Thread cancellation — What to do about resources allocated to threads that need to be cancelled?
- Asynchronous events (signals, asynchronous procedure calls) — Which thread(s) get the event? How are the events handled?

- Thread pools — Threads may be less expensive than processes, but creating/maintaining them still entails performance and memory costs; a *thread pool* maintains a finite set of "ready-to-go" threads that can be recycled as requests come in, with extra requests being queued until a thread becomes available
- Thread-specific (or -local) data — Threads generally share the data of the overall process that created them, but occasionally it makes sense to have "local variables" in thread scope
- Thread interaction with the underlying kernel — usually done through an intermediate data structure called a *lightweight process* (LWP) or *virtual processor* which binds a thread to an underlying kernel thread