Operating Systems

Threads

Lecture 4
Michael O'Boyle

Overview

- Process vs threads
  - how related
- Concurrency
  - why threads
- Design space of process/threads
  - a simple taxonomy
- Kernel threads
  - more efficient
- User-level threads
  - even faster

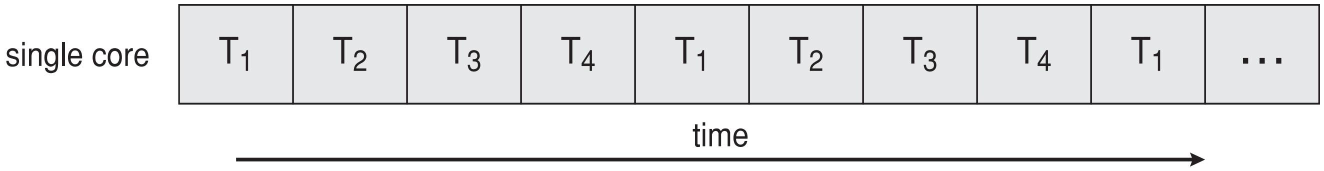
What’s “in” a process?

- A process consists of (at least):
  - An address space, containing
    - the code (instructions) for the running program
    - the data for the running program
  - Thread state, consisting of
    - The program counter (PC), indicating the next instruction
    - The stack pointer register (implying the stack it points to)
    - Other general purpose register values
  - A set of OS resources
    - open files, network connections, sound channels, …
- Decompose …
  - address space
  - thread of control (stack, stack pointer, program counter, registers)
  - OS resources

Thread: Concurrency vs. Parallelism

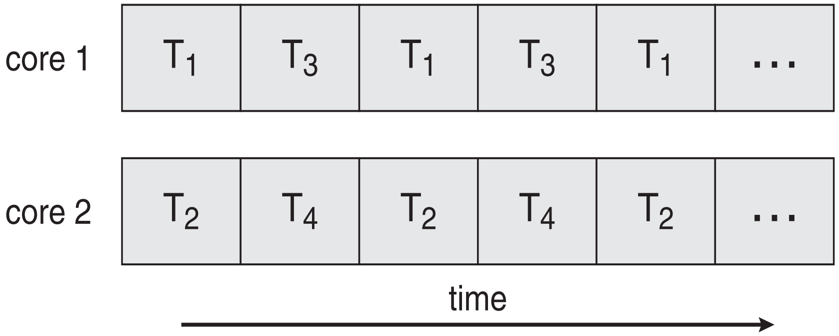
- Threads are about concurrency and parallelism

- **Concurrent execution on single-core system:**



single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	…

- **Parallelism on a multi-core system:**



core 1	T_1	T_3	T_1	T_3	T_1	…
core 2	T_2	T_4	T_2	T_4	T_2	…

Motivation

- Threads are about concurrency and parallelism
- One way to get concurrency and parallelism is to use multiple processes
  - The programs (code) of distinct processes are isolated from each other
- Threads are another way to get concurrency and parallelism
  - Threads “share a process” – same address space, same OS resources
  - Threads have private stack, CPU state – are schedulable

What’s needed?

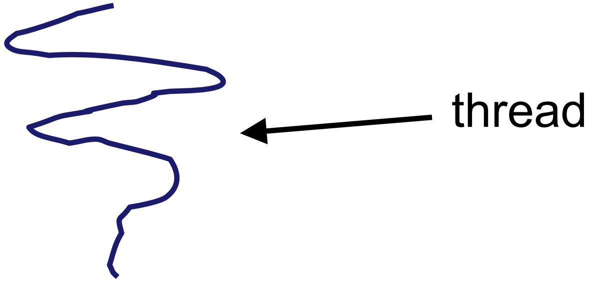
- In many cases
  - Everybody wants to run the same code
  - Everybody wants to access the same data
  - Everybody has the same privileges
  - Everybody uses the same resources (open files, network connections, etc.)
- But you’d like to have multiple hardware execution states:
  - an execution stack and stack pointer (SP)
    - traces state of procedure calls made
  - the program counter (PC), indicating the next instruction
  - a set of general-purpose processor registers and their values

How could we achieve this?

- Given the process abstraction as we know it:
  - fork several processes
  - cause each to *map* to the same physical memory to share data
    - see the **`shmget()`** system call for one way to do this
- This is really inefficient
  - space: PCB, page tables, etc.
  - time: creating OS structures, fork/copy address space, etc.

Can we do better?

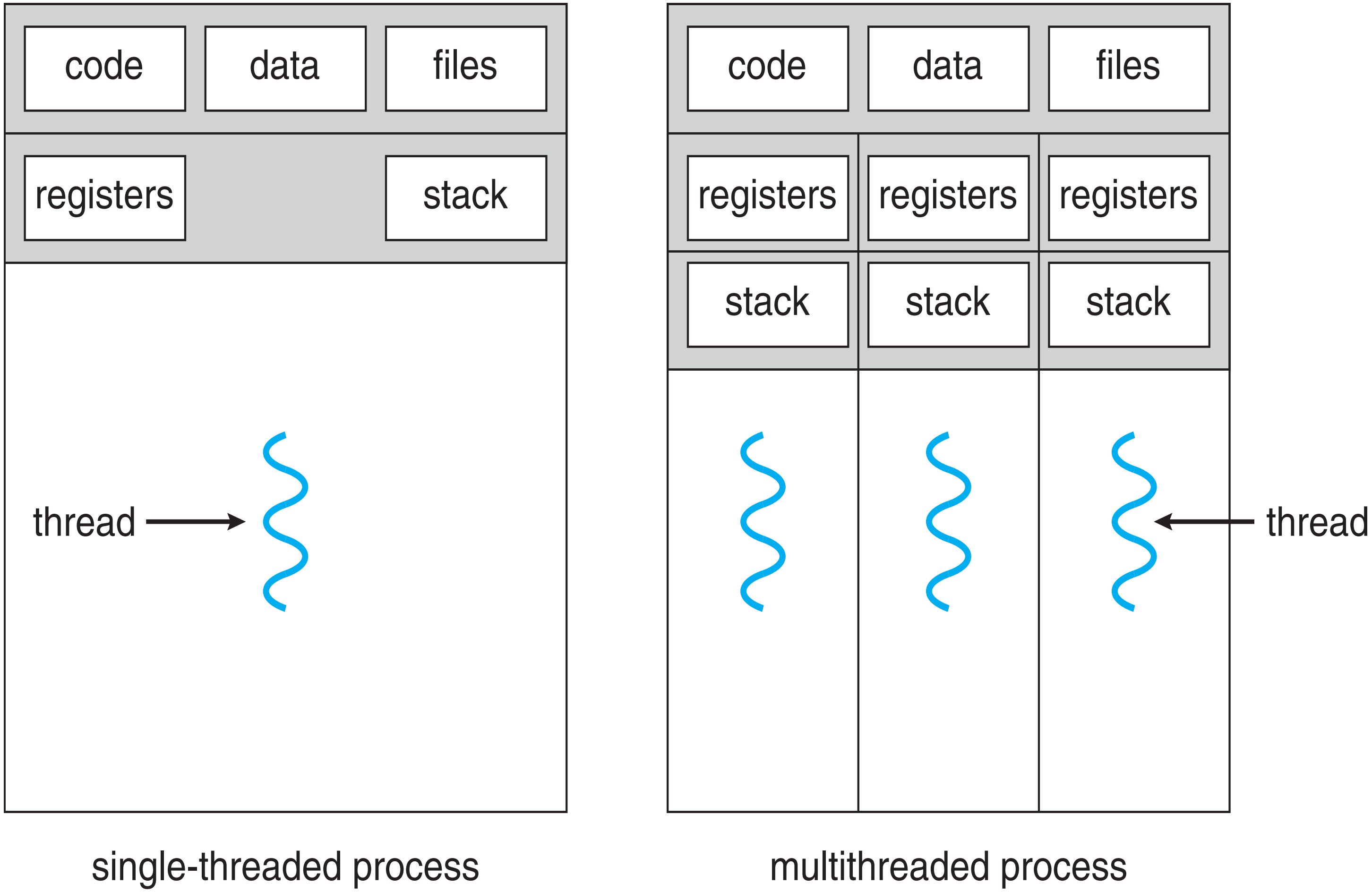
- Key idea:
  - separate the concept of a process (address space, OS resources)
  - … from that of a minimal "thread of control" (execution state: stack, stack pointer, program counter, registers)
- This execution state is usually called a thread, or sometimes, a lightweight process



Threads and processes

- Most modern OS's (Mach (Mac OS), Chorus, Windows, UNIX) therefore support two entities:
  - the process, which defines the address space and general process attributes (such as open files, etc.)
  - the thread, which defines a sequential execution stream within a process
- A thread is bound to a single process / address space
  - address spaces, however, can have multiple threads executing within them
  - sharing data between threads is cheap: all see the same address space
  - creating threads is cheap too!
- Threads become the unit of scheduling
  - processes / address spaces are just containers in which threads execute

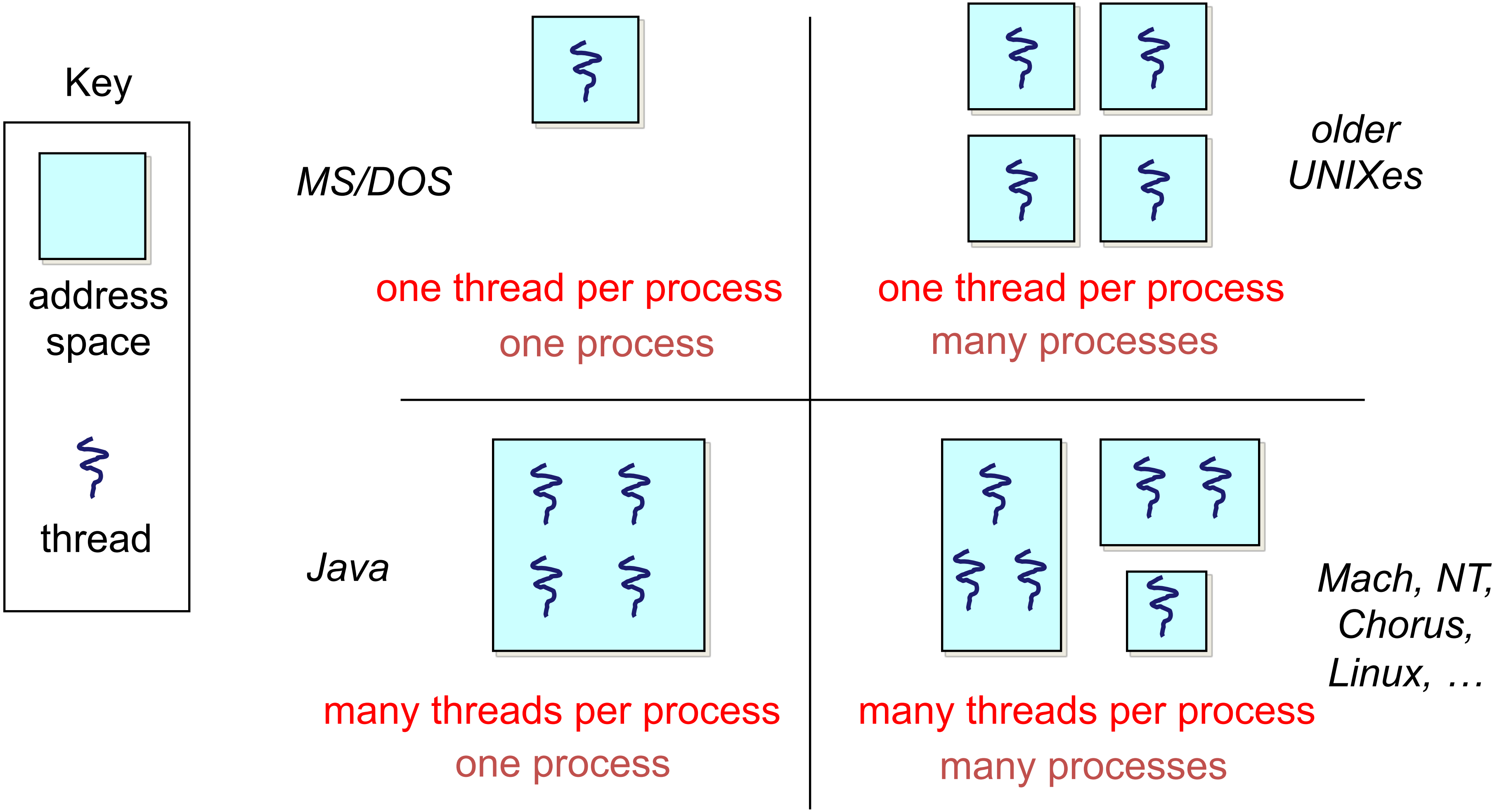
Single and Multithreaded Processes



Communication

- Threads are <u>concurrent executions sharing an address space</u> (and some OS resources)
- Address spaces provide isolation
  - If you can’t name it, you can’t read or write it
- Hence, communicating between processes is expensive
  - Must go through the OS to move data from one address space to another
- Because threads are in the same address space, communication is simple/cheap
  - Just update a shared variable!

The design space



Key

address space

thread

MS/DOS

one thread per process

one process

older UNIXes

one thread per process

many processes

Java

many threads per process

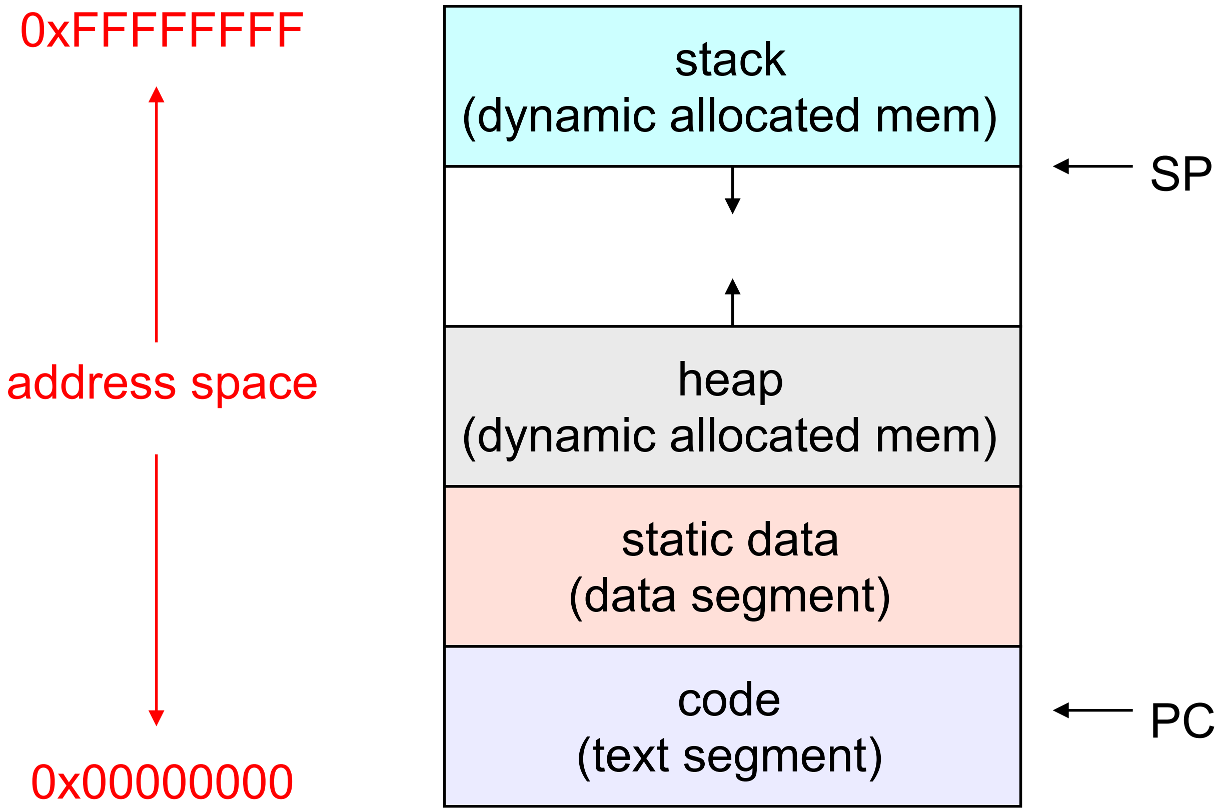
one process

Mach, NT, Chorus, Linux, …

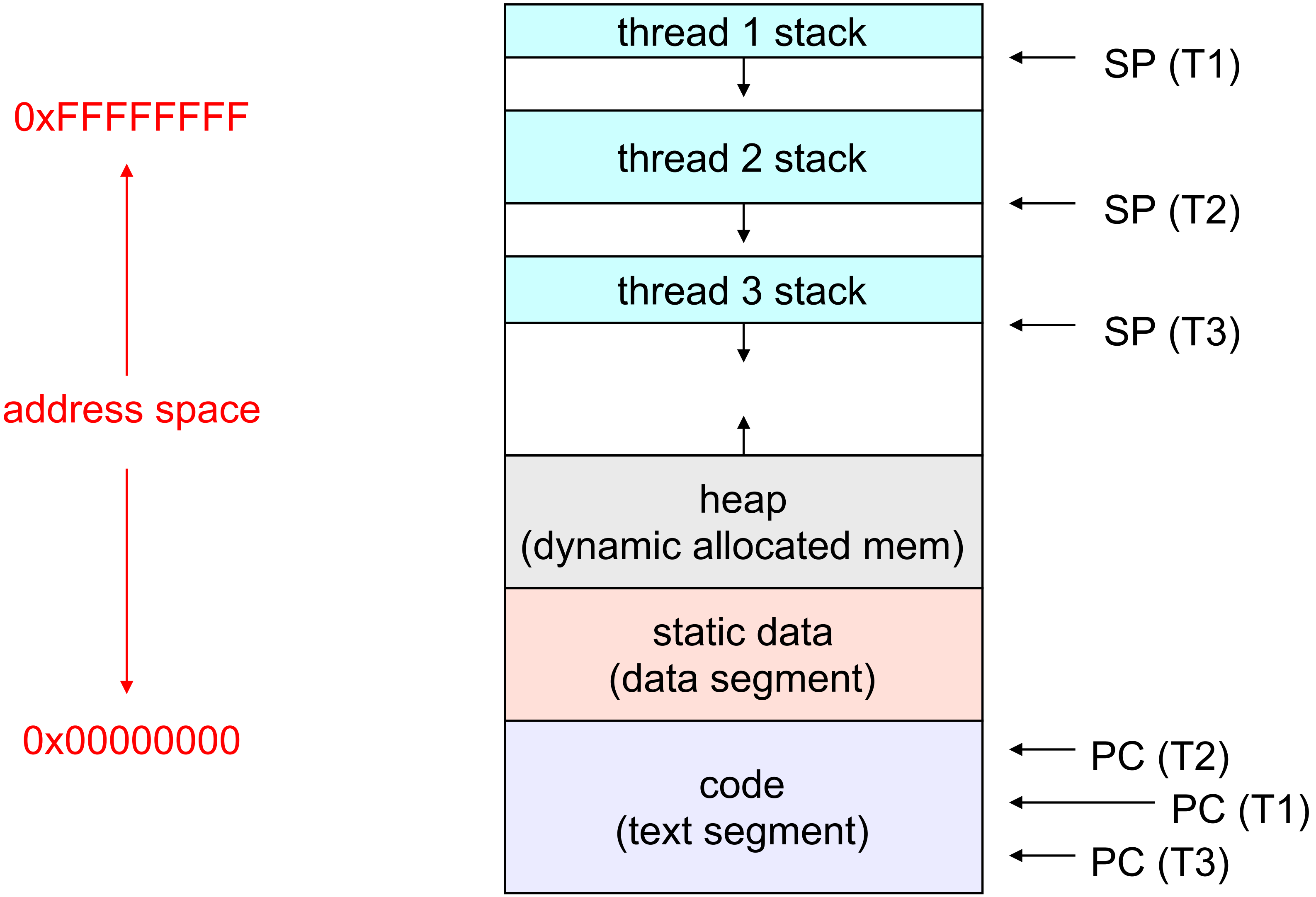
many threads per process

many processes

(old) Process address space



(new) Address space with threads



Process/thread separation

- Concurrency (multithreading) is useful for:
  - handling concurrent events (e.g., web servers and clients)
  - building parallel programs (e.g., matrix multiply, ray tracing)
  - improving program structure (the Java argument)
- Multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time
- Supporting multithreading – that is, separating the concept of a process (address space, files, etc.) from that of a minimal thread of control (execution state), is a big win
  - creating concurrency does not require creating new processes
  - "faster / better / cheaper"

Terminology

- Just a note that there's the potential for some confusion …
  - Old : "process" == "address space + OS resources + single thread"
  - New: "process" typically refers to an address space + system resources + all of its threads …
    - When we mean the "address space" we need to be explicit

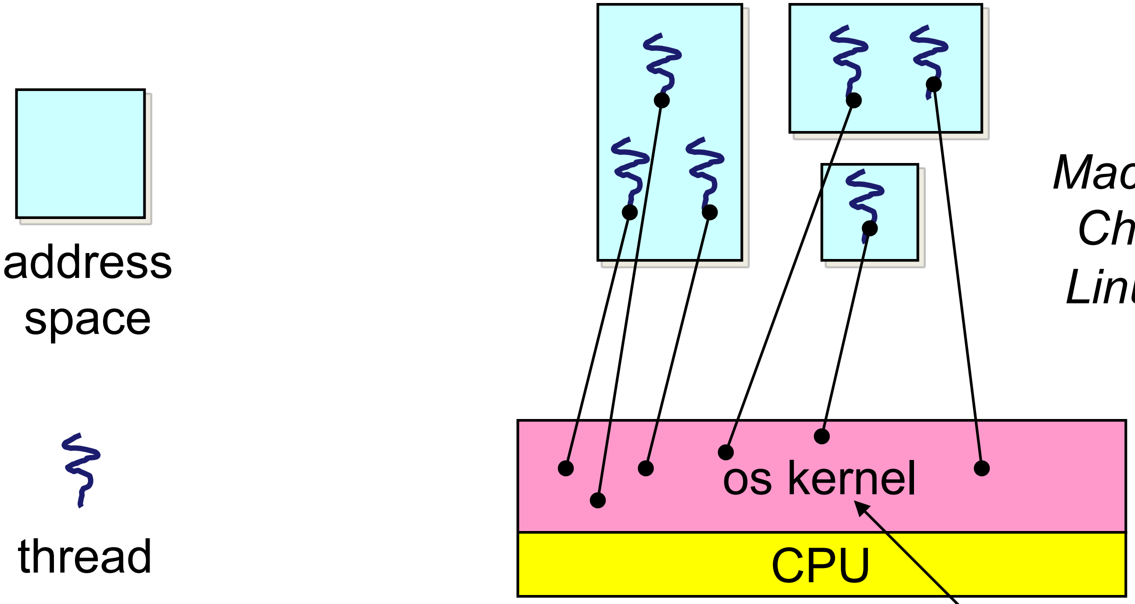
  "thread" refers to a single thread of control within a process / address space

- A bit like "kernel" and "operating system" …
  - Old: "kernel" == "operating system" and runs in "kernel mode"
  - New: "kernel" typically refers to the microkernel; lots of the operating system runs in user mode

Where do threads come from?

- Natural answer: the OS is responsible for creating/managing threads
  - For example, the kernel call to create a new thread would
    - allocate an execution stack within the process address space
    - create and initialize a Thread Control Block
      - stack pointer, program counter, register values
    - stick it on the ready queue
- We call these kernel threads
  - There is a "thread name space"
    - Thread id's (TID's)
    - TID's are integers

Kernel threads



(thread create, destroy, signal, wait, etc.)

Mach, NT, Chorus, Linux, …

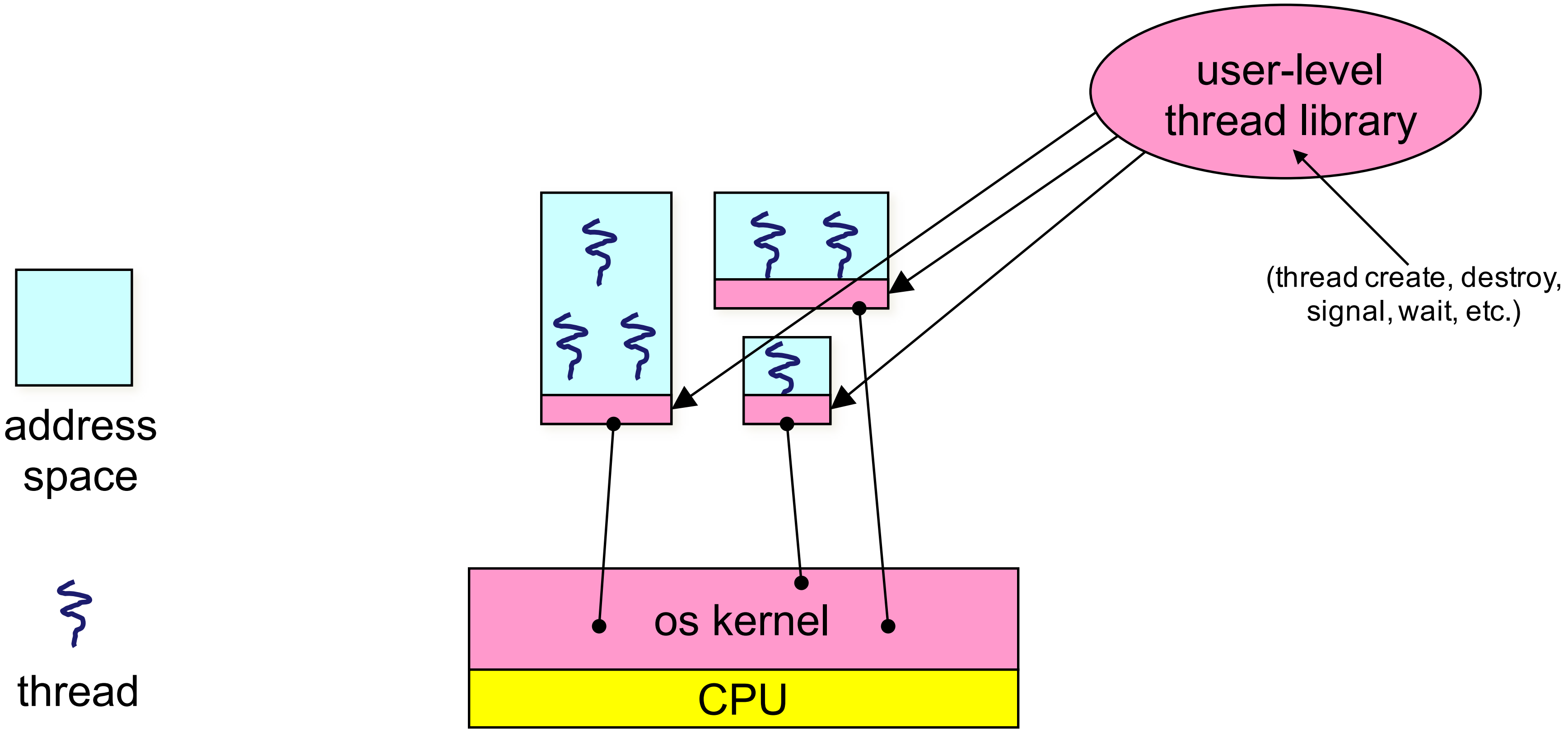
Kernel threads

- OS now manages threads *and* processes / address spaces
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - possible to overlap I/O and computation inside a process
- Kernel threads are cheaper than processes
  - less state to allocate and initialize
- But, they're still pretty expensive for fine-grained use
  - orders of magnitude more expensive than a procedure call
  - thread operations are all **system calls**
    - context switch
    - argument checks
  - must maintain kernel state for each thread

Cheaper alternative

- There is an alternative to kernel threads
- Threads can also be managed at the user level (within the process)
  - a library linked into the program manages the threads
    - the thread manager doesn't need to manipulate address spaces (which only the kernel can do)
    - threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
    - the thread package multiplexes user-level threads on top of kernel thread(s)
    - each kernel thread is treated as a "virtual processor"
  - we call these user-level threads

User-level threads



user-level thread library

(thread create, destroy, signal, wait, etc.)

address space

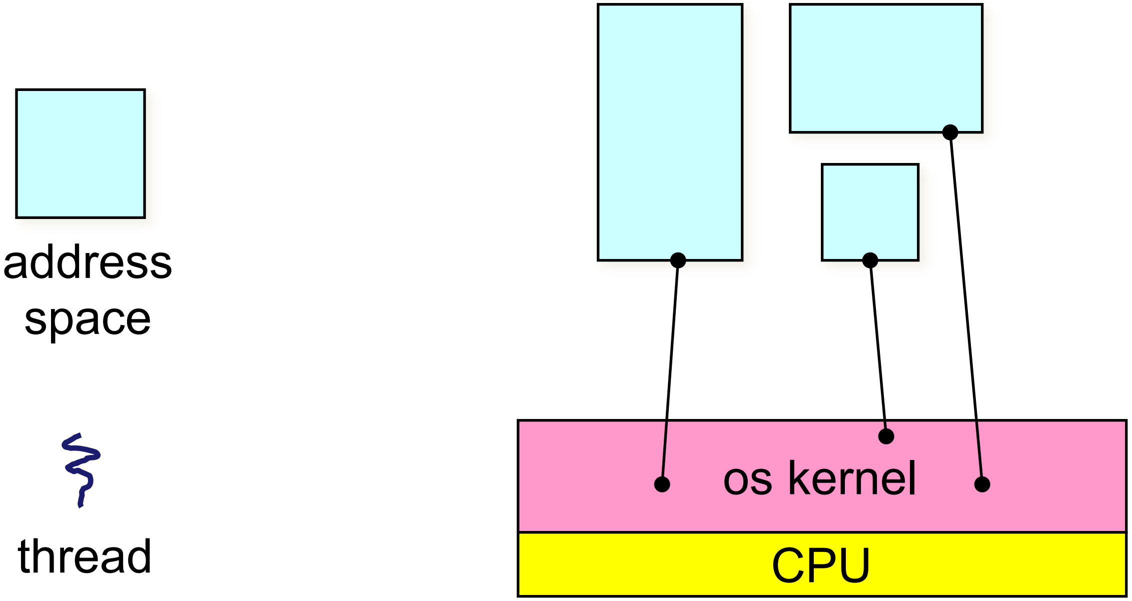
thread

os kernel

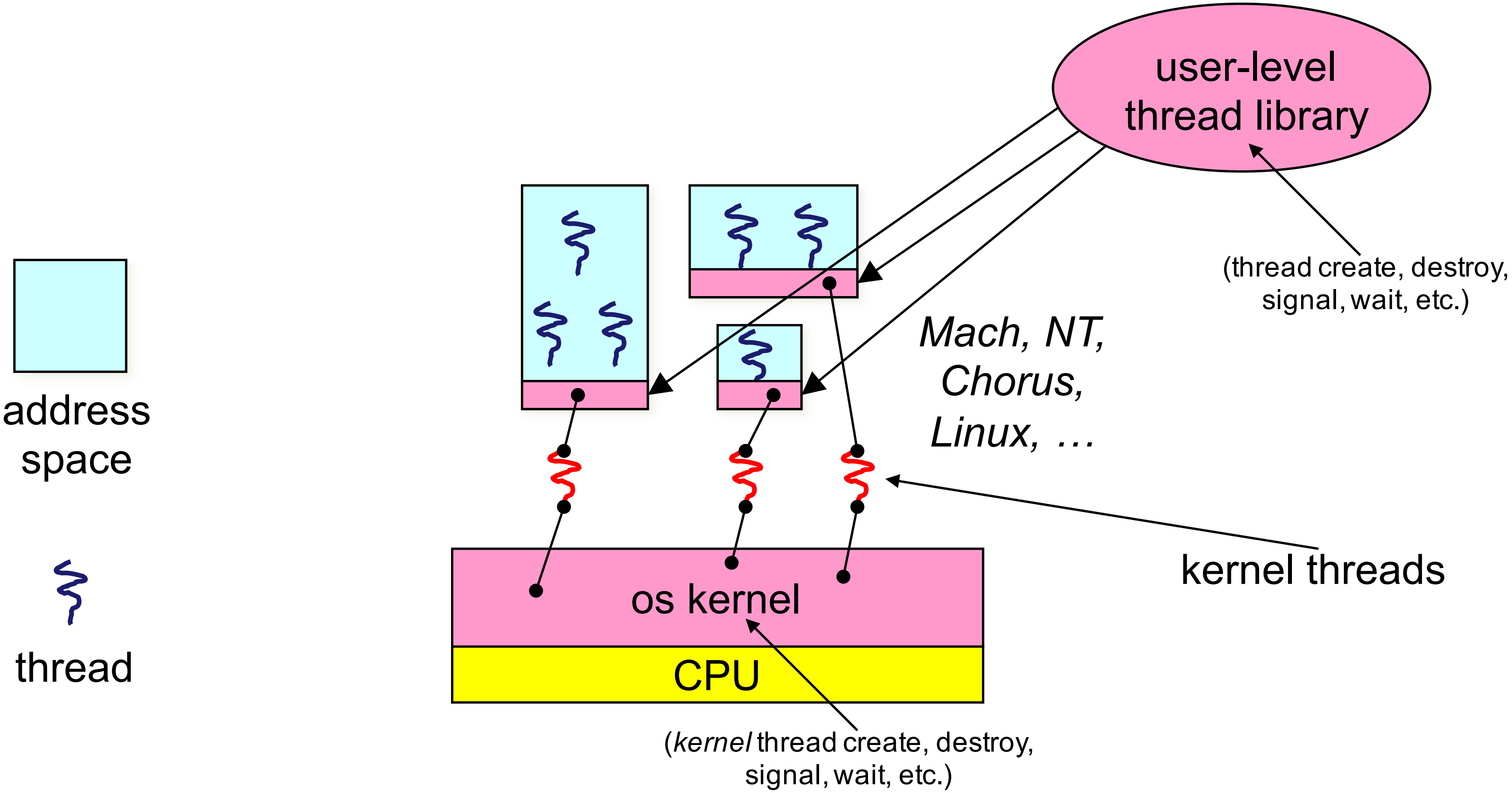
CPU

Now thread id is unique within the context of a process, not unique system-wide

User-level threads: what the kernel sees



User-level threads



address space

thread

user-level thread library

(thread create, destroy, signal, wait, etc.)

Mach, NT, Chorus, Linux, …

kernel threads

os kernel

CPU

(*kernel* thread create, destroy, signal, wait, etc.)

One problem: If a user-level thread blocked due to I/O, all other blocked

User-level threads

- User-level threads are small and fast
  - managed entirely by user-level library
    - E.g., pthreads (**`libpthreads.a`**)
  - each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  - creating a thread, switching between threads, and synchronizing threads are done via procedure calls
    - no kernel involvement is necessary!
- User-level thread operations can be 10-100x faster than kernel threads as a result

OLD Performance example

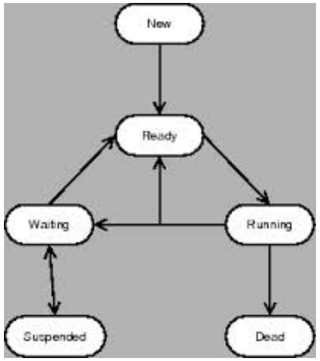
- On a 700MHz Pentium running Linux 2.2.16 (only the relative numbers matter; ignore the ancient CPU!):
  - Processes
    - **`fork/exit`**: 251 μs
  - Kernel threads
    - **`pthread_create()/pthread_join()`**: 94 μs **(2.5x faster)**

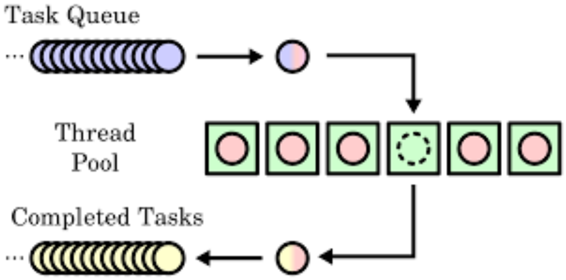
      Why?
  - User-level threads
    - **`pthread_create()/pthread_join`**: 4.5 μs **(another 20x faster)**

      Why?

User-level thread implementation

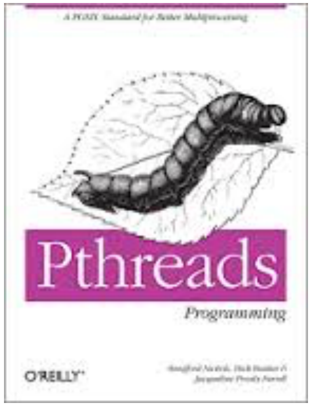
- The OS schedules the kernel thread
- The kernel thread executes user code, including the thread support library and its associated thread scheduler
- The thread scheduler determines when a user-level thread runs
  - it uses queues to keep track of what threads are doing: run, ready, wait
    - just like the OS and processes
    - but, implemented at user-level as a library



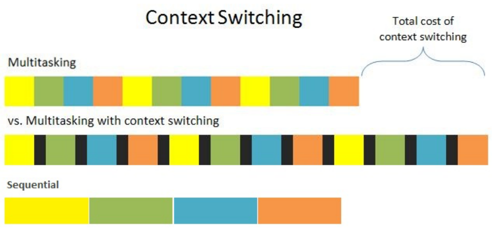


Thread interface

- This is taken from the POSIX `pthreads` API:
  - `rcode = pthread_create(&t, attributes, start_procedure)`
    - creates a new thread of control
    - new thread begins executing at start_procedure
  - `pthread_cond_wait(condition_variable, mutex)`
    - the calling thread blocks, sometimes called thread_block()
  - `pthread_signal(condition_variable)`
    - starts a thread waiting on the condition variable
  - `pthread_exit()`
    - terminates the calling thread
  - `pthread_join(t)`
    - waits for the named thread to terminate



Thread context switch



- Very simple for user-level threads:
  - save context of currently running thread
    - push CPU state onto thread stack
  - restore context of the next thread
    - pop CPU state from next thread's stack
  - return as the new thread
    - execution resumes at PC of next thread
  - Note: no changes to memory mapping required
- This is all done in assembly language
  - it works at the level of the procedure calling convention

How to keep a user-level thread from hogging the CPU?

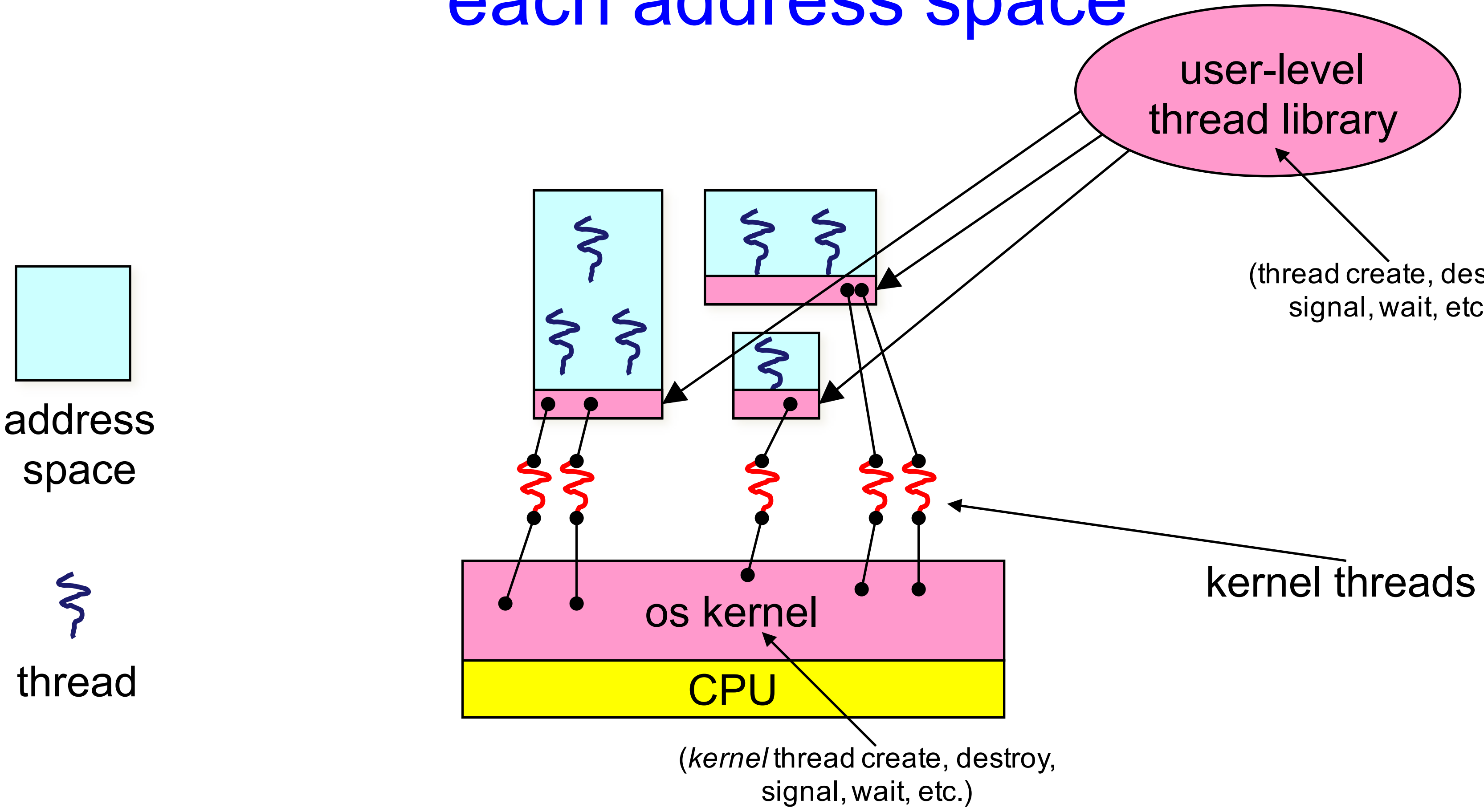
- Strategy 1: force everyone to cooperate
  - a thread willingly gives up the CPU by calling **`yield()`**
  - **`yield()`** calls into the scheduler, which context switches to another ready thread
  - what happens if a thread never calls **`yield()`**?

- Strategy 2: use preemption
  - scheduler requests that a timer interrupt be delivered by the OS periodically
    - usually delivered as a UNIX signal (`man signal`)
    - signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
  - at each timer interrupt, scheduler gains control and context switches as appropriate

What if a thread tries to do I/O?

- The kernel thread “powering” it is lost for the duration of the (synchronous) I/O operation!
  - The kernel thread blocks in the OS, as always
  - It maroons with it the state of the user-level thread
- Could have one kernel thread “powering” each user-level thread
  - “common case” operations (e.g., synchronization) would be quick
- Could have a limited-size “pool” of kernel threads “powering” all the user-level threads in the address space
  - the kernel will be scheduling these threads, obliviously to what’s going on at user-level

Multiple kernel threads "powering" each address space



Summary

- Multiple threads per address space
- Kernel threads are much more efficient than processes, but still expensive
  - all operations require a kernel call and parameter validation
- User-level threads are:
  - much cheaper and faster
  - great for common-case operations
    - creation, synchronization, destruction
  - can suffer in uncommon cases due to kernel obliviousness
    - I/O
    - preemption of a lock-holder