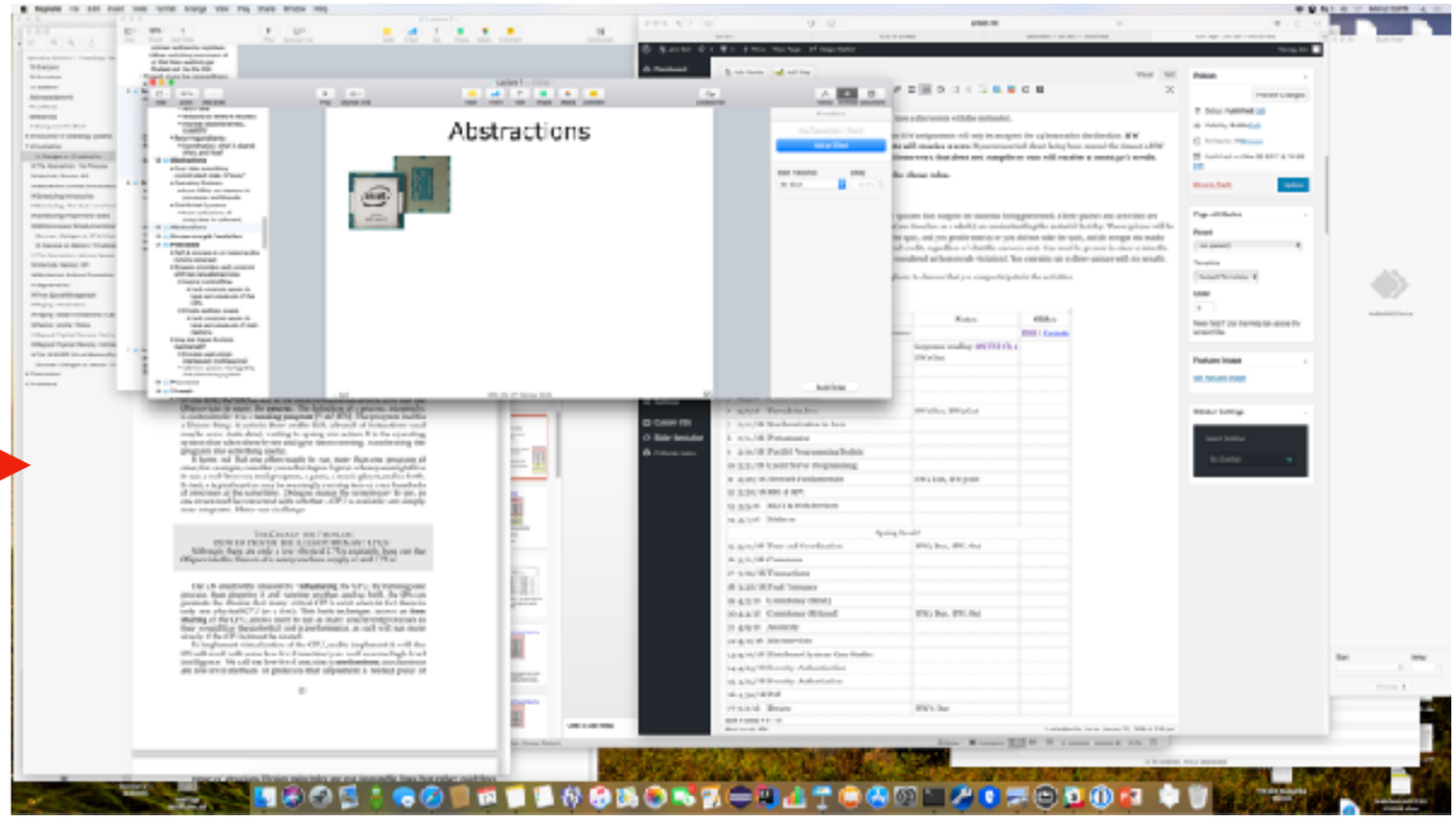
Processes

CS 475, Spring 2018
Concurrent & Distributed Systems

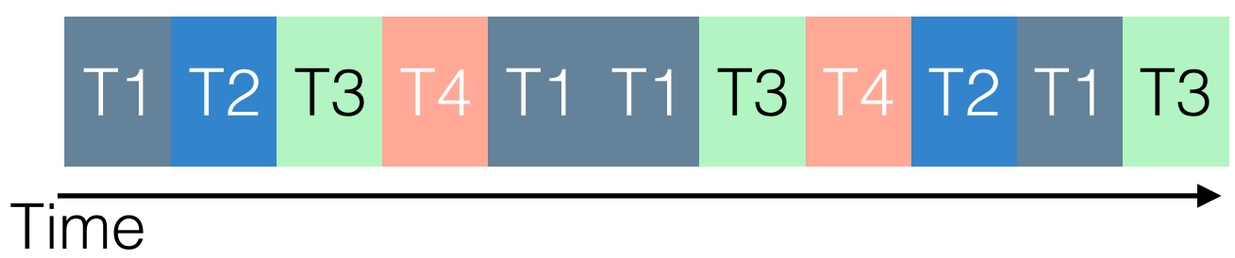
Review: Abstractions



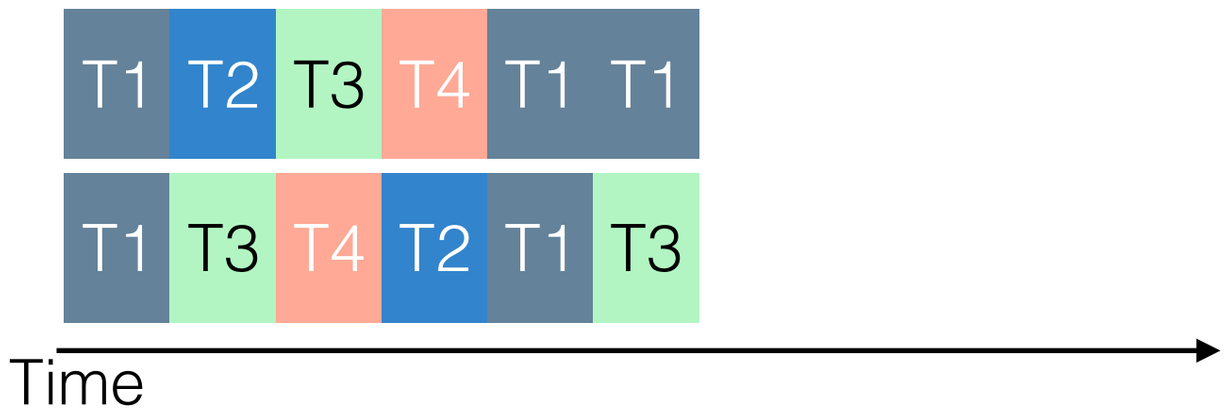
Review: Concurrency & Parallelism

4 different things: T1 T2 T3 T4

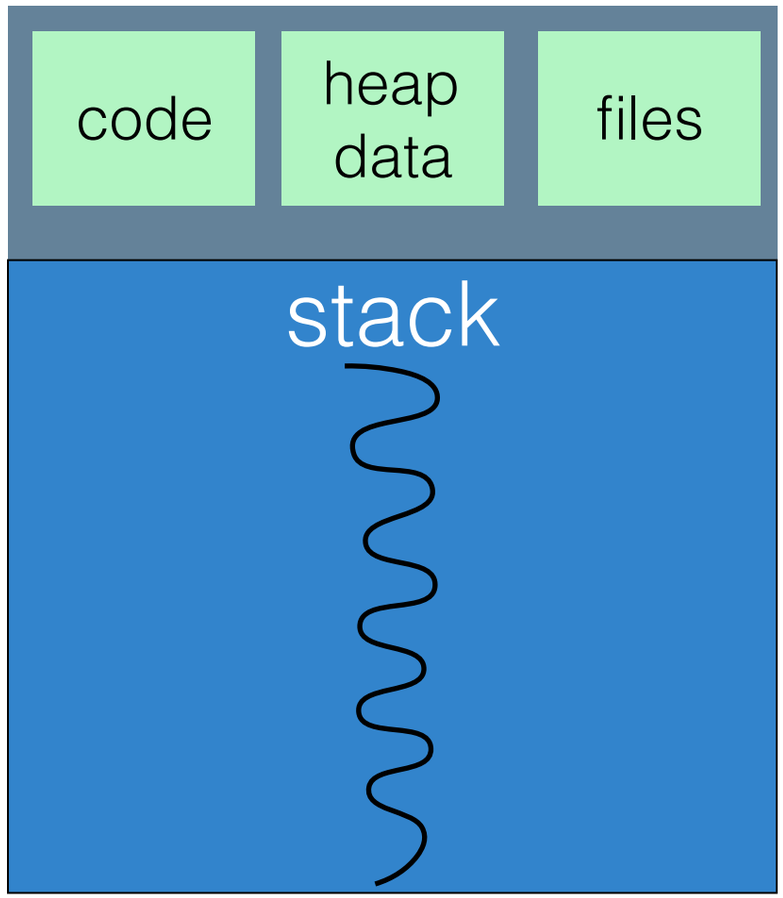
Concurrency:
(1 processor)



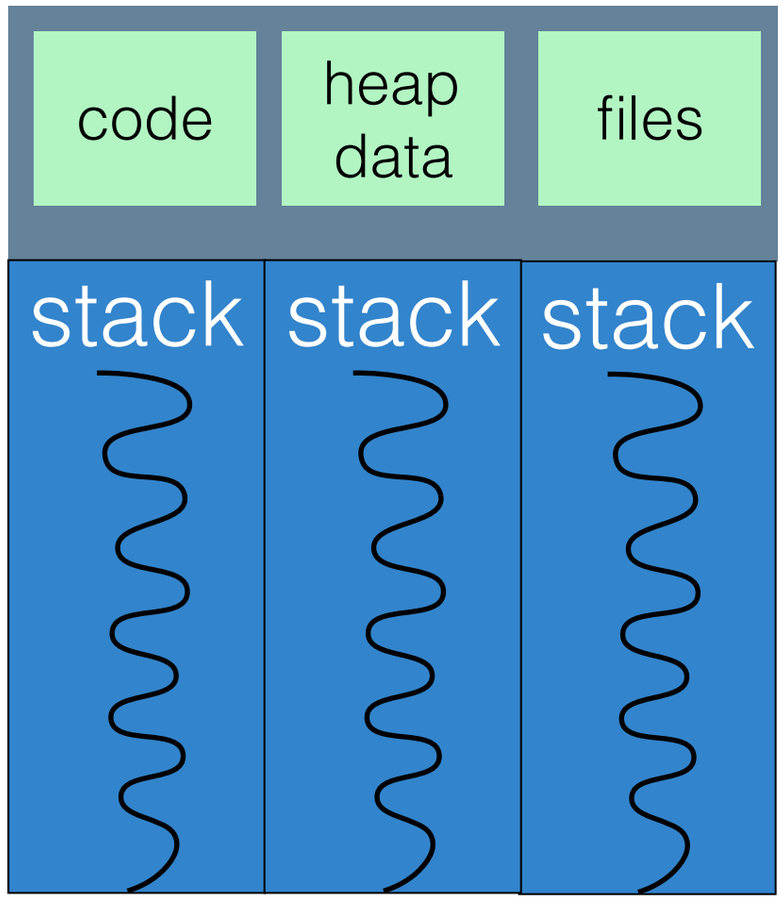
Parallelism:
(2 processors)



Review: Processes vs Threads



Single-Threaded Process



Multi-Threaded Process

Review: Processes vs Threads

- Context Switching
  - Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
    - When switching processes, **all** of that data needs to get flushed out (by the OS)
- Threads share the same address space: no need to do this switch

Today

- How does the OS manage concurrency?
- Processes!
  - A little bit of OS
  - Basic concepts
  - Linux internals
  - Context switching
  - Discuss HW1
- Additional readings:
  - OS TEP Ch 4, 6
  - OS TEP Ch 5 might be useful for HW1
- Slide acknowledgements:
  - OS Concepts 9th Ed, Silberschatz, Galvin & Gagne

Process as a concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
  - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

Process as a concept

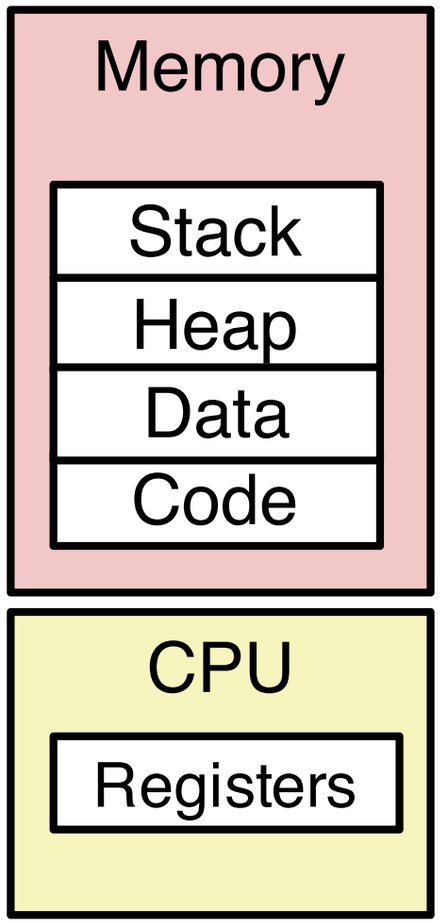
- Program is passive entity stored on disk (executable file), process is active
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process State

- As a process executes, it changes state
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

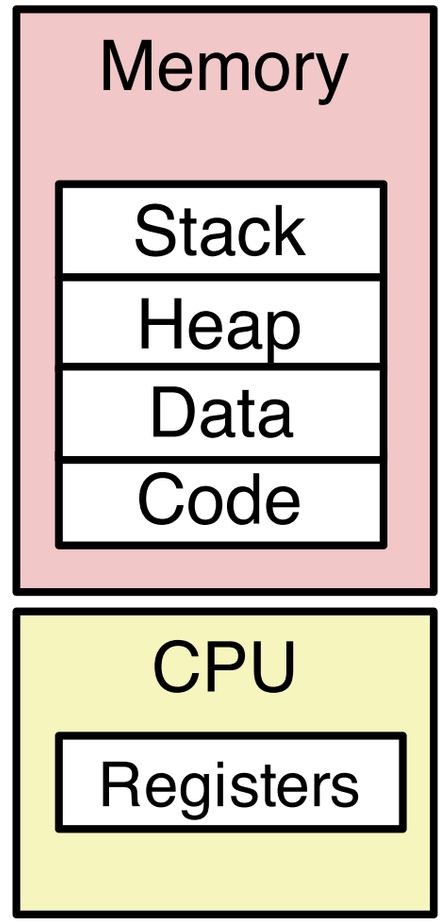
Process Representation

- A process has some mapping into the physical machine (machine state)
- Provide two key abstractions to programs:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching
  - Private address space
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called virtual memory

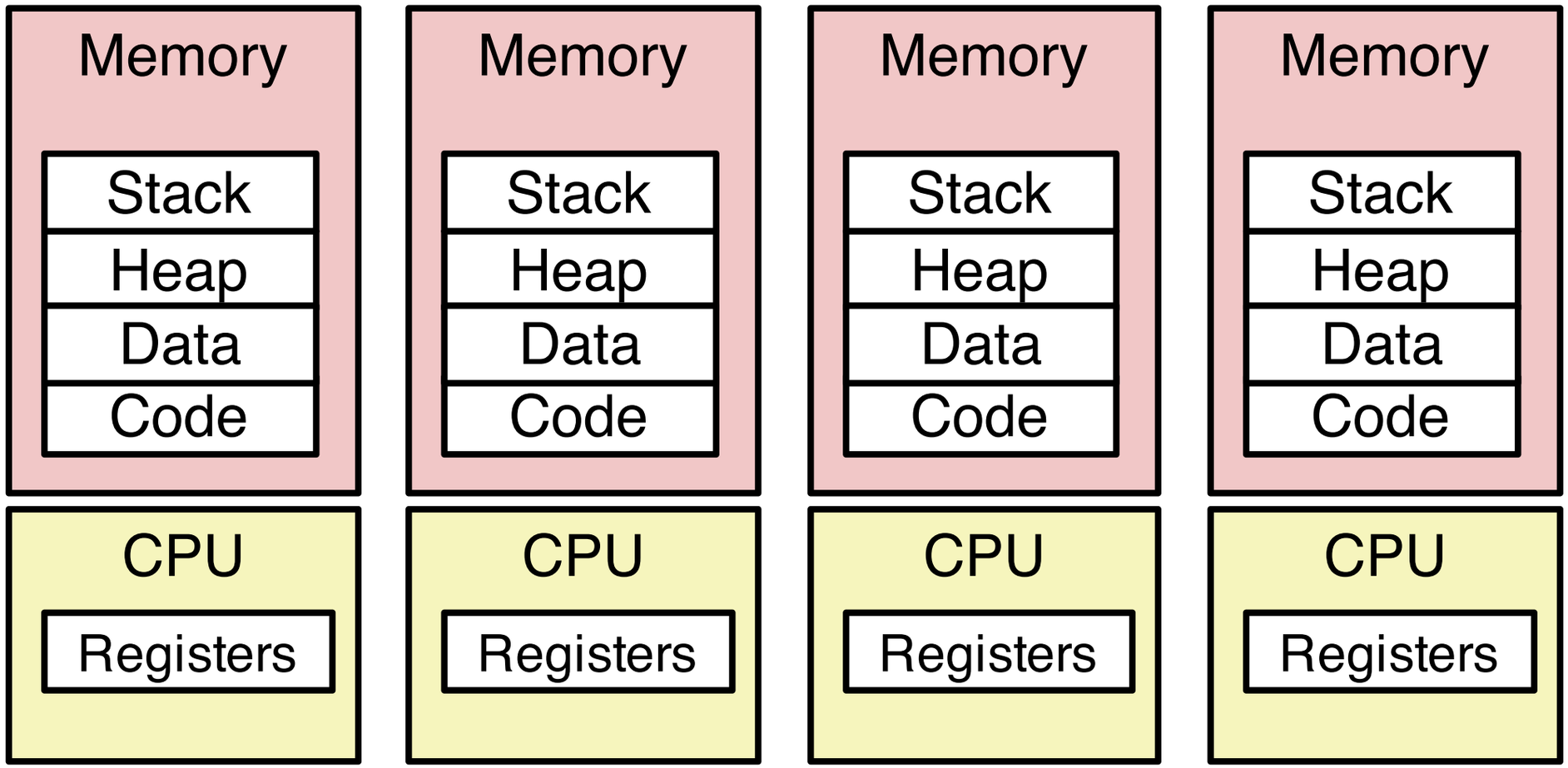


Process Control Block (PCB)

- The OS has one big data structure that stores all of the information about each process, including:
  - Process state – running, waiting, etc
  - Program counter – location of instruction to next execute
  - CPU registers – contents of all process-centric registers
  - CPU scheduling information- priorities, scheduling queue pointers
  - Memory-management information – memory allocated to the process
  - Accounting information – CPU used, clock time elapsed since start, time limits
  - I/O status information – I/O devices allocated to process, list of open files

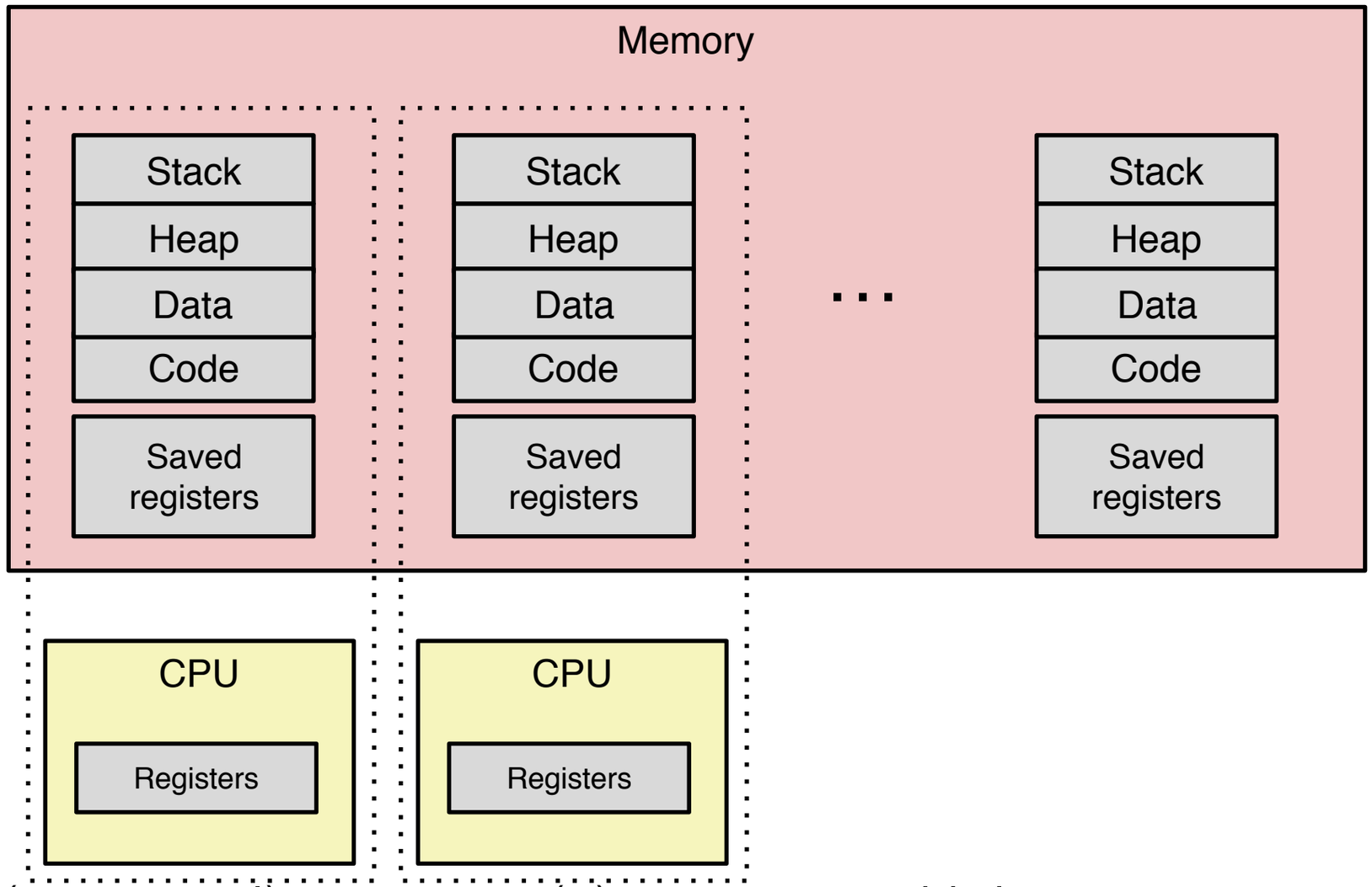


Virtual View of Multiprocessing



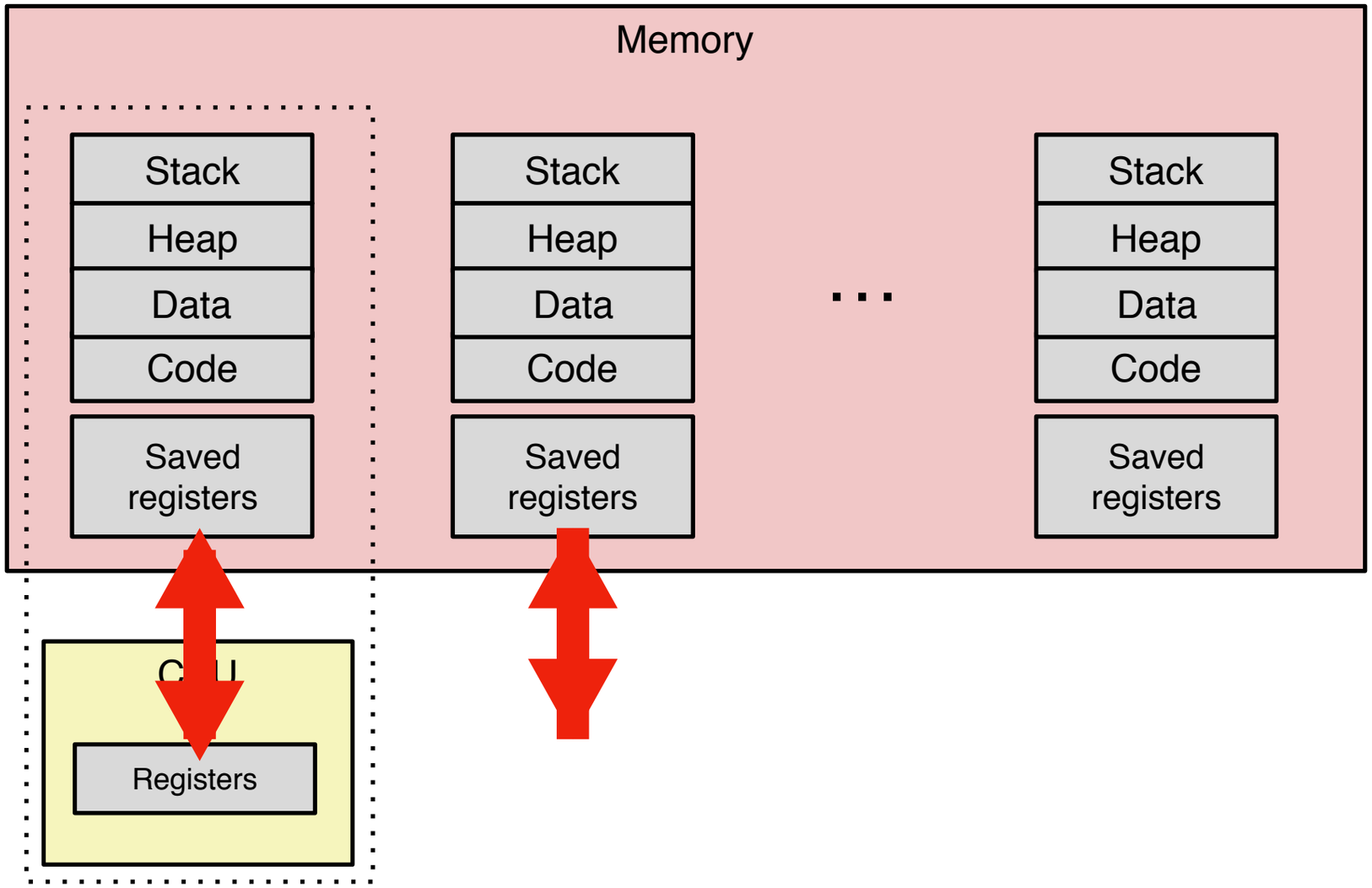
- Goal: make it look like all processes are running at once, with infinite processors

Actual View of Multiprocessing



- Single (or several) processor(s) execute multiple processes **concurrently**
  - Executions are interleaved (multitasking)
  - Address spaces for each process managed by virtual memory system (see OS)
  - Registers for non-executing processes saved in memory

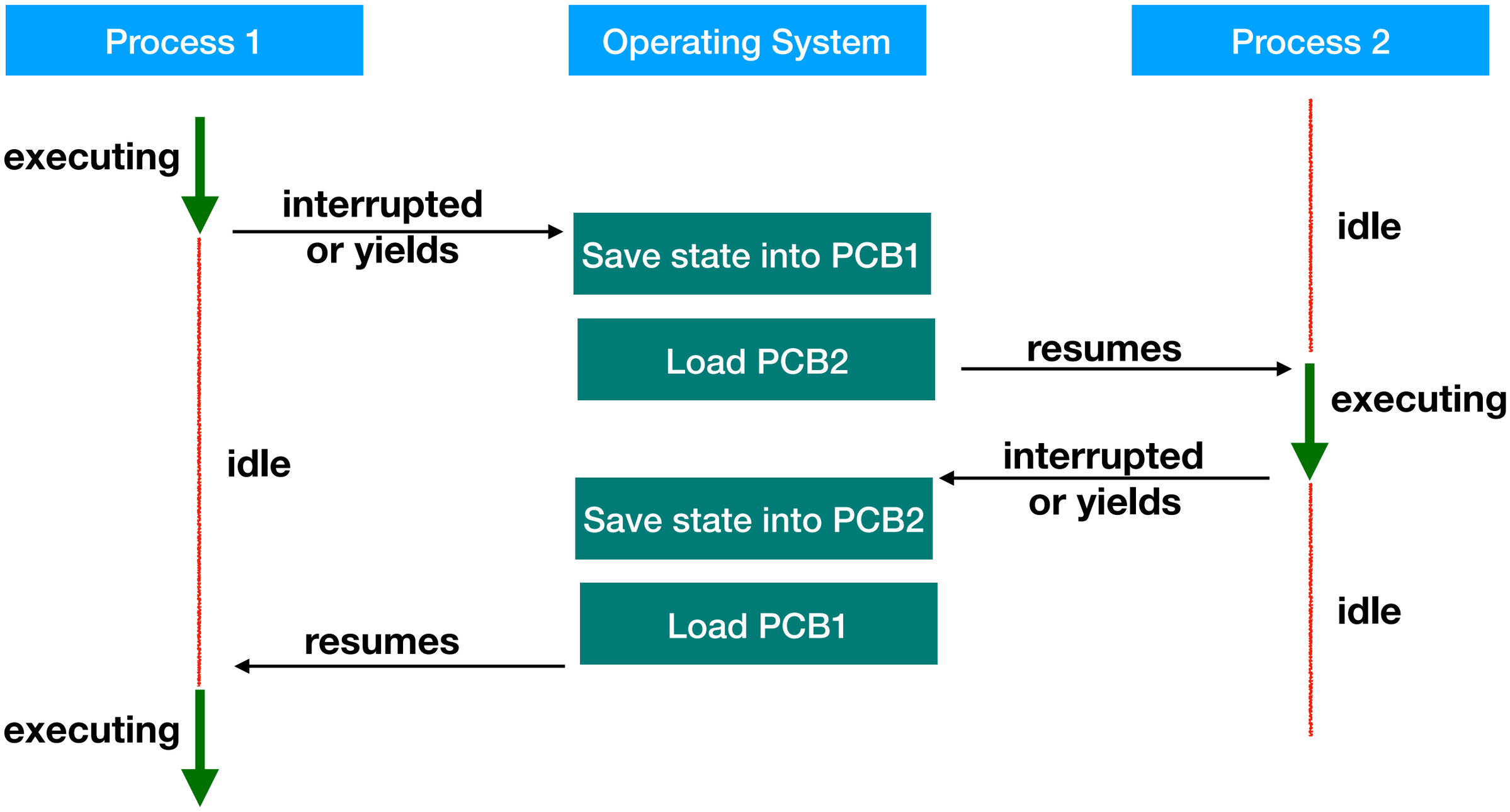
CPU Switching from Process to Process



Context Switching

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
    - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
    - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

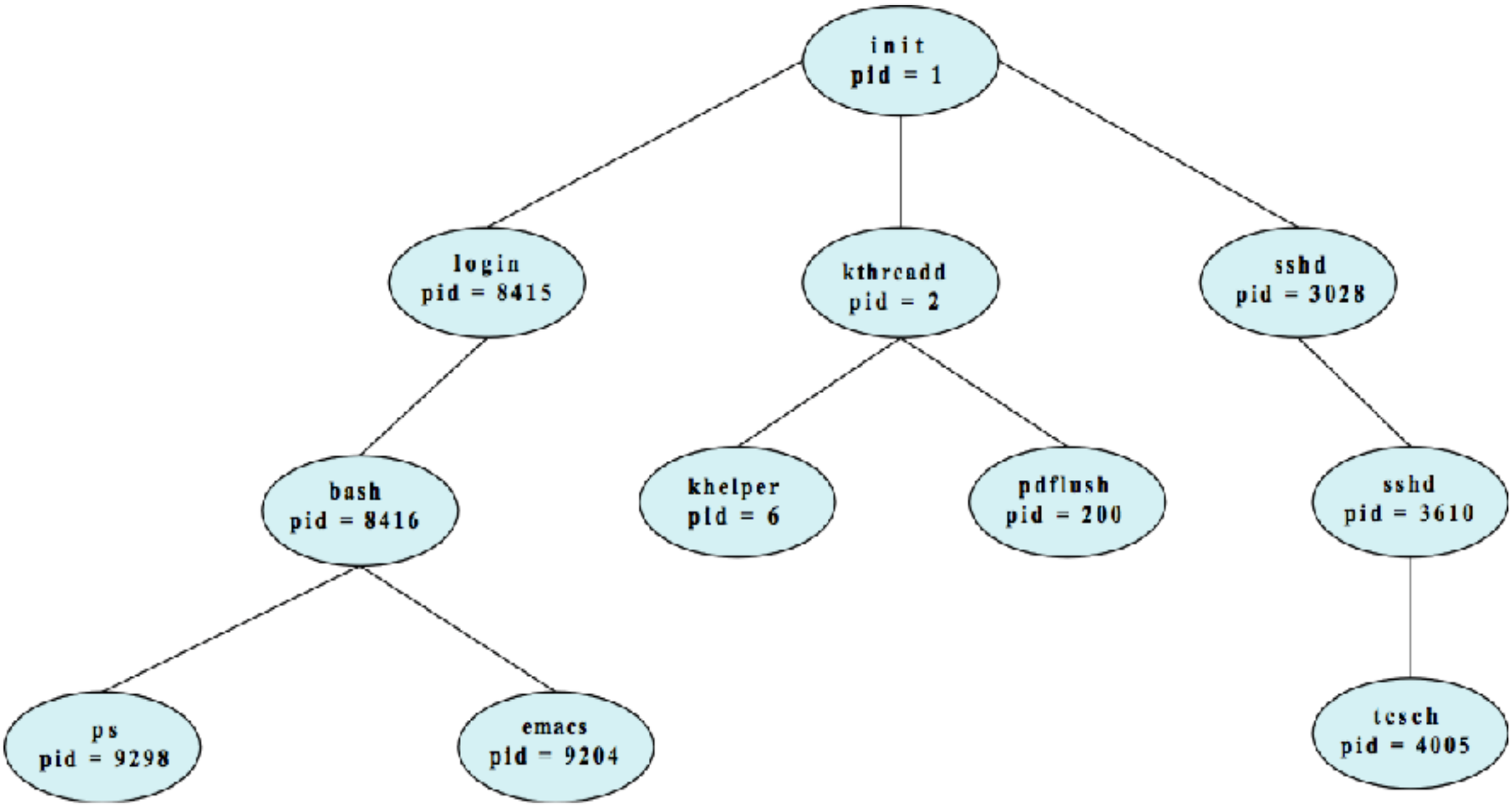
Context Switching



Creating Processes

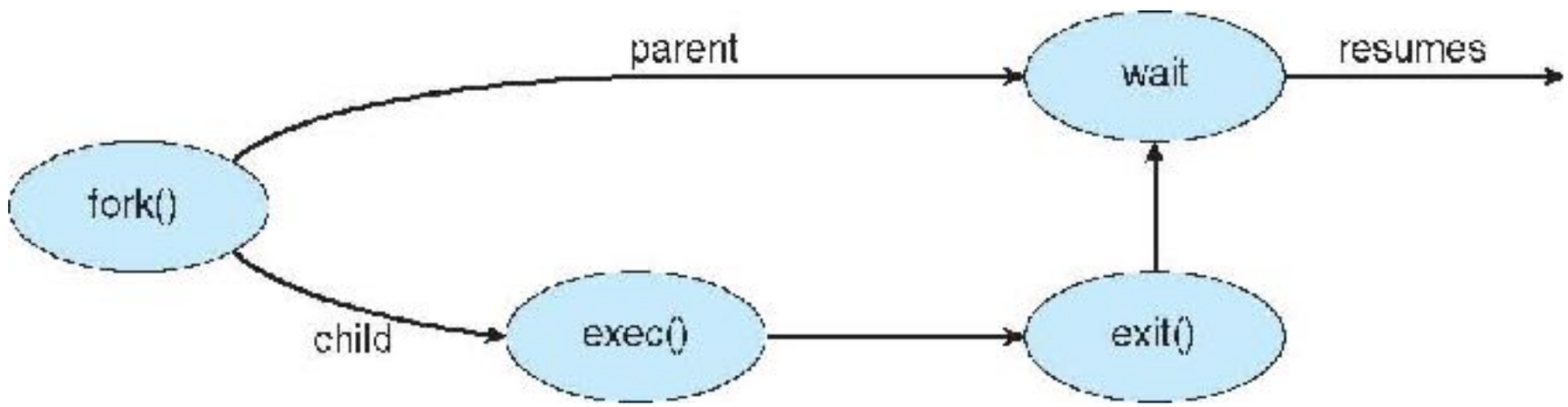
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
  - Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Tree



Creating Processes

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program



Fork

- `int fork(void)`
- creates a new process (child process) that is identical to the calling process (parent process)
- returns 0 to the child process
- returns child's `pid` to the parent process

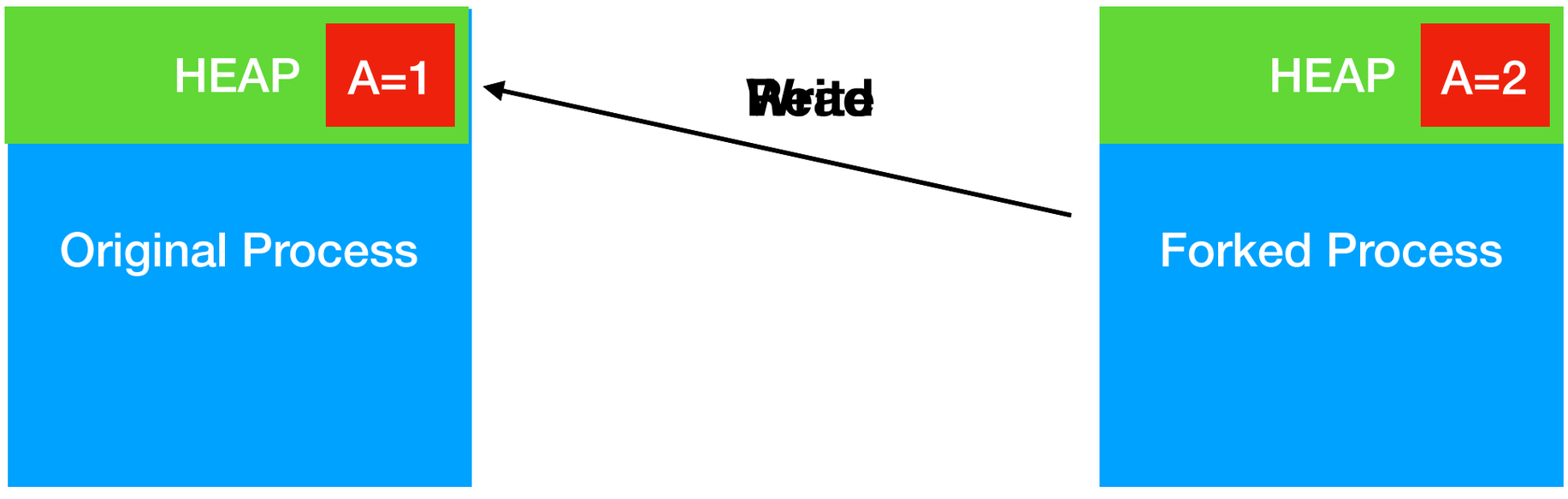
```
if (fork() == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called once but returns twice

Fork

- Under the hood: is NOT actually copying everything
- Fork'ed memory is "copy-on-write" protected
  - Memory is not copied up front
  - If fork'ed process reads, then OK
  - If fork'ed process writes, then make copy
- Possible from OS' implementation of virtual memory (but that's another course)

Copy on write



Exec

- `int exec(....)`
- Never returns (unless an error)
- Loads the code from some executable (passed as a parameter)
- Re-initializes all of memory of that process to be clean

Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - cascading termination. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
  - The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

  - If no parent waiting (did not invoke `wait()`) process is a zombie
  - If parent terminated without invoking `wait` , process is an orphan

Process Scheduling

- Key for concurrency
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
  - Job queue – set of all processes in the system
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute
  - Device queues – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Checkpoint

Go to socrative.com and select “Student Login” (works well on laptop, tablet or phone)

Room Name: CS475

ID is your @gmu.edu email

Remember: this is a checkpoint for you, it is only graded for attendance

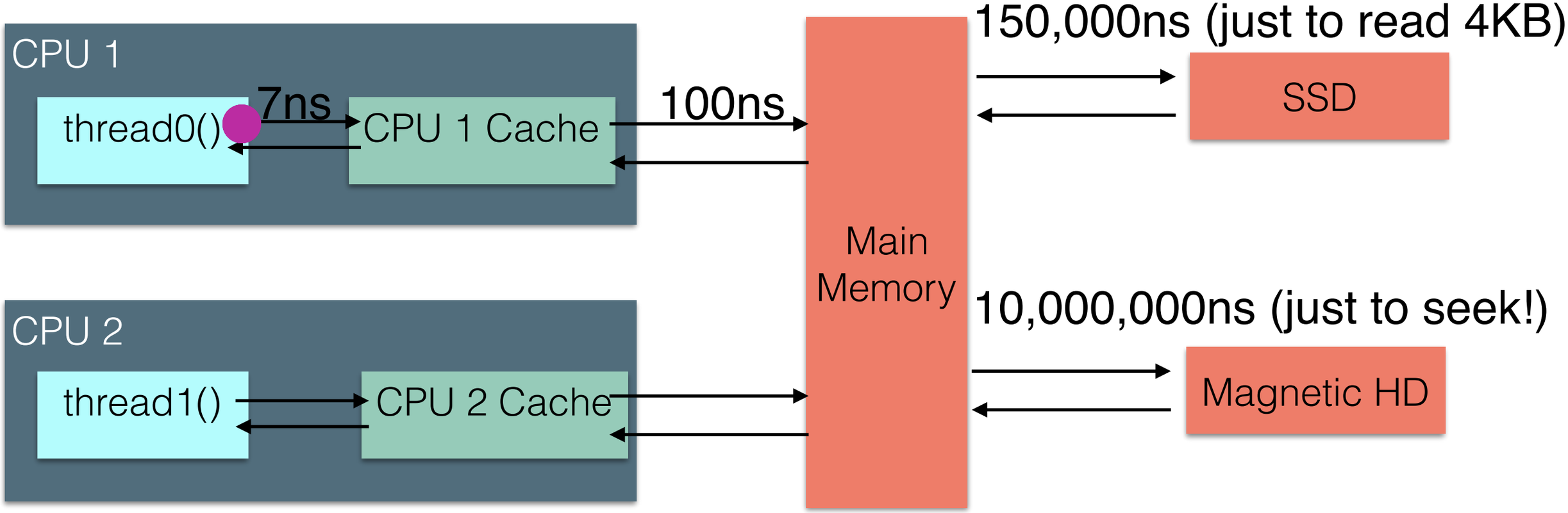
Schedulers

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the degree of multiprogramming

Different Kinds of Processes

- Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; few very long CPU bursts
  - Long-term scheduler strives for good process mix
  - This will come up for distributed systems too!

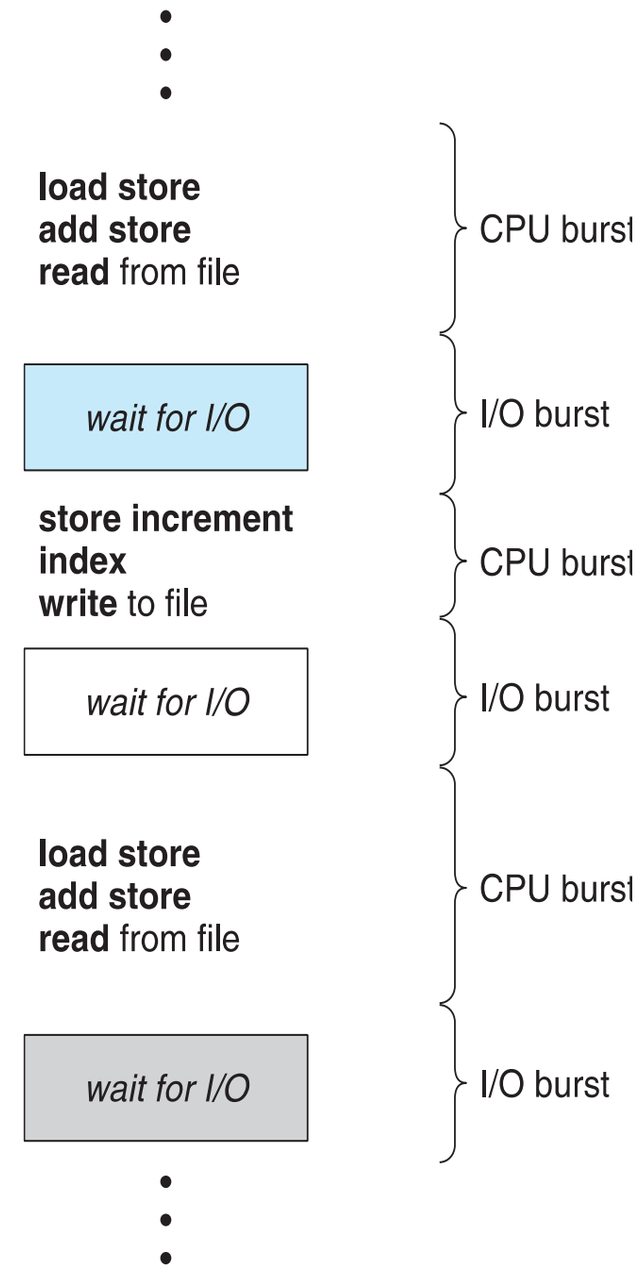
Sidebar: I/O Speeds



Takeaway: it's REALLY important to have other stuff to do while I/O happens

Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern



Basic CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  - Switches from running to waiting state
  - Terminates
  - Switches from running to ready state
  - Switches from waiting to ready

Preemption

- In the "old days" (before me), there were **non-preemptive** schedulers
- Run job from start to end (or until process yields)
- Virtually all schedulers are, in fact, **preemptive**
  - These schedulers can force a process to yield and perform a context switch

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- Ideally we want to minimize on these criteria

FCFS Scheduling

Process	Actual Burst Time
P1	24
P2	3
P3	3

- What if they arrive in this order? P1, P2, P3
  - Waiting time for P1: 0; P2=24; P3=27
  - Average waiting time: 17
- What if they arrive P2, P3, P1?
  - Waiting time for P2: 0, P3: 3, P1: 6
  - Average waiting time: 3
- “Convoy effect” - short processes get stuck behind long processes and have to wait

Obvious fix: "Shortest Job First"

- IF we know how long everything will take, we can optimize on waiting time by running the short jobs first (the second case from FCFS)
- This would be **optimal**
- But: how do we know how long each process will take?

Process	Actual Burst Time
P1	24
P2	3
P3	3

Estimating CPU time

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define: $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n$.

- Commonly, α set to ½
- Preemptive version called shortest-remaining-time-first

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
  - Shortest Job First is priority scheduling where priority is the inverse of predicted next CPU burst time
  - Problem ≡ **Starvation** – low priority processes may never execute
  - Solution ≡ **Aging** – as time progresses increase the priority of the process

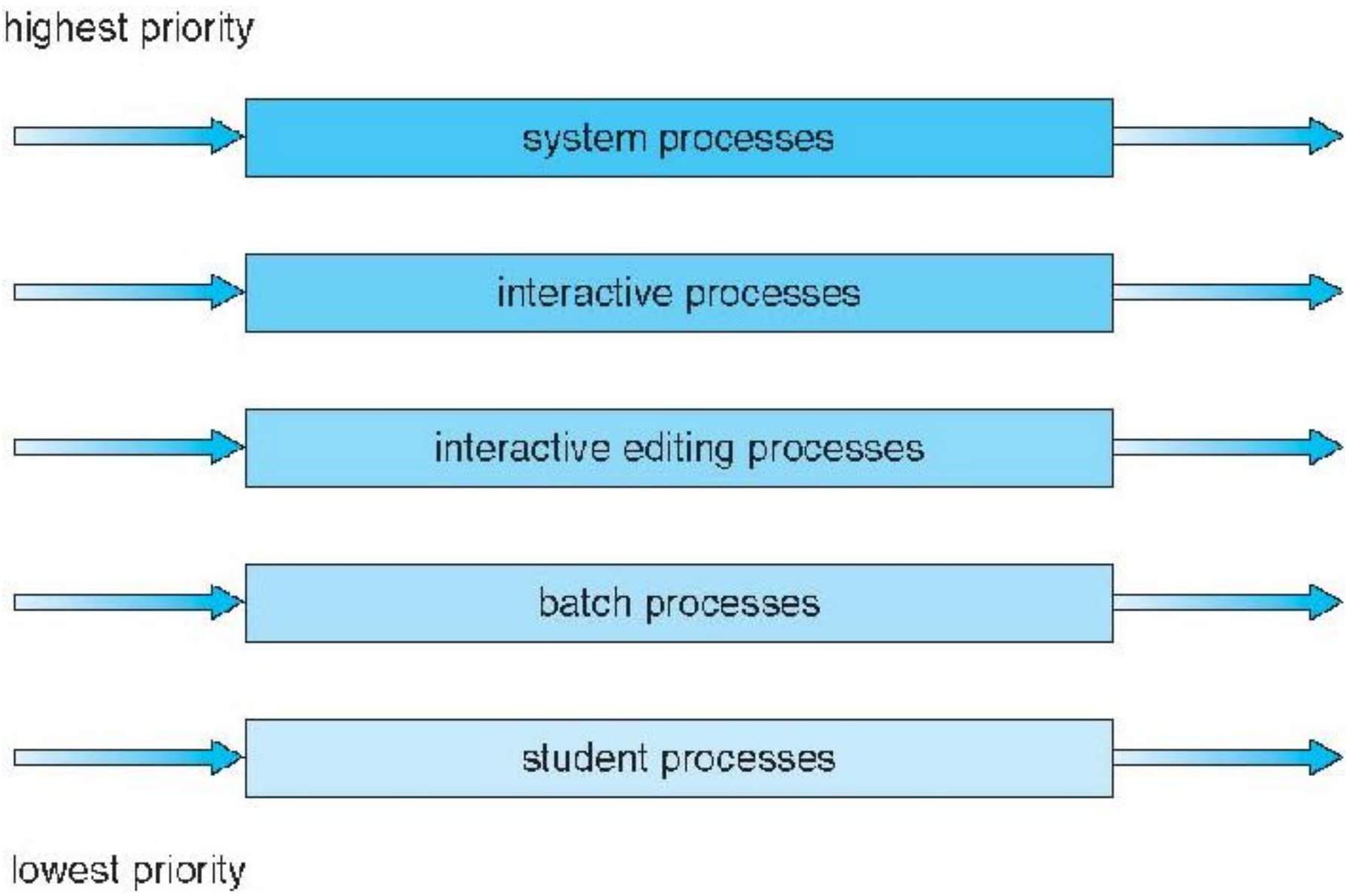
Round Robin Scheduling

- Each process gets a small unit of CPU time (time quantum q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high
- Good on response time (each process starts quick), bad on turnaround time

Multilevel Queue

- Combine Round Robin + priority scheduling
- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue



Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single foreground process- controlled via user interface
  - Multiple background processes– in memory, running, but not on the display, and with limits
    - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a service to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Assignment 1 Discussion

- http://www.jonbell.net/gmu-cs-475-spring-2018/homework-1/
- https://autolab.cs.gmu.edu/courses/CSTEST/assessments

Roadmap

- Next week:
  - How can we get multiple processes to work together?
  - Threads