CS202 – Advanced Operating Systems

Threads

February 3, 2025

Check your understanding

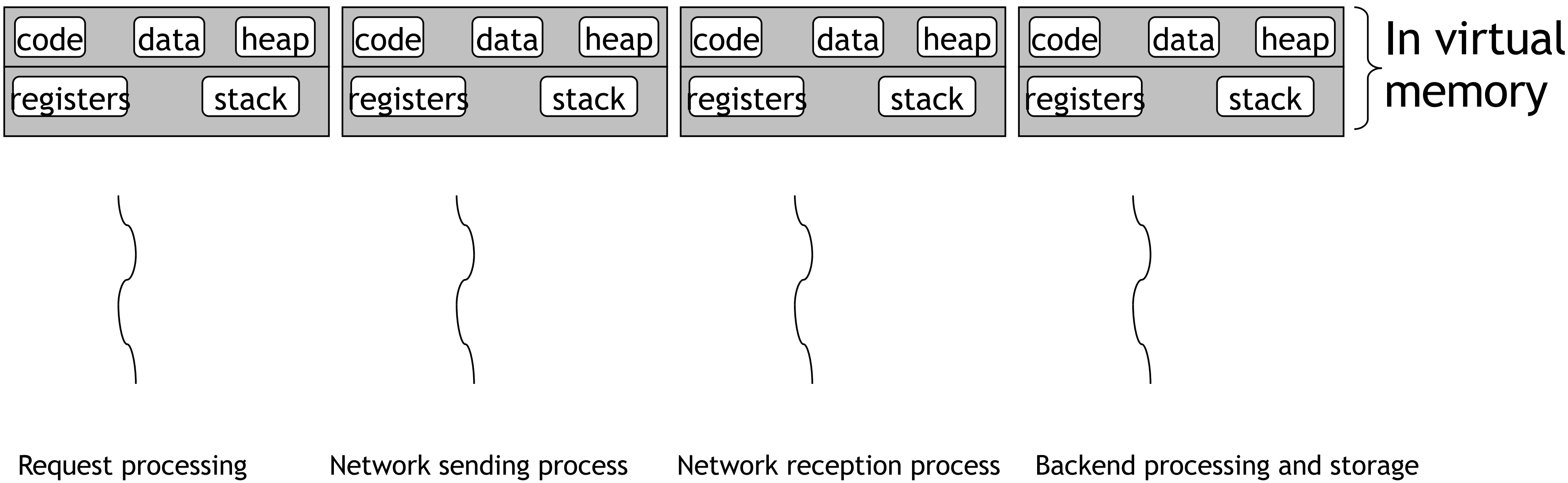
- True or False: a non-preemptive CPU scheduler can be invoked on every mode switch (i.e., trap or interrupt)
  - No, cannot preempt a running process until it gives up the CPU (I/O)
- True or False: we should schedule CPU-bound processes by giving them a higher priority because they will use the CPU
  - No, we typically want to bias higher priorities toward I/O bound processes since they will be more responsive and get out of the way
- How is scheduling related information stored?
  - In queues of Process Control Blocks for each state (running, ready) – and Thread Control Blocks for threads (more tonite)

Advantages of Threads

- Improve Responsiveness
  - Ideally, a thread is always ready
- Resource Sharing
  - All the stuff is easily accessible
- Economy of Resources
  - Thread resources are cheaper than process resources
- Utilization of Multiprocessors
  - Get all of them running

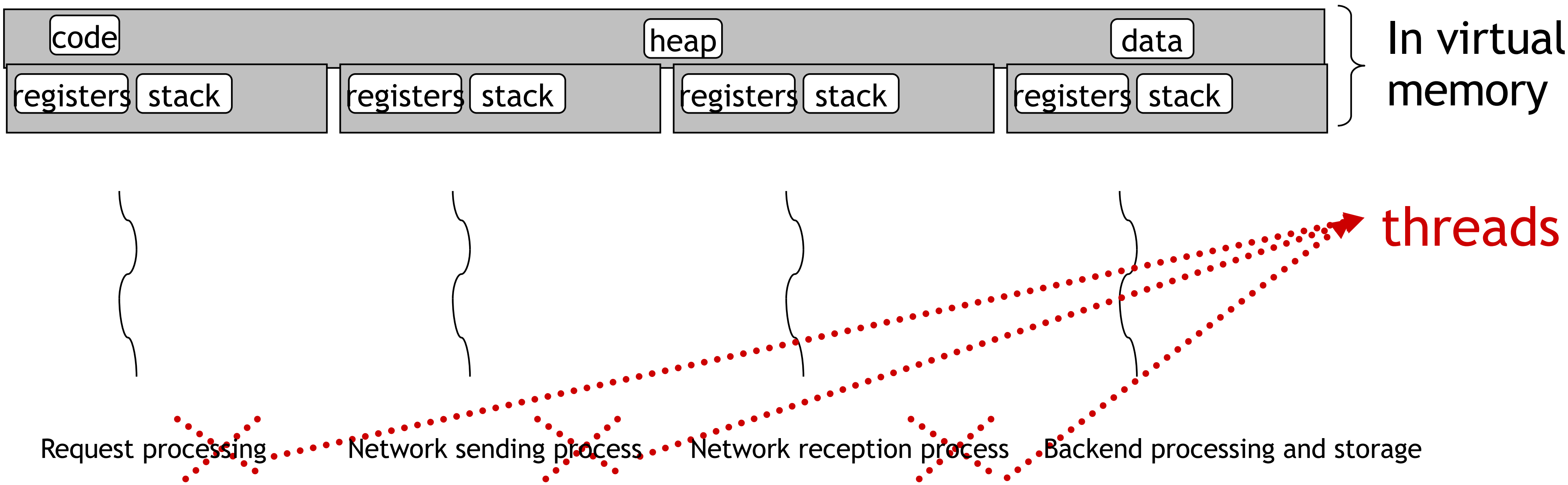
Old Approach:
Multiple Cooperating Processes

E.g., Server Process



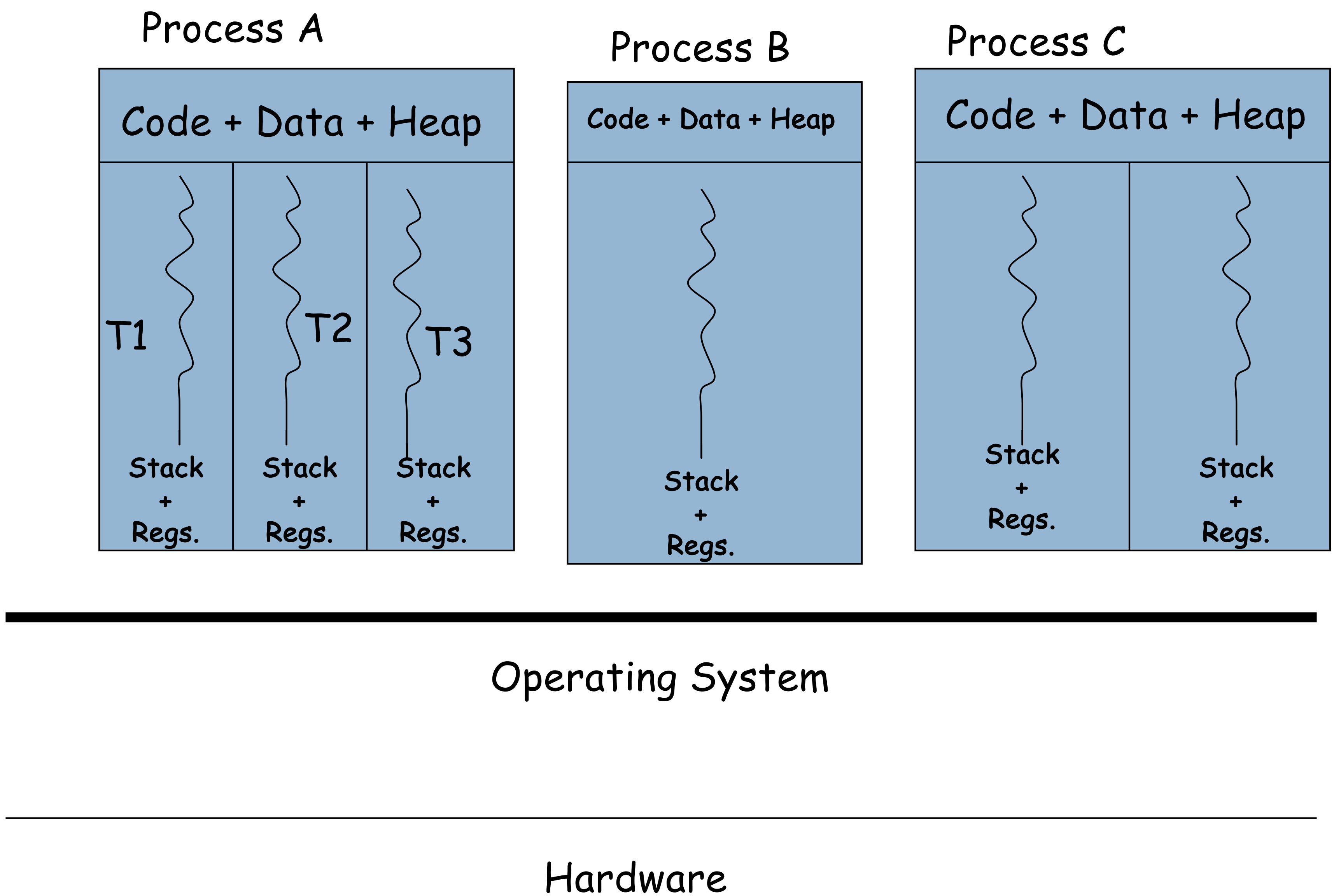
New Approach: Multiple Cooperating Threads in One Process

Share!

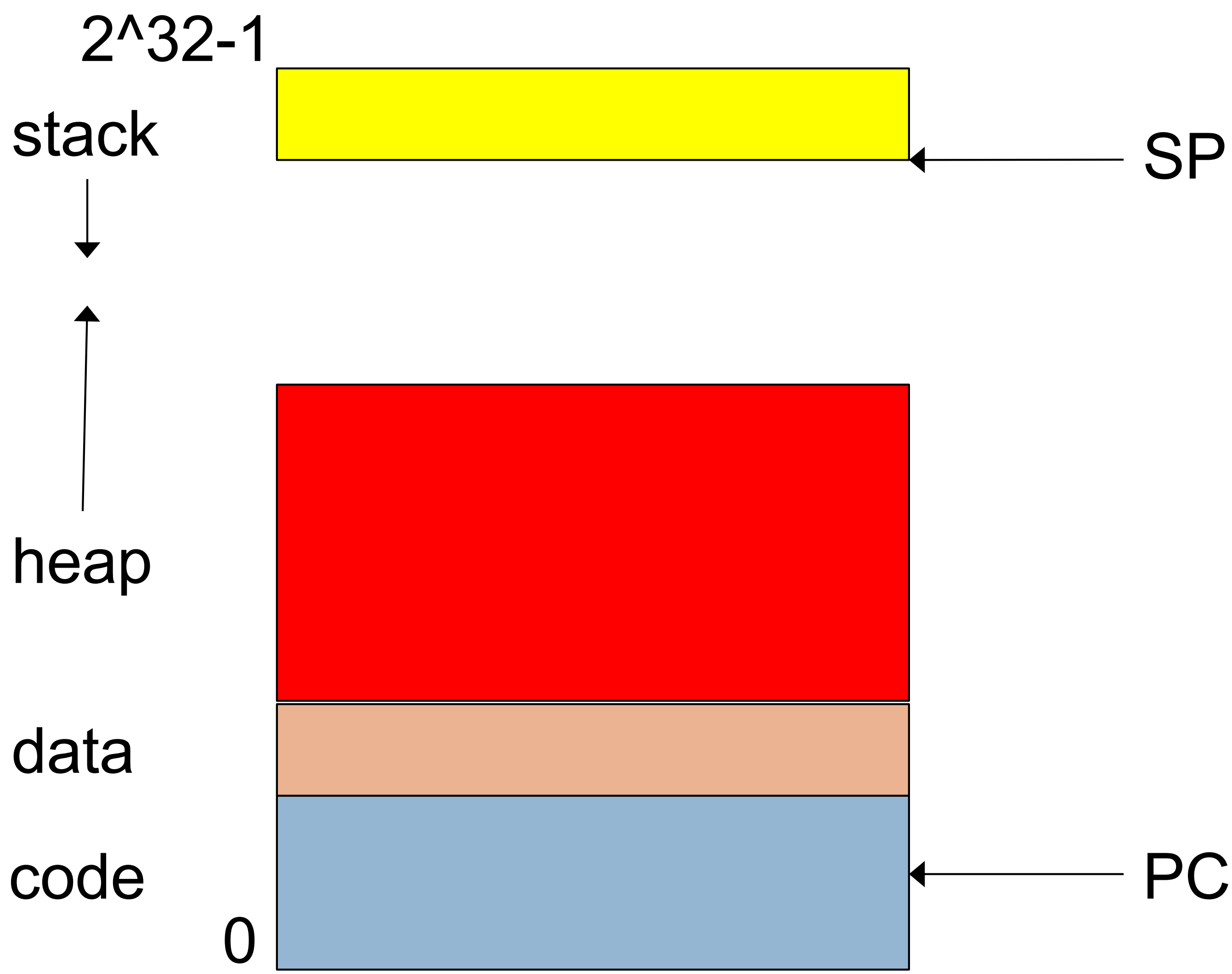


- Share code, data, heap in same address space
  - Only registers and stack must be per thread
  - Why?

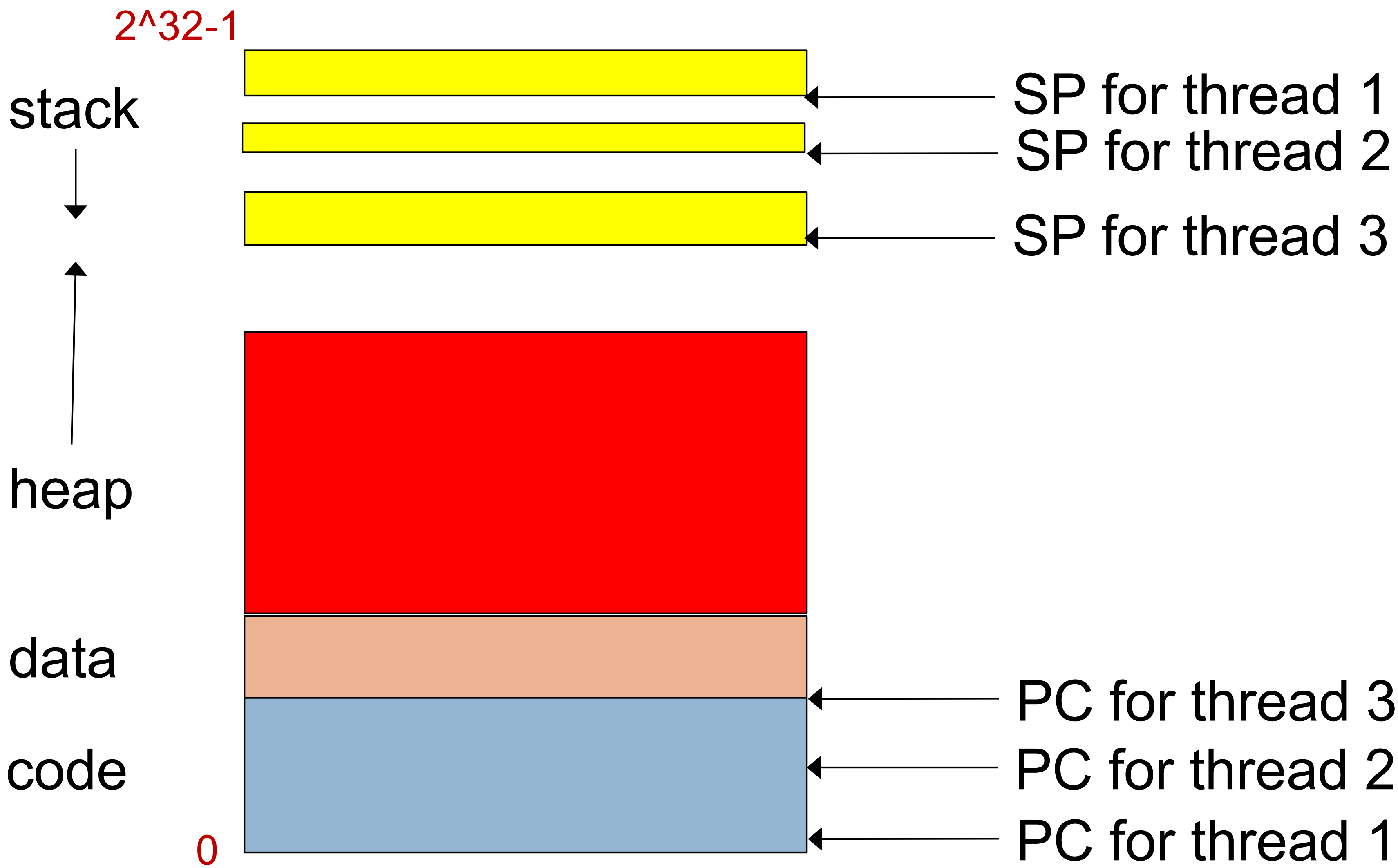
Threads


Process A
Code + Data + Heap
T1
T2
T3
Stack + Regs.
Stack + Regs.
Stack + Regs.
Process B
Code + Data + Heap
Stack + Regs.
Process C
Code + Data + Heap
Stack + Regs.
Stack + Regs.
Operating System
Hardware

(Old) Process Address Space



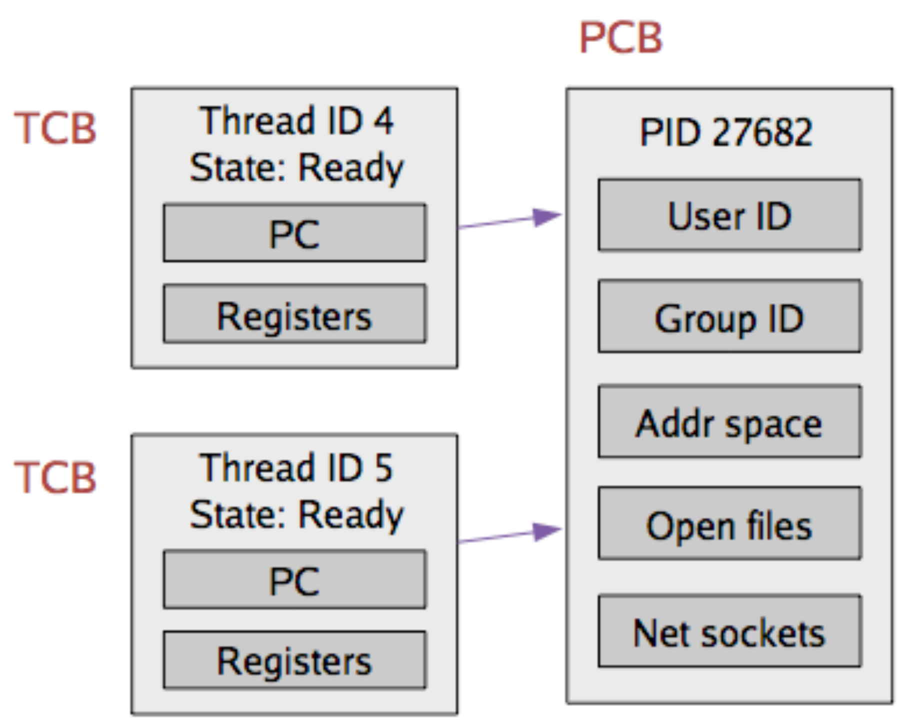
(New) Address Space w/ Threads



All threads in a process share the same address space

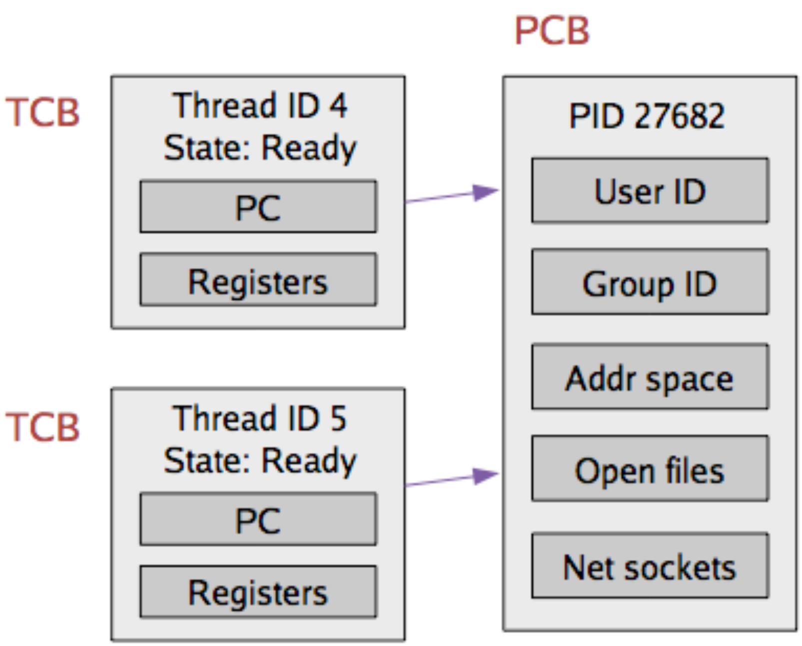
Implementing Threads

- Given what we know about processes, implementing threads is “easy”
- Idea: Break the PCB into two pieces:
  - Thread-specific stuff: Processor state
  - Process-specific state: Address space and OS resources (e.g., open files)



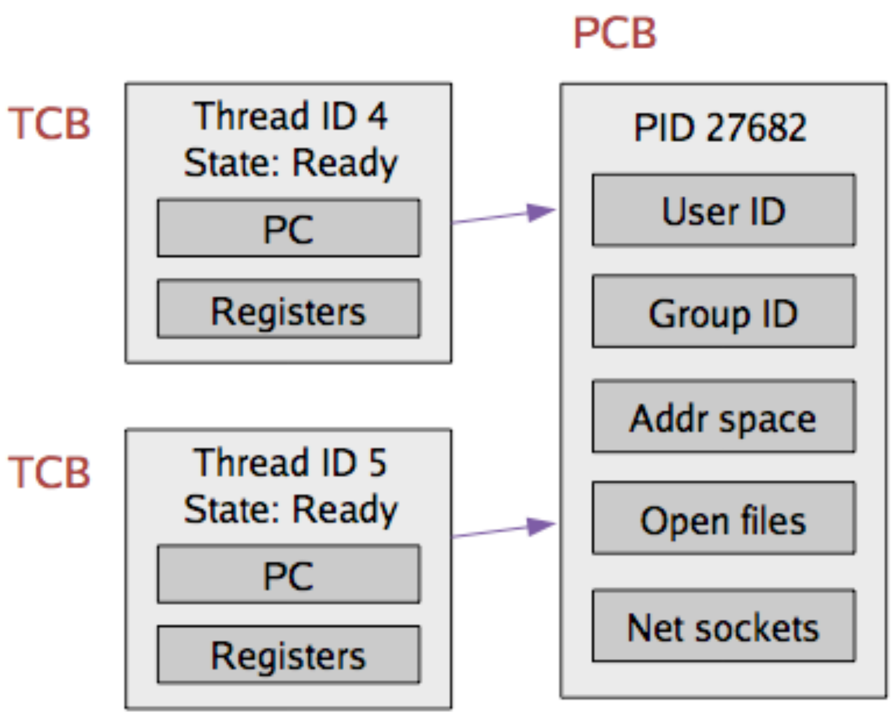
Thread Control Block (TCB)

- TCB contains info on a single thread
  - Thread id
  - Scheduling state
  - H/W context (registers)
  - A pointer to corresponding PCB
- PCB contains info on the containing process
  - Address space and OS resources, but NO processor state!



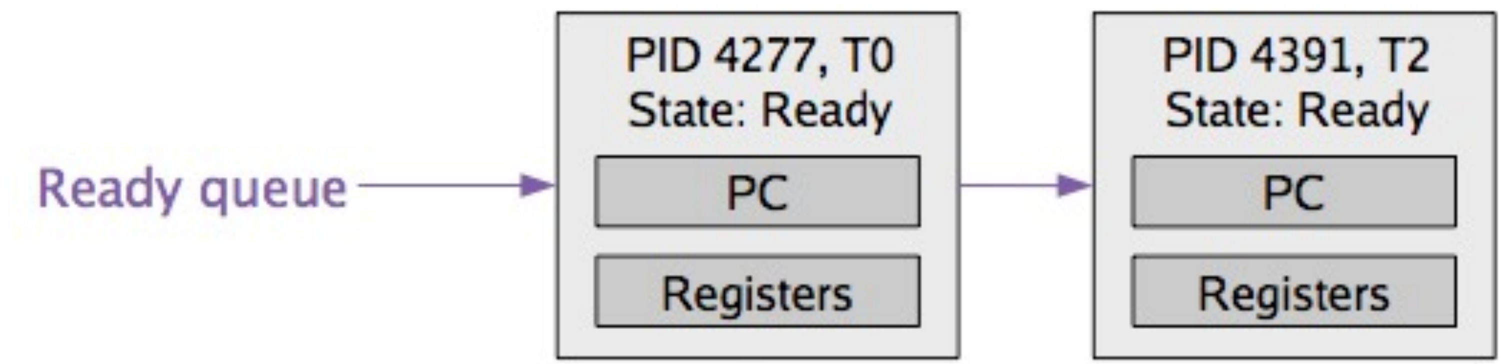
Thread Control Block (TCB)

- TCBs are smaller and cheaper than PCBs
  - E.g., For some recent version of Linux:
    - Linux TCB (thread_struct) has 24 fields
    - Linux PCB (task_struct) has 106 fields



Context Switching

- TCB is now the unit of a context switch
  - Ready queue, wait queues, etc. now contain pointers to TCBs
  - Context switch causes CPU state to be copied to/from the TCB



- Switch between two threads of the same process
  - No need to change address space
    - No TLB flush
- Switch between two threads of different processes
  - Must change address space, causing cache and TLB pollution

Security

- What about security?
- What happens when a bug occurs
  - in one process of a set of cooperating processes?
  - in one thread in a process?
- Depends on bug impact
  - Crash
  - Exploit

- Good news is that new hardware features are being developed to obtain some isolation among threads in the same process

Threading Models

- Programming: **Library or system call interface**
  - User-Space Threading
    - Thread management support in user-space library
    - Linked into your program
  - Kernel Threading
    - Thread management support in the kernel
    - Invoked via system call

User-Space Threads

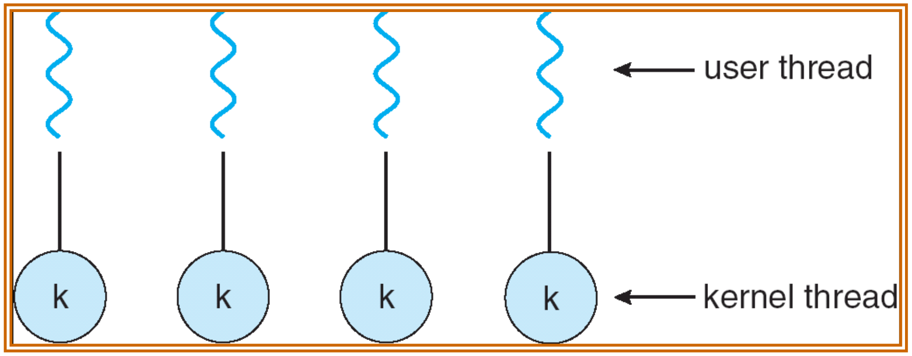
- Thread management support in user-space library
  - Sets of functions for creating, invoking, and switching among threads
- Linked into your program
  - Thread libraries
- Examples
  - POSIX Threads (PThreads)
  - Win32 Threads
  - Java Threads

Kernel Threads

- Thread management support in kernel
  - Sets of system calls for creating, invoking, and switching among threads
- Supported and managed directly by the OS
  - Thread objects in the kernel
- Nearly all OS support a notion of threads
  - Linux -- thread and process abstractions are mixed
  - Solaris
  - Mac OS X
  - Windows
  - …

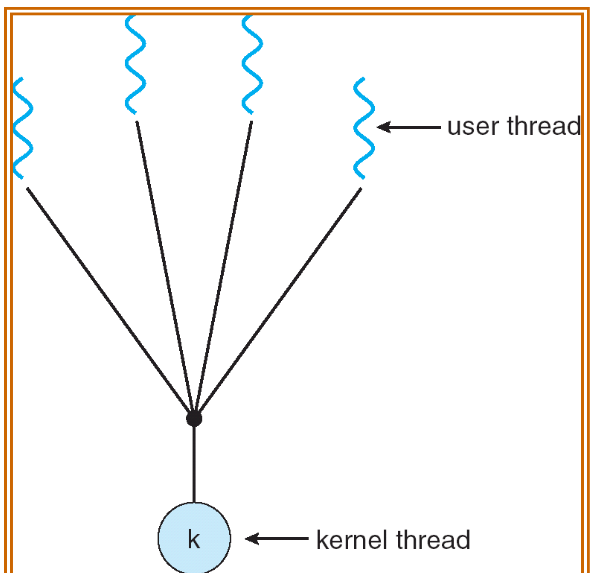
One-to-One Thread Model

- One user-level thread per kernel thread
  - A kernel thread is allocated for every user-level thread
  - Must get the kernel to allocate resources for each new user-level thread
- How does it work?
  - Create new thread, including system call to kernel
  - Upon *yield*, switch to another thread in system
    - Kernel is aware
  - Upon *wait*, another thread in the process may run
    - Only the single kernel thread is blocked
    - Kernel is aware there are other options in this process



Many-to-One Thread Model

- Many user-level threads correspond to a single kernel thread
  - Kernel is not aware of the mapping
  - Handled by a thread library
- How does it work?
  - Create and execute a new thread
  - Upon *yield*, switch to another thread in the same process
    - Kernel is unaware
  - Upon *wait*, all threads are blocked
    - Kernel is unaware there are other options
    - Can't wait and run at the same time



SCHEDULER ACTIVATIONS

Context

- Neither user-level threads nor kernel-level threads work ideally
  - User-level threads have application information
    - They are also cheap
    - But not visible to kernel
  - Kernel-level threads
    - Expensive
    - Lack application information

Idea

- Abstraction: threads in a shared address space
  - Others possible?
- Can be implemented in two ways
  - Kernel creates and dispatches threads
    - Expensive and inflexible
  - User level
    - One kernel thread for each virtual processor

User level on top of kernel threads

- Each application gets a set of virtual processors
  - Each corresponds to a kernel level thread

- User level threads implemented in user land
  - Any user thread can use any kernel thread (virtual processor)
    - Fast thread creation and switch – no system calls
    - Fast synchronization!

  - What happens when a thread blocks?
  - Any other issues?

Goals (from paper)

- Functionality
  - No processor idles when there are ready threads
  - No priority inversion (high priority thread waiting for low priority one) when it’s ready
  - When a thread blocks, the processor can be used by another thread
- Performance
  - Closer to user threads than kernel threads
- Flexibility
  - Allow application-level customization or even a completely different concurrency model

Problems

- User thread does a blocking call?
  - Application loses a processor!
- Scheduling decisions at user and kernel not coordinated
  - Kernel may de-schedule a thread at a bad time (e.g., while holding a lock)
  - Application may need more or less computing
- Solution?
  - Allow coordination between user and kernel schedulers

Scheduler activations

- Allow user-level threads to act like kernel-level threads
  - Via virtual processors
- Notify user-level scheduler (runtime) of relevant kernel events
  - Like what?
- Provide space in kernel to save context of user thread when kernel stops it
  - E.g., for I/O or to run another application

Kernel upcalls

- New processor available
  - Reaction? Runtime picks user thread to use it
- Activation blocked (e.g., for page fault)
  - Reaction? Runtime runs a different thread on the activation
- Activation unblocked
  - Activation now has two contexts
  - Running activation is preempted – why?
- Activation lost processor
  - Context remapped to another activation
- What do these accomplish?

Runtime->Kernel

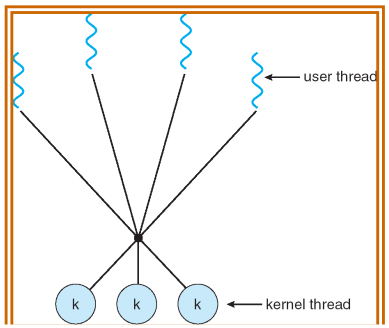
- Informs kernel when it needs more resources, or when it is giving up some
- Could cause the kernel to preempt low priority threads
  - Only kernel can preempt
- Almost everything else is user level!
  - Performance of user level, with the advantages of kernel threads!

Preemptions in critical sections

- Runtime checks during upcall whether preempted user thread was running in a critical section
  - Continues the user thread using a user-level context switch in this case
    - Once lock is released, it switches back to original thread
    - Keep track of critical sections using a hash table of section begin/end addresses

Many-to-Many Thread Model

- A pool of user-level threads maps to a pool of kernel threads
  - Pool sizes can be different (kernel pool is no larger)
  - A kernel thread is pool is allocated for every user-level thread
  - No need for the kernel to allocate resources for each new user-level thread
- How does it work?
  - Create new thread (may map to kernel thread dynamically)
  - Upon *yield*, switch to another thread in system
    - Kernel is aware
  - Upon *wait*, another thread in the process may run
    - If a kernel thread is available to be scheduled to that process
    - Kernel is aware of the mapping between process threads and kernel threads



Conclusions

- Today was a review of threading
- A program may be run using multiple threads
  - Threads can use the CPU and other resources more efficiently
- Threads share the heap, code, and global data
  - But have their own stacks and registers
  - Context switching requires a per-thread data structure called a thread control block
- Discussed scheduler activations
  - Many-to-many thread model managed by user space
  - User threads make scheduling decisions given kernel thread resources

Questions