# Computer Systems

CSE 410 Winter 2022

16 –Threads

# Review: What's "in" a process?

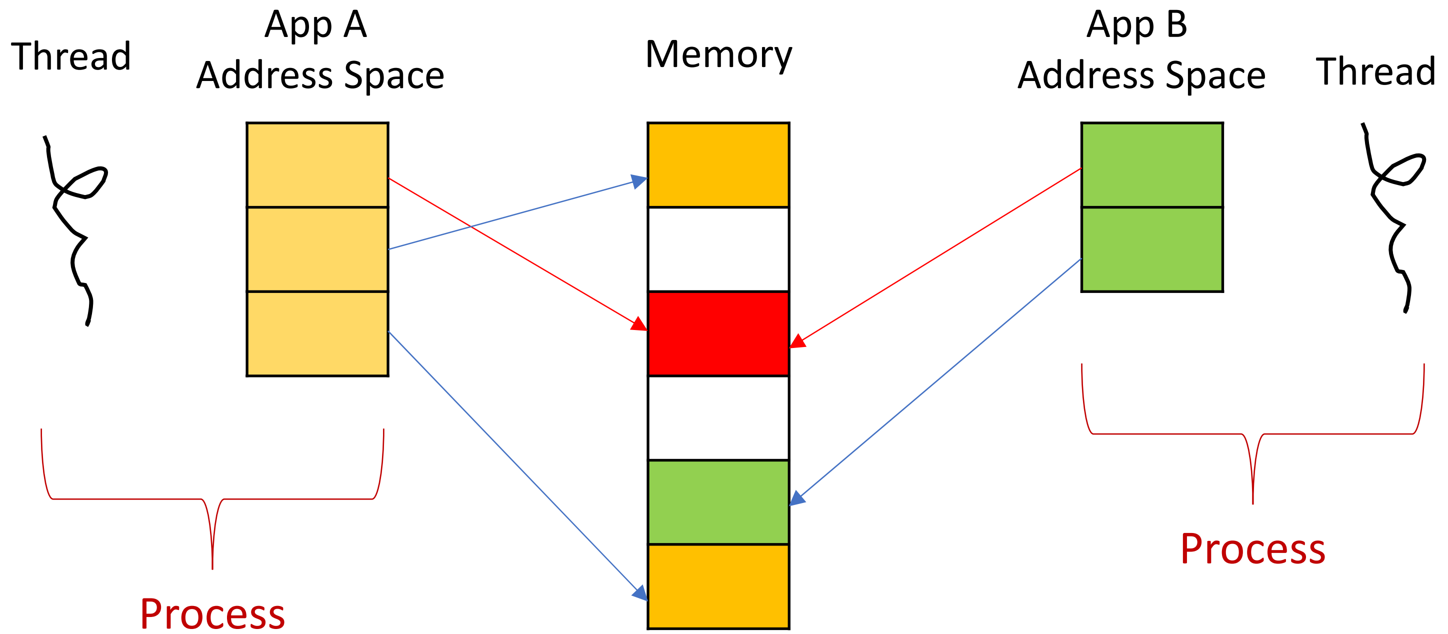
- A process consists of (at least):
  - An address space, containing
    - the code (instructions) for the running program
    - the data for the running program
  - Thread state, consisting of
    - The program counter (PC), indicating the next instruction
    - The stack pointer register (implying 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

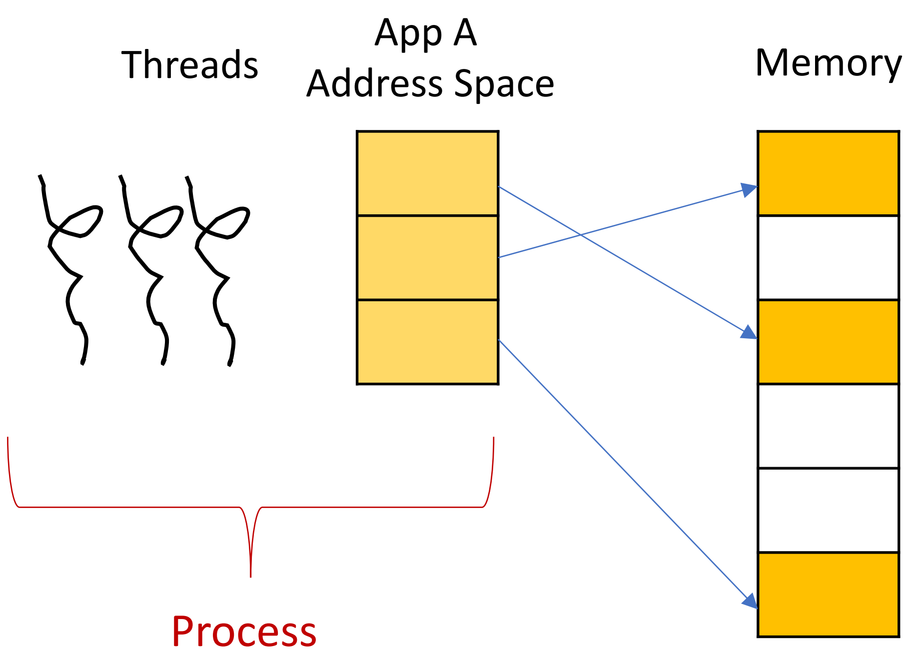
- Threads are about concurrency and parallelism
  - Parallelism:  physically simultaneous operations for performance
  - Concurrency:  logically (and possibly physically) simultaneous operations for convenience/simplicity
- One way to get concurrency and parallelism is to use multiple processes
  - The 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

# Parallelism/Concurrency and Communication

- Communicating between processes can be slow because one explicit goal of the process abstraction is isolation
- We can get fast communication by sharing memory between address spaces



# One Process, Multiple Threads




- Each thread is a flow of control
- All threads share all of memory (both virtual and physical)

- Threads execute the same code
- Threads operate on the same variables, except...
- By convention, local variables (stack variables) are touched only the thread that creates them

# Concurrency/Parallelism via Threads

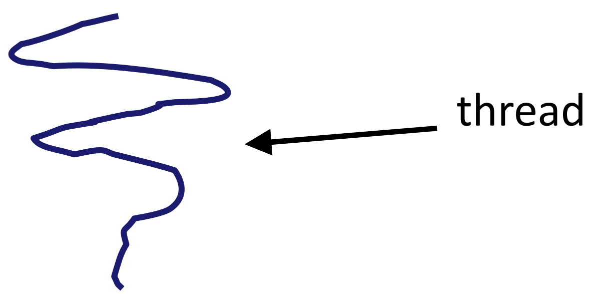
- Imagine a web server, which might like to handle multiple requests concurrently
  - While waiting for the credit card server to approve a purchase for one client, it could be retrieving the data requested by another client from disk, and assembling the response for a third client from cached information
- 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 to support concurrent execution?

- 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

# Threading

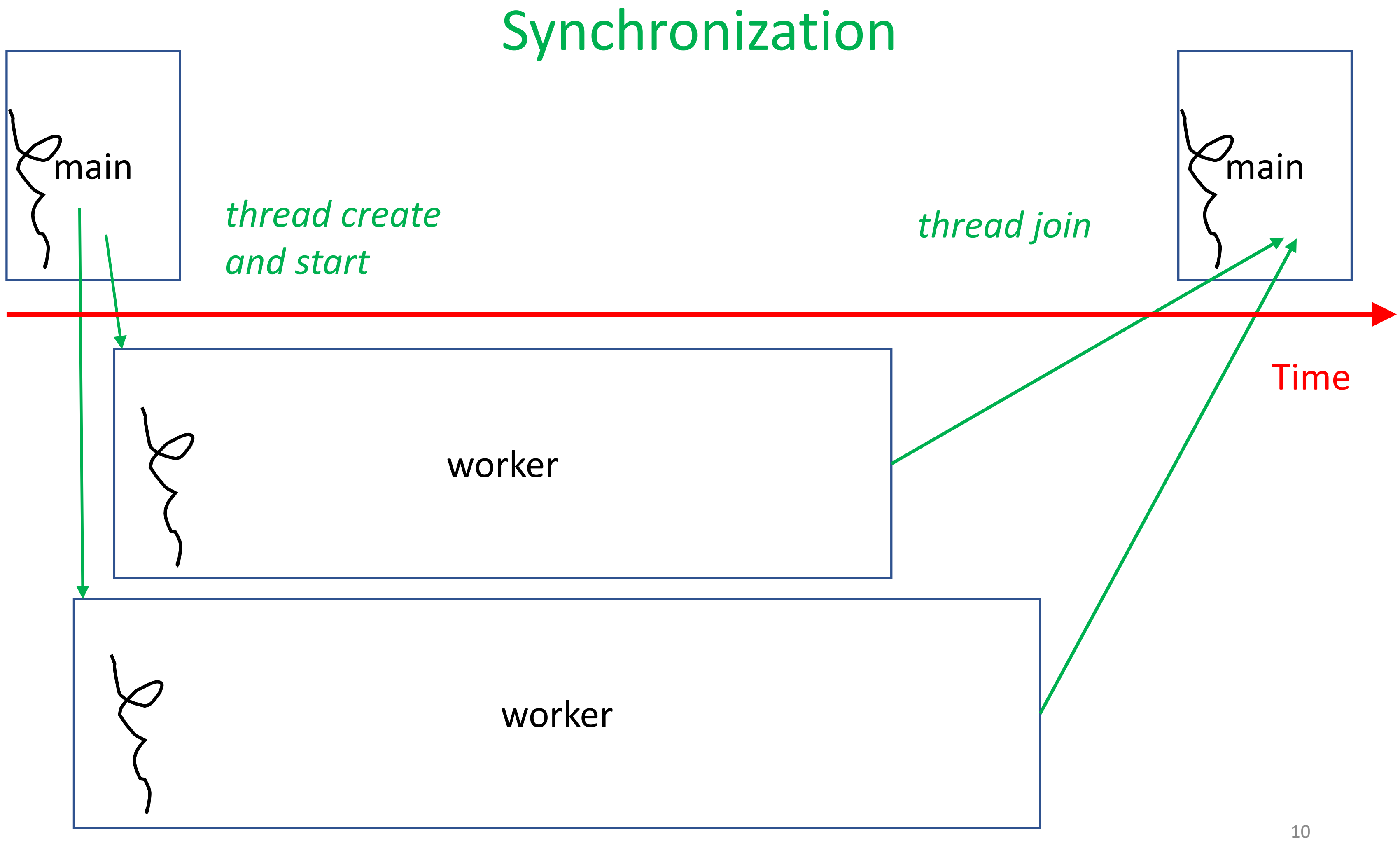
- Key idea:
  - separate the concept of a process (address space, OS resources)
  - … from that of a minimal "thread of control" (execution state: stack, stack pointer, program counter, registers)
- This execution state is usually called a thread, or sometimes, a lightweight process



# A Java Example

- *This code is available at klaatu:/courses/cse410/22wi/lect16Files*
- The example has a main(). main() is executed by the single thread that is created when the process is created (i.e., when the program is run)
- main() creates two worker threads. The threads run concurrently – that is, logically at the same time (and possibly physically at the same time)
  - There is no specific order in which instructions executed by the two threads related to each other
  - Each run of the program can (and will) result in different orderings of instructions executed by different threads
    - The instructions executed by a single thread follow the normal control flow rules

# Example: main() and workers



# Worker code

```
public class Worker extends Thread {
   private String myName;

   Worker(String name) {
      this.myName = name;
   }

   public void run() {
      try {
         for (int i=0; i<10; i++) {
            // pretend to do some work
            int sleepTime = (int)(Math.random() * 2000.0); // time to sleep in msec.
            Thread.sleep(sleepTime);

            // print some output to show progress
            System.out.println("Thread " + this.myName + " here");
         }
      } catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

# main() code

```
public static void main(String[] args) {
    Worker wA = new Worker("A");   // create threads
    Worker wB = new Worker("B");

    System.out.println("Starting worker threads");

    try {

      wA.start();  // start threads executing
      wB.start();

      wA.join();   // wait until threads are done executing
      wB.join();
    } catch (Exception e) {
      System.out.println(e);
    }

    System.out.println("Worker threads have terminated");
  }
```

# Example Executions

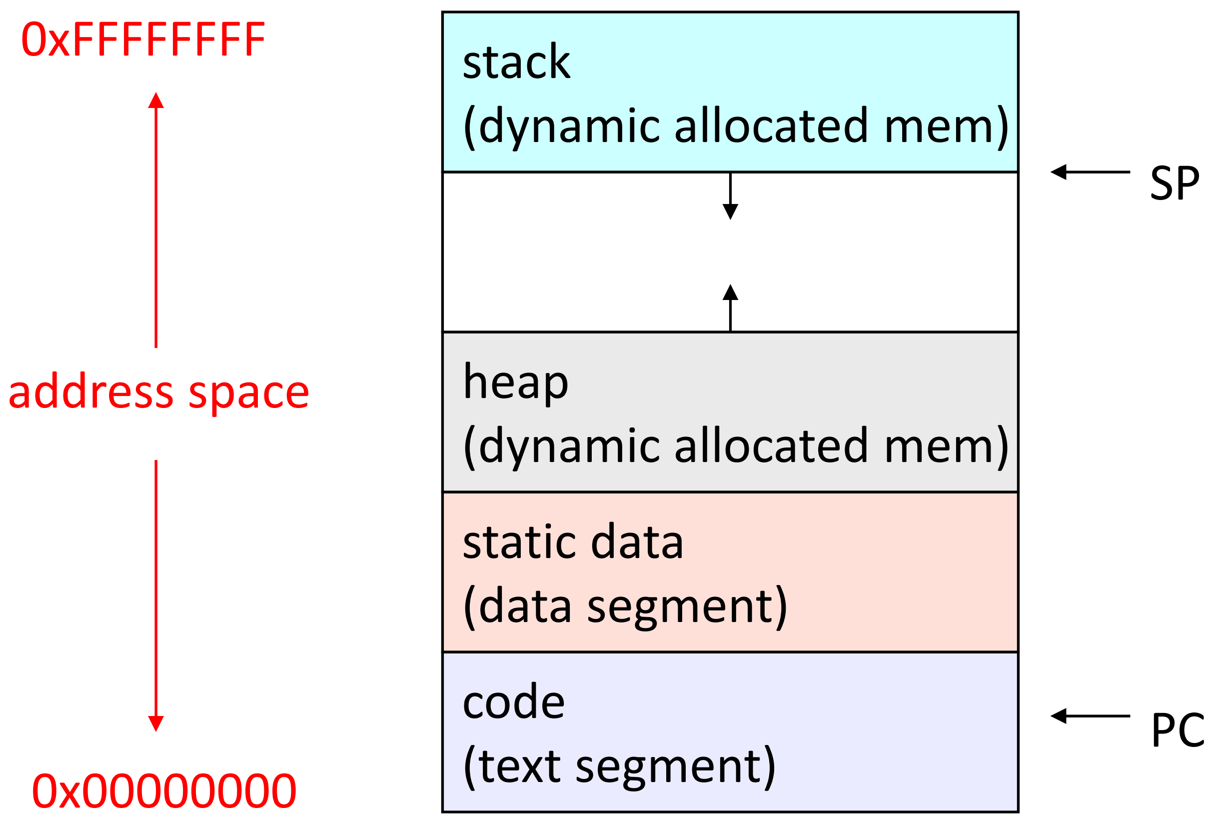
```
Starting worker threads
Thread A here
Thread B here
Thread B here
Thread B here
Thread A here
Thread A here
Thread B here
Thread A here
Thread B here
Thread B here
Thread A here
Thread B here
Thread A here
Thread B here
Thread A here
Thread B here
Thread A here
Thread A here
Thread B here
Thread A here
Worker threads have terminated
```

```
Starting worker threads
Thread A here
Thread B here
Thread A here
Thread A here
Thread B here
Thread A here
Thread A here
Thread B here
Thread B here
Thread B here
Thread A here
Thread B here
Thread A here
Thread B here
Thread B here
Thread A here
Thread A here
Thread B here
Thread B here
Thread A here
Worker threads have terminated
```

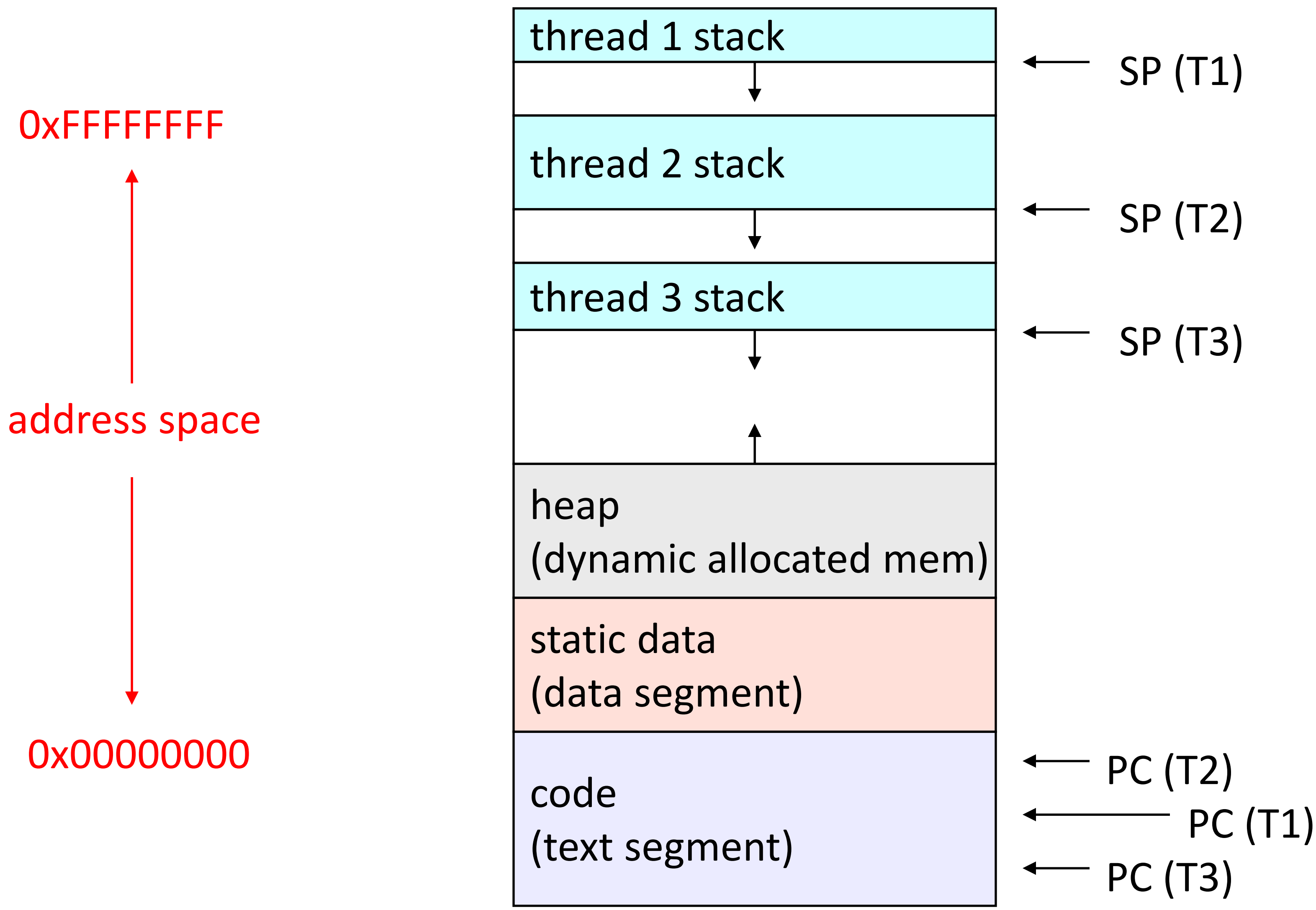
# OS: Implementing threads and processes

- 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
- 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

# (old) Process address space



# (new) Address space with threads



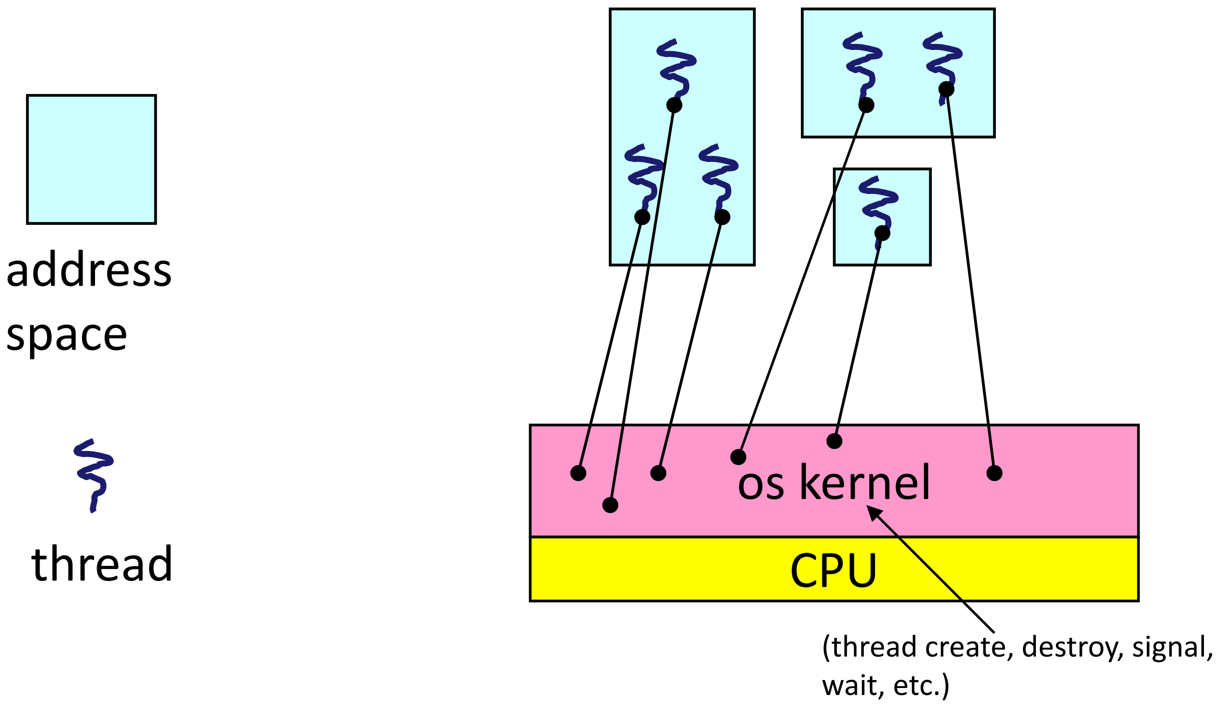
# Process/thread separation

- 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
  - useful program structuring mechanism
- 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?"

- 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
    - put 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



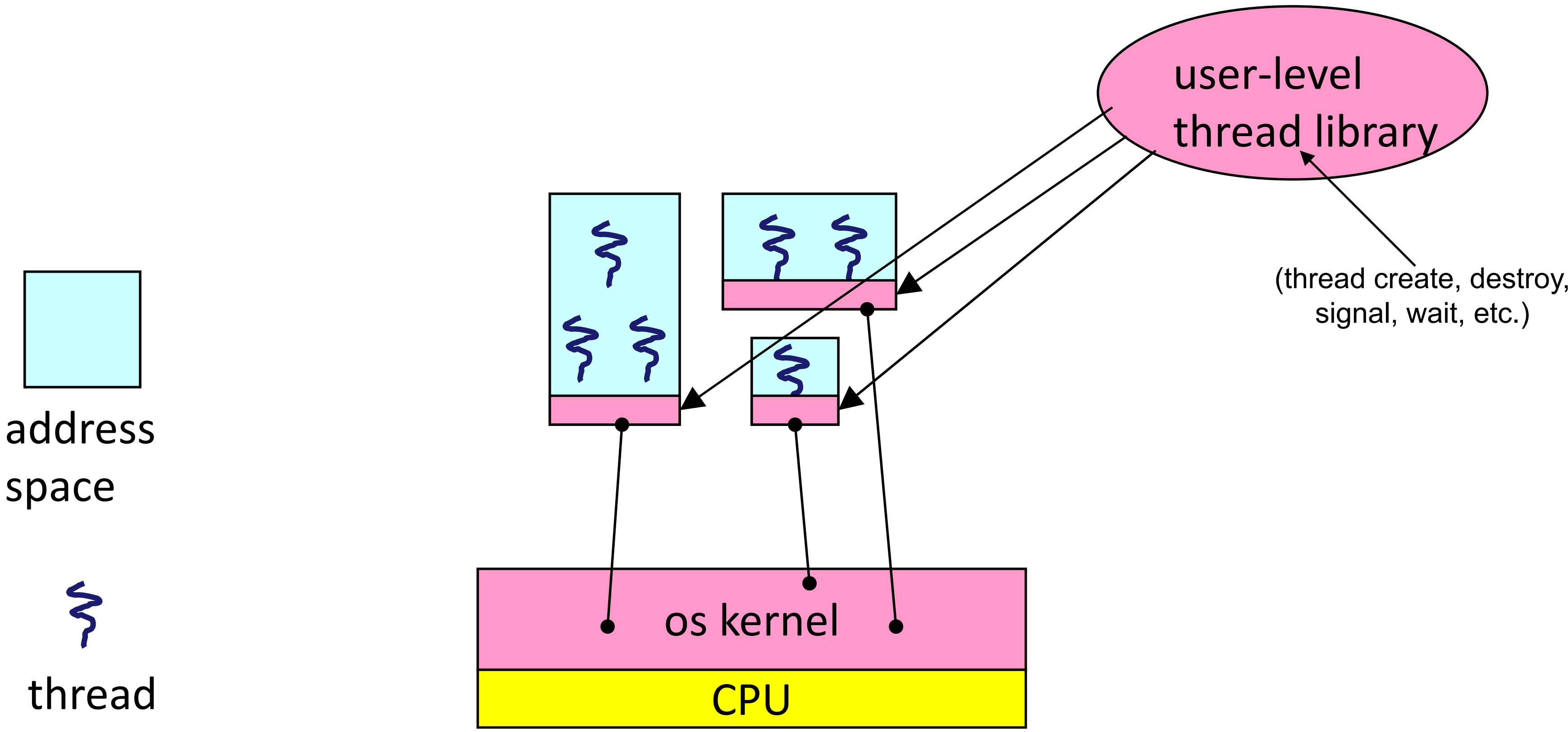
# Kernel threads

- OS now manages threads *and* processes / address spaces
  - sthread 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

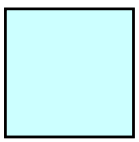
# “Where do threads come from?” (Part 2)

- 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

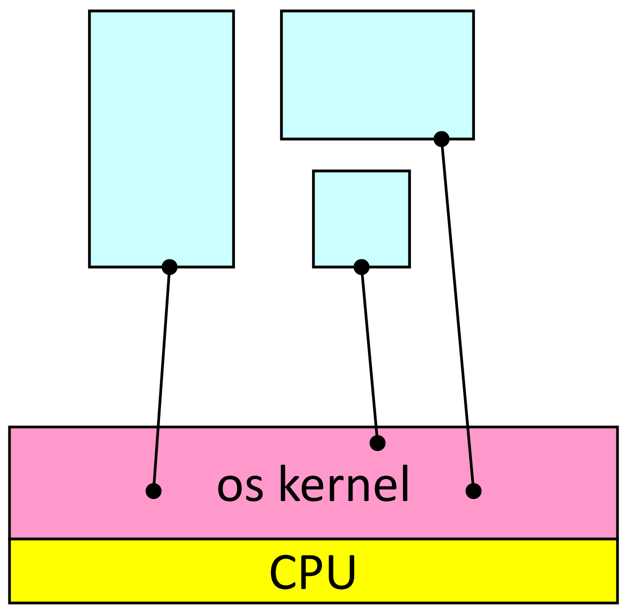


# User-level threads: what the kernel sees

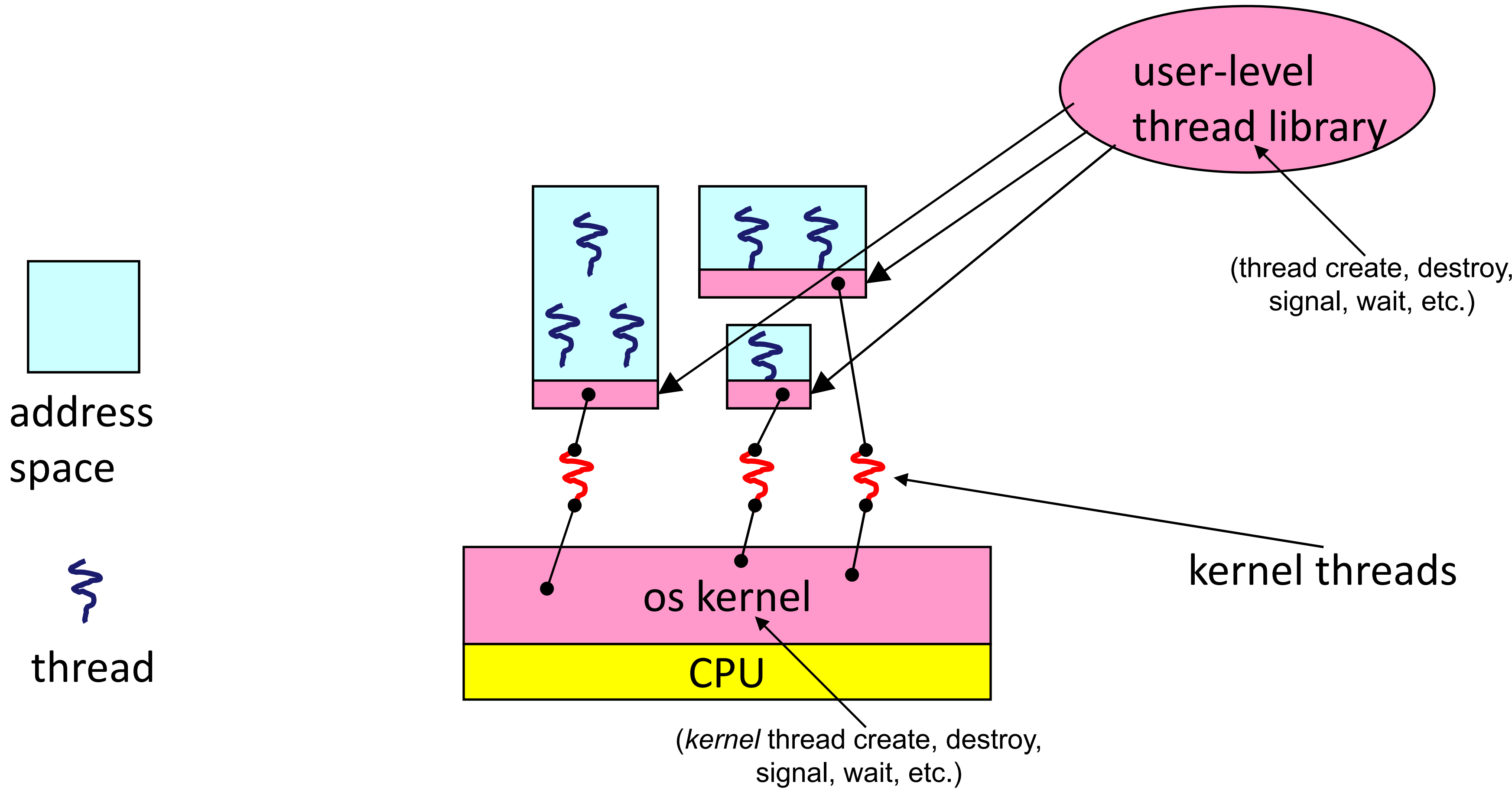


address space

thread



# User-level threads: the full story



# 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

# Summary

- You often 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