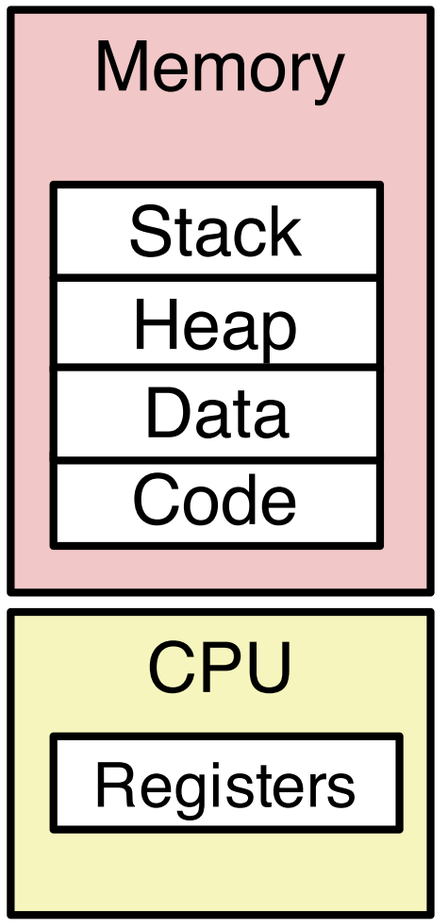
Concurrent Processes

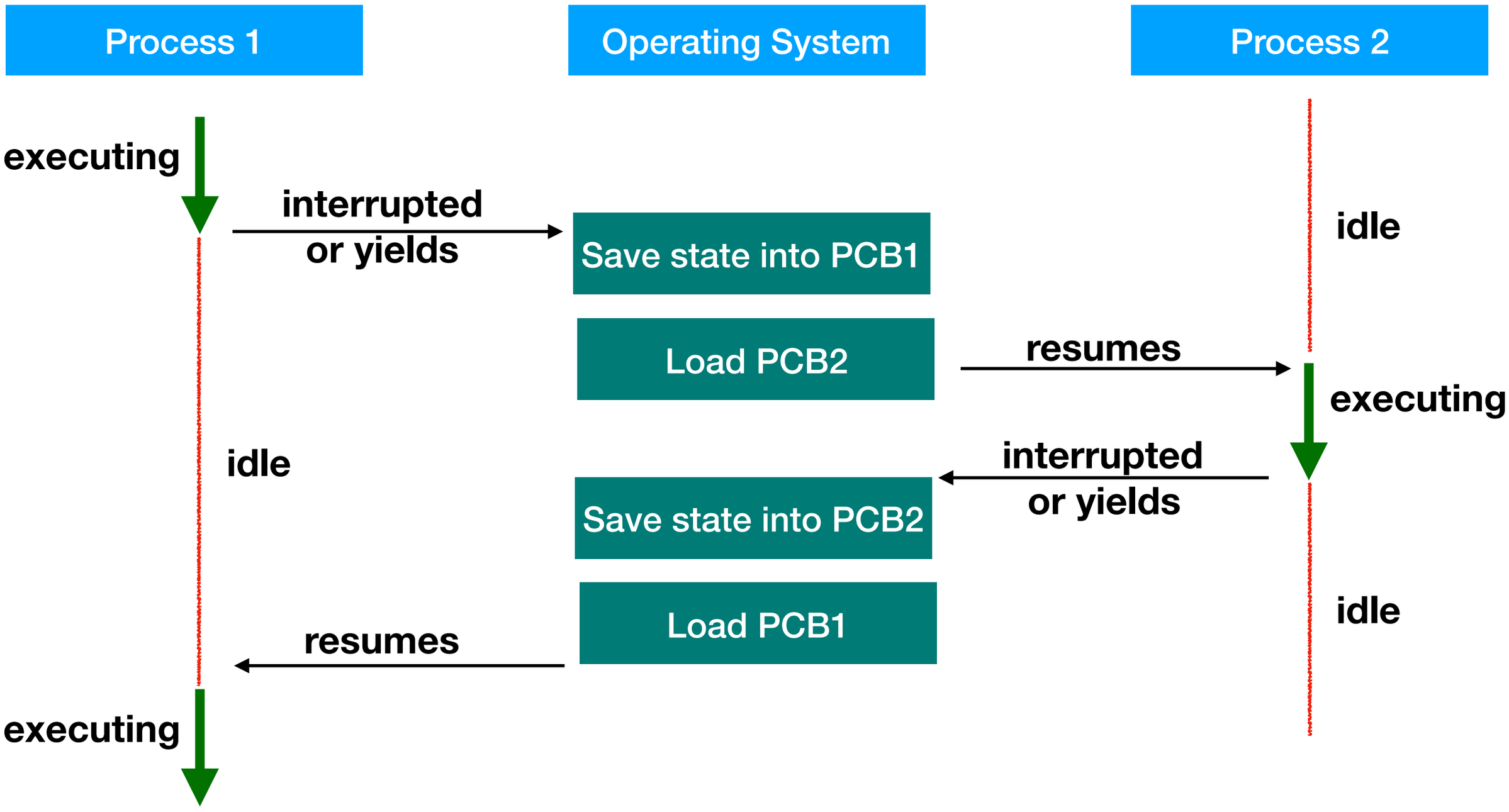
CS 475, Spring 2018
Concurrent & Distributed Systems

Review: 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



Review: Context Switching



Review: 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");
}
```

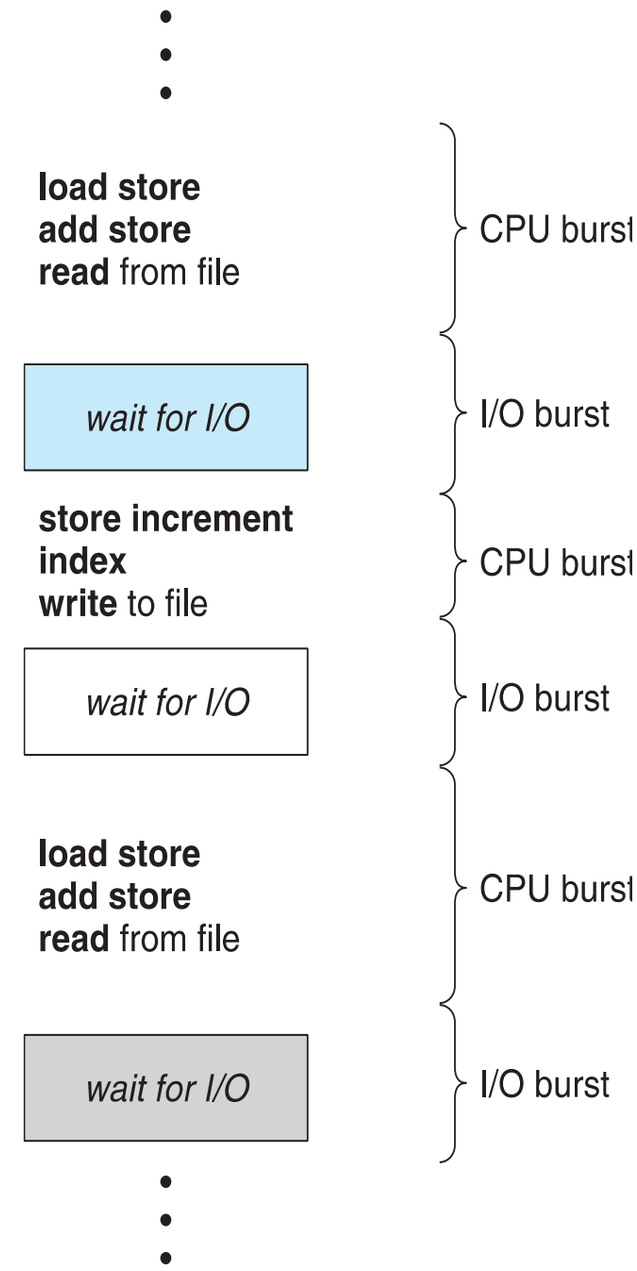
Fork is interesting (and often confusing) because it is called once but returns twice

Review: 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

Review: Scheduling

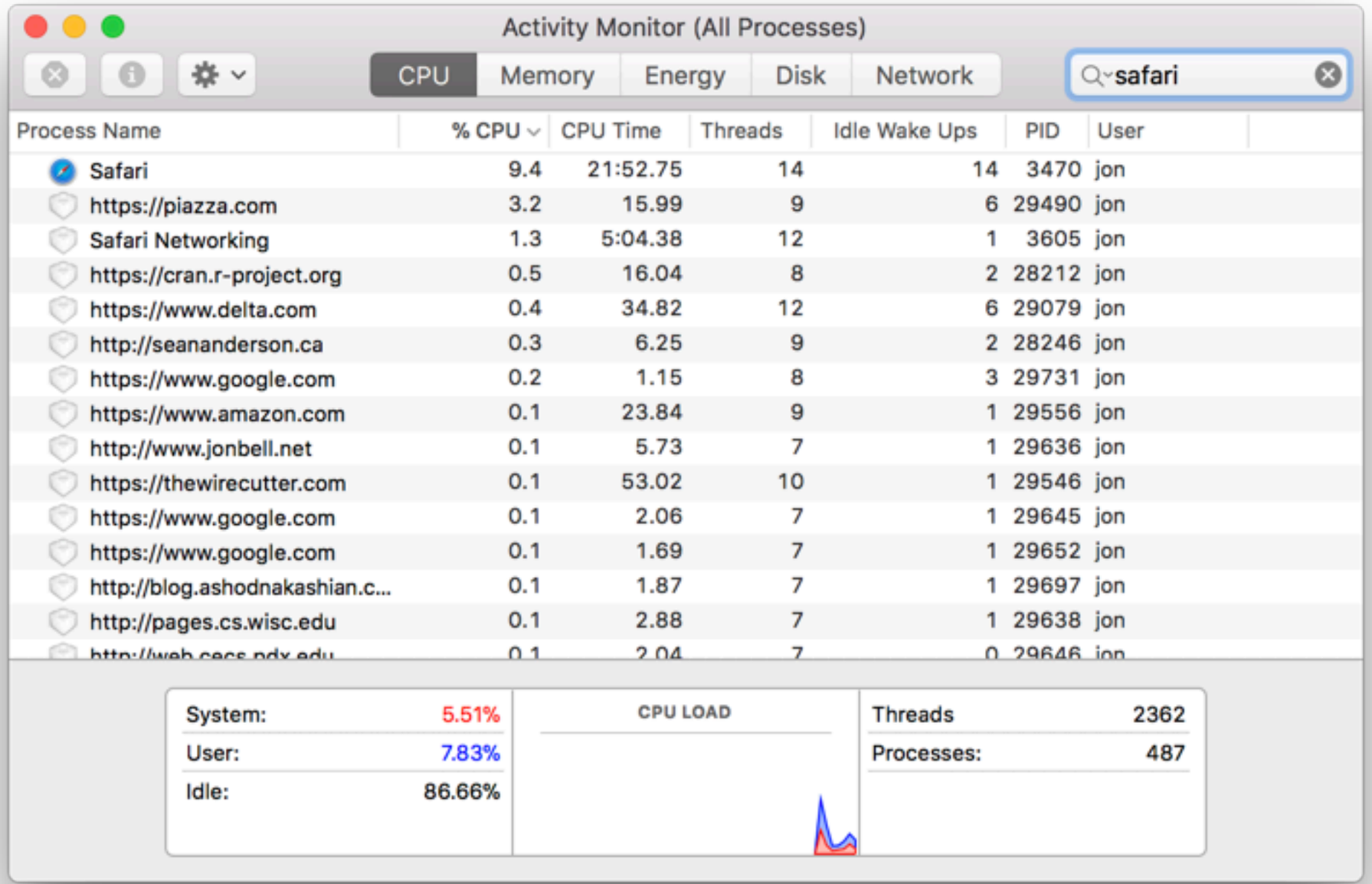
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern



Today

- How to coordinate multiple concurrent processes
  - Signals
  - Shared memory
  - Message passing
- Discuss HW1
- Additional readings:
  - Tannenbaum 4.3
- Slide acknowledgements:
  - OS Concepts 9th Ed, Silberschatz, Galvin & Gagne

Motivation


Activity Monitor (All Processes)
CPU
Memory
Energy
Disk
Network
safari
Process Name
% CPU
CPU Time
Threads
Idle Wake Ups
PID
User
Safari 9.4 21:52.75 14 14 3470 jon
https://piazza.com 3.2 15.99 9 6 29490 jon
Safari Networking 1.3 5:04.38 12 1 3605 jon
https://cran.r-project.org 0.5 16.04 8 2 28212 jon
https://www.delta.com 0.4 34.82 12 6 29079 jon
http://seananderson.ca 0.3 6.25 9 2 28246 jon
https://www.google.com 0.2 1.15 8 3 29731 jon
https://www.amazon.com 0.1 23.84 9 1 29556 jon
http://www.jonbell.net 0.1 5.73 7 1 29636 jon
https://thewirecutter.com 0.1 53.02 10 1 29546 jon
https://www.google.com 0.1 2.06 7 1 29645 jon
https://www.google.com 0.1 1.69 7 1 29652 jon
http://blog.ashodnakashian.c... 0.1 1.87 7 1 29697 jon
http://pages.cs.wisc.edu 0.1 2.88 7 1 29638 jon
System: 5.51%
User: 7.83%
Idle: 86.66%
CPU LOAD
Threads 2362
Processes: 487

Signals

- OS has a mechanism to **interrupt** regular execution of a process
- We've discussed this in the context of **preemption** - when a round-robin scheduler decides a processes' quantum is up
- Signals notify a process of some message (the signal) and then allow the process to do something

Signals

- A signal is a small message that notifies a process that an event of some type has occurred in the system
- Akin to exceptions and interrupts
- Sent from the kernel (sometimes at the request of another process) to a process
- Signal type is identified by small integer ID's (1-30)
- Only information in a signal is its ID and the fact that it arrived

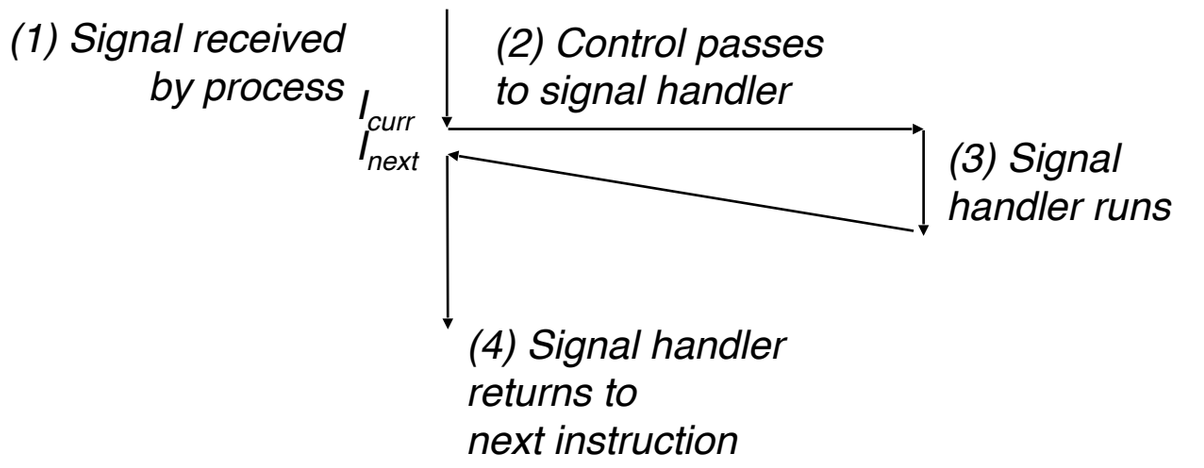
ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signal Concepts

- Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process
- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the kill system call to explicitly request the kernel to send a signal to the destination process

Signal Delivery

- A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Some possible ways to react:
  - Ignore the signal (do nothing)
  - Terminate the process (with optional core dump)
  - Catch the signal by executing a user-level function called signal handler
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt:



Signal Delivery

- A signal is pending if sent but not yet received
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded
- A process can block the receipt of certain signals
  - Blocked signals can be delivered, but will not be received until the signal is unblocked
- A pending signal is received at most

Signal Delivery

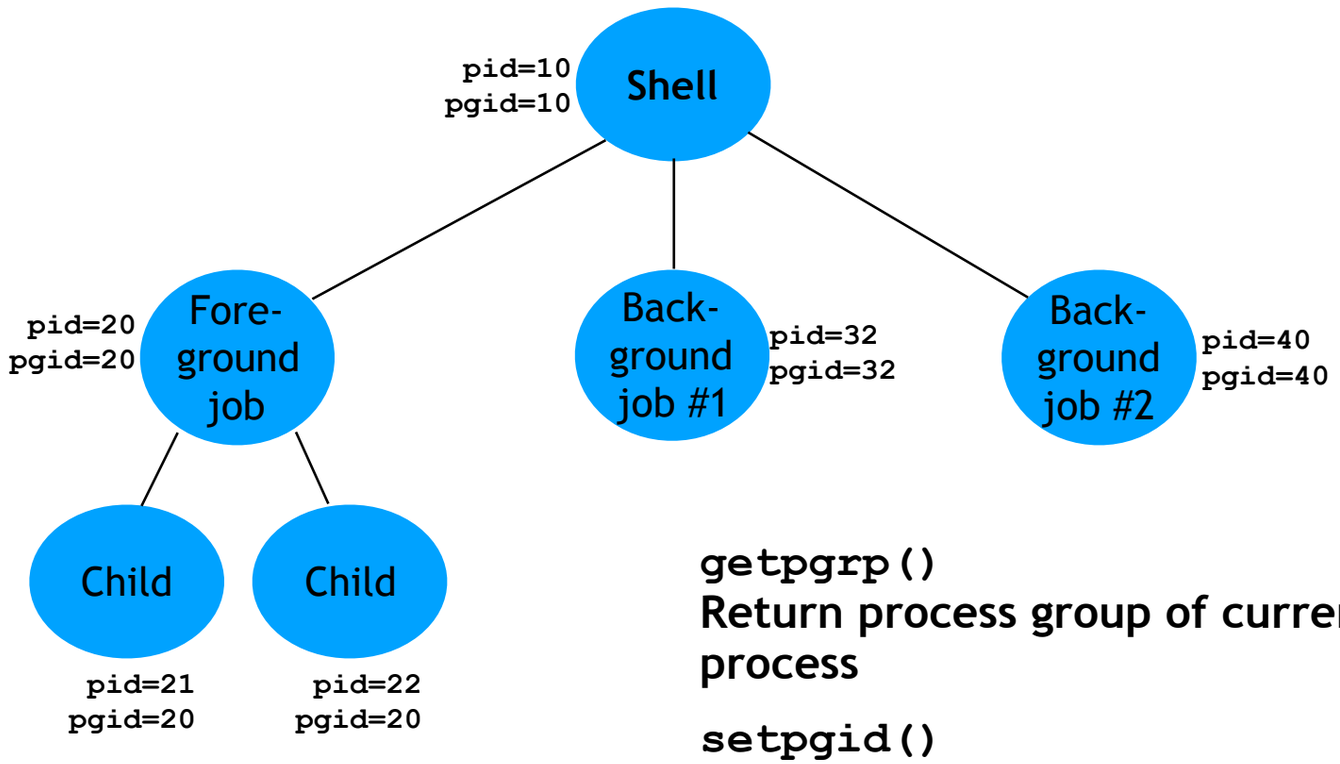
- A pending signal is received at most once
- Signals might not be delivered! (why?)
- We will discuss at least once/exactly once/at most once in detail in a few weeks

Signal Delivery

- Kernel maintains pending and blocked bit vectors in the context of each process
  - pending: represents the set of pending signals
    - Kernel sets bit k in pending when a signal of type k is delivered
    - Kernel clears bit k in pending when a signal of type k is received
  - blocked: represents the set of blocked signals
    - Can be set and cleared by using the sigprocmask function
    - Also referred to as the signal mask.

Sidebar: Process Groups

- By default, every time a process fork's, its child inherits the parent's **process group**
- Can request a new one, also request a new one
- Signals are delivered to a single process, or to a process group



`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process

Sending Signals

- You likely already know how to send at least one signal!
- Typing ctrl-c (ctrl-z) causes the kernel to send a SIGINT (SIGTSTP) to every job in the foreground process group.
  - SIGINT – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process

Sending Signals

- `/bin/kill` program sends arbitrary signal to a process or process group (not just the kill signal!)
- Or, from your (C) code: `int kill(pid_t pid, int sig);`
- Examples
  - `/bin/kill -9 24818`
    Send SIGKILL to process `24818`
  - `/bin/kill -9 -24817`
    Send SIGKILL to every process in process group `24817`

Receiving and Handling Signals

- Signal delivery generally* happens when OS starts executing a process
- If there are no signals to be delivered:
  - Start executing the next instruction in the process (this is the usual context-switch story)
- If there ARE signals to be delivered
  - Take the first signal pending, force process to receive that signal
  - After all signals delivered, if process still running, then resume execution as normal

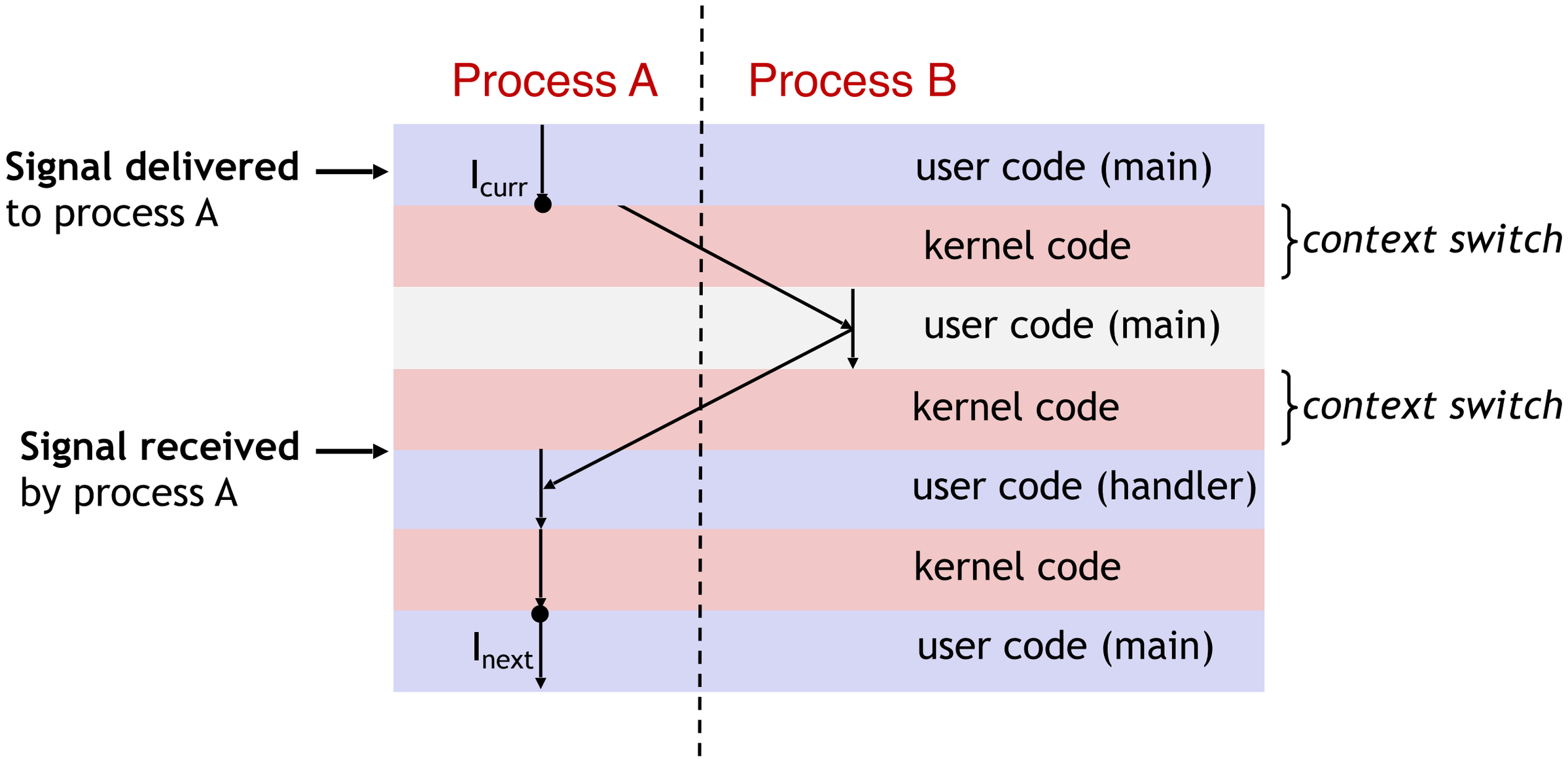
Default Signal Handlers

- Each signal type has a predefined default action, which may be one of:
  - The process terminates
  - The process stops until restarted by a SIGCONT signal
  - The process ignores the signal

Creating Signal Handlers

- The signal function modifies the default action associated with the receipt of signal signum:
  - handler_t *signal(int signum, handler_t *handler)
- Different values for handler:
  - SIG_IGN: ignore signals of type signum
  - SIG_DFL: revert to the default action on receipt of signals of type signum
  - Otherwise, handler is the address of a user-level signal handler
    - Called when process receives signal of type signum
    - Referred to as “installing” the handler
    - Executing handler is called “catching” or “handling” the signal
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

Signal Handlers as Concurrency



Signal Caveats

- Handlers can be interrupted by other handlers
- Because they run concurrently with main program, can inadvertently corrupt data structures
- Not all signals can be handled (SIGKILL, SIGSTOP)
- Relatively high overhead (>10,000 clock cycles)
- No queues (only can have 1 pending signal for each signal type)

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

Interprocess Communication

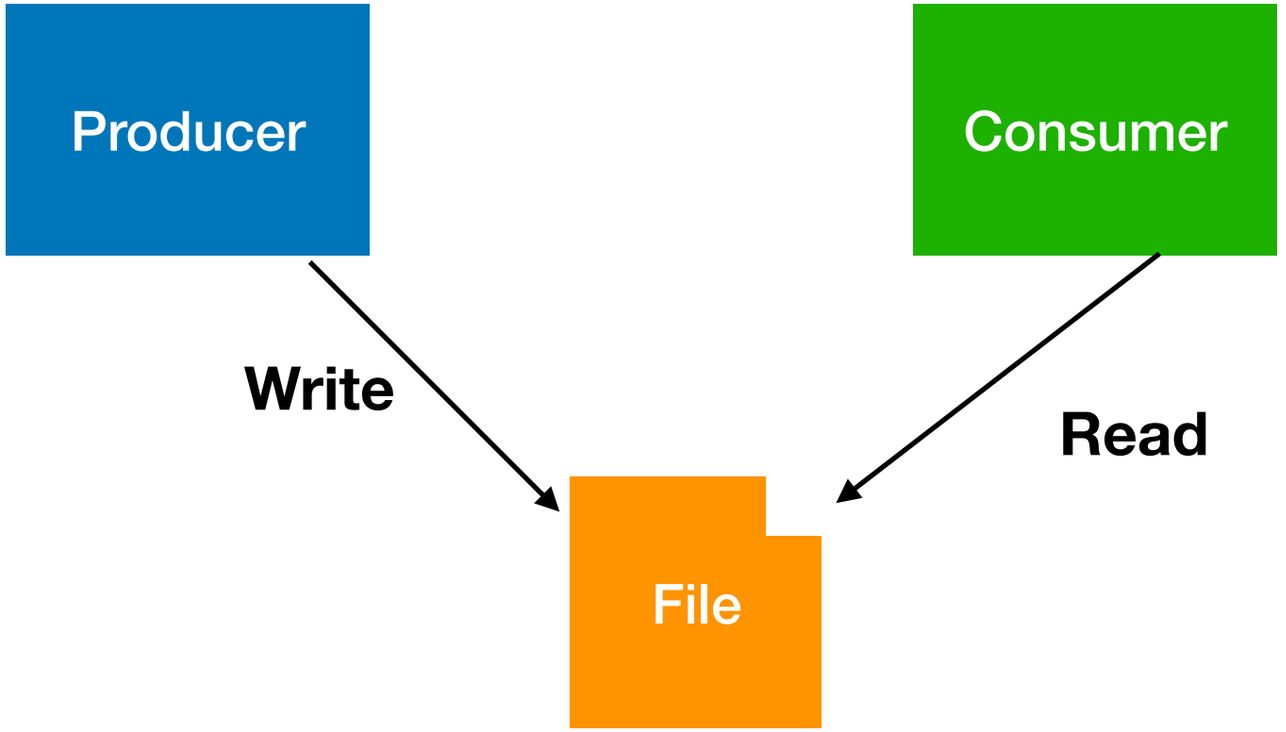
- We might want two processes to seriously work together
- For example:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Signals are very, very NOT sufficient for these purposes
- What we need is **interprocess communication (IPC)**

Producer-Consumer Model

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size
- Producer writes to a buffer, consumer reads
- Buffer is just a chunk of memory

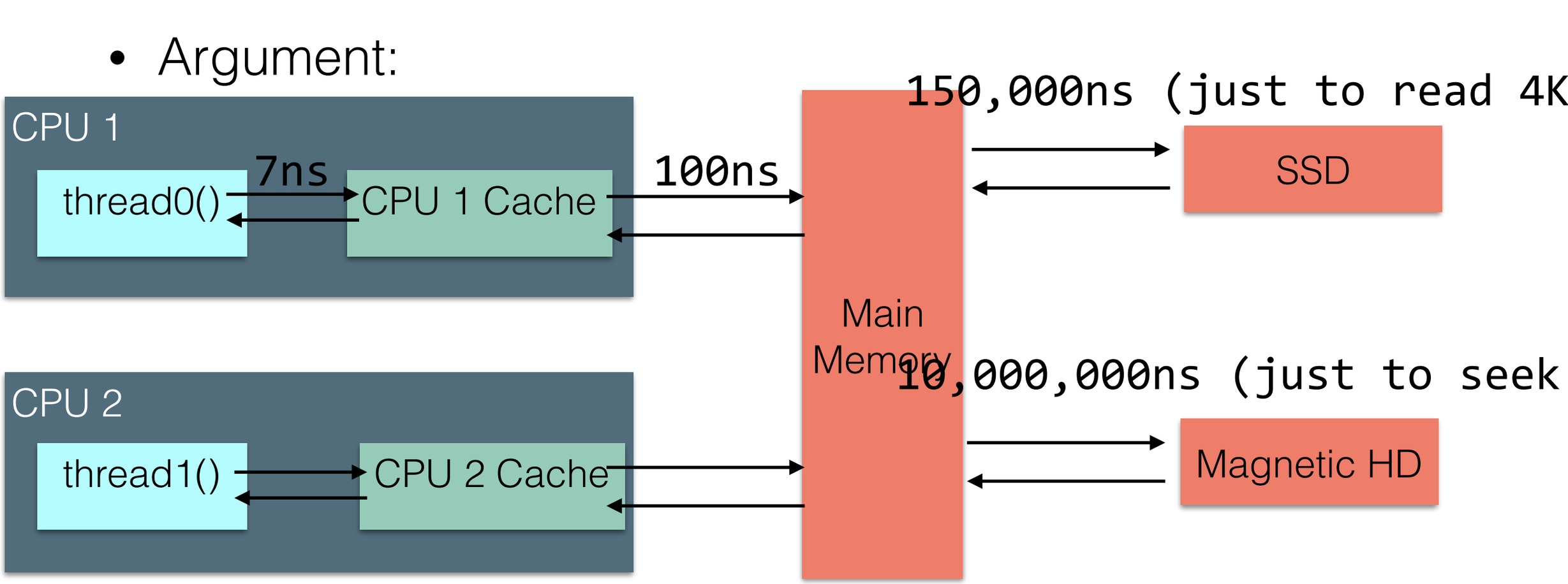
Strawman IPC

- Producer writes to a file
- Consumer reads from same file



Strawman IPC

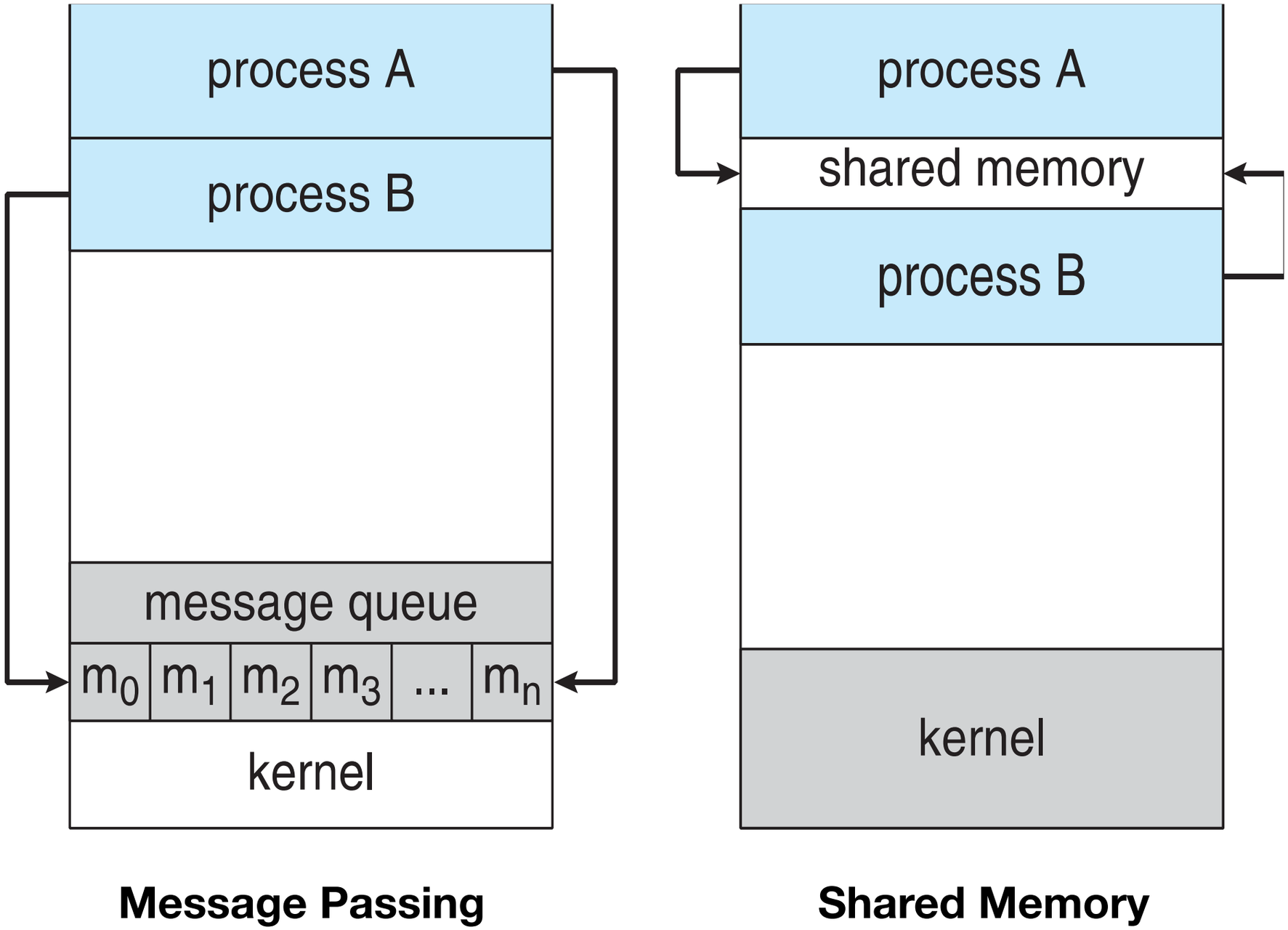
- Does it work? Yes
- Is it cumbersome (and perhaps error-prone)? Yes
  - What happens if consumer reads while producer is writing?
- Is it efficient?
  - No
  - Argument:



Improving on the Strawman

- Shared memory
  - Strawman, but the “file” is just a hunk of memory that’s shared between processes
- Message Passing
  - Abstraction on top of shared memory: producer sends messages to consumer

Message Passing & Shared Memory



Message Passing **Shared Memory**

Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization will be discussed in very great detail in the next few weeks

Shared Memory Pseudocode

```
//Producer
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out);
    /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
//Consumer
item next_consumed;
while (true) {
    while (in == out)
    ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

Shared Memory in POSIX

- Process first creates shared memory segment

```
shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
```

- Also used to open an existing segment to share it
- Set the size of the object

```
ftruncate(shm fd, 4096);
```

- Memory map the shared memory

```
shm_ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd,
0);
```

- Write to shared memory

```
sprintf(shm_ptr, "Writing to shared memory");
```

Shared Memory Summary

- As high performance as you can get
  - Each process directly reads/writes memory, which happens to be shared
- Can become confusing to program (correctly)
  - Which variables exactly are shared?
  - What happens if I copy a pointer to (non-shared) memory into shared memory?
  - What happens if producer/consumer read/write simultaneously?

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message)
  - receive(message)
- The message size is either fixed or variable
- Messaging system can be arbitrarily complex, adding additional features

Message Passing

- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive
- On a single machine, this is usually done by creating a named mailbox (or "port")
- Key implementation questions:
  - Are sending and/or receiving blocking, or non-blocking?
  - Is there a message queue?

Syncronous and Asynchronous

- Message passing may be either blocking or non-blocking
- **Blocking** is considered synchronous
  - Blocking send -- the sender is blocked until the message is received
  - Blocking receive -- the receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send -- the sender sends the message and continue
  - Non-blocking receive -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - E.g. both send and receive are blocking, only one, neither

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     ->Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of n messages
     ->Sender must wait if link full
  3. Unbounded capacity – infinite length
     ->Sender never waits

Message Passing Pseudocode

```
message next_produced;
while (true) {
    /* produce an item in next produced */
    send(next_produced);
}
```

```
message next_consumed;
while (true) {
    receive(next_consumed);

    /* consume the item in next consumed */
}
```

Message Passing in Mach

- (MacOS X, iOS)
- Mach communication is entirely message based - Even system calls are messages
- Each task gets two mailboxes at creation- Kernel and Notify
- Only three system calls needed for message transfer
  - msg_send(), msg_receive(), msg_rpc()
- Mailboxes needed for communication, created via
  - port_allocate()
- Send and receive are flexible, for example four options if mailbox full:
  - Wait indefinitely
  - Wait at most n milliseconds
  - Return immediately
  - Temporarily cache a message

Message Passing Systems

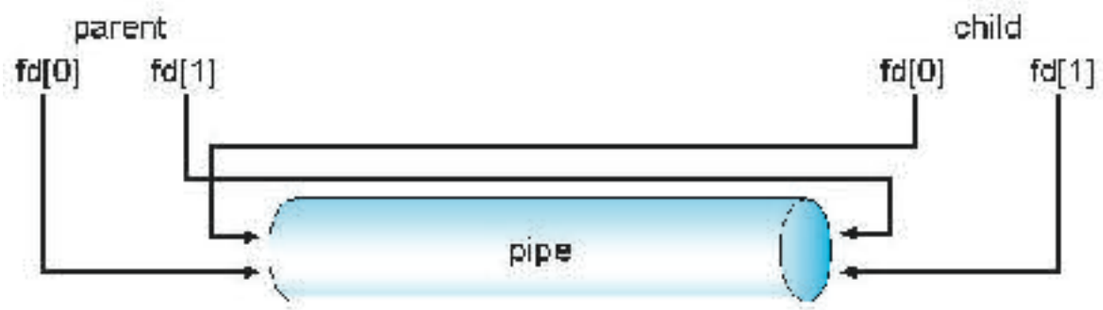
- Messaging is a higher level abstraction than shared memory
- Messaging is a common communication pattern in distributed systems too
- When scaling up, there are additional questions that we will think about (later):
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Messaging Tradeoffs

- Requires no programmer effort to do synchronization
- MIGHT be slower than shared memory (there's more "stuff" to do)
- MIGHT be faster than shared memory (that "stuff" is written by someone smart who spent a lot of time on it)
- Can be abstracted even further: messaging between computers

Pipes

- Used in some client-server systems
- Not as generic as shared memory or message passing - ordinary pipes can only be used between child and parent processes
- Ordinary pipes allow communication in standard producer-consumer style
  - Producer writes to one end (the write-end of the pipe)
  - Consumer reads from the other end (the read-end of the pipe)
  - Ordinary pipes are therefore unidirectional



Pipes

- Consider as message passing: producer writes to pipe, consumer reads from pipe
- Compare to shared memory: in shared memory, can read/write any part of the memory block
- Pipes are 1:1 (message passing might have multiple producers/consumers)

IPC Summary

- Different models suit different needs
- Examples (real world):
  - Communication via postcard
  - Speaking in a room with two people
  - Instant messaging
  - Speaking in this classroom (with 60 people)

Assignment 1 Discussion

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

Roadmap

- Weds: Threads
  - Shared memory, but sharing all memory, within a process
- Next week: Synchronization
  - Sharing memory safely