- The scope of multithreading

LWPs, kernel and user threads

- kernel-level threads – supported by the kernel
  - Solaris, Linux, Windows XP/2000
  - all scheduling, synchronization, thread structures maintained in kernel
  - could write apps using kernel threads, but would have to go to kernel for everything
- user-level threads – supported by a user-level library
  - Pthreads, Java threads, Win32…
  - sched. & synch can often be done fully in user space; kernel doesn't need to know there are many user threads
  - problem with blocking on a system call

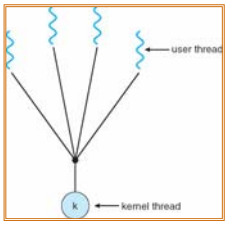
- LightWeight Processes - LWP
  - these are "virtual CPUs", can be multiple per process
  - the scheduler of a threads library schedules user-level threads to these virtual CPUs
  - kernel threads implement LWPs => visible to the kernel, and can be scheduled
    - sometimes LWP & kernel threads used interchangeably, but there can be kernel threads without LWPs

Multithreading models

- There are 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)
- similar models can apply on scheduling kernel threads to real CPUs

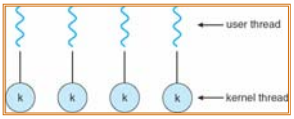
Many to one

- 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



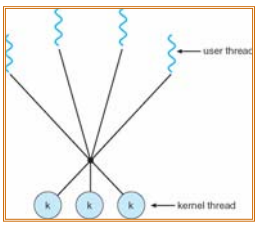
One to one

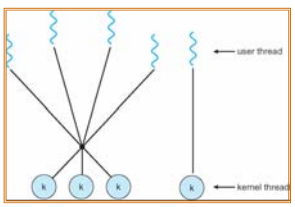
- In this model, the library maps each thread to a different lightweight process
- Advantages:
  - can exploit parallelism, blocking system calls
- Disadvantages:
  - thread creation involves LWP creation
  - each thread takes up kernel resources
  - limiting the number of total threads
- Used in LinuxThreads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



Many to many

- 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)





High-Level Program Structure Ideas

- Boss/workers model
- Pipeline model
- Up-calls
- Keeping shared information consistent using version stamps

Thread Design Patterns

Common ways of structuring programs using threads

- Boss/workers model
  - boss gets assignments, dispatches tasks to workers
  - variants (thread pool, single thread per connection…)
- Pipeline model
  - do some work, pass partial result to next thread
- Up-calls
  - fast control flow transfer for layered systems
- Version stamps
  - technique for keeping information consistent

Boss/Workers

Boss:

```
forever {
    get a request
    switch(request)
      case X: Fork (taskX)
    case Y: Fork (taskY)
    …
}
```

Worker:

```
taskX();
```

- Advantage: simplicity
- Disadvantage: bound on number of workers, overheard of threads creation, contention if requests have interdependencies
- Variants: fixed thread pool (aka *workpile*, *workqueue*), producer/consumer relationship, workers determine what needs to be performed…

Pipeline

- Each thread completes portion of a task, and passes results
- like an assembly line or a processor pipeline
- Advantages: trivial synchronization, simplicity
- Disadvantages: limits degree of parallelism, throughput driven by slowest stage, handtuning needed

Up calls

- Layered applications, e.g. network protocol stacks have top-down and bottom-up flows
- Up-calls is a technique in which you structure layers so that they can expect calls from below
- Thread pool of specialized threads in each layer
  - essentially an up-call pipeline per connection
- Advantages: best when used with fast, synchronous control flow transfer mechanisms or program structuring tool
- Disadvantages: programming becomes more complicated, synchronization required for top-down

Version Stamps

- (not a programming structure idea but useful technique for any kind of distributed environment)
- maintain “version number” for shared data
  - keep local cached copy of data
  - check versions to determine if changed