Computer Systems

CSE 410 Spring 2012

21 – Processes & Threads

Processes – Programmer's View

- **Definition: A *process* is an instance of a running program**
  - One of the most important ideas in computer science
  - Not the same as "program" or "processor"

- **Process provides each program with *two key abstractions*:**
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory

- **But what's really going on underneath?**

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 (static data, heap data, stack)
  - CPU state, consisting of
    - The program counter (PC), indicating the next instruction
    - The stack pointer
    - Other general purpose register values
  - A set of OS resources
    - open files, network connections, sound channels, …
- **In other words, it's all the stuff you need to run the program**
  - or to re-start it, if it's interrupted at some point

The OS's process namespace

- **(Like most things, the particulars depend on the specific OS, but the principles are general)**
- **The name for a process is called a process ID (PID)**
    - An integer
- **The PID namespace is global to the system**
    - Only one process at a time has a particular PID
- **Operations that create processes return a PID**
    - E.g., fork()
- **Operations on processes take PIDs as an argument**
    - E.g., kill(), wait(), nice()

Representation of processes by the OS

- **The OS maintains a data structure to keep track of a process's state**
  - Called the process control block (PCB) or process descriptor
  - Identified by the PID
- **OS keeps all of a process's execution state in (or linked from) the PCB when the process isn't running**
  - PC, SP, registers, etc.
  - when a process is unscheduled, the state is transferred out of the hardware into the PCB
  - (when a process is running, its state is spread between the PCB and the CPU)
- **Note: It's natural to think that there must be some esoteric techniques being used**
  - fancy data structures that you'd never think of yourself

Wrong! It's pretty much just what you'd think of!

The PCB

- **The PCB is a data structure with many, many fields:**
  - process ID (PID)
  - parent process ID
  - execution state
  - program counter, stack pointer, registers
  - address space info
  - UNIX user id, group id
  - scheduling priority
  - accounting info
  - pointers for state queues
- **In Linux:**
  - defined in `task_struct` (`include/linux/sched.h`)
  - over 95 fields!!!

PCBs and CPU state (1)

- **When a process is running, its CPU state is inside the CPU**
  - PC, SP, registers
  - CPU contains current values
- **When the OS gets control because of a …**
  - Trap: Program executes a syscall
  - Exception: Program does something unexpected (e.g., page fault)
  - Interrupt: A hardware device requests service

the OS saves the CPU state of the running process in that process's PCB

PCBs and CPU state (2)

- **When the OS returns the process to the running state, it loads the hardware registers with values from that process's PCB – general purpose registers, stack pointer, instruction pointer**
- **The act of switching the CPU from one process to another is called a context switch**
  - systems may do 100s or 1000s of switches/sec.
  - takes a few microseconds on today's hardware
- **Choosing which process to run next is called scheduling**

The OS kernel is not a process

- It's just a block of code!
- (In a microkernel OS, many things that you normally think of as the operating system execute as user-mode processes. But the OS kernel is just a block of code.)

Process ID

Pointer to parent

List of children

Process state

Pointer to address space descriptor

Program counter
stack pointer
(all) register values

uid (user id)
gid (group id)
euid (effective user id)

Open file list

Scheduling priority

Accounting info

Pointers for state queues

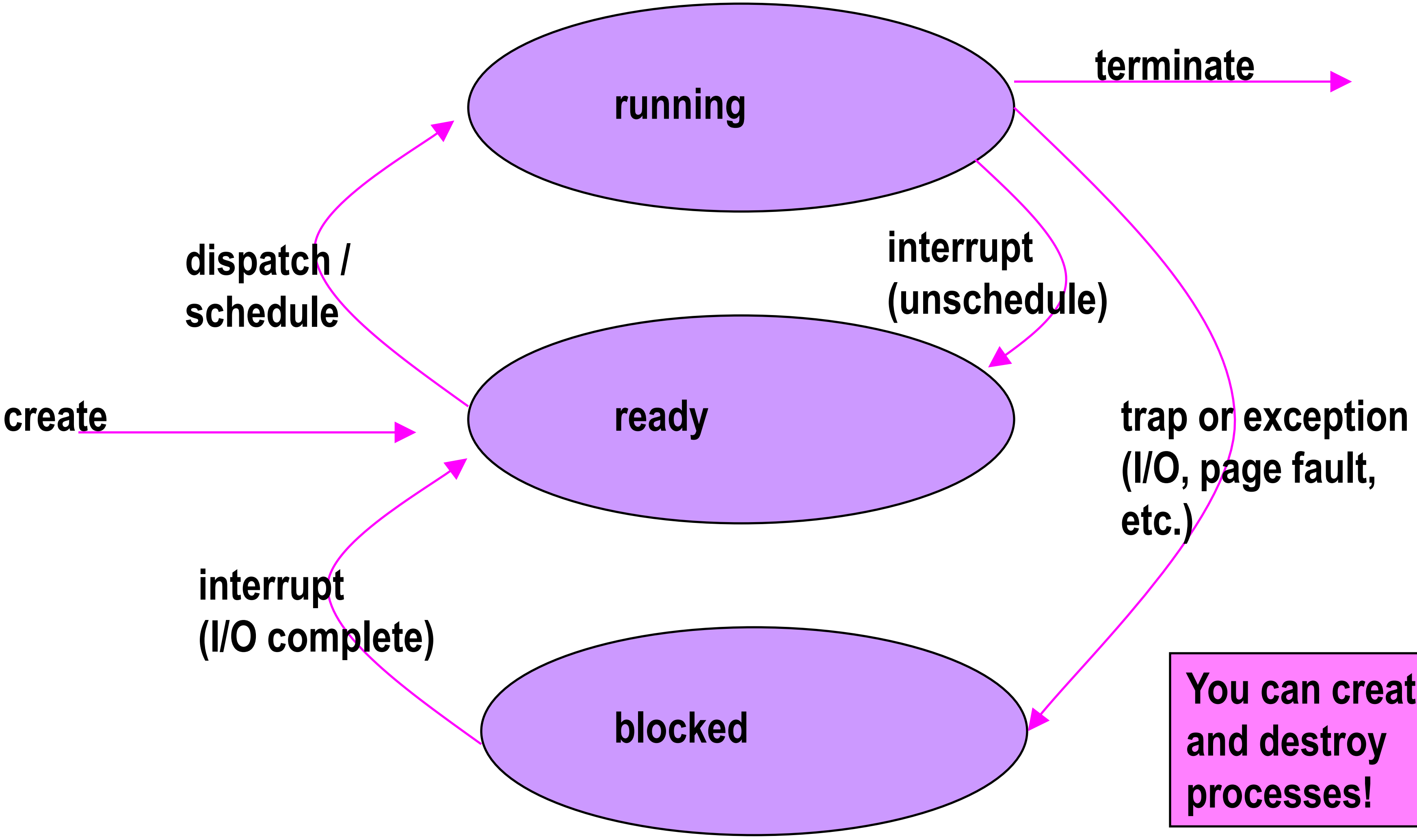
Exit (“return”) code value

This is (a simplification of) what each of those PCBs looks like inside!

Process execution states

- **Each process has an execution state, which indicates what it's currently doing**
  - ready: waiting to be assigned to a CPU
    - could run, but another process has the CPU
  - running: executing on a CPU
    - it's the process that currently controls the CPU
  - waiting (aka "blocked"): waiting for an event, e.g., I/O completion, or a message from (or the completion of) another process
    - cannot make progress until the event happens
- **As a process executes, it moves from state to state**
  - UNIX: run `ps`, STAT column shows current state
  - which state is a process in most of the time?

Process states and state transitions

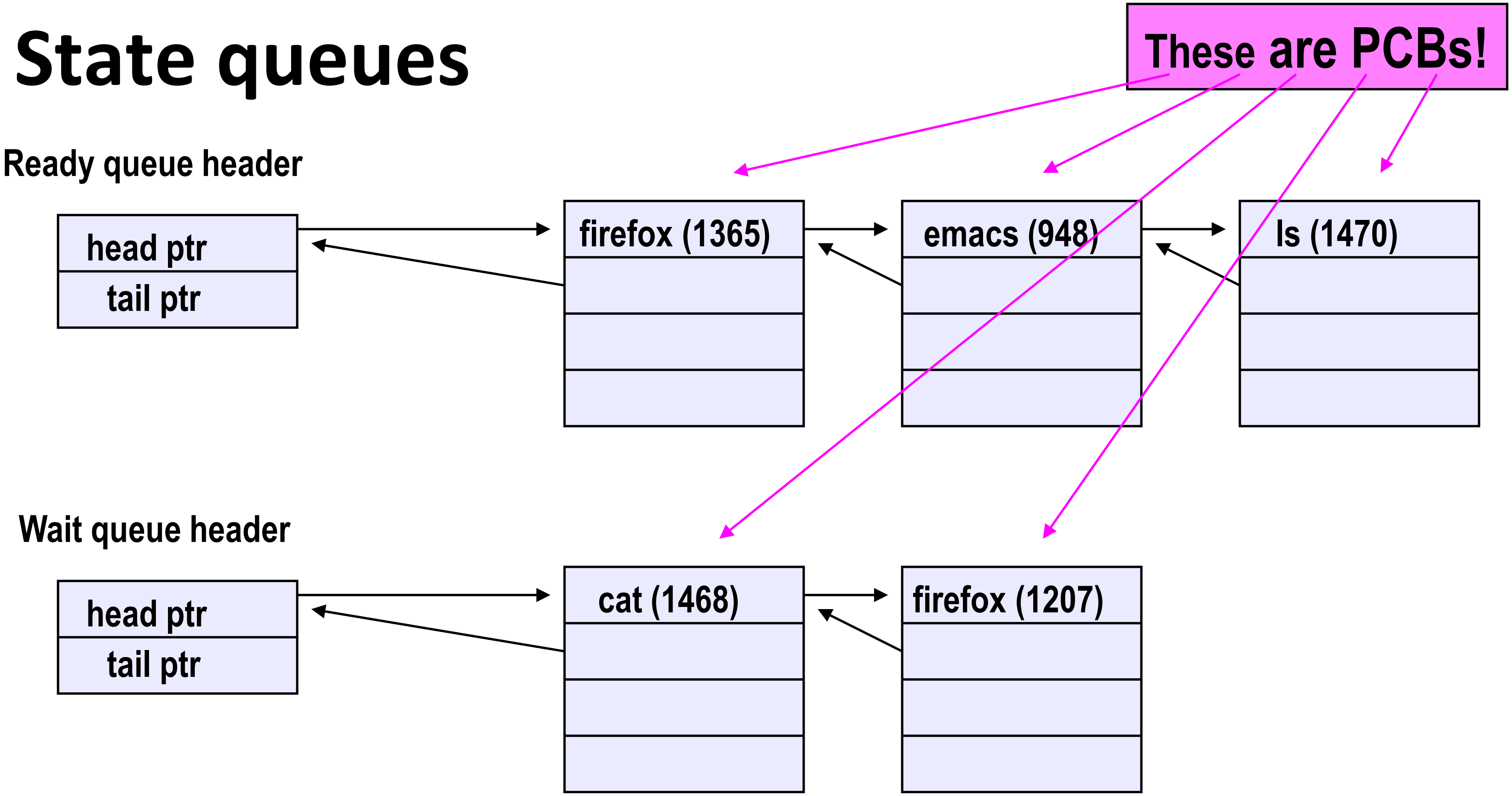


You can create and destroy processes!

State queues

- **The OS maintains a collection of queues that represent the state of all processes in the system**
  - typically one queue for each state
    - e.g., ready, waiting, …
  - each PCB is queued onto a state queue according to the current state of the process it represents
  - as a process changes state, its PCB is unlinked from one queue, and linked onto another
- **Once again, *this is just as straightforward as it sounds!* The PCBs are moved between queues, which are represented as linked lists. *There is no magic!***

State queues



- There may be many wait queues, one for each type of wait (particular device, timer, message, …)

PCBs and state queues

- **PCBs are data structures**
  - dynamically allocated inside OS memory
- **When a process is created:**
  - OS allocates a PCB for it
  - OS initializes PCB
  - (OS does other things not related to the PCB)
  - OS puts PCB on the correct queue
- **As a process computes:**
  - OS moves its PCB from queue to queue
- **When a process is terminated:**
  - PCB may be retained for a while (to receive signals, etc.)
  - eventually, OS deallocates the PCB

Review: 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, …
- **That’s a lot of concepts bundled together!**
- **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
- **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

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 commands, 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

What’s needed?

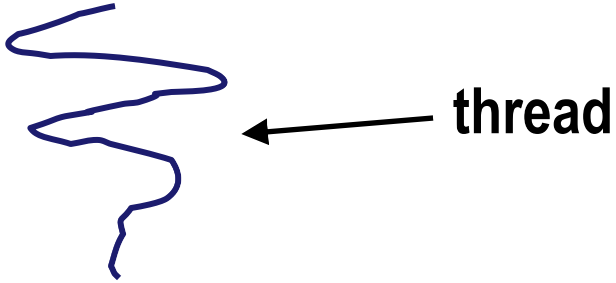
- **In each of these examples of concurrency (web server, web client, parallel program):**
    - Everybody wants to run the same code
    - Everybody wants to access the same data
    - Everybody has the same privileges
    - Everybody uses the same resources (open files, network connections, etc.)
- **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 (kind of)
- **This is like making a pig fly – it's really inefficient**
  - space: PCB, page tables, etc.
  - time: creating OS structures, fork/copy address space, etc.
- **Some equally bad alternatives for some of the examples:**
  - Entirely separate web servers
  - Manually programmed asynchronous programming (non-blocking I/O) in the web client (browser)

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**



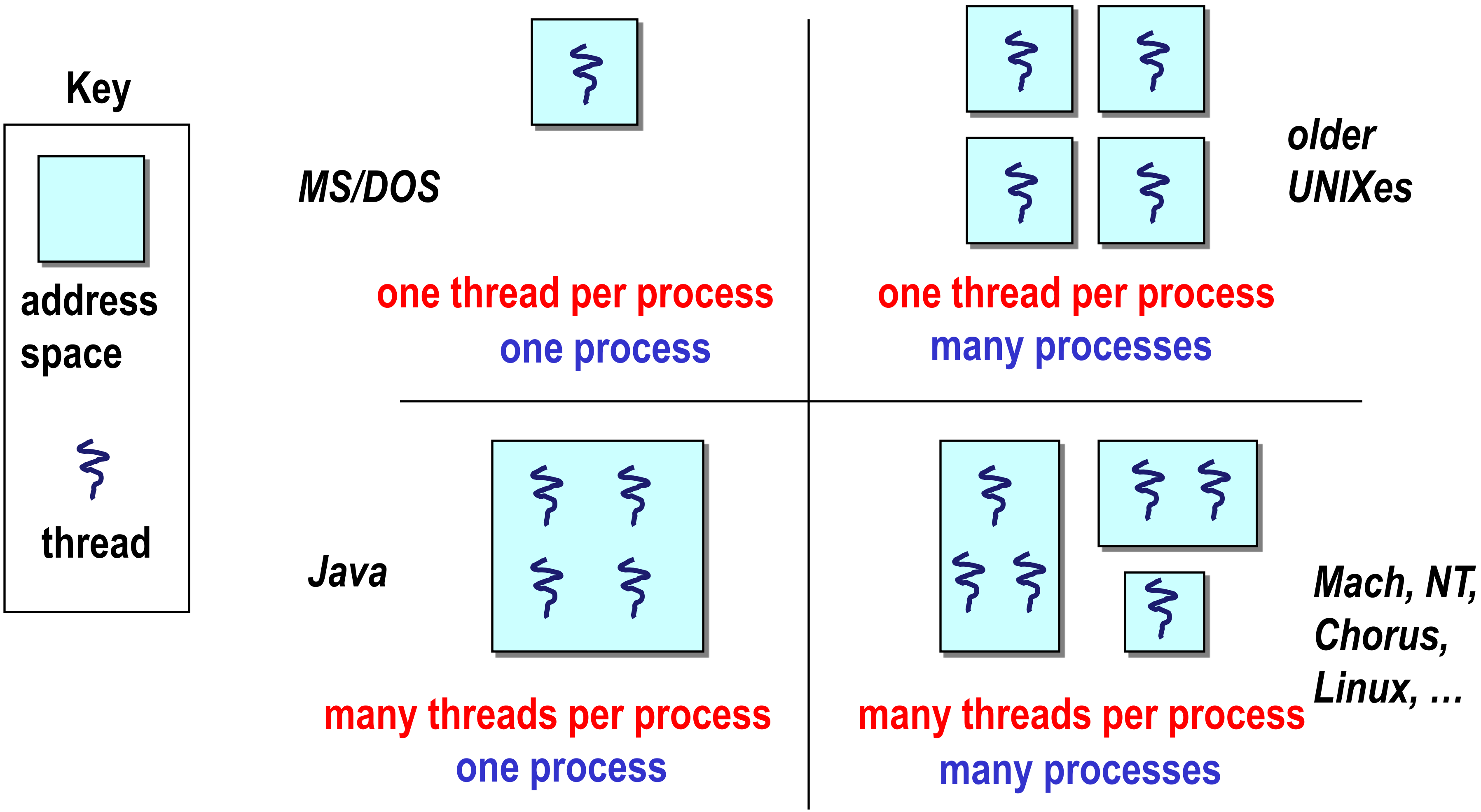
Threads and processes

- **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

Threads

- **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

one thread per process
many processes

older UNIXes

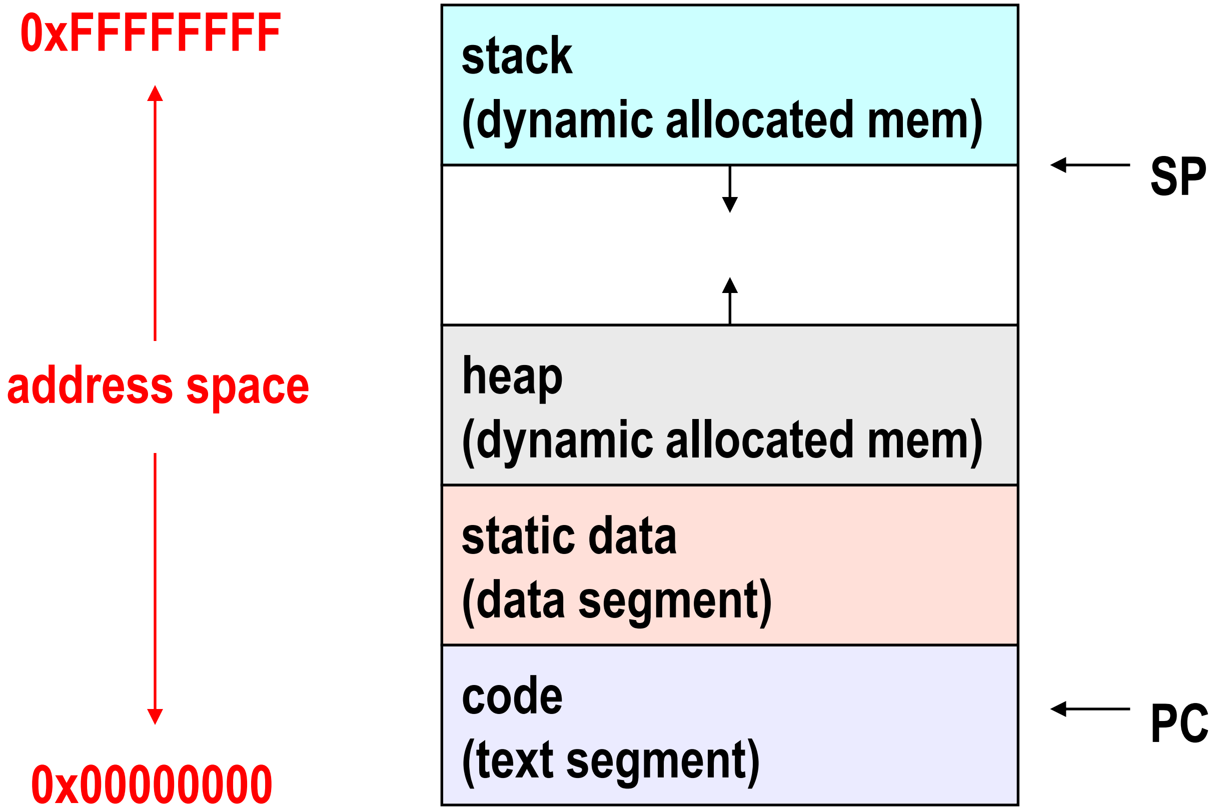
Java

many threads per process
one process

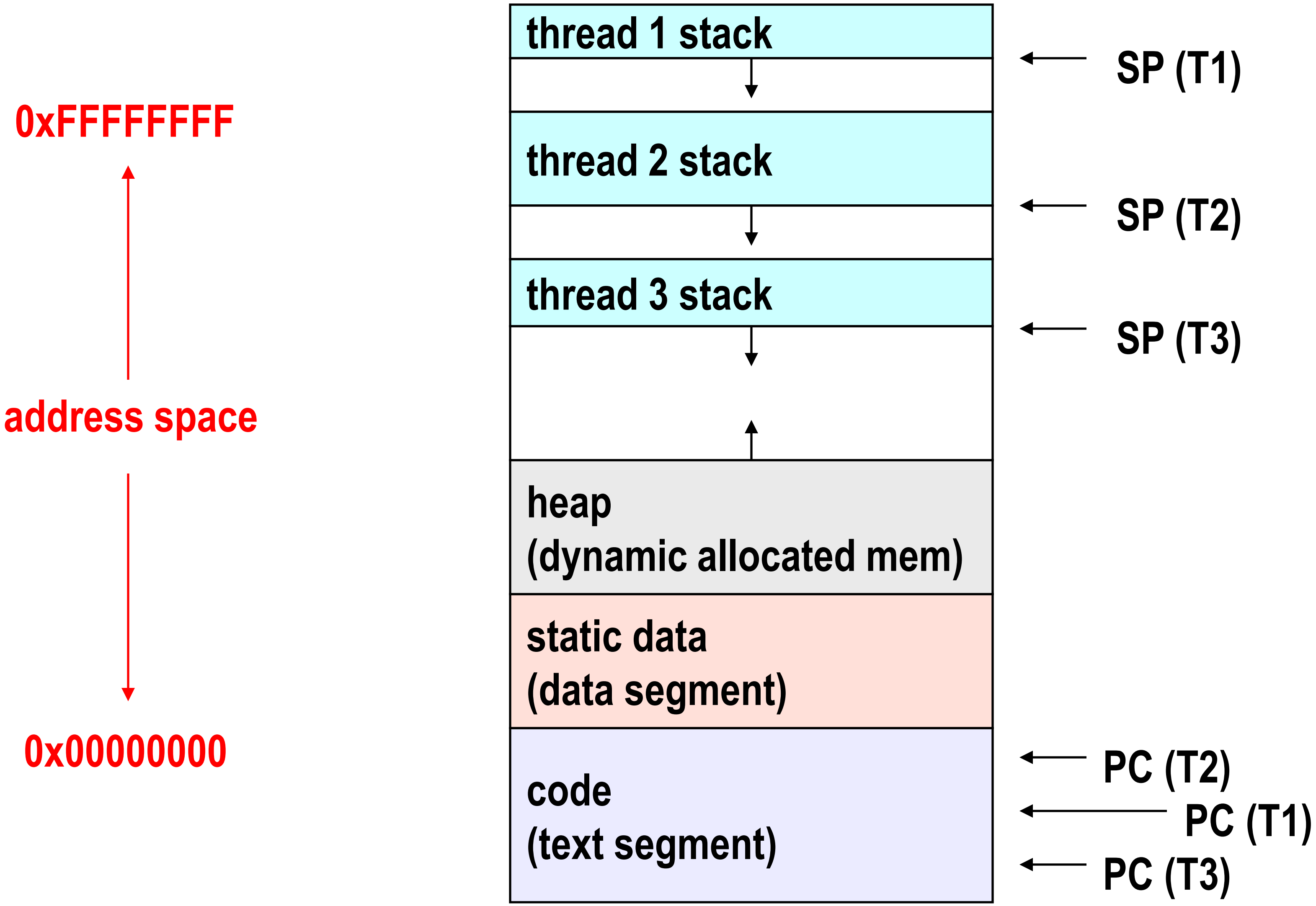
many threads per process
many processes

Mach, NT, Chorus, Linux, …

(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 world: “process” == “address space + OS resources + single thread”
  - New world: “process” typically refers to an address space + system resources + all of its threads …
    - When we mean the “address space” we need to be explicit

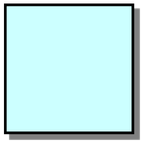
  “thread” refers to a single thread of control within a process / address space

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

“Where do threads come from, Mommy?”

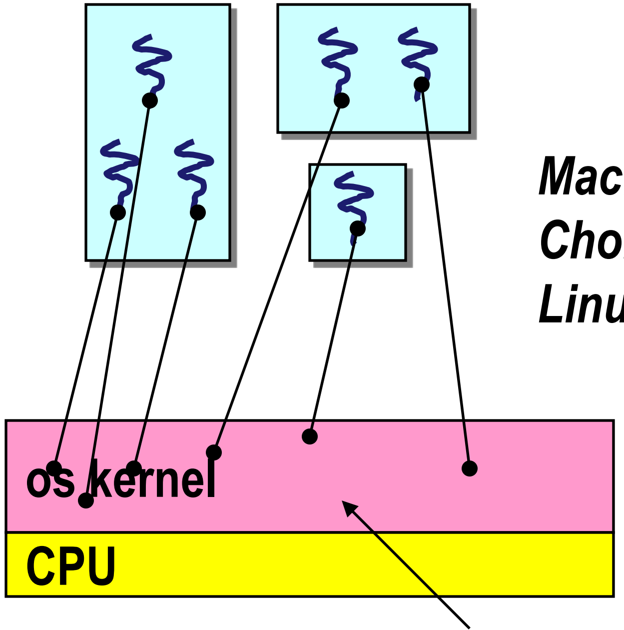
- **Natural answer: the OS is responsible for creating/managing threads**
  - For example, the kernel call to create a new thread would
    - allocate an execution stack within the process address space
    - create and initialize a Thread Control Block
      - stack pointer, program counter, register 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 (surprise!)

Kernel threads



address space

thread



Mach, NT, Chorus, Linux, …

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

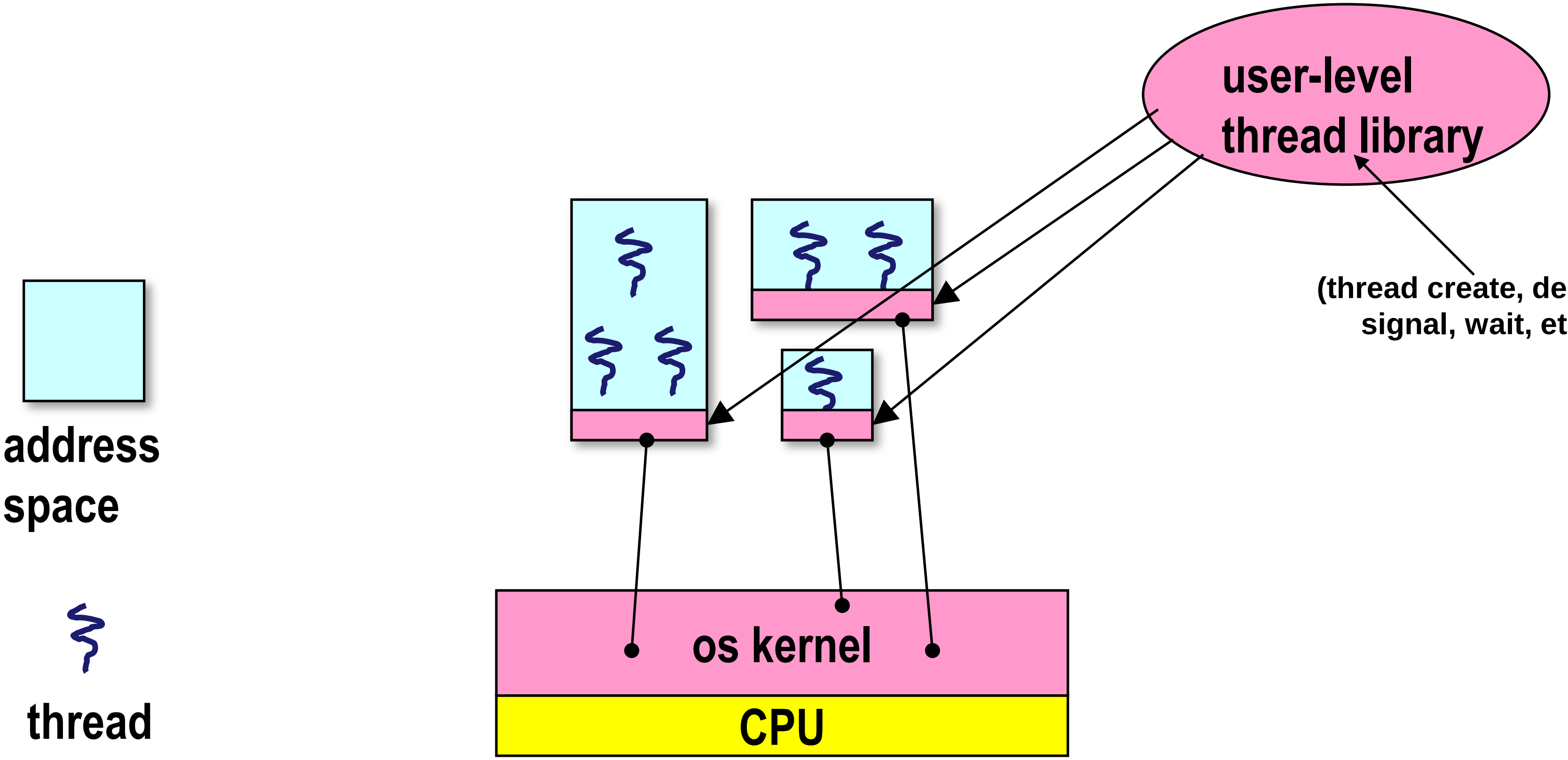
Kernel threads

- **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

“Where do threads come from, Mommy?” (2)

- **There is an alternative to kernel threads**
- **Threads can also 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), 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



address space

thread

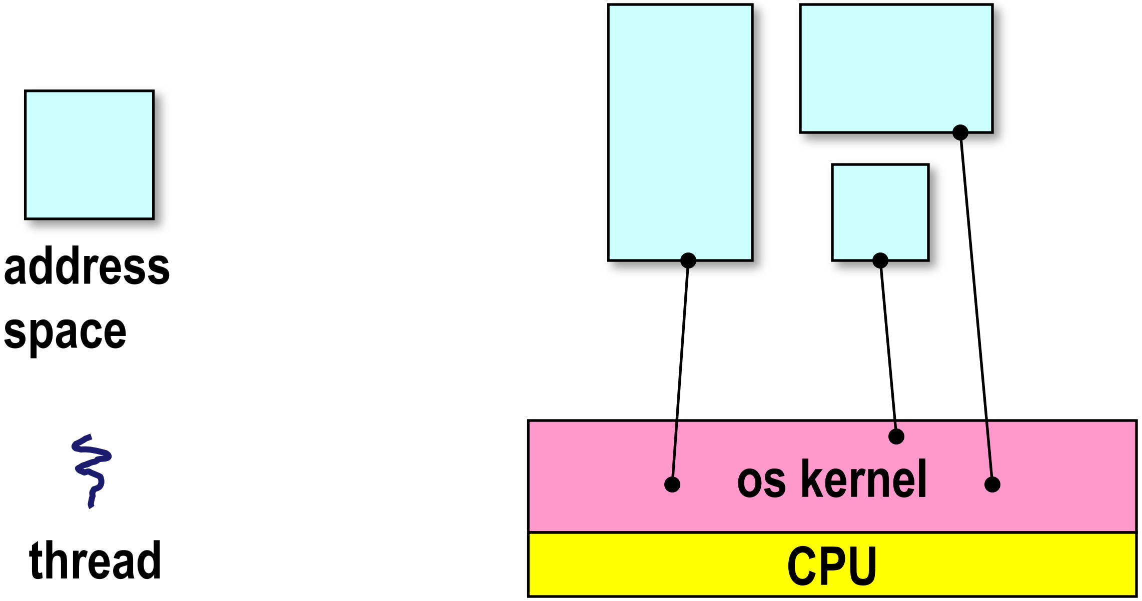
user-level thread library

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

os kernel

CPU

User-level threads: what the kernel sees



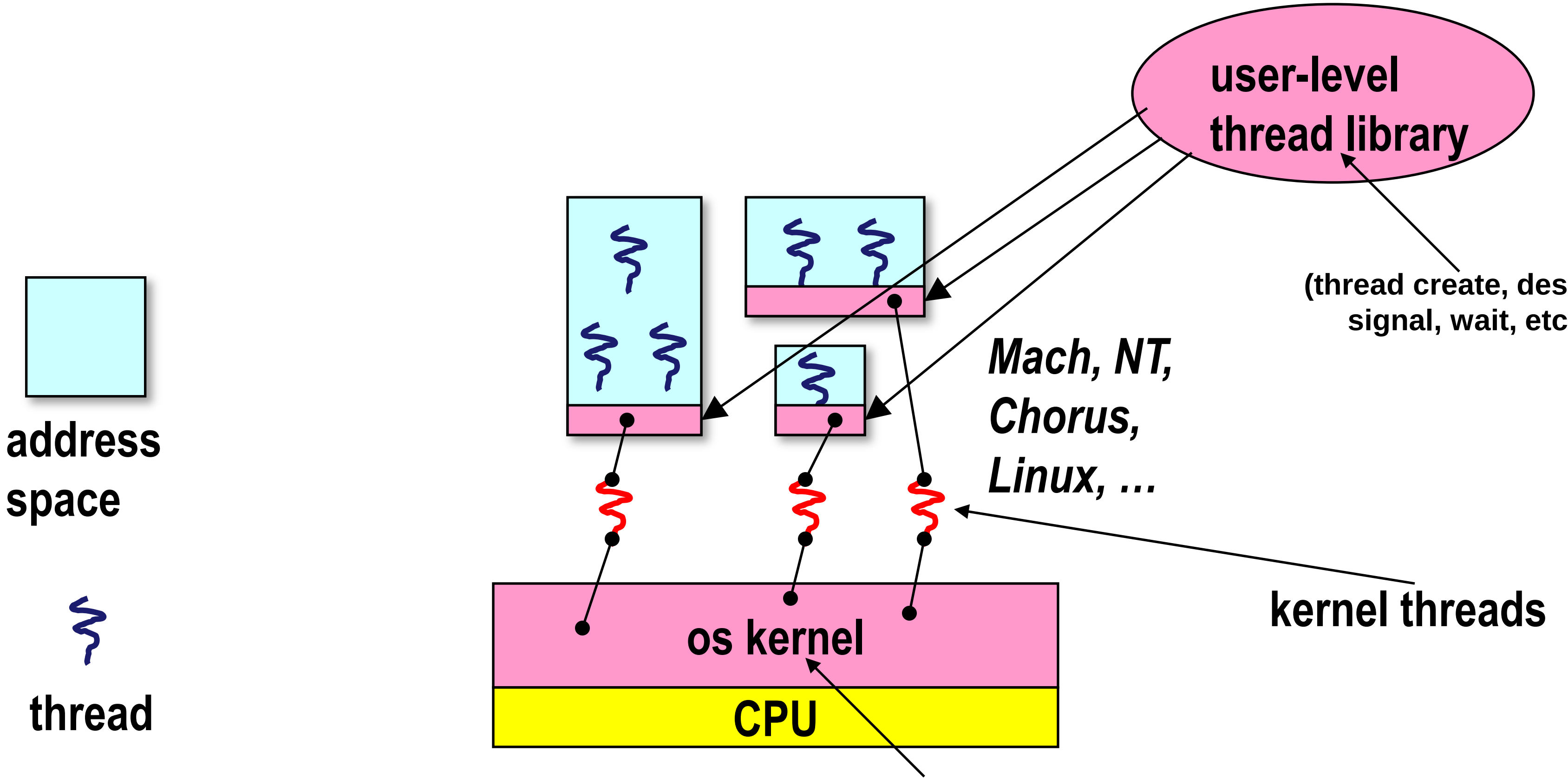
address space

thread

os kernel

CPU

User-level threads: the full story



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

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

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 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

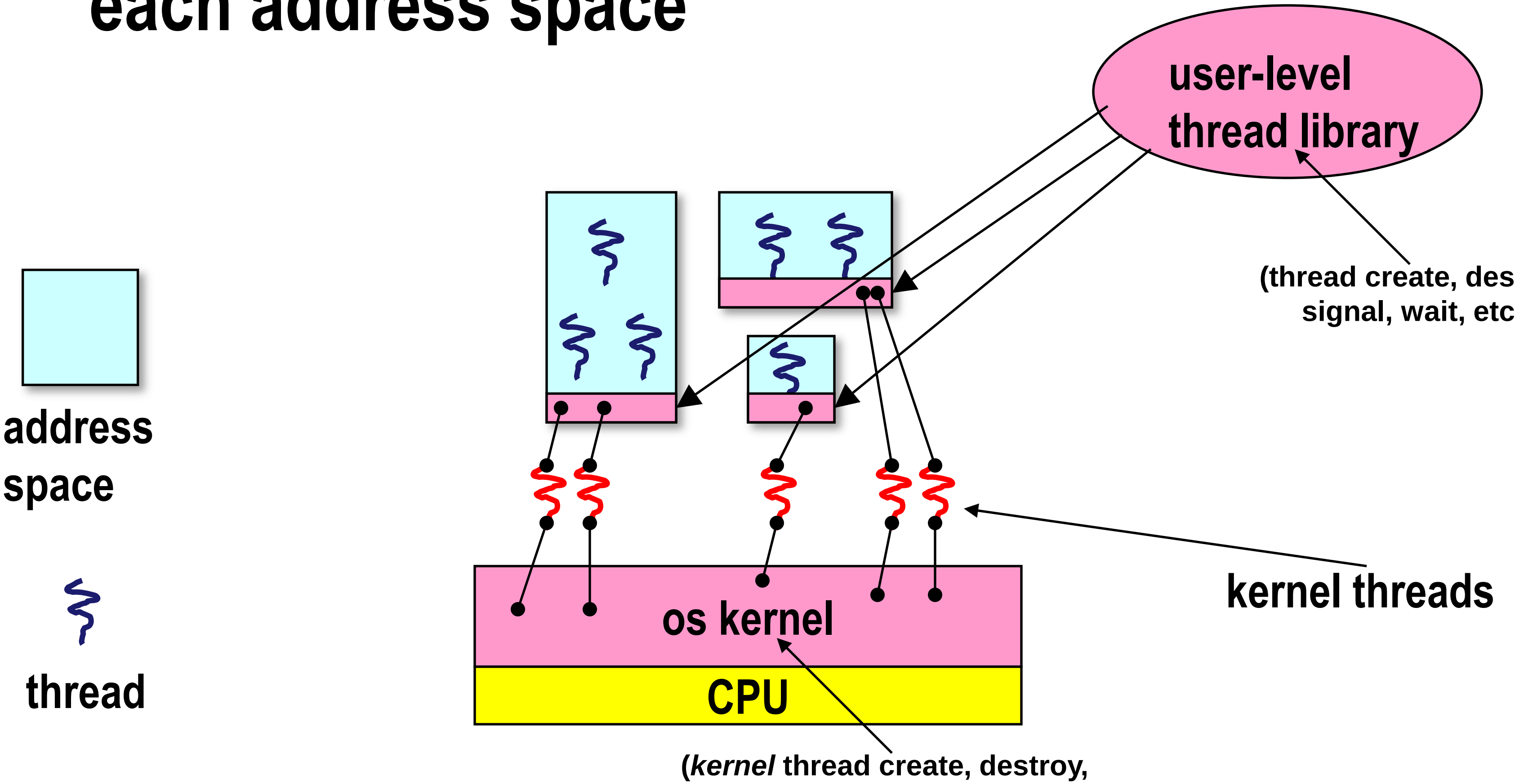
How to keep a user-level thread from hogging the CPU?

- **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



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

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

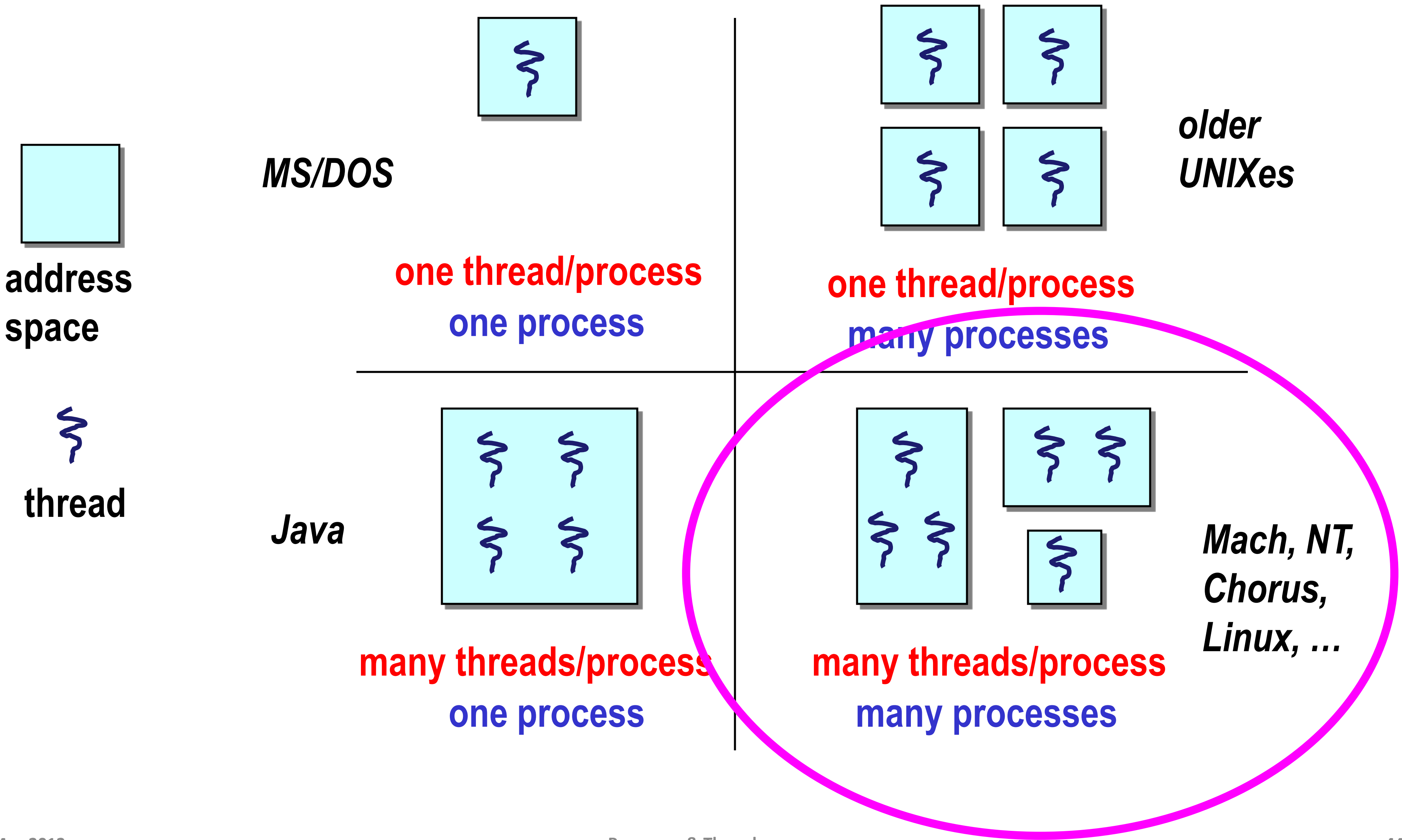
Addressing these problems

- **Effective coordination of kernel decisions and user-level threads requires OS-to-user-level communication**
  - OS notifies user-level that it is about to suspend a kernel thread
- **This is called "scheduler activations"**
    - a research paper from UW with huge effect on practice
    - each process can request 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
    - *ACM TOCS 10*,1

Summary

- **You really want multiple threads per address space**
- **Kernel threads are much more efficient than processes, but they're still not cheap**
  - all operations require a kernel call and parameter validation
- **User-level threads are:**
  - really fast/cheap
  - great for common-case operations
    - creation, synchronization, destruction
  - can suffer in uncommon cases due to kernel obliviousness
    - I/O
    - preemption of a lock-holder
- **Scheduler activations are an answer**
  - pretty subtle though

The design space



address space

thread

MS/DOS

one thread/process
one process

older UNIXes

one thread/process
many processes

Java

many threads/process
one process

Mach, NT, Chorus, Linux, …

many threads/process
many processes