Threads

CSE 410, Spring 2009

Computer Systems

http://www.cs.washington.edu/410

Reading and References

- Reading
  - » Read sec. 4.1-4.2, rest of ch. 4 as background, *Operating System Concepts*, Silberschatz, Galvin, and Gagne

What's in a process?

- A process consists of (at least):
  - » an address space
  - » the code for the running program
  - » the data for the running program
  - » 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
  - » a set of OS resources
    - open files, network connections, sound channels, …
- That's a lot of concepts bundled together!

Concurrency

- 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 46 “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 concurrently employ multiple processors
  - » For example, multiplying a large matrix – 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* address space to share data
- This is like making a pig fly – it's really inefficient
  - » space: PCB, page tables, etc.
  - » time: creating OS structures, fork and copy addr space, etc.
- Some equally bad alternatives for some of the cases:
  - » Entirely separate web servers
  - » Asynchronous programming (explicit programming of non-blocking I/Os) in the web client (browser)

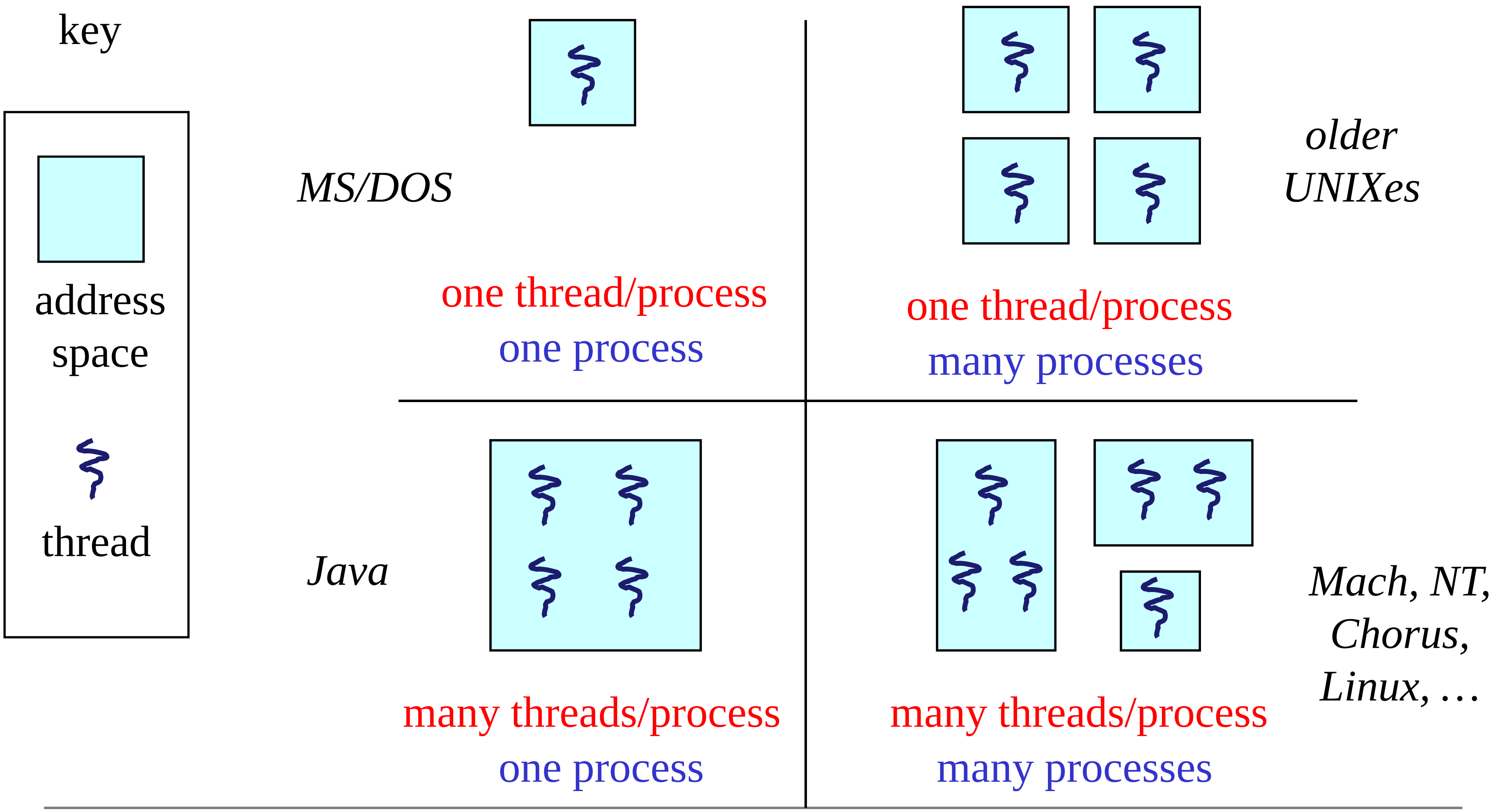
Can we do better?

- Key idea:
  - » separate the concept of a process (address space, etc.)
  - » from that of a minimal "thread of control" (execution state: PC, etc.)
- This execution state is usually called a thread, or sometimes, a lightweight process

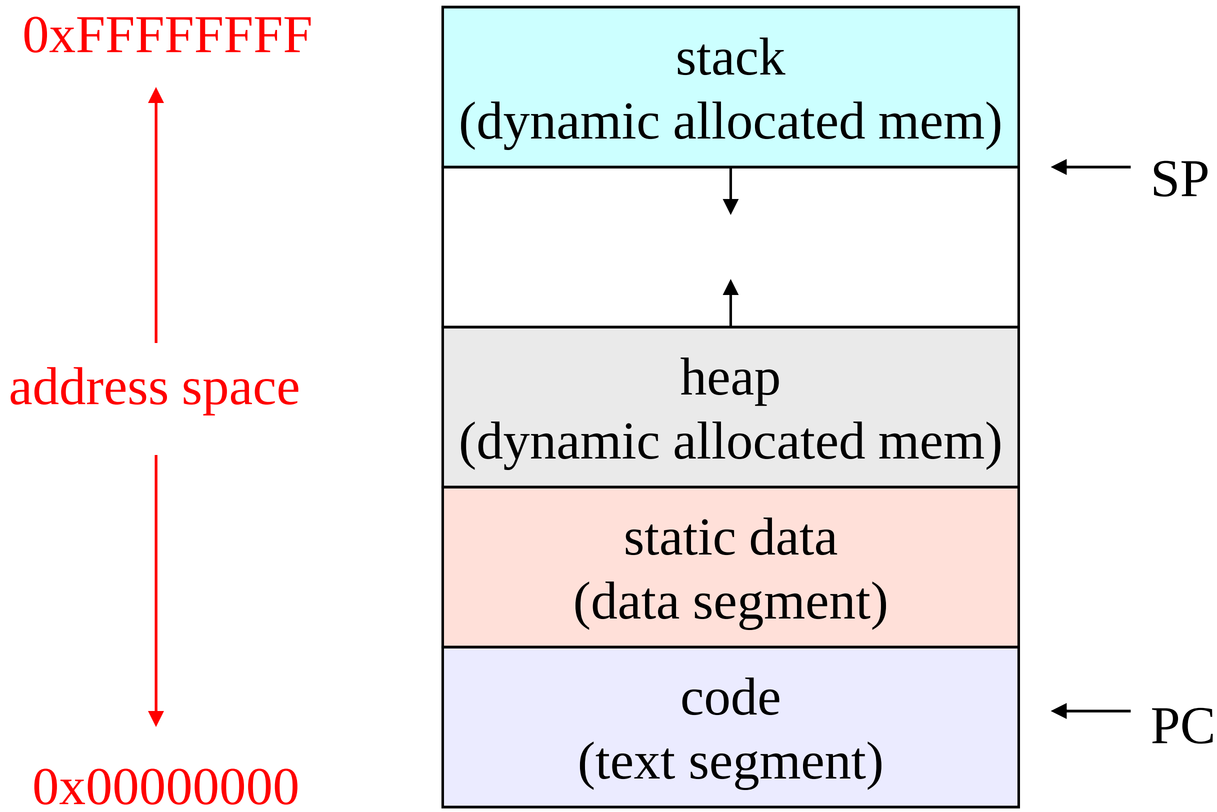
Threads and processes

- Most modern OS's (Mach, Chorus, Win/XP, modern 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
  - » processes, however, can have multiple threads executing within them
  - » sharing data between threads is cheap: all see same address space
- Threads become the unit of scheduling
  - » processes are just containers in which threads execute

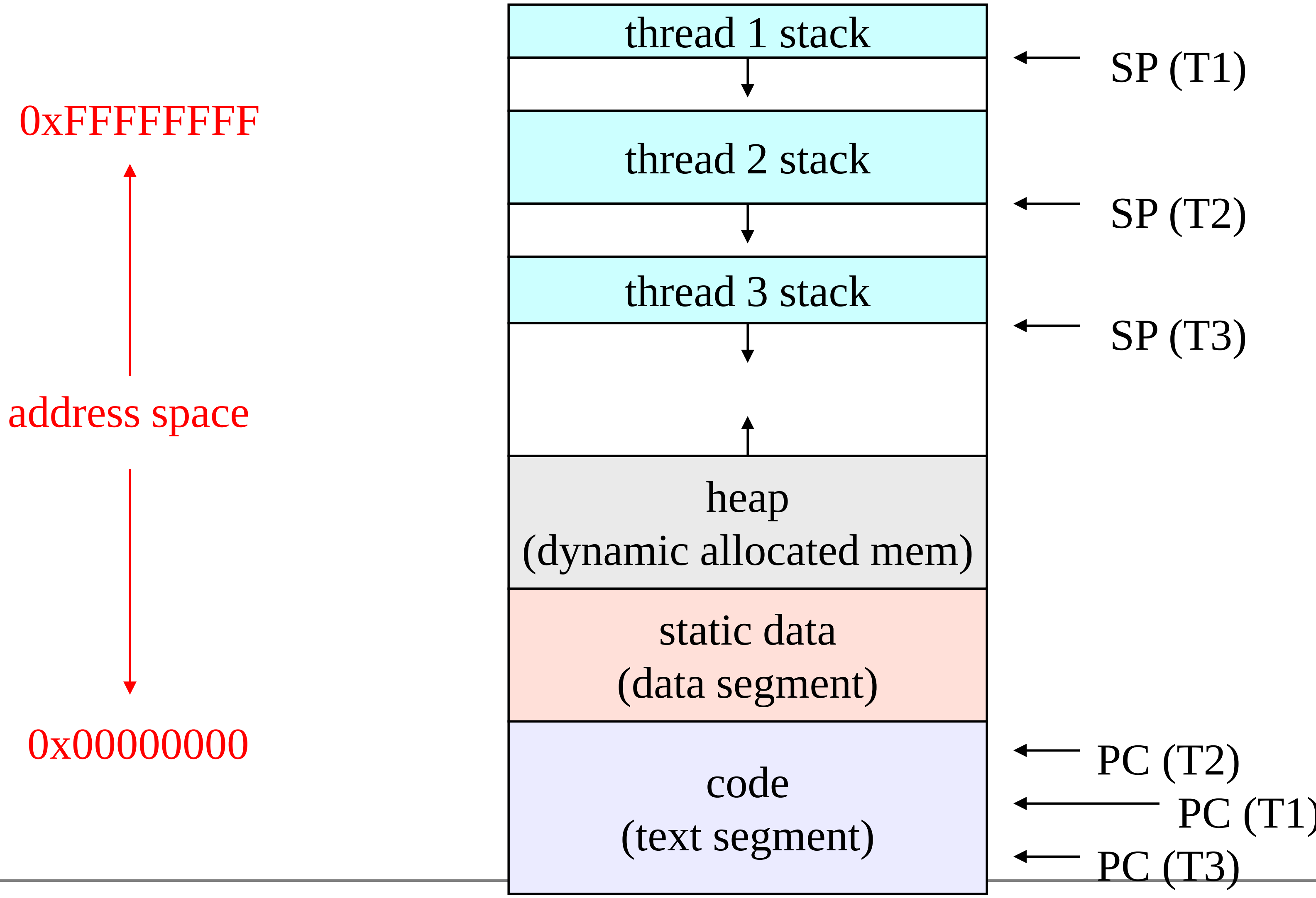
The design space



(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"

“Where do threads come from, Mommy?”

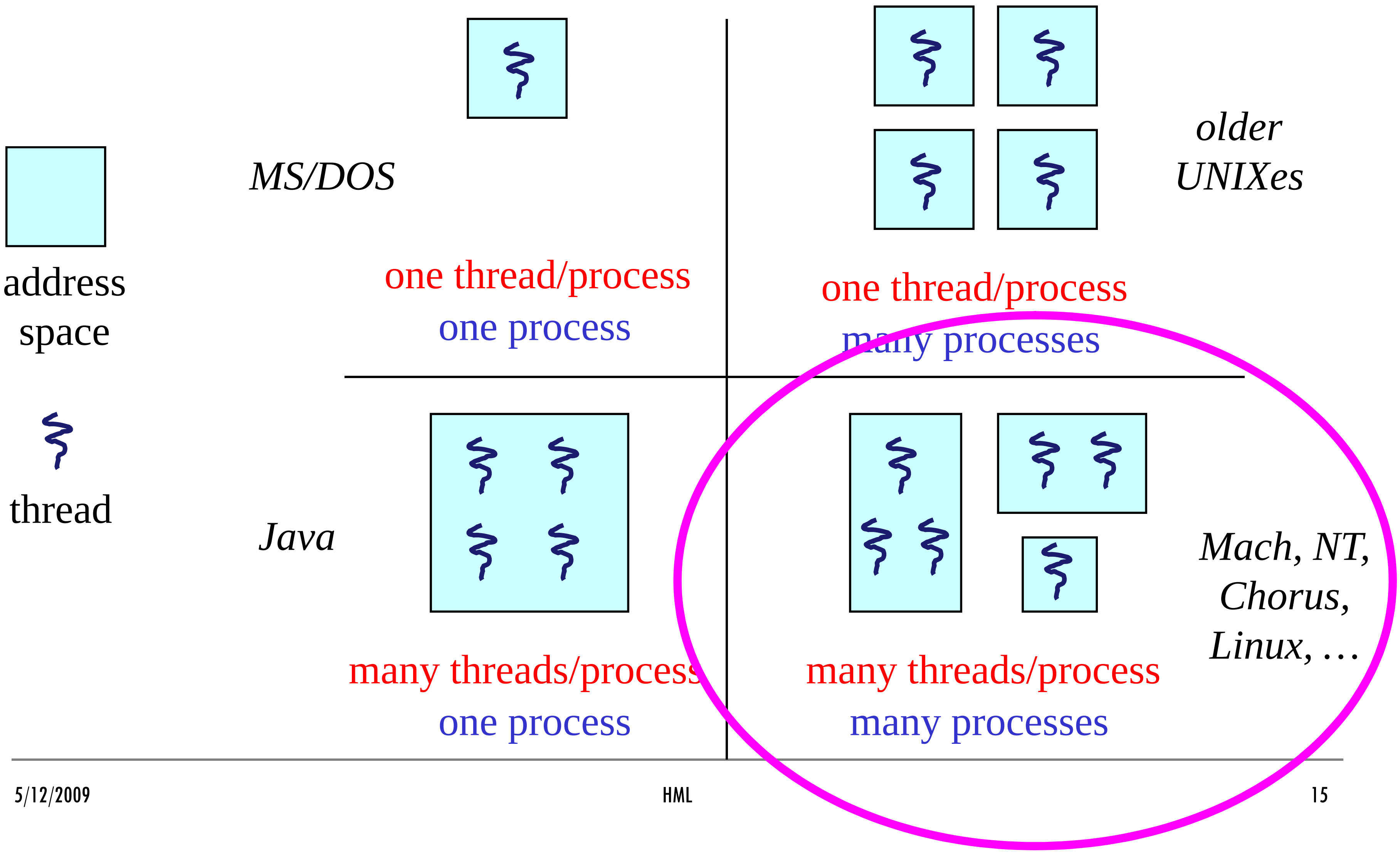
- Natural answer: the kernel is responsible for creating/managing threads
  - » for example, the kernel call to create a new thread would
    - allocate an execution stack within the process address space
    - create and initialize a Thread Control Block

      stack pointer, program counter, register values
    - stick it on the ready queue
  - » we call these kernel threads

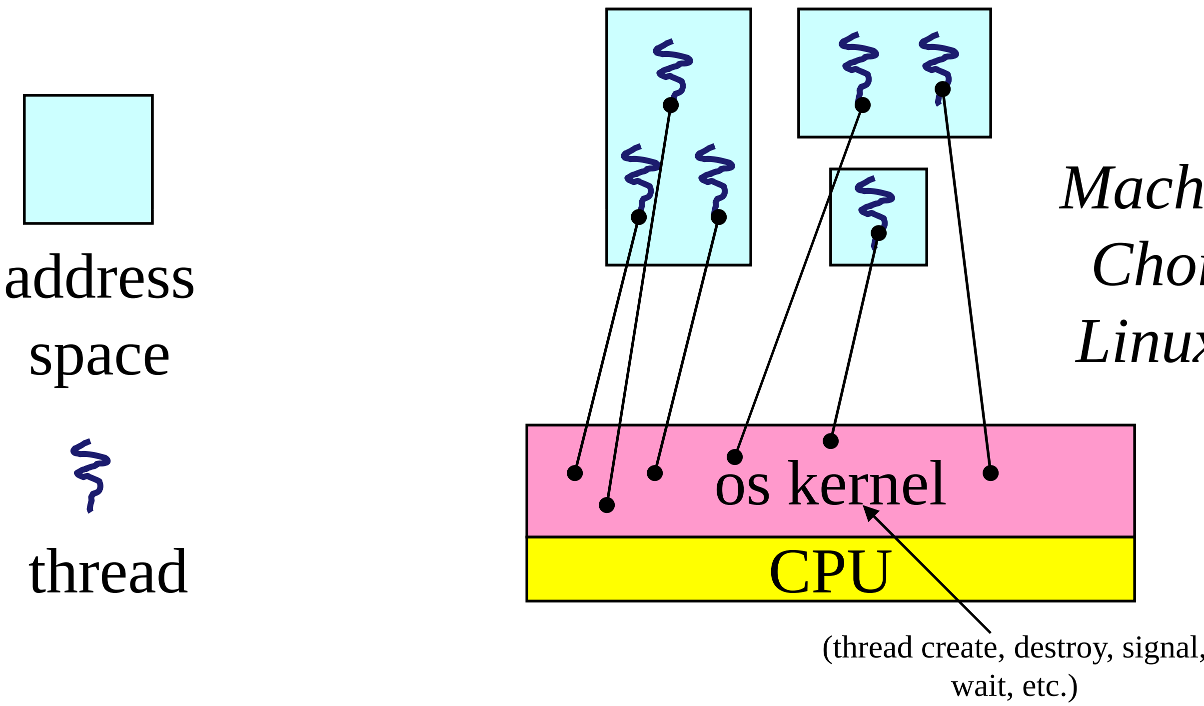
User-Level Threads

- 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
    - Thread package multiplexes user-level threads on top of kernel thread(s), which it treats as "virtual processors"
  - » we call these user-level threads

The design space

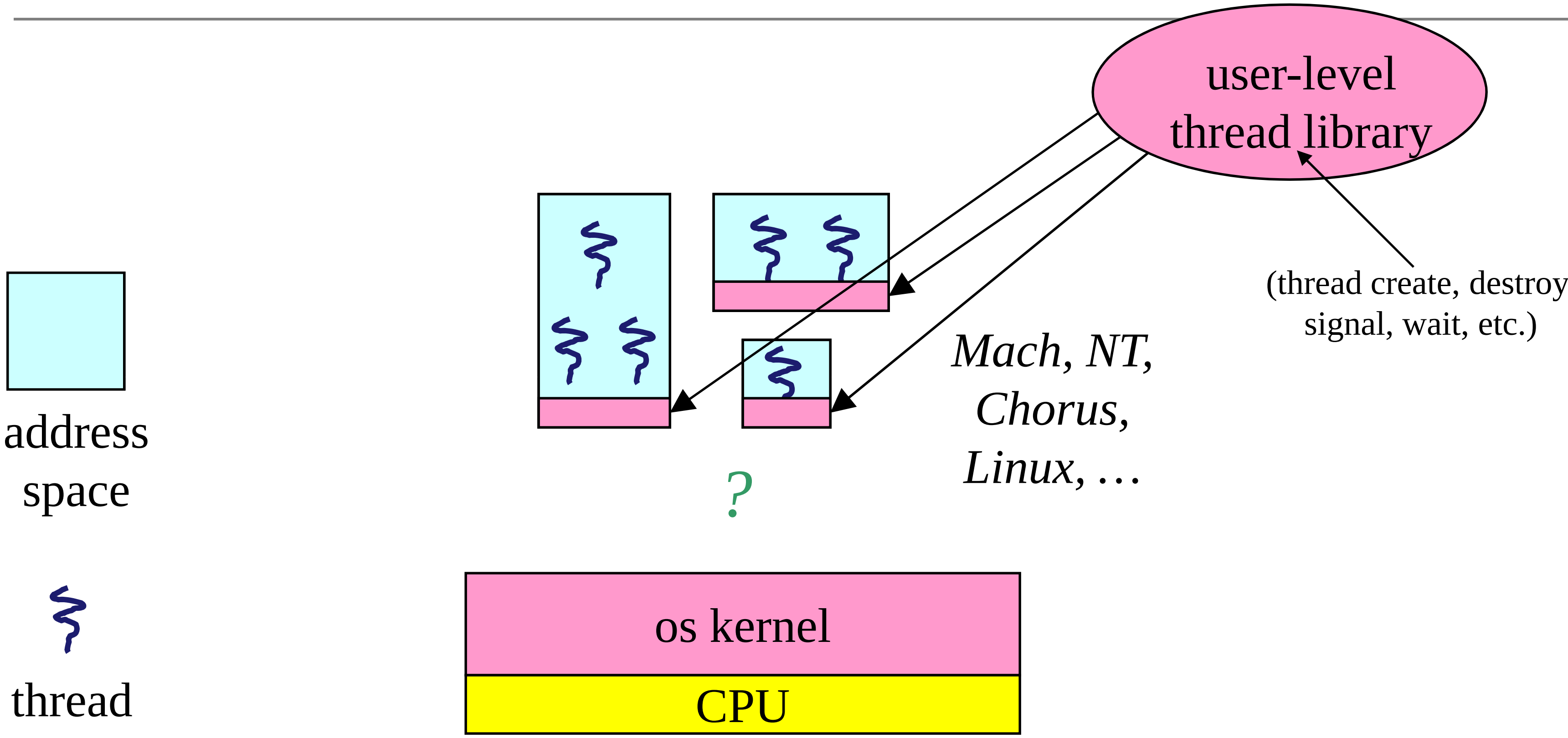


Kernel threads

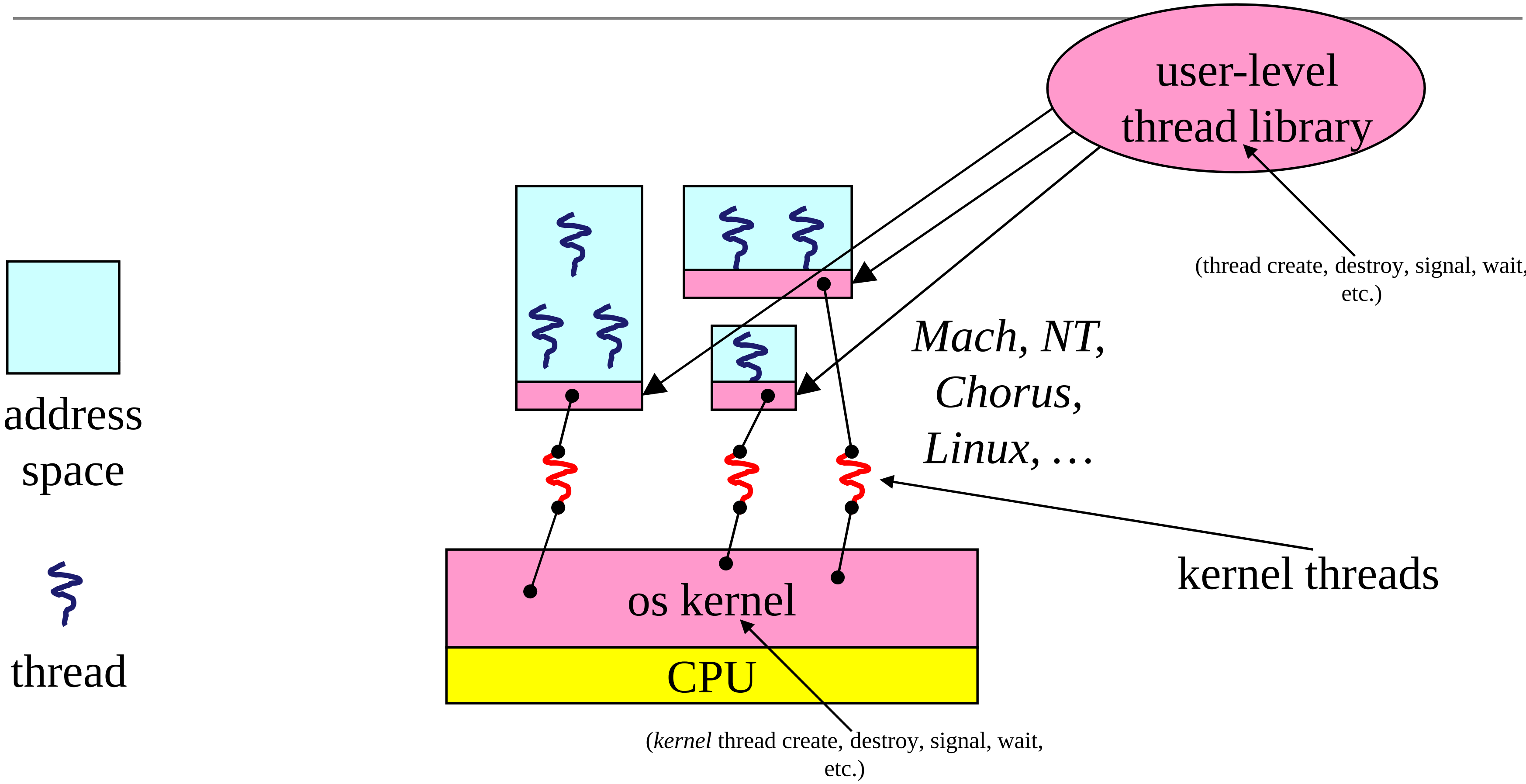


Mach, NT, Chorus, Linux, …

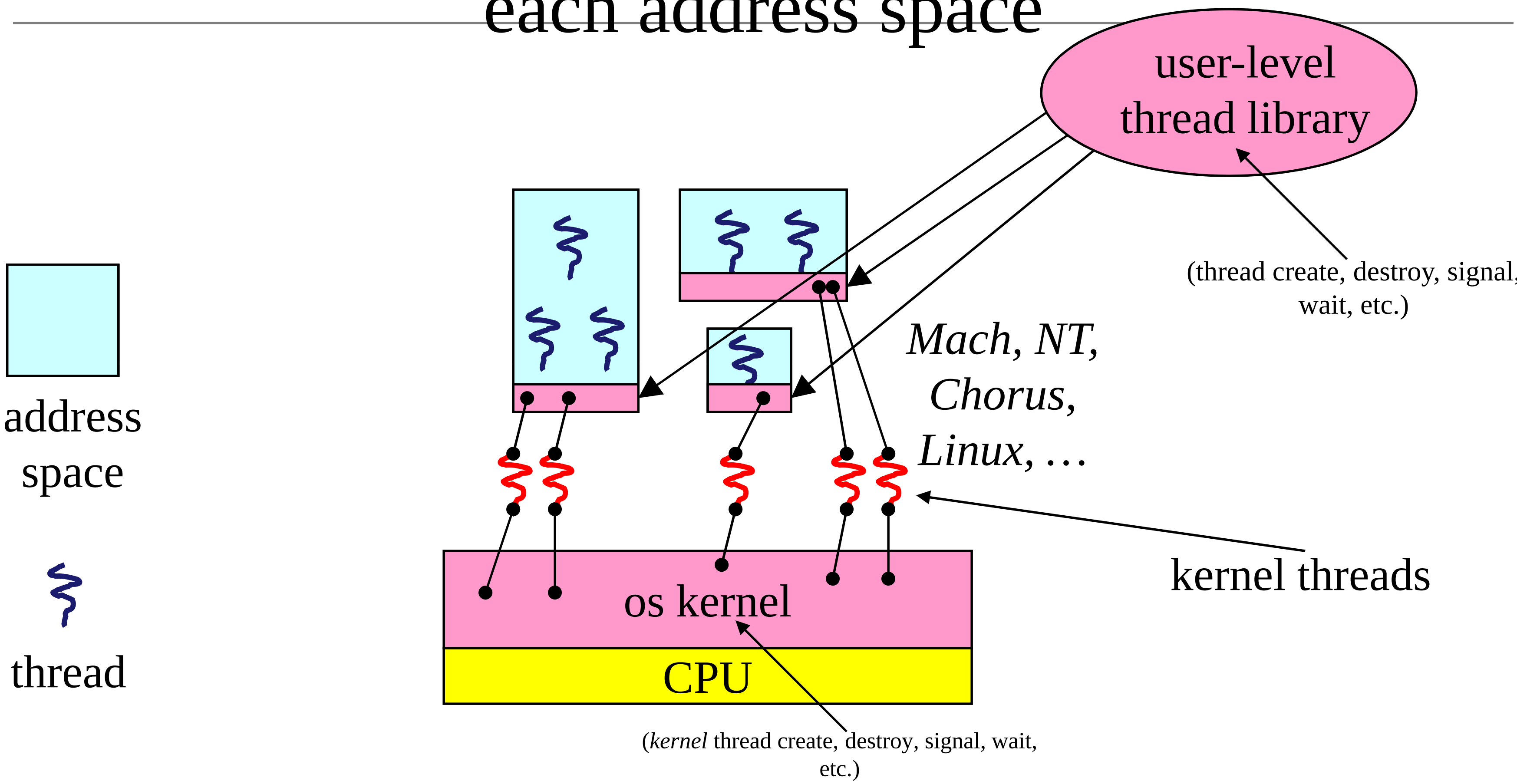
User-level threads, conceptually



User-level threads, *really*



Multiple kernel threads "powering" each address space



Kernel threads

- OS now manages threads *and* processes
  - » 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 (e.g., 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

User-level threads

- To make threads cheap and fast, they need to be implemented at the user level
  - » managed entirely by user-level library, e.g. `libpthreads.a`
- User-level threads are small and fast
  - » 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

Performance example

- On a 700MHz Pentium running Linux 2.2.16:

  » Processes
    - `fork/exit:` 251 μs

  » Kernel threads
    - `pthread_create()/pthread_join():` 94 μs

  » User-level threads
    - `pthread_create()/pthread_join`: 4.5 μs

User-level thread implementation

- The kernel thread (the kernel-controlled executable entity associated with the address space) executes the code in the address space
- This code includes the thread support library and its associated thread scheduler
- The thread scheduler determines when a 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

How to keep a 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
  - » User-level scheduler requests that a timer interrupt be delivered by the OS periodically
  - » at each timer interrupt, scheduler gains control and context switches as appropriate

Thread context switch

- Very simple for user-level threads:
  - » save context of currently running thread
    - push machine state onto thread stack
  - » restore context of the next thread
    - pop machine state from next thread's stack
  - » return as the new thread
    - execution resumes at PC of next thread
- 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

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!
- 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 its threads obliviously to what's going on at user-level

What if the kernel preempts a thread holding a lock?

- Other threads will be unable to enter the critical section and will block (stall)
  - » tradeoff, as with everything else
- Solving this requires coordination between the kernel and the user-level thread manager
  - » “scheduler activations”
    - a research paper from UW with huge effect on industry
    - 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 verification
- User-level threads are:
  - » fast as blazes
  - » 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 the answer
  - » pretty subtle though