Threads

Operating Systems

Department of Computer Science
Rutgers University

Threads

- Thread of execution: stack + registers (which includes the PC)
  - Informally: where an execution stream is currently at in the program and the routine call chain that brought the stream to the current place
  - Example: A called B which called C which called B which called C
  - The PC should be pointing somewhere inside C at this point
  - The stack should contain 5 activation records: A/B/C/B/C
  - Thread for short
- Process model discussed last time implies a single thread

Multi-Threading

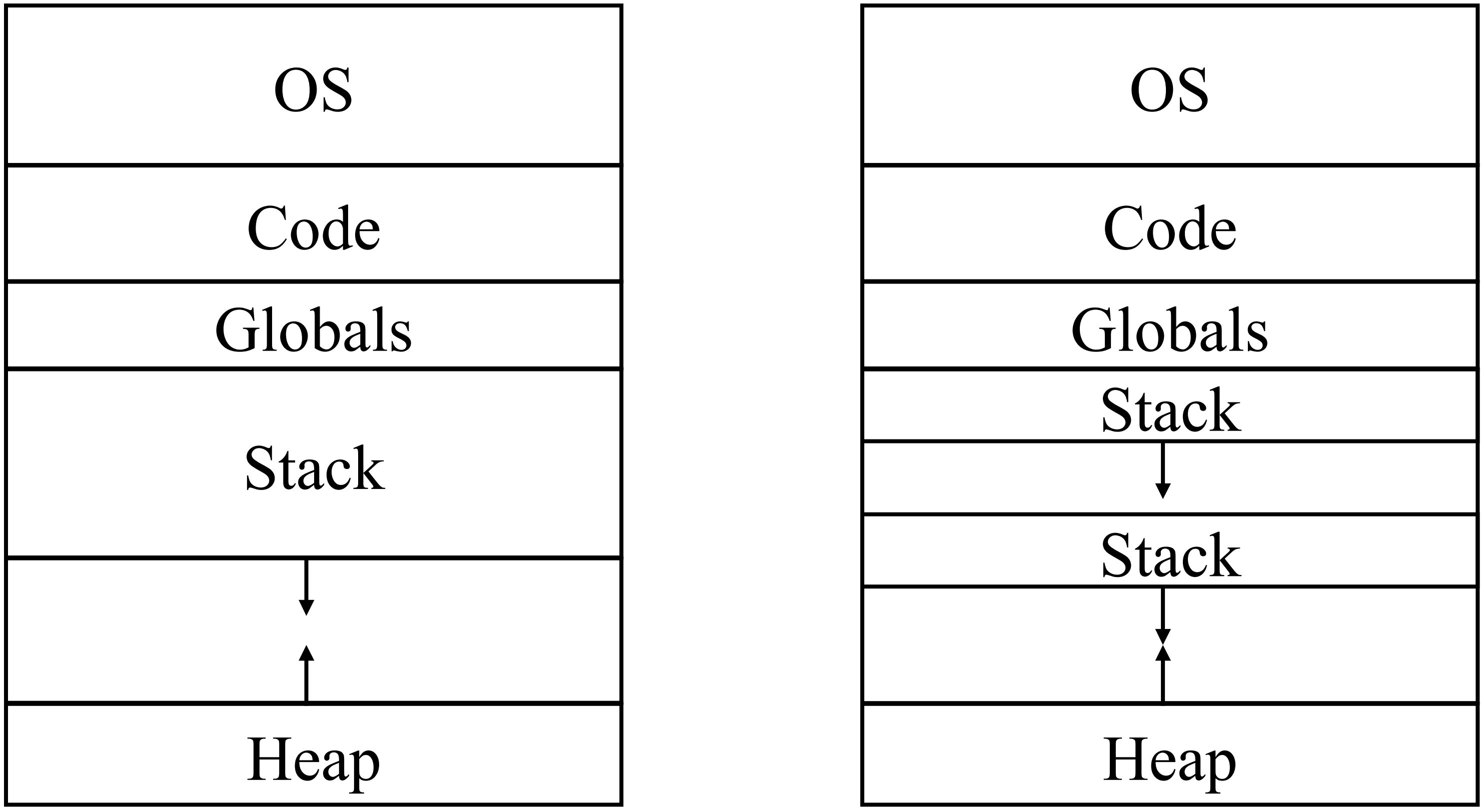
Why limit ourselves to a single thread?

- Think of a web server that must service a large stream of requests
- If only have one thread, can only process one request at a time
- What to do when reading a file from disk?

Multi-threading model

- Each process can have multiple threads
- Each thread has a private stack
  - Registers are also private
- All threads of a process share the code, globals, and heap
  - Objects to be shared across multiple threads should be allocated on the heap or in the globals area

Process Address Space Revisited



(a) Single-threaded address space (b) Multi-threaded address space

Multi-Threading (cont)

Implementation

Each thread is described by a *thread-control block* (TCB)

A TCB typically contains

- Thread ID
- Space for saving registers
- Thread-specific info (e.g., signal mask, scheduling priority)

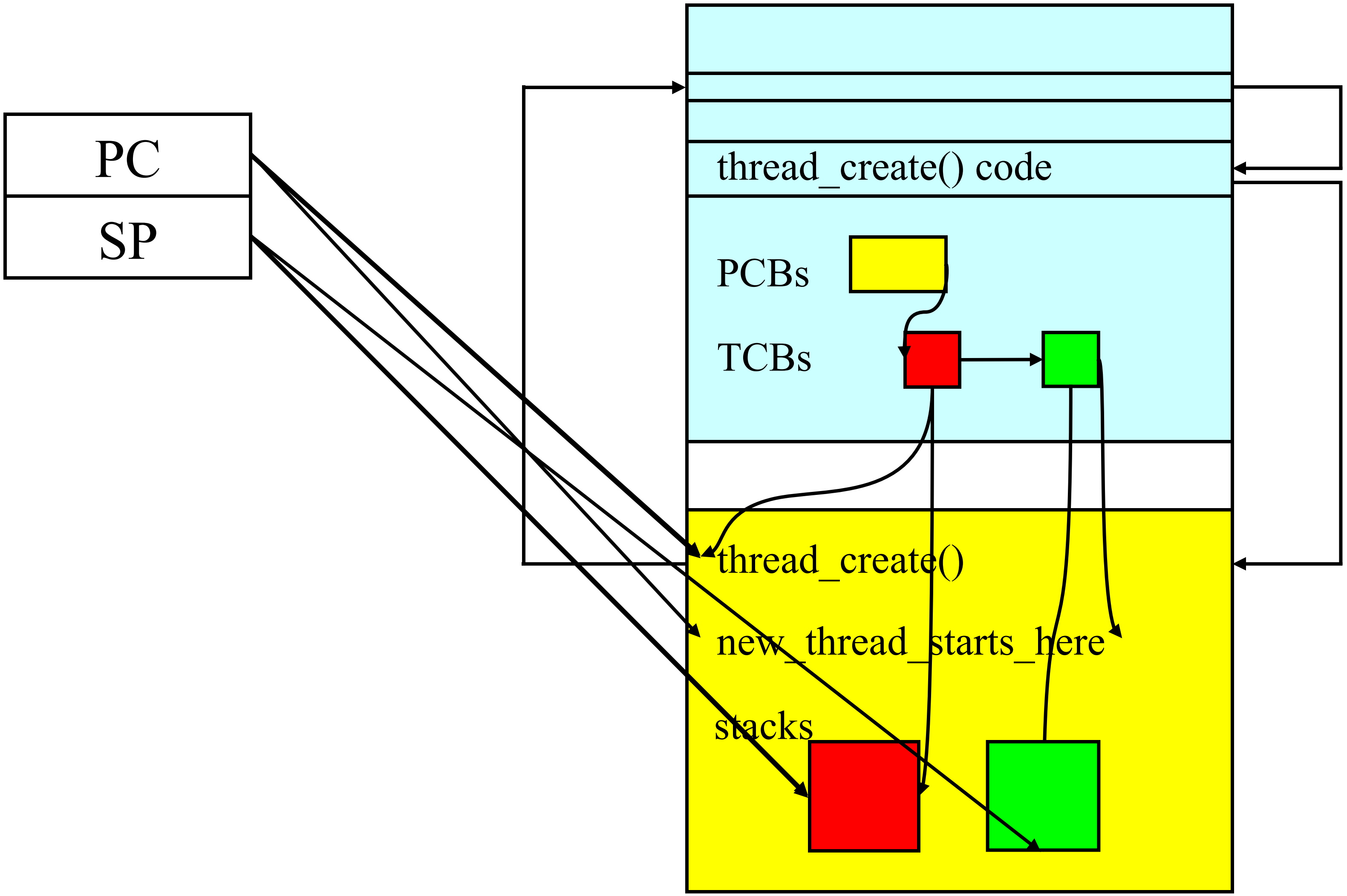
Observation

Although the model is that each thread has a private stack, threads actually share the process address space

⇒ There's no memory protection!

⇒ Threads could potentially write into each other's stack

Thread Creation



Context Switching

Suppose a process has multiple threads on a machine with a single non-multithreaded CPU core … what to do?

In fact, even if we only had one thread per process, we would have to do something about running multiple processes …

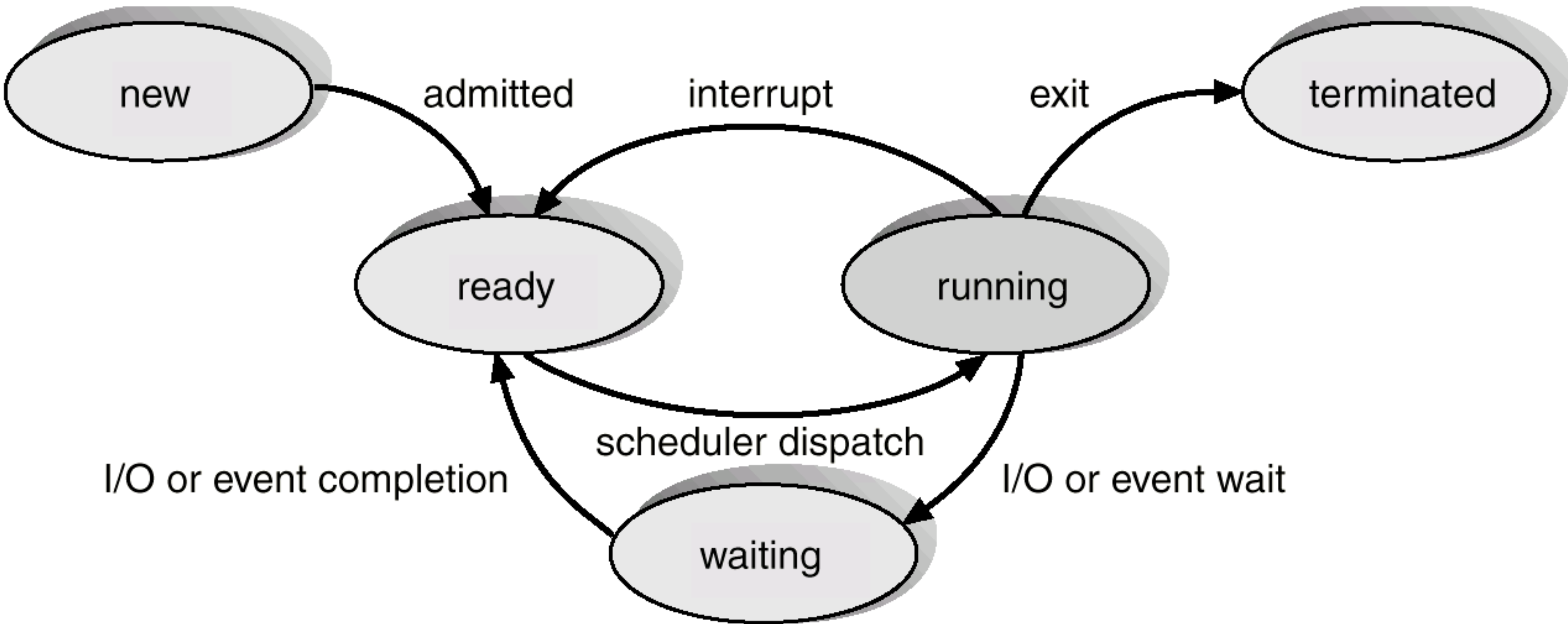
We *multiplex* the multiple threads on the core

At any point in time, only one thread is *running* (again, assuming a single non-multithreaded core)

At some point, the OS may decide to stop the currently running thread and allow another thread to run

This switching from one running thread to another is called *context switching*

Diagram of Thread State



Context Switching (cont)

How to do a context switch?

Save state of currently executing thread

- Copy all "live" registers to thread control block
- For register-only machines, need at least 1 scratch register
  - points to area of memory in thread control block that registers should be saved to

Restore state of thread to run next

- Copy values of live registers from thread control block to registers

When does context switching take place?

Context Switching (cont)

When does context switching occur?

- When the OS decides that a thread has run long enough and that another thread should be given the CPU
- When a thread performs an I/O operation and needs to block to wait for the completion of this operation
- To wait for some other thread
  - Thread synchronization: we'll talk about this lots in a couple of lectures

How Is the Switching Code Invoked?

user thread executing → clock interrupt → PC modified by hardware to "vector" to interrupt handler → user thread state is saved for restart → clock interrupt handler is invoked → disable interrupt checking → check whether current thread has run "long enough" → if yes, post asynchronous software trap (AST) → enable interrupt checking → exit clock interrupt handler → enter "return-to-user" code → check whether AST was posted → if not, restore user thread state and return to executing user thread; if AST was posted, call context switch code

Why need AST?

How Is the Switching Code Invoked? (cont)

user thread executing → system call to perform I/O → PC modified by hardware to “vector” to trap handler → user thread state is saved for restart → OS code to perform system call is invoked → disable interrupt checking → I/O operation started (by invoking I/O driver) → set thread status to waiting → move thread’s TCB from run queue to wait queue associated with specific device → enable interrupt checking → exit trap handler → call context switching code

Context Switching

At entry to CS, the return address is either in a register or on the stack (in the current activation record)

CS saves this return address to the TCB instead of the current PC

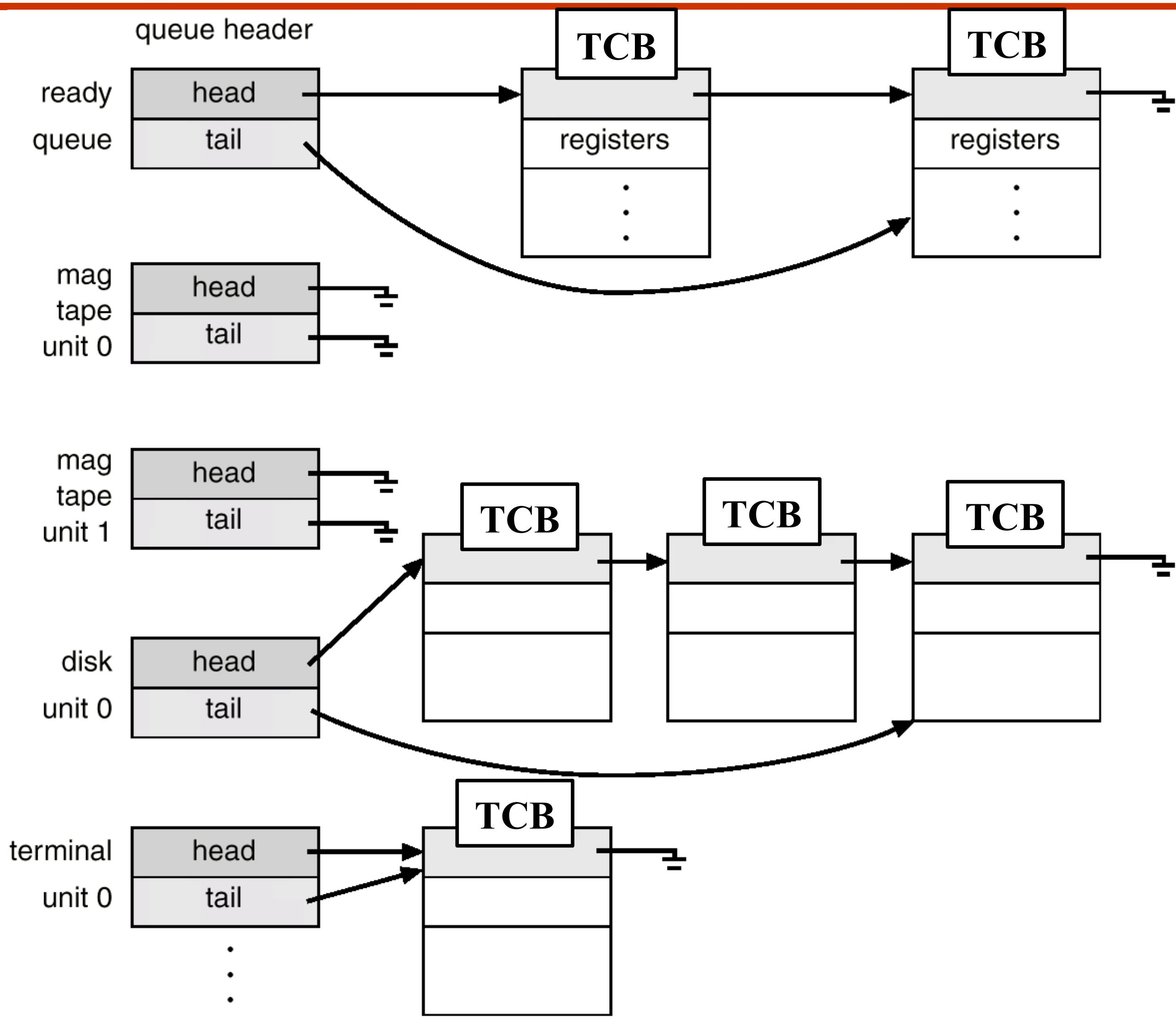
To thread, it looks like CS just took a while to return!

- If the context switch was initiated from an interrupt, the thread never knows that it has been context switched out and back in unless it looks at the “wall” clock

Context Switching (cont)

- Even that is not quite the whole story
- When a thread is switched out, what happens to it?
- How do we find it to switch it back in?
- This is what the TCB is for. System typically has
  - A run queue that points to the TCBs of threads ready to run
  - A blocked queue per device to hold the TCBs of threads blocked waiting for an I/O operation on that device to complete
  - When a thread is switched out at a timer interrupt, it is still ready to run so its TCB stays on the run queue
  - When a thread is switched out because it is blocking on an I/O operation, its TCB is moved to the blocked queue of the device

Ready Queue And Various I/O Device Queues



Switching Between Threads of Different Processes

What if switching to a thread of a different process?

Caches, TLB, page table, etc.?

- Caches
  - Physical addresses: no problem
  - Virtual addresses: cache must either have process tag or must flush cache on context switch
- TLB
  - Each entry must have process tag or must flush TLB on context switch
- Page table
  - Typically have page table pointer (register) that must be reloaded on context switch

Threads & Signals

- What happens if kernel wants to signal a process when all of its threads are blocked?
- When there are multiple threads, which thread should the kernel deliver the signal to?
  - OS writes into process control block that a signal should be delivered
  - Next time any thread from this process is allowed to run, the signal is delivered to that thread as part of the context switch
  - What happens if kernel needs to deliver multiple signals?

Thread Implementation

Kernel-level threads

Kernel sees multiple execution contexts

Thread management done by the kernel

User-level threads

Implemented as a thread library which contains the code for thread creation, termination, scheduling and switching

Kernel sees one execution context and is unaware of thread activity

Can be preemptive or not

User-Level vs. Kernel-Level Threads

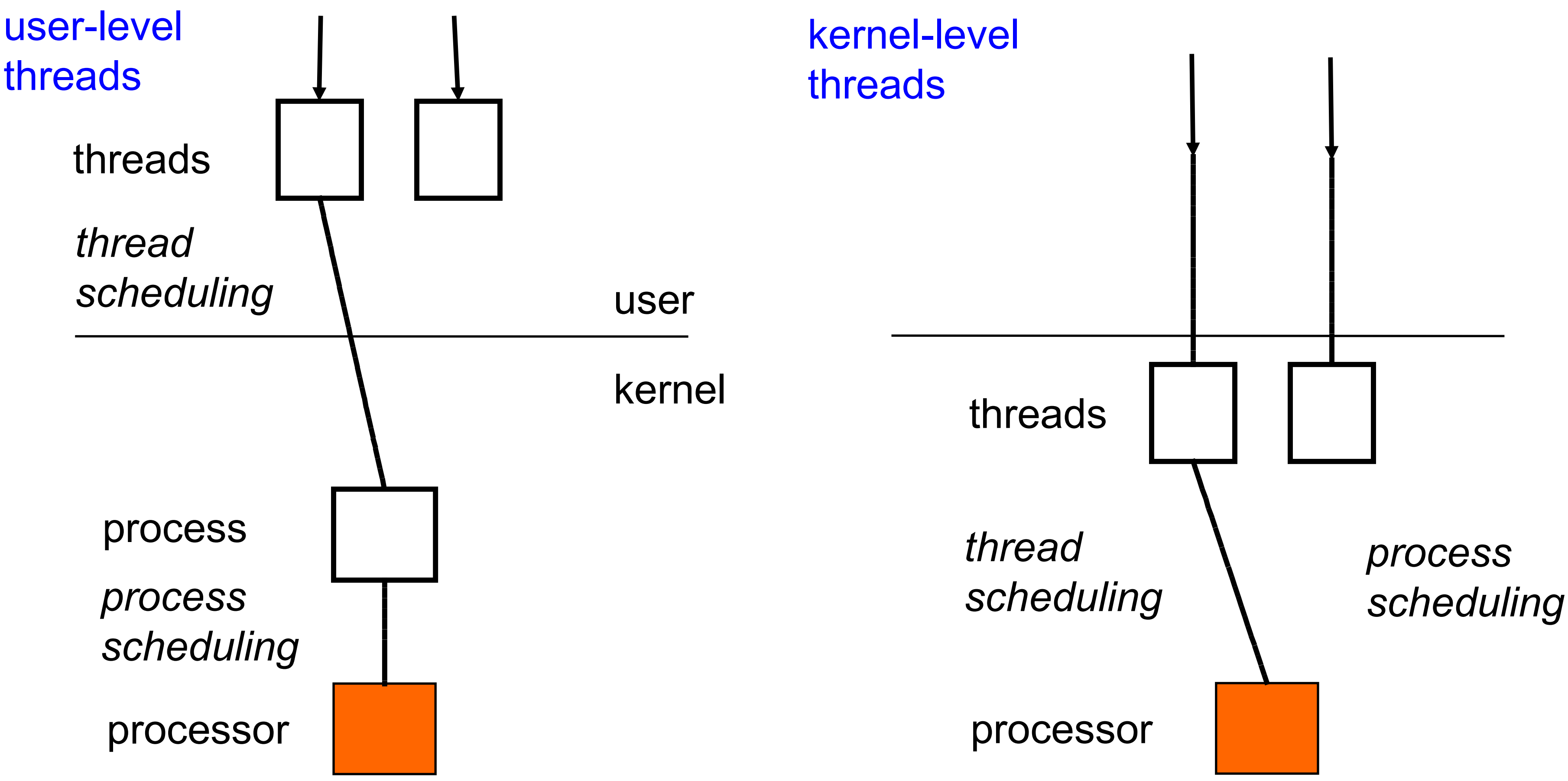
Advantages of user-level threads

- Performance: low-cost thread operations (do not require crossing protection domains)
- Flexibility: scheduling can be application-specific
- Portability: user-level thread library easy to port

Disadvantages of user-level threads

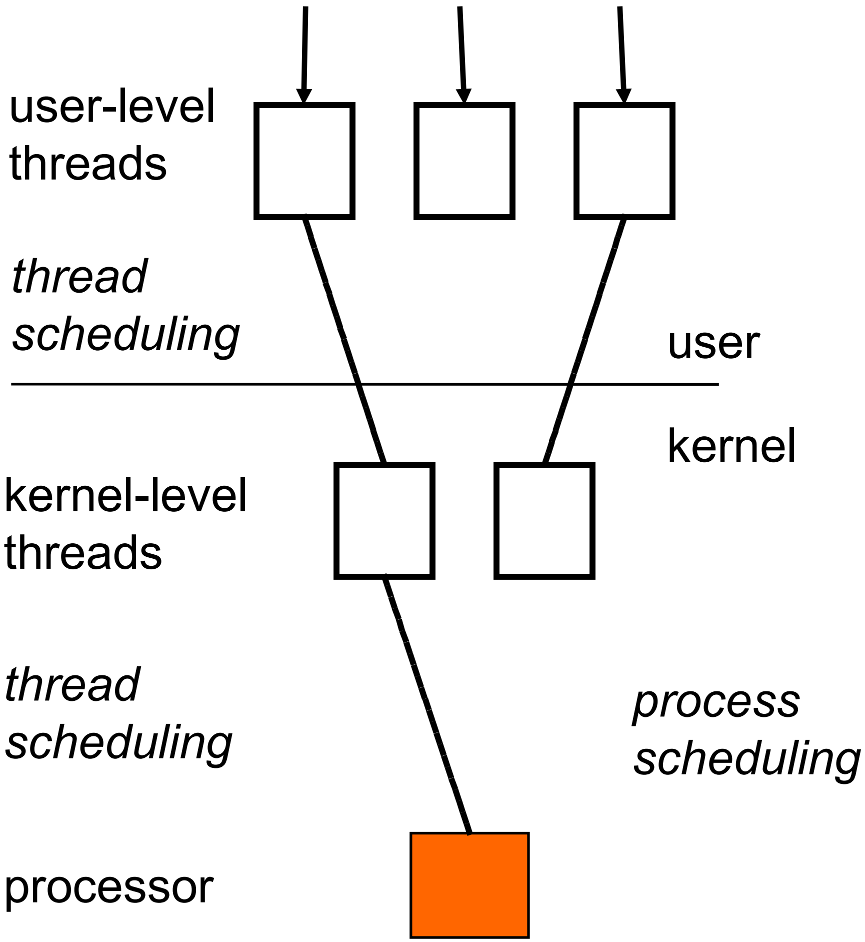
- If a user-level thread is blocked in the kernel, the entire process (all threads of that process) are blocked
- Cannot take advantage of multiprocessing (the kernel assigns one process to only one processor)

User-Level vs. Kernel-Level Threads



User-Level vs. Kernel-Level Threads

- No reason why we shouldn't have both
- Most systems now support kernel threads
- User-level threads are available as linkable libraries



Thread Implementation in Real OSes

Lightweight processes (Solaris only)

- LWPs create an extra layer between user-level and kernel-level threads
- An LWP runs in user-space on top of a kernel-level thread; multiple user-level threads can be created on top of each LWP
- LWPs of the same process share data

Process ~= Thread (Linux only)

- The schedulable entities are processes (called "tasks" in Linux lingo)
- A process can be seen as a single thread, but a process can contain multiple threads that share code + data
- In the Pthreads library (Native POSIX Thread Library or NPTL) for Linux, each thread created corresponds to a kernel schedulable entity

Threads vs. Processes

Why multiple threads?

Can't we use multiple processes to do whatever that is that we do with multiple threads?

Of course, we need to be able to share memory (and other resources) between multiple processes …

But this sharing is already supported

Operations on threads (creation, termination, scheduling, etc) are cheaper than the corresponding operations on processes

This is because thread operations do not involve manipulations of other resources associated with processes

Inter-thread communication is supported through shared memory without kernel intervention

Why not? Have multiple other resources, why not threads

Thread/Process Operation Latencies

Operation	User-level Thread (μs)	Kernel Threads (μs)	Processes (μs)
Null fork	34	948	11,300
Signal-wait	37	441	1,840

VAX uniprocessor running UNIX-like OS, 1992.

Operation	Kernel Threads (μs)	Processes (μs)
Null fork	45	108

2.8-GHz Pentium 4 uniprocessor running Linux, 2004.

Next Time

Synchronization