Lecture 9

- Administration

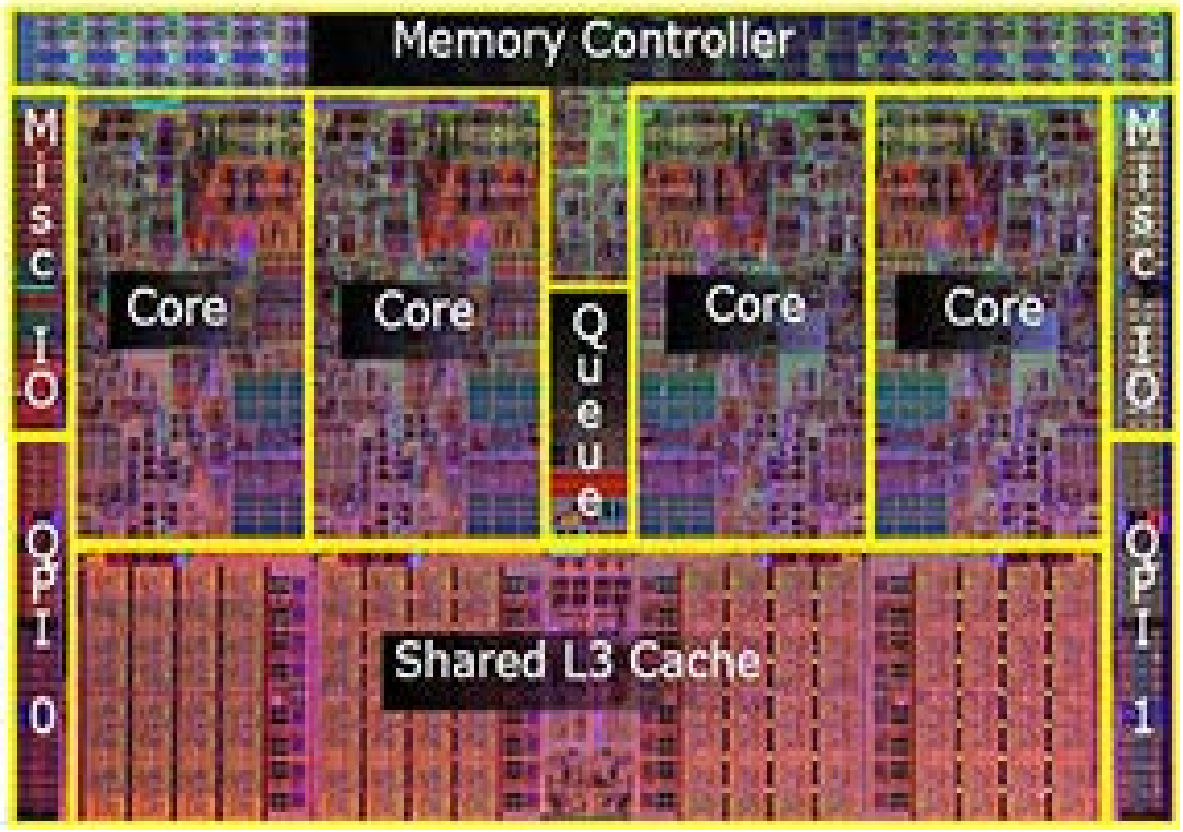
- Pthreads
- Exceptions in C

https://computing.llnl.gov/tutorials/pthreads/

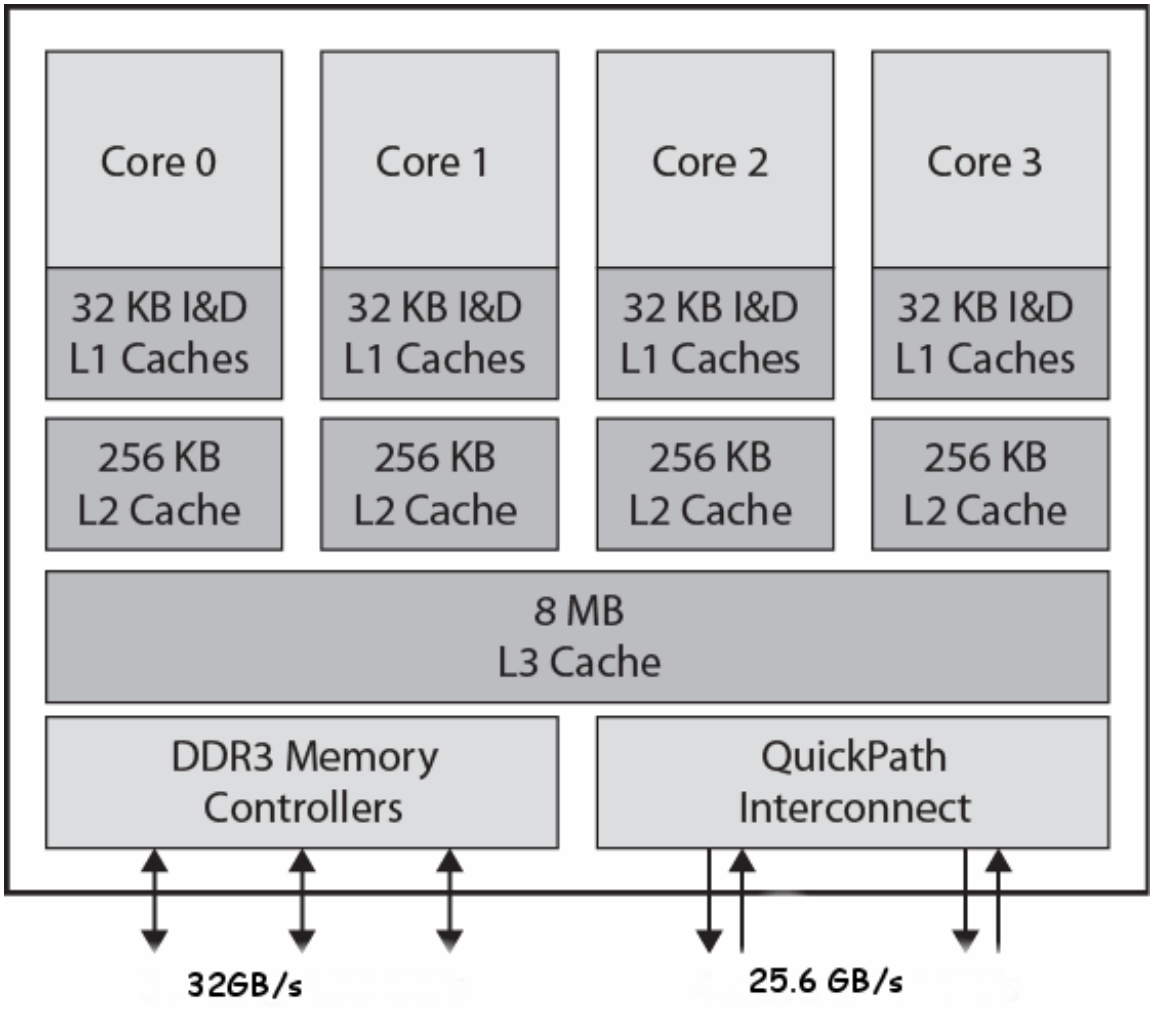
Intel Core i7

approx 45x45 mm

45 nm feature size

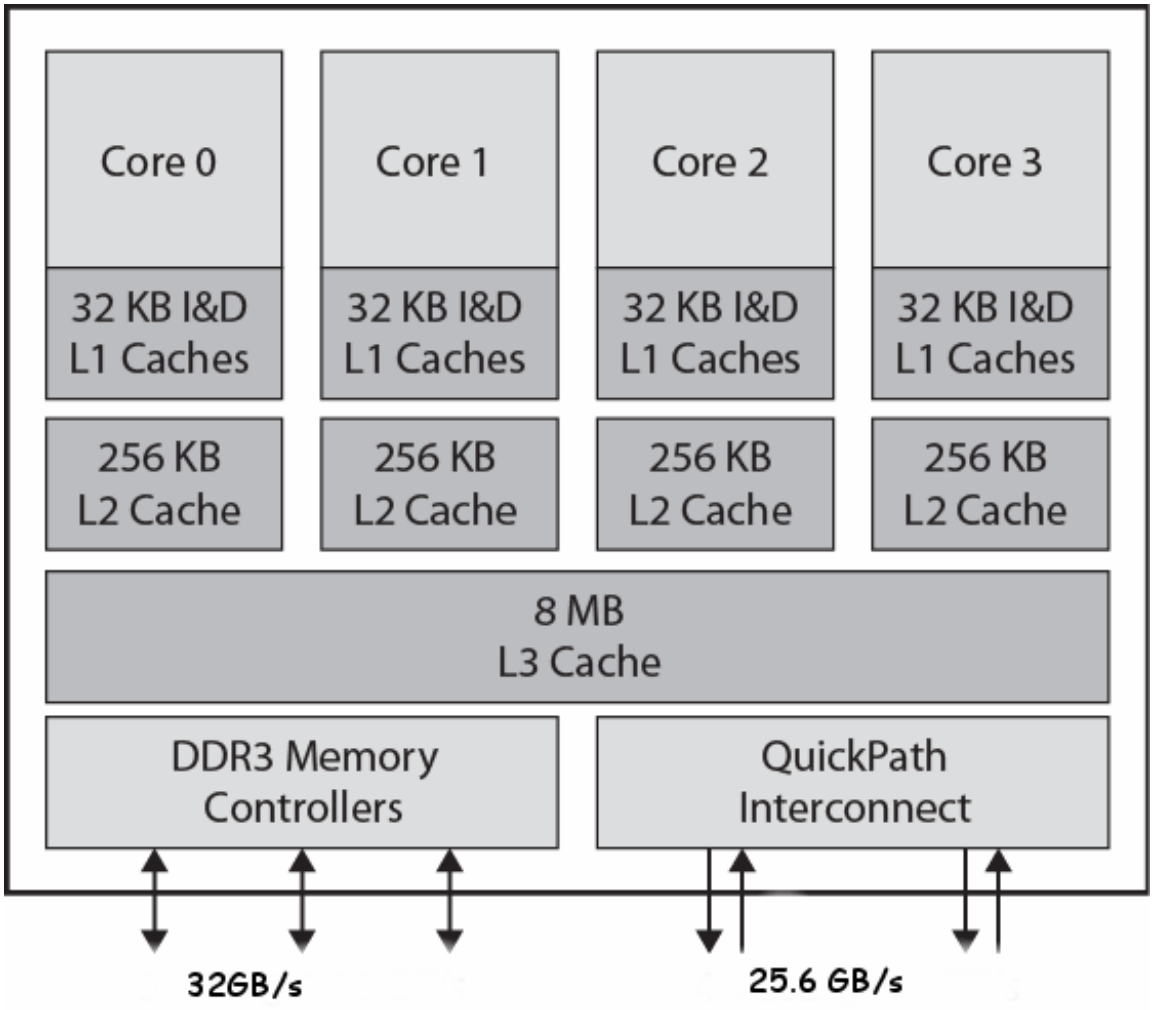


Intel Core i7 Block Diagram



.3 ns/B !

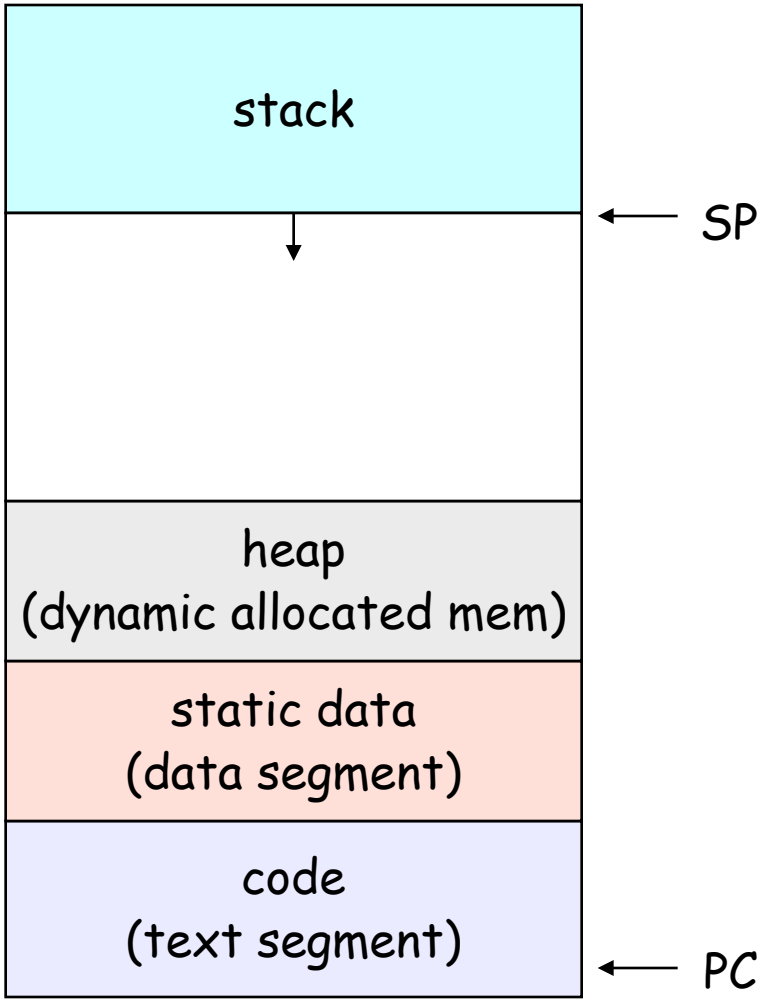
Intel Core i7 Block Diagram



.3 ns/B !

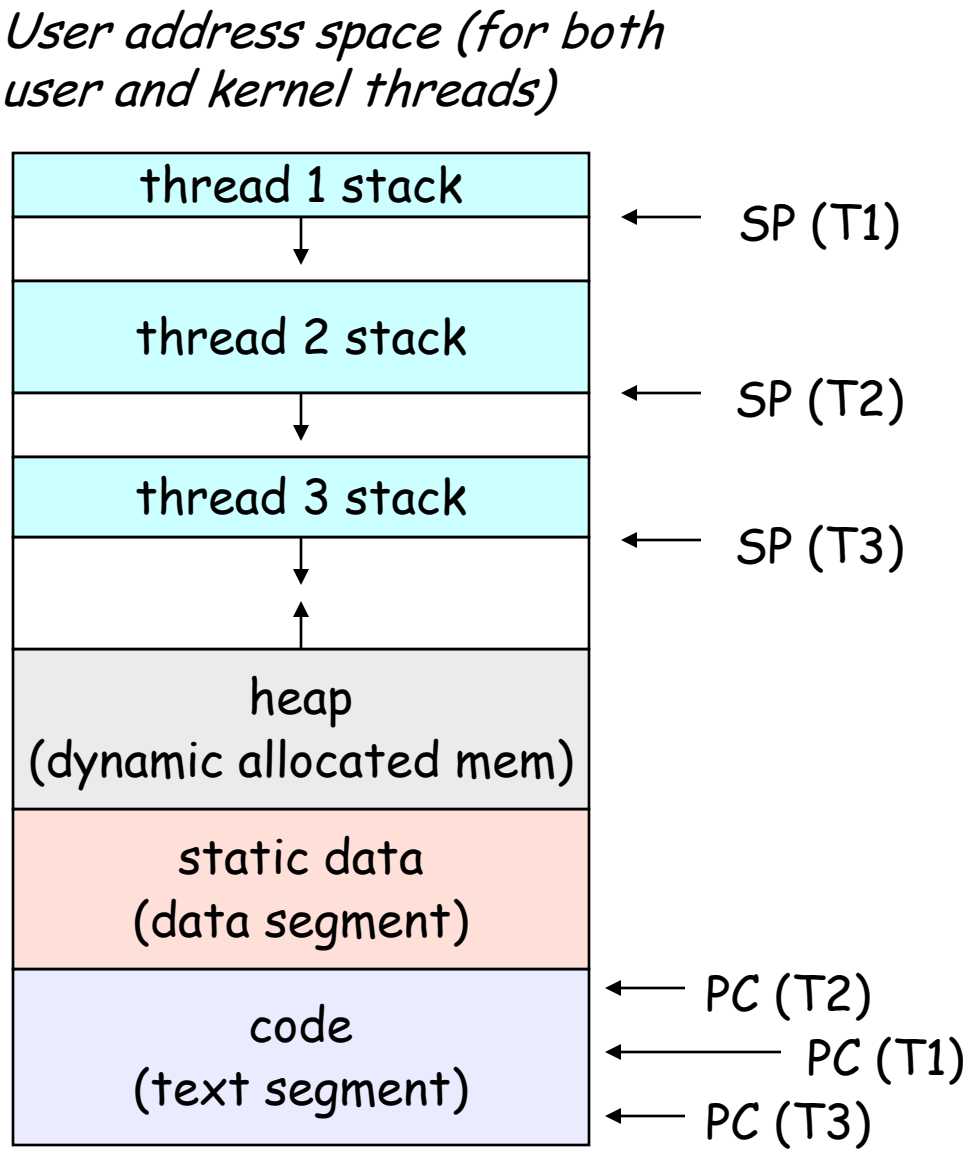
Process Address Space Review

- Every process has a user stack and a program counter
- In addition, each process has a kernel stack and program counter
  - (not shown here)



Threaded Address Space

- Every thread *always* has its own user stack and program counter
  - For both user, kernel threads
- For user threads, there is only a single kernel stack, program counter, PCB, etc.

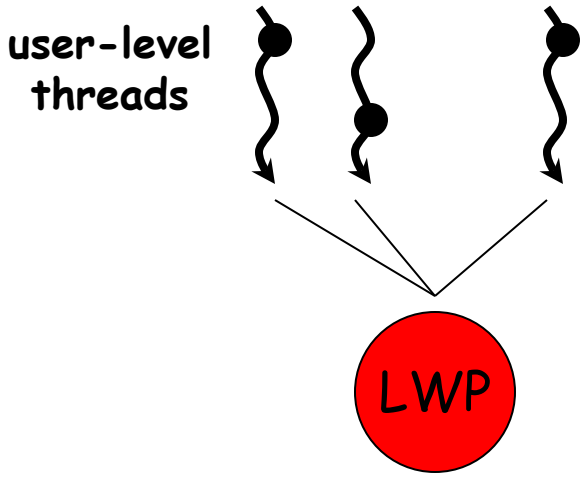


Light versus Heavy weight

- Why Pthreads?
  - The primary motivation for using Pthreads is to realize potential program performance gains.
  - Fork needs IPC (Inter-Process Communication communicate)
  - For example, the following table compares timing results for the fork() subroutine and the pthreads_create() subroutine. Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Many-to-One Model

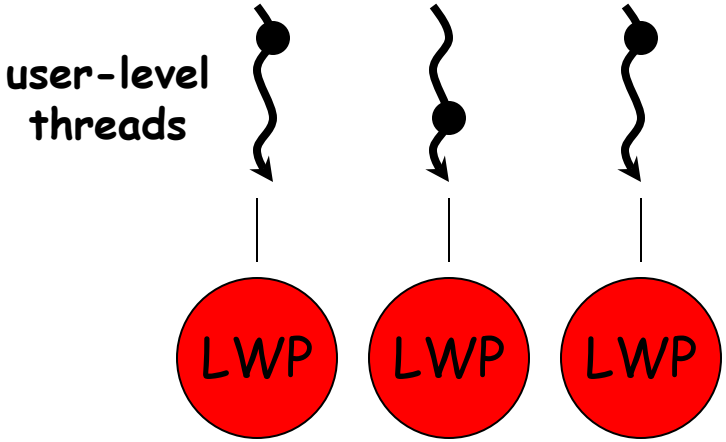


Thread creation, scheduling, synchronization done in user space.

Mainly used in language systems, portable libraries

- Fast - no system calls required
- Few system dependencies; portable
- No parallel execution of threads - can't exploit multiple CPUs
- All threads block when one uses synchronous I/O

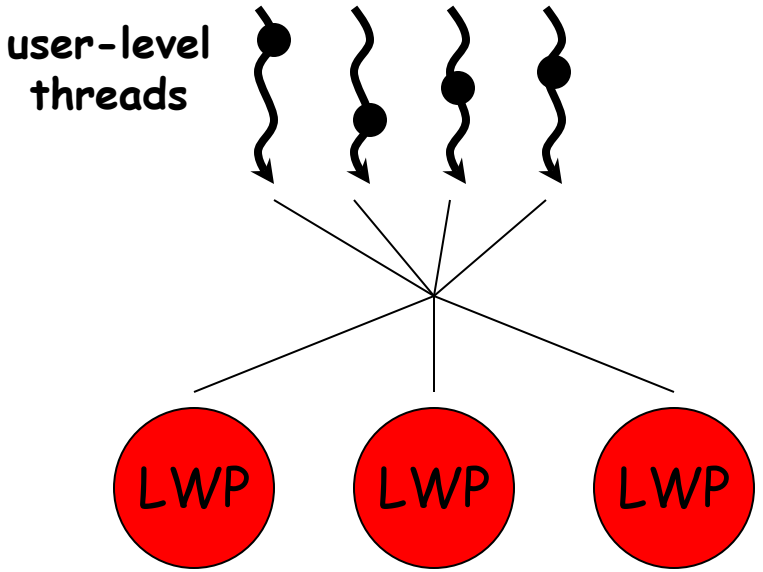
One-to-one Model



Thread creation, scheduling, synchronization require system calls
Used in Linux Threads, Windows NT, Windows 2000, OS/2

- More concurrency
- Better multiprocessor performance
- Each user thread requires creation of kernel thread
- Each thread requires kernel resources; limits number of total threa

Many-to-Many Model

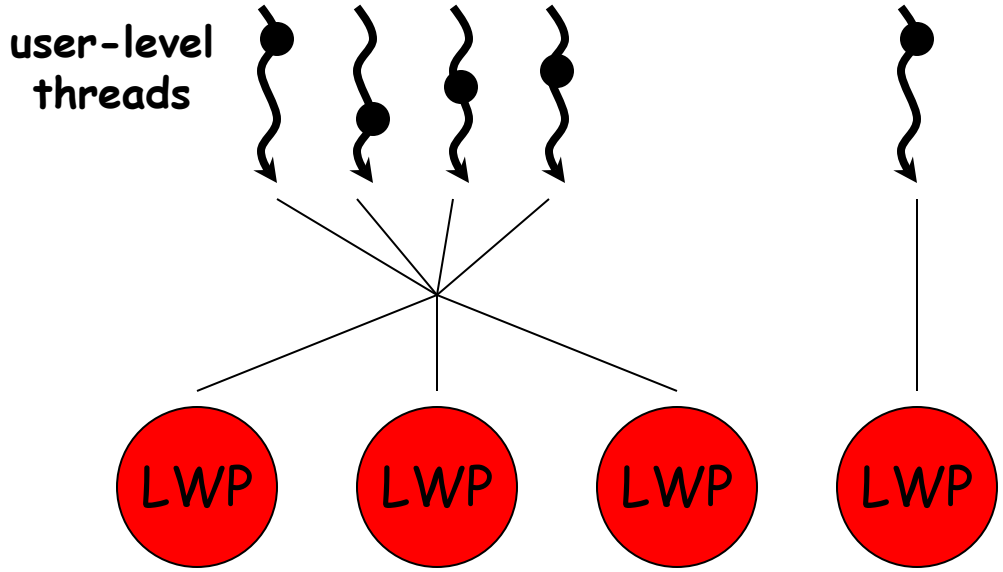


If U < L? No benefits of multithreading
If U > L, some threads may have to wait for an LWP to run

- Active thread - executing on an LWP
- Runnable thread - waiting for an LWP

A thread gives up control of LWP under the following:
– synchronization, lower priority, yielding, time slicing

Two-level Model



- Combination of one-to-one + "strict" many-to-many models
- Supports both bound and unbound threads
  - Bound threads - permanently mapped to a single, dedicated LWP
  - Unbound threads - may move among LWPs in set
- Thread creation, scheduling, synchronization done in user space
- Flexible approach, "best of both worlds"
- Used in Solaris implementation of Pthreads and several other Unix implementations (IRIX, HP-UX)

User-Level vs. Kernel Threads

User-Level

- Managed by application
- Kernel not aware of thread
- Context switching cheap
- Create as many as needed
- Must be used with care

Kernel-Level

- Managed by kernel
- Consumes kernel resources
- Context switching expensive
- Number limited by kernel resources
- Simpler to use

Key issue: kernel threads provide virtual processors to user-level threads, but if all of kthreads block, then all user-level threads will bloc*even* if the program logic allows them to proceed

When Would User Threads Be Useful?

- The π calculator?
- The web server?
- The Fibonacci GUI?

Problems with Many-to-Many Threads

- Lack of coordination between user and kernel schedulers
  - "Left hand not talking to the right"
- Specific problems
  - Poor performance
    - e.g., the OS preempts a thread holding a crucial lock
  - Deadlock
    - Given K kernel threads, at most K user threads can block
      - Other runnable threads are starved out!

Multi-processor Scheduling

- Preemptive
- Non-preemptive