CSE 451: Operating Systems

Spring 2021

Module 5

Threads

John Zahorjan

Module overview

- Big picture: Achieving concurrency/parallelism
- Kernel threads
- User-level threads

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
  - A set of OS resources
    - open files, network connections, sound channels, …
  - 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
- Today: decompose …
  - address space
  - thread of control (stack, stack pointer, program counter, registers)
  - OS resources

The Big Picture

- Threads are about concurrency and parallelism
  - Parallelism: physically simultaneous operations for performance
  - Concurrency: logically (and possibly physically) simultaneous operations for convenience/simplicity/correctness
- One way to get concurrency and parallelism is to use multiple processes
  - The programs (code) of distinct processes are isolated from each other
  - *Why aren't processes all we need (for concurrency and parallelism)?*
- 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

Concurrency/Parallelism

- Imagine a web server, which might like to handle multiple requests concurrently
  - *While waiting for the credit card server to approve a purchase for one client, it could be retrieving the data requested by another client from disk, and assembling the response for a third client from cached information*
- Imagine a web client (browser), which might like to initiate multiple requests concurrently
  - *The CSE home page has dozens of "src= …" html references, each of which is going to involve a lot of sitting around! Wouldn't it be nice to be able to launch these requests concurrently?*
- Imagine a parallel program running on a multiprocessor, which might like to employ "physical concurrency"
  - *For example, multiplying two large matrices – split the output matrix into k regions and compute the entries in each region concurrently, using k processors*

Why Do We Think We Need Threads?

- If I want more than one thing running at a time, I already have a mechanism to achieve that
  - Processes

- What's right with processes?
  - Convenient abstraction of CPU, memory, I/O
  - Failure isolation
  - Potentially independent development by independent groups

- What's wrong with processes?
  - IPC (inter-process communication) is slow
    - Why?
  - Why does it matter?

Threads: Essential Idea

- You'd like to be able to preserve multiple, independent hardware execution states and rotate physical resources (cores) among them
  - Hey, just like processes!
  - Except this time we separate abstracting the CPU from abstracting memory
- Per-thread hardware execution state consists of:
  - 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
- That's a *thread*
  - A thread can be *running* (allocated a core)
  - A thread can be *runnable* (waiting for a core)
  - A thread can be *blocked* (waiting for something else)
- Up to now it was a process, because we assumed one thread per process
  - Now we're separating the idea of thread so that we can have more than one per process

Thread vs. Single-Threaded Process Abstraction

- Q: What is the "full execution context" of a thread? *(I made up that term, so don't use it outside this class!)*

  A: The things on the previous slide (stack, SP, PC, other registers) *plus* all the values the thread can access in virtual memory

- Both processes and threads are about multiplexing physical cores over "abstractions of cores"
  - To do that we save all CPU state
  - We don't save the "full execution context" though

- If a single threaded process loses its core, the contents of its address space "can't" be modified until it runs again

- If a thread loses its core, the address space contents can change before it runs again
  - Race conditions
  - Why don't we provide mechanism so that the address space contents can't change?

Multiple processes vs. One Process with Multiple Threads

- If I want concurrency/parallelism, why not just use multiple processes?
  - There was a time when that's what we did (that's not very important)
  - It's possible to do that and achieve whatever you want (that's a modestly important idea)
  - So why add another abstraction/implemented resource?

- Threads in a process share the virtual address space
  - Communication can be done by reading/writing main memory
  - Fast!
  - Race conditions!

- You'd like to have very fast communication among the separate execution contexts
  - Use shared memory
    - All threads run in the same virtual address space

Sanity Preserving Programming Convention

- Assume all globals are shared
  - It's global because we want access to it from everywhere/everyone
  - That's directly applicable in C/C++
  - What it means to be "global" in other languages is "something that is shared"...

- Assume locals are "private"
  - Stacks are (intended to be) private to a thread
  - That's directly applicable in C/C++/Java/most conventional languages

- There is no enforcement of this!
  - Well, the usual language semantics mean that
    - a fresh instance of a local variable is allocated when a thread enters a procedure
    - that instance is accessible only to that thread...
    - unless that thread explicitly makes it accessible to some other thread

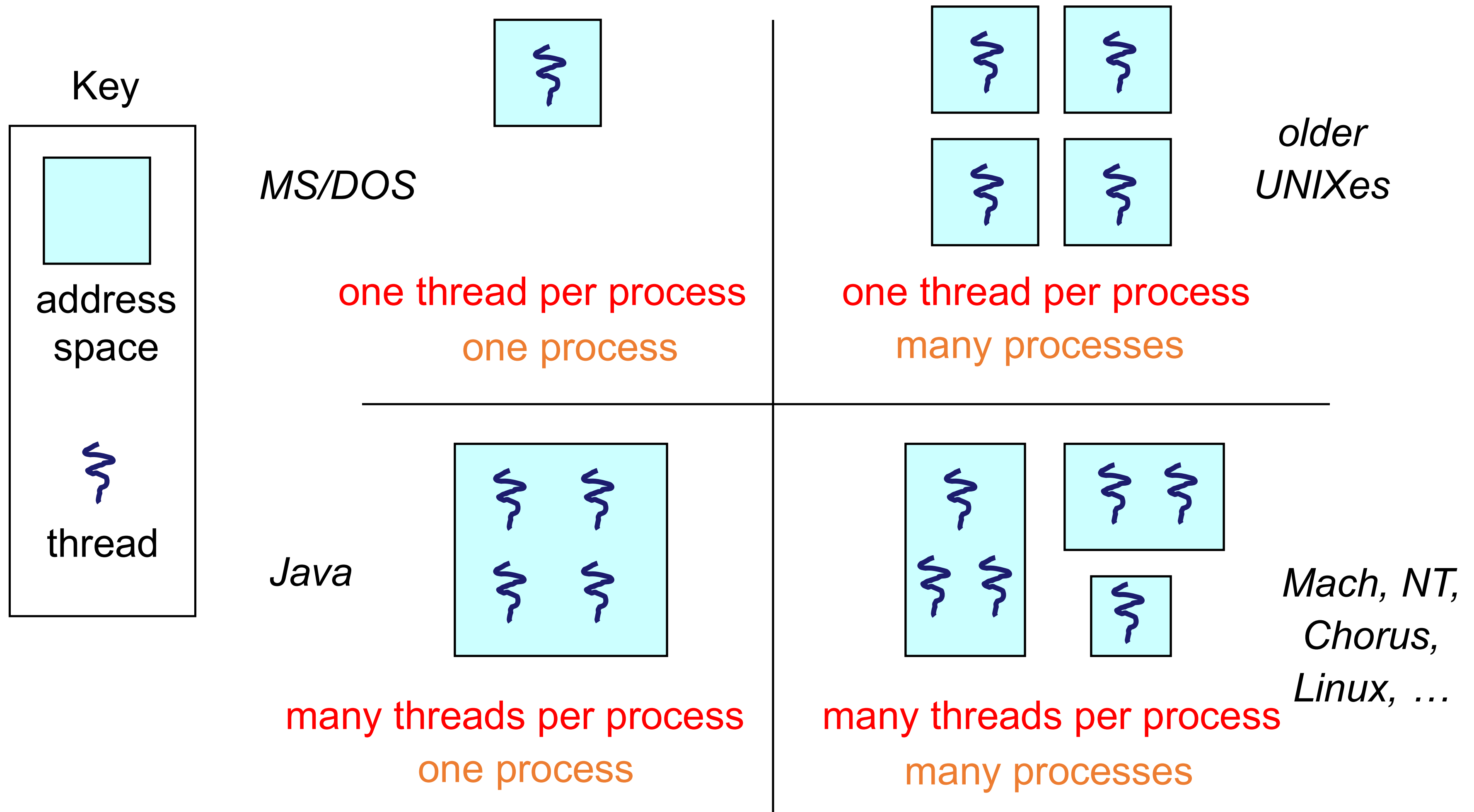
    - all memory is accessible to all threads, but at the language level there's no (good) name for the local variables of other threads, so there's no runtime enforcement making locals private to a thread

Threads and processes

- OS's 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

- Threads become the unit of scheduling
  - processes / address spaces are just containers in which threads execute
  - if you're scheduling (allocating) a core, you might be inclined to pick a thread in the same address space as the one that had been executing there
    - why? don't have to change address space; potential that main memory caches contain useful data (avoid misses)
  - but you can't always make that decision
    - why?

The design space



OS (Linux) fork()

fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following points:

<deleted by me>

Note the following further points:

- The child process is created with a single thread—the one that called fork(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of pthread_atfork(3) may be helpful for dealing with problems that this can cause.
- The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see open(2)) as the corresponding file descriptor in the parent. This means that the two descriptors share open file status flags, current file offset, and signal-driven I/O attributes (see the description of F_SETOWN and F_SETSIG in fcntl(2)).
- The child inherits copies of the parent's set of open message queue descriptors (see mq_overview(7)). Each descriptor in the child refers to the same open message queue description as the corresponding descriptor in the parent. This means that the two descriptors share the same flags (mq_flags).
- The child inherits copies of the parent's set of open directory streams (see opendir(3)). POSIX.1-2001 says that the corresponding directory streams in the parent and child may share the directory stream positioning; on Linux/glibc they do not.

OS (Linux) clone()

clone() creates a new process, in a manner similar to fork(2).

This page describes both the glibc clone() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.

Unlike fork(2), clone() allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of CLONE_PARENT below.)

The main use of clone() is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space.

When the child process is created with clone(), it executes the function fn(arg). (This differs from fork(2), where execution continues in the child from the point of the fork(2) call.) The fn argument is a pointer to a function that is called by the child process at the beginning of its execution. The arg argument is passed to the fn function.

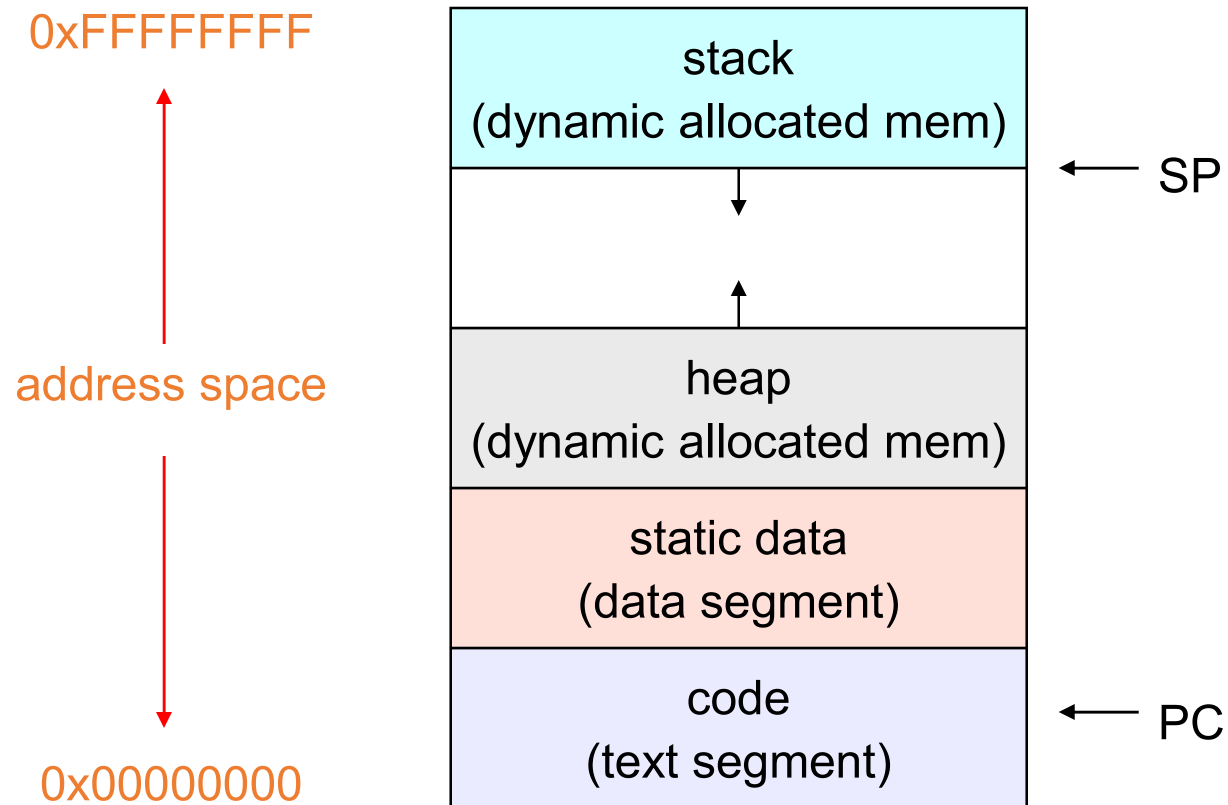
When the fn(arg) function application returns, the child process terminates. The integer returned by fn is the exit code for the child process. The child process may also terminate explicitly by calling exit(2) or after receiving a fatal signal.

The child_stack argument specifies the location of the stack used by the child process. Since the child and calling process may share memory, it is not possible for the child process to execute in the same stack as the calling process. The calling process must therefore set up memory space for the child stack and pass a pointer to this space to clone(). Stacks grow downward on all processors that run Linux (except the HP PA processors), so child_stack usually points to the topmost address of the memory space set up for the child

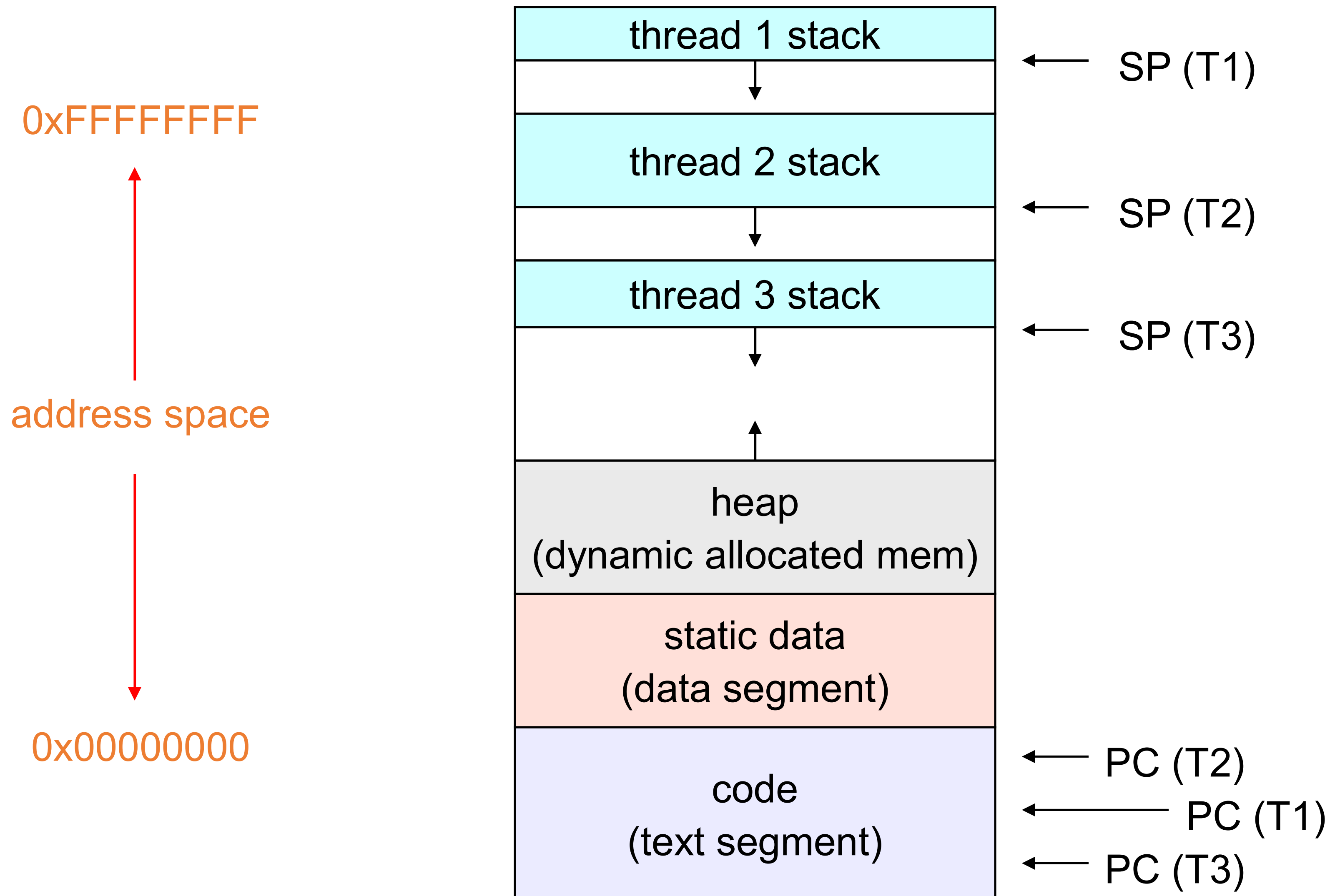
Portable Application-level Thread Interface

- This is taken from the POSIX `pthreads` API:
  - `pthread_create(&t, attributes, start_procedure)`
    - creates a new thread of control
    - new thread begins executing at start_procedure
  - `pthread_exit()`
    - terminates the calling thread
  - `pthread_wait(t)`
    - waits for the named thread to terminate
  - `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
  - …

(old) Process virtual address space



(new) Virtual address space with threads



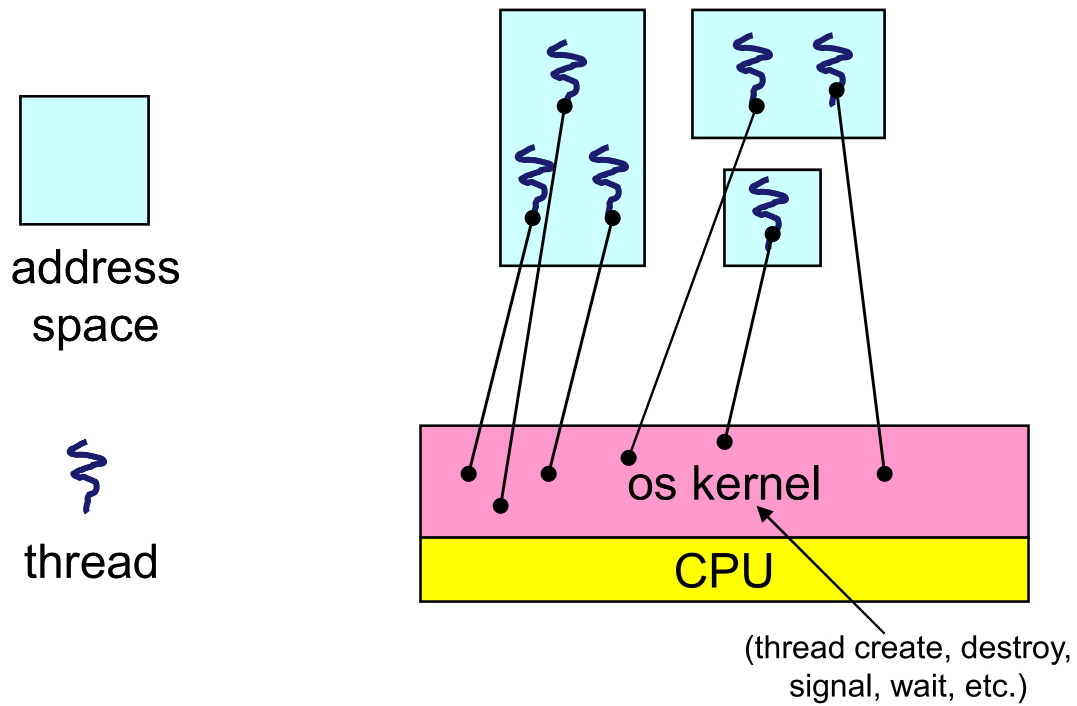
Value of process/thread separation

- 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 single core processor
  - even though only one thread can run at a time
  - Why?
- 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”

Where do threads come from? Kernel threads

- 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
      - Where in the address space and how big might be arguments to the creation call
    - 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 (surprise!)
    - What's the scope?
- Why "must" the kernel be responsible for creating/managing threads?

Kernel threads



Kernel threads

- OS now manages threads *and* processes / address spaces
- All thread operations are implemented in the kernel
  - create, destroy, synchronize
- 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 another thread (from that process or others) on that core
    - concurrency
  - possible to overlap I/O and computation, or computation and computation, inside a single process
    - that is, under programmer control
    - parallelism

Kernel Threads / Granularity

- Granularity refers to the amount of work that is done per (thread creation) operation
  - overhead = (cost of creation) / (cost of creation + cost of useful work)

- Thread operations are cheaper than process operations
  - less state to allocate and initialize

- But they're still pretty expensive for fine-grained use
  - creation is orders of magnitude more expensive than a procedure call
  - all thread operations are system calls
    - context switch
    - argument checks

- Is there a way to reduce the cost of thread operations?
  - If there were, threads would be useful even for more fine-grained applications

Where do threads come from? User-Level Threads

- There is an alternative to implementing threads in the kernel
  - Note that thread management isn't doing anything that feels privileged
  - Well, maybe switching threads on a core feels privileged...
    - Re-allocating a core among processes IS privileged
    - Re-allocating a core among threads in a single process isn't

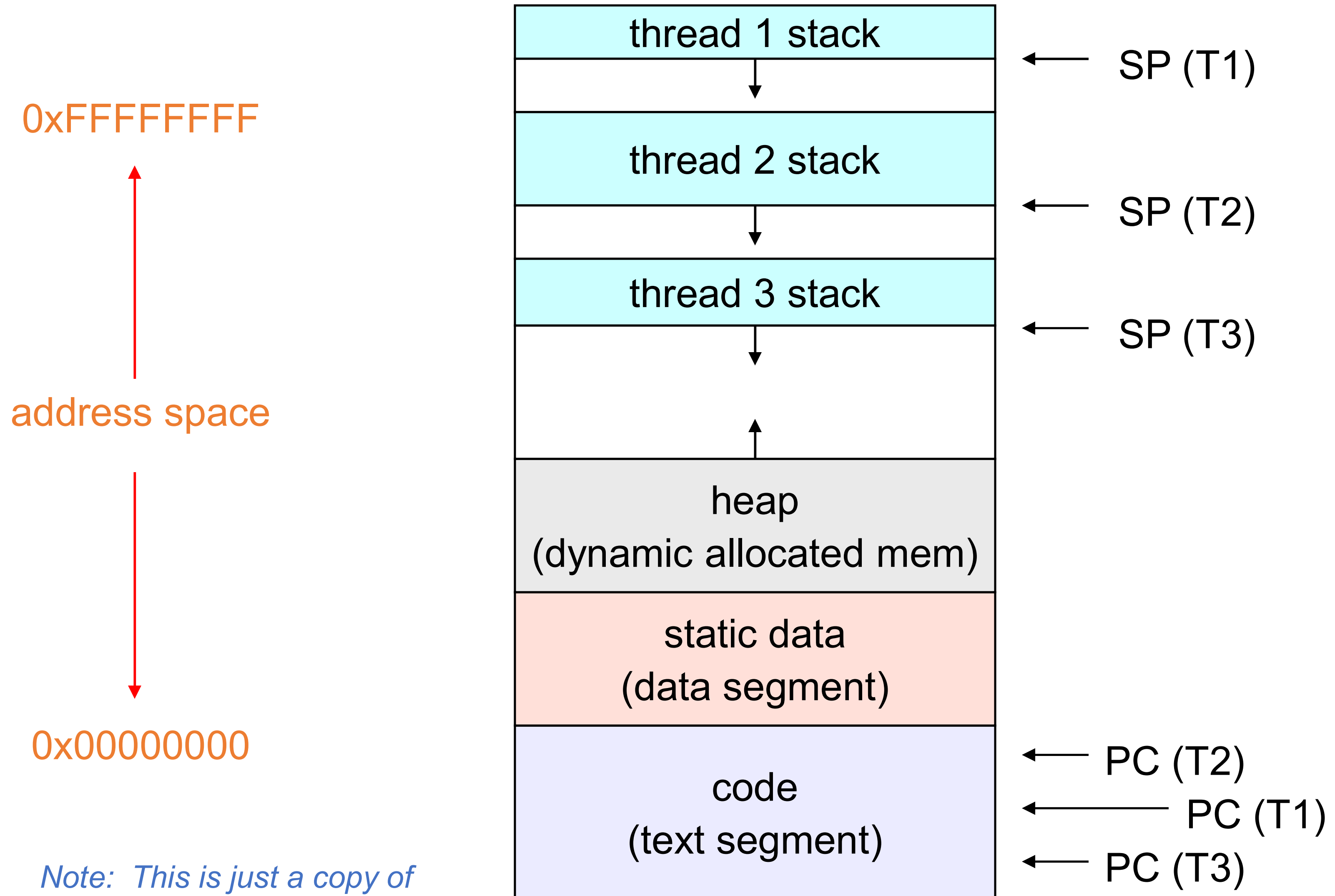
- Threads can be managed at the user level (that is, entirely from within the process)
  - a library linked into the program manages the threads
    - because threads share the same address space, the thread manager doesn't need to manipulate address spaces (which only the kernel can do)
    - threads differ (roughly) only in hardware contexts (PC, SP, registers)
      - those things 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 Implementation

- Of course we can allocate a stack and space to save registers in regular old code
  - It’s just memory, nothing special
- It turns out we can switch threads in regular old (assembler) code as well
  - Just need to save the current PC, SP, and general purpose registers to save state of the currently running thread
  - Then need to restore the PC, SP, and general purpose registers of the thread we want to resume
- *(We can’t reallocate cores, but we can switch the thread that is running on a core we’ve been allocated)*

Address space with user-level threads



Note: This is just a copy of the slide for kernel threads…

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 by assembly language
  - it works at the level of the procedure calling convention
    - thus, it cannot be implemented using procedure calls

Thread switch: the xk implementation

- xk implements thread switch in the kernel
  - So it isn't user-level threads
  - But when you see the code it will be clear there's nothing privileged about it
- Thread switching is done by an explicit procedure call
  - If at user-level, you could imagine that the application code calls yield(), which performs the context switch
    - User-level threads can also support asynchronous switching
    - Set a timer, receive a signal when it goes off, have the handler call yield()
- In xk, the calls look like this:
  - `swtch(&mycpu()->scheduler, p->context);`
  - The arguments are pointers to memory in the process control block used to save registers

swtch.S

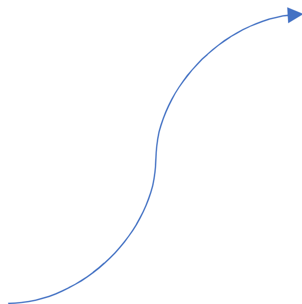
```
# arg0 -- context (in struct process) for the old process
# arg1 -- context (in struct process) for the new process

swtch:
  push %rbp
  push %rbx
  push %r11
  push %r12
  push %r13
  push %r14
  push %r15

  mov %rsp, (%rdi)
  mov %rsi, %rsp

  pop %r15
  pop %r14
  pop %r13
  pop %r12
  pop %r11
  pop %rbx
  pop %rbp

  ret
```



swtch(&mycontext, &nextcontext)

```
swtch:
  push %rbp
  push %rbx
  push %r11
  push %r12
  push %r13
  push %r14
  push %r15

  mov %rsp, (%rdi)
  mov %rsi, %rsp

  pop %r15
  pop %r14
  pop %r13
  pop %r12
  pop %r11
  pop %rbx
  pop %rbp

  ret
```

---

swtch(&somecontext, &mycontext)

```
swtch:
  push %rbp
  push %rbx
  push %r11
  push %r12
  push %r13
  push %r14
  push %r15

  mov %rsp, (%rdi)
  mov %rsi, %rsp

  pop %r15
  pop %r14
  pop %r13
  pop %r12
  pop %r11
  pop %rbx
  pop %rbp

  ret
```

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, blocking a thread, and terminating a thread are done via procedure calls
  - no kernel involvement is necessary!

- because blocking is done at user level, synchronization primitives can also be implemented at user level
  - E.g., join() [thread wait], lock(), unlock(), etc.

- user-level thread operations can be 10-100x faster than kernel threads as a result

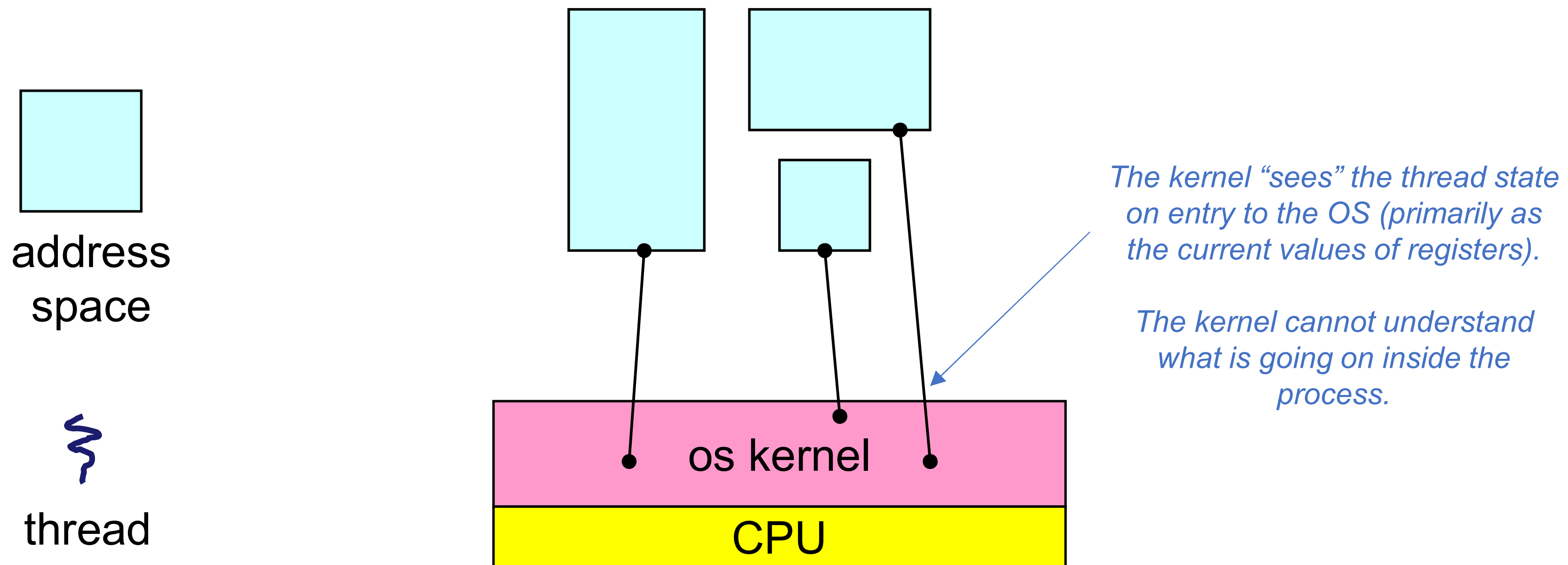
Performance example

- 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 than process fork)**
  - User-level threads
    - **`pthread_create()/pthread_join`**: 4.5 μs **(another 20x faster than kernel threads)**

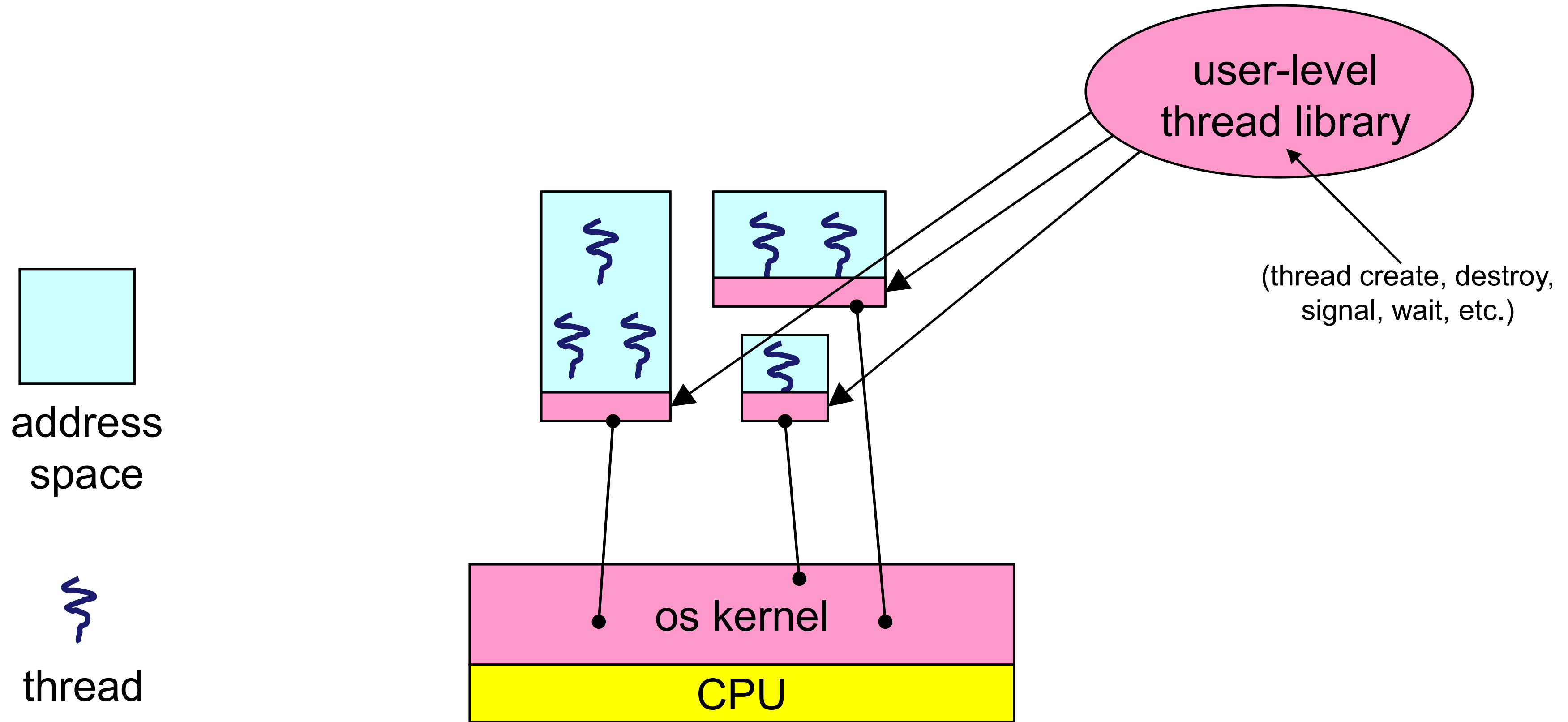
Why?

Why?

User-level threads: what the kernel sees



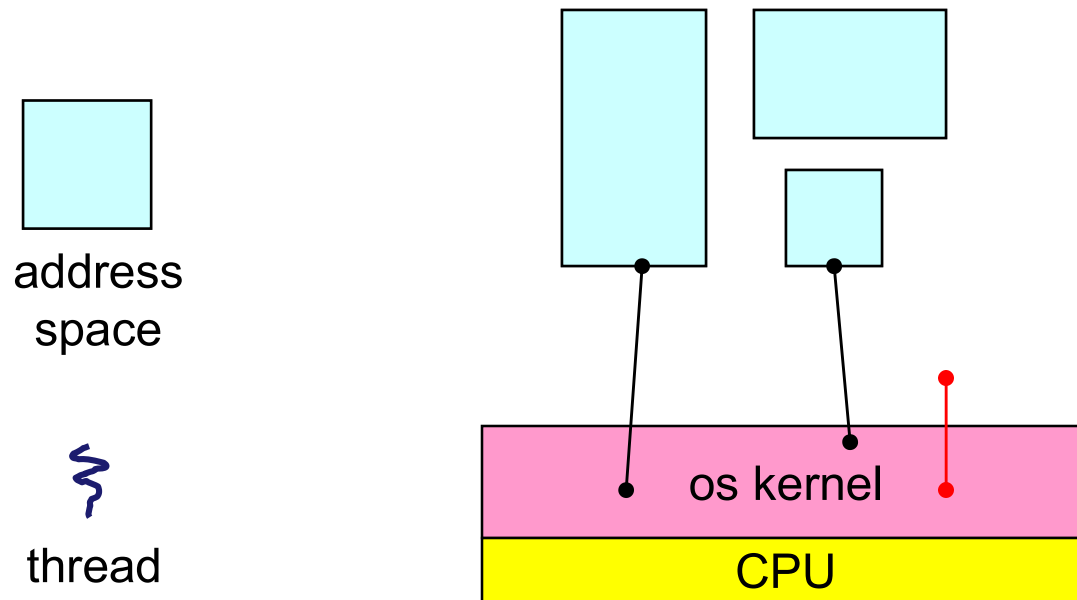
User-level threads



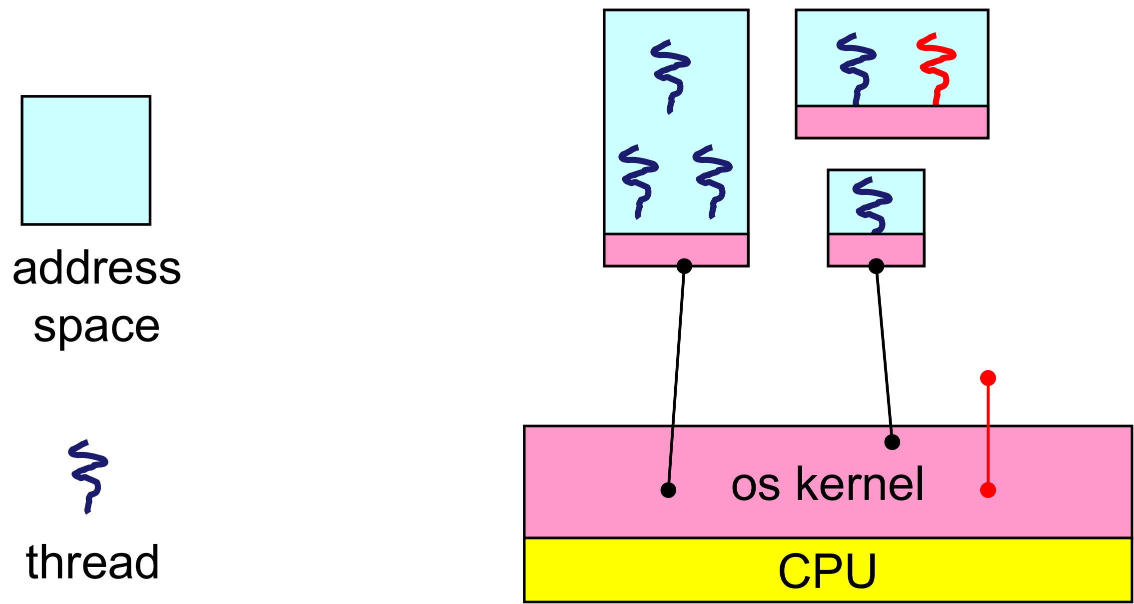
User-level thread scheduling

- The OS schedules the kernel thread
  - Takes a core and loads the kernel state onto it, then lets it run
- The kernel thread executes user code, just like always
  - except that now this includes the thread support library and its associated thread scheduler
  - that code creates a thread abstraction at user level, using data structures defined at user level
- The user-level 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 with kernel threads
- The kernel thread is sometimes running in the context of user-level thread A and sometimes in the context of user-level thread B, etc.
  - What does that mean?

Problem: What if a user-level thread tries to do IO?



What if a user-level thread tries to do IO?



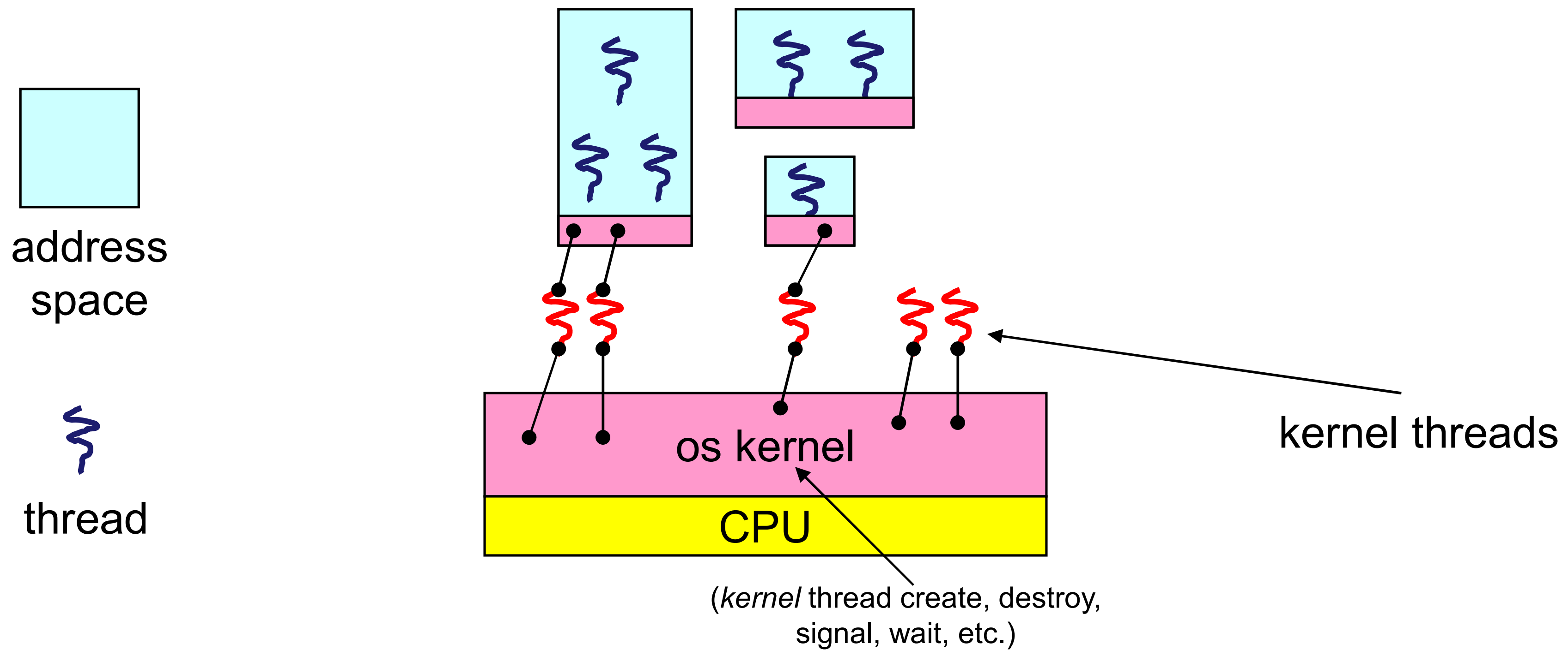
What if a user-level thread does I/O?

- 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
    - Why can’t the OS “un-maroon” the user-level thread?
    - Why can’t the user-level thread package un-maroon the user-level thread?
- Why is this a problem?

How to deal with kernel thread blocking

- Could have one kernel thread "powering" each user-level thread
  - when one user-level thread blocks, all other user-level threads could potentially continue to run
    - depends on how many cores are allocated to the process
  - "common case" operations (e.g., synchronization) could be quick
  - creation/termination wouldn't be quick
- But, if a kernel thread is blocked (e.g., loses its core) while the user-level thread its running holds a user-level lock, synchronization can be very slow
  - lock can't be freed until that kernel thread is re-scheduled
- Could have a limited-size "pool" of kernel threads "powering" all the user-level threads in the address space
  - How might this help?
  - the kernel will be scheduling these threads, obliviously to what's going on at user-level

Multiple kernel threads "powering" each address space



What is the problem?

- Basically, the issue is that policy decisions are being made at the wrong level
  - at a level that doesn't have enough information
- The OS is in charge of the policy of how many cores to give to each process
- The OS is in charge of which kernel threads within a process should be running (have a core), given the number of threads and number of available cores
- The thread library should be in charge of which user-level threads should be running given the number of kernel threads the process has
  - Note that whether a kernel thread has a core or not is not easily visible to the thread library!
- So, kernel policy is effectively trumping user-level policy

Addressing this problem

- Effective coordination of kernel decisions and user-level threads requires OS-to-user-level communication
  - OS notifies user-level that it has suspended a kernel thread, or re-started one
    - mechanism
  - Code at the user-level decides what effect that should have on user-level thread scheduling decisions
    - policy
- This is called "scheduler activations"
    - a research paper from UW
- Each process can create one or more kernel threads
  - process is given responsibility for mapping user-level threads onto kernel threads
  - kernel promises to notify user-level before it suspends or destroys a kernel thread

Scheduler Activations

- OS events of interest to the user-level thread library are
  - I've allocated another core to your process
  - I've taken a core away from your process
- When allocating a new core
  - OS does an upcall to user-level thread library handler routine
  - user-level thread library looks at its list of threads waiting for a core, picks one, and switches to it
- When taking away a core (that is running kernel thread A)
  - OS does an upcall to user-level thread library
  - It uses some other kernel-level thread, say B, that is already allocated to that process to send the upcall
  - The arguments to the upcall mean "I've blocked A and B. Here are the contents of their registers when I blocked them. Now you do whatever you think is best.'

Summary

- Multiple threads per address space are important both for convenience in writing code and for performance
  - → kernel threads
- Kernel threads are much more efficient than processes, but they're still not cheap
  - → user-level threads
- User-level threads can suffer in some cases due to kernel obliviousness
  - I/O
  - preemption of a lock-holder
- Scheduler activations are an answer
  - improved mechanism/policy separation