CSE 15L: Software Tools and Techniques Laboratory

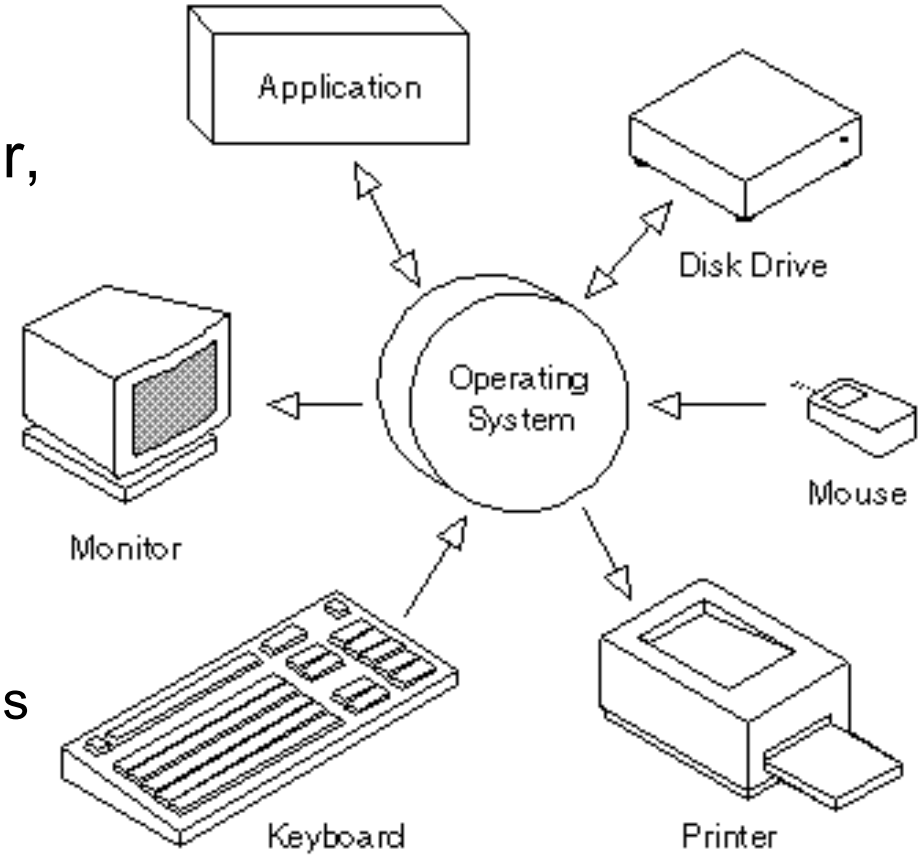
Winter 2021 - http://ieng6.ucsd.edu/~cs15x

Instructors: Gary Gillespie Keith Muller

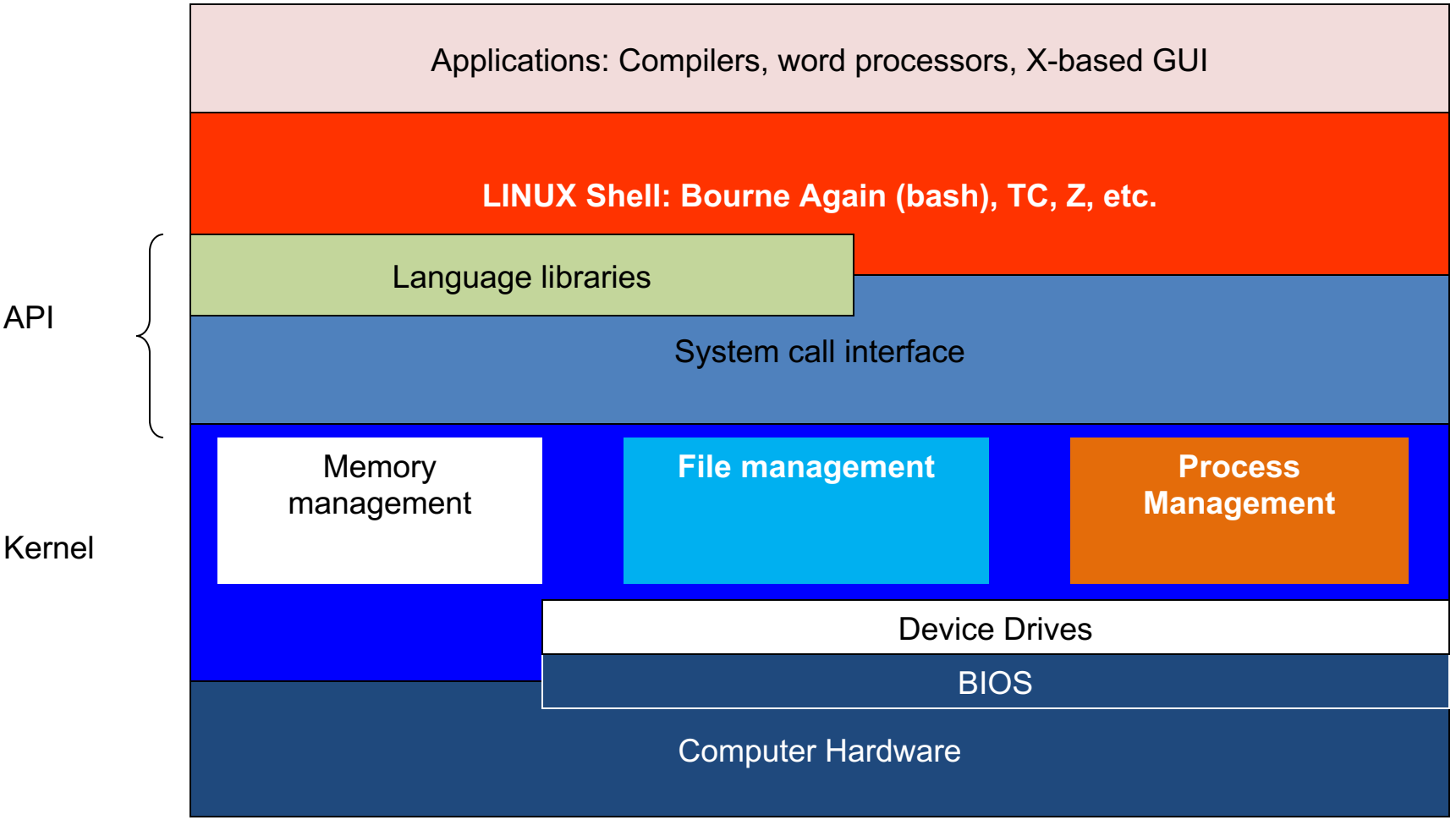
Class sessions will be recorded and made available to students asynchronously.

Operating System

- To facilitate easy, efficient, fair, orderly, and secure use of resources
  - Provide a user interface
  - Organize files on disk
  - Allocating resource to different users with security control
  - Co-ordinate programs to work with devices and other programs



Abstraction Layers



Abstraction Layers

- Kernel
  - The part of an OS where the real work is done
- Library and System call interfaces
  - Comprise a set of functions (often known as Application Progarmmer's Interface API) that can be used by the applications and library routines to use the services provided by the kernel
- File Management
  - Control the creation, removal of files and provide directory maintenance
  - For a multiuser system, every user should have its own right to access files and directories

Abstraction Layers

- Memory management software
  - Memory in a computer is divided into main memory (RAM) and persistent secondary storage (usually refer to disk/flash storage)

- Device driver software
  - Interfaces between the kernel and the BIOS
  - Different device has different driver

Abstraction Layers

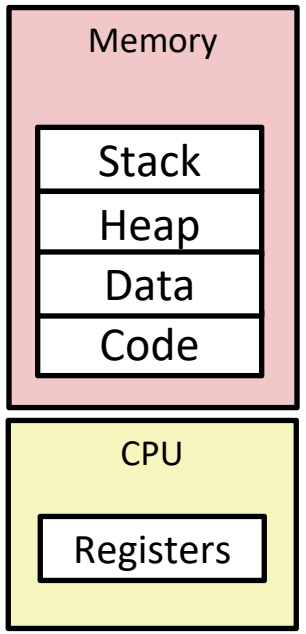
- Process Management
  - In a multitasking system, multiple programs are executing simultaneously in the system
  - When a program starts to execute, it becomes a process
  - The same program executing at two different times will become two different processes
  - The Kernel manages processes in terms of creating, suspending, and terminating them
  - A process is protected from other processes and can communicate with the others

Processes

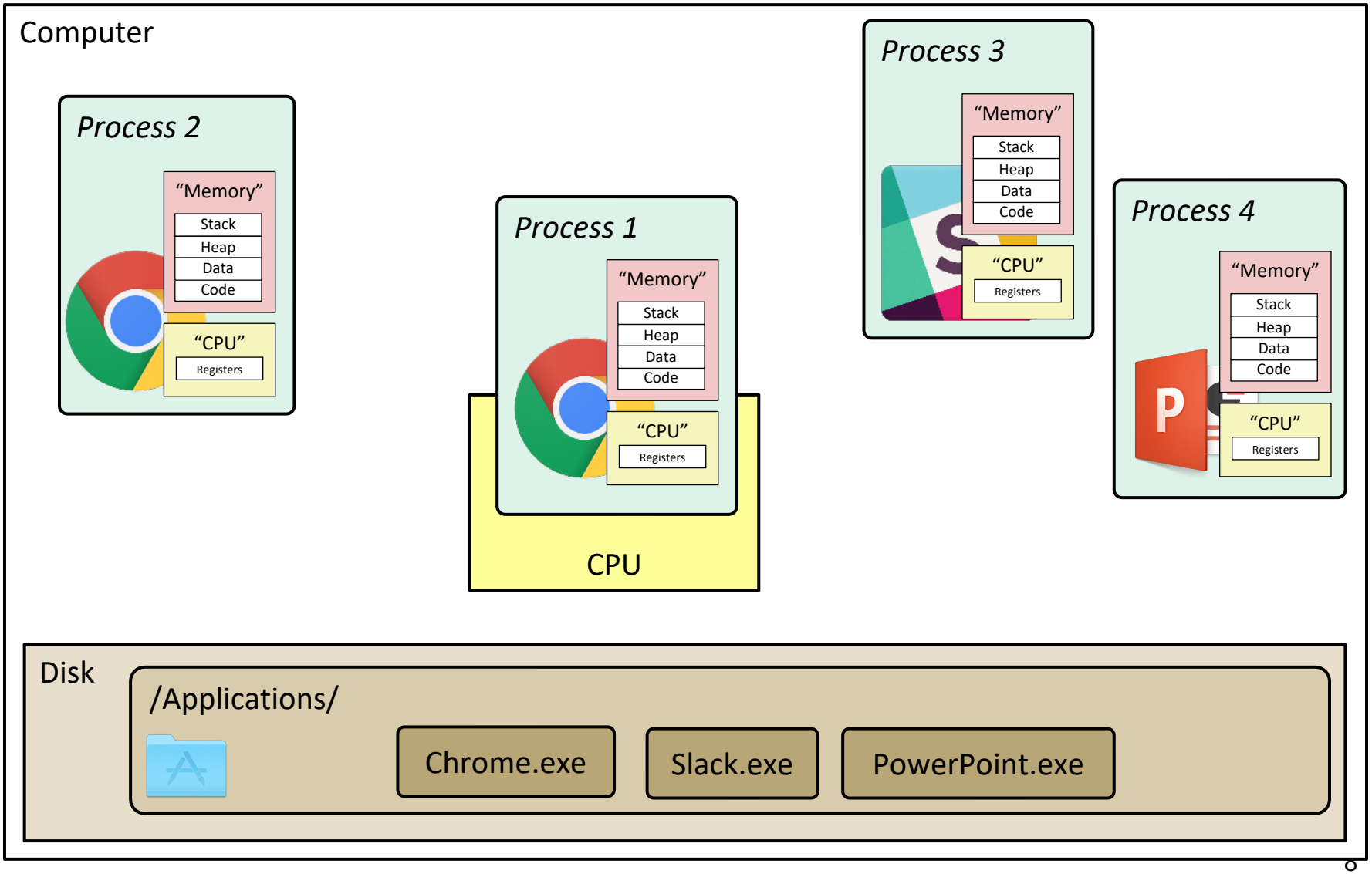
- Informal definition:

  A process is a program in execution.

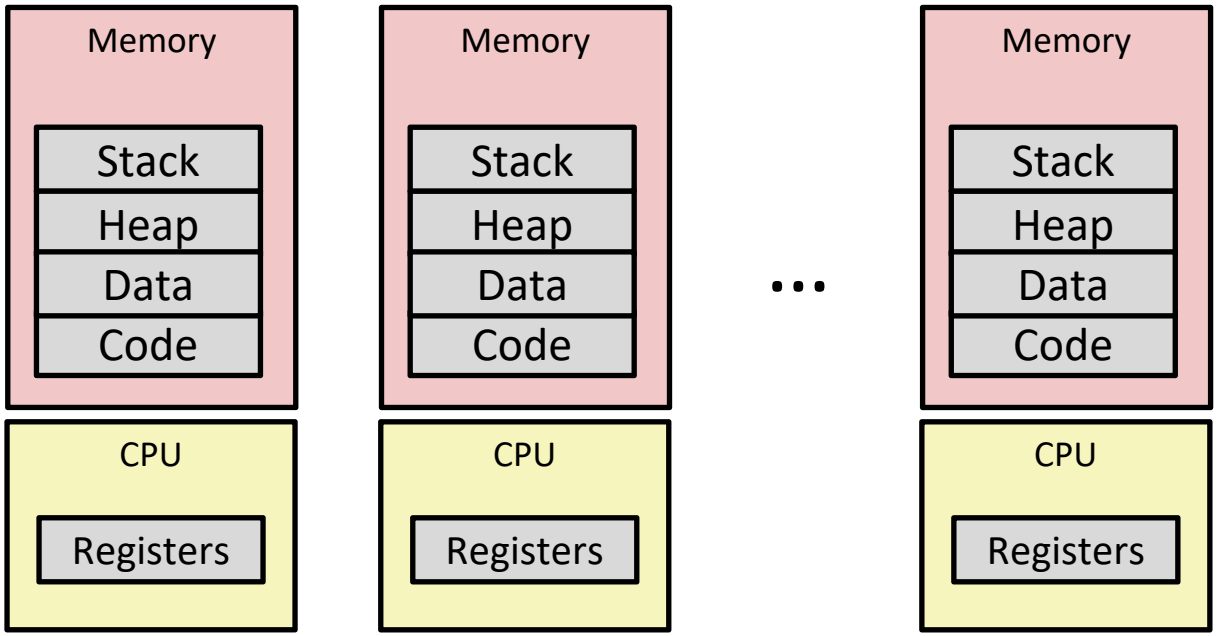
- Process is not the same as a program.
  - Program is a passive entity stored in disk
  - Program (code) is just one part of the process.
- How to start a process?
  - Execute a utility, program, or script!



What is a process?

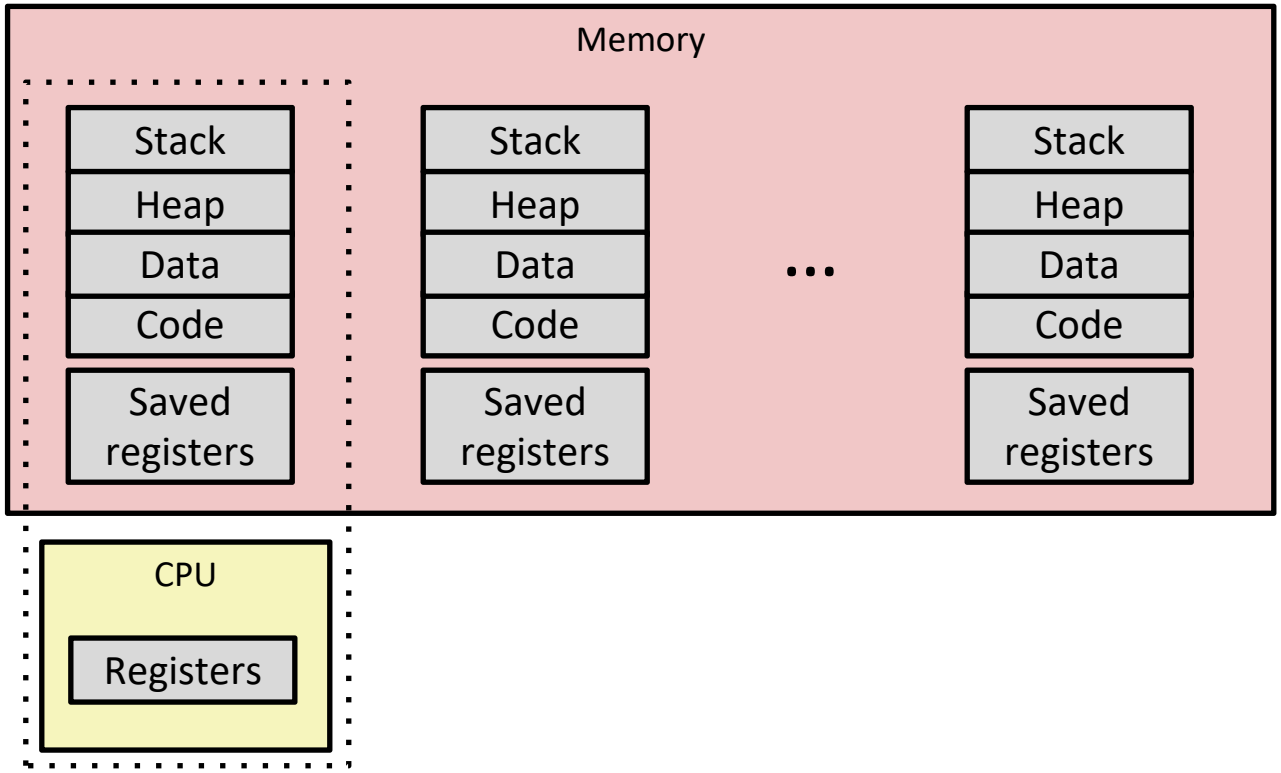


Multiprocessing: The Illusion



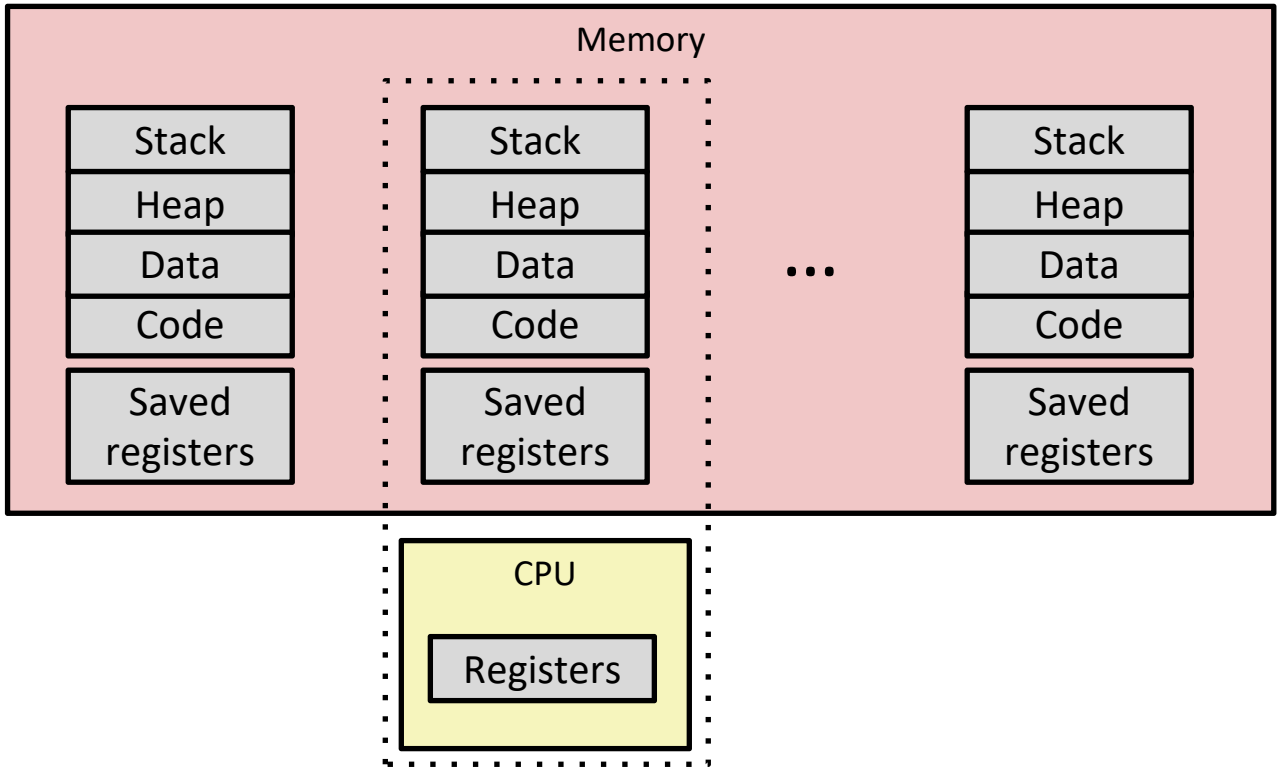
- Computer appears to runs processes concurrently
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing: The Reality



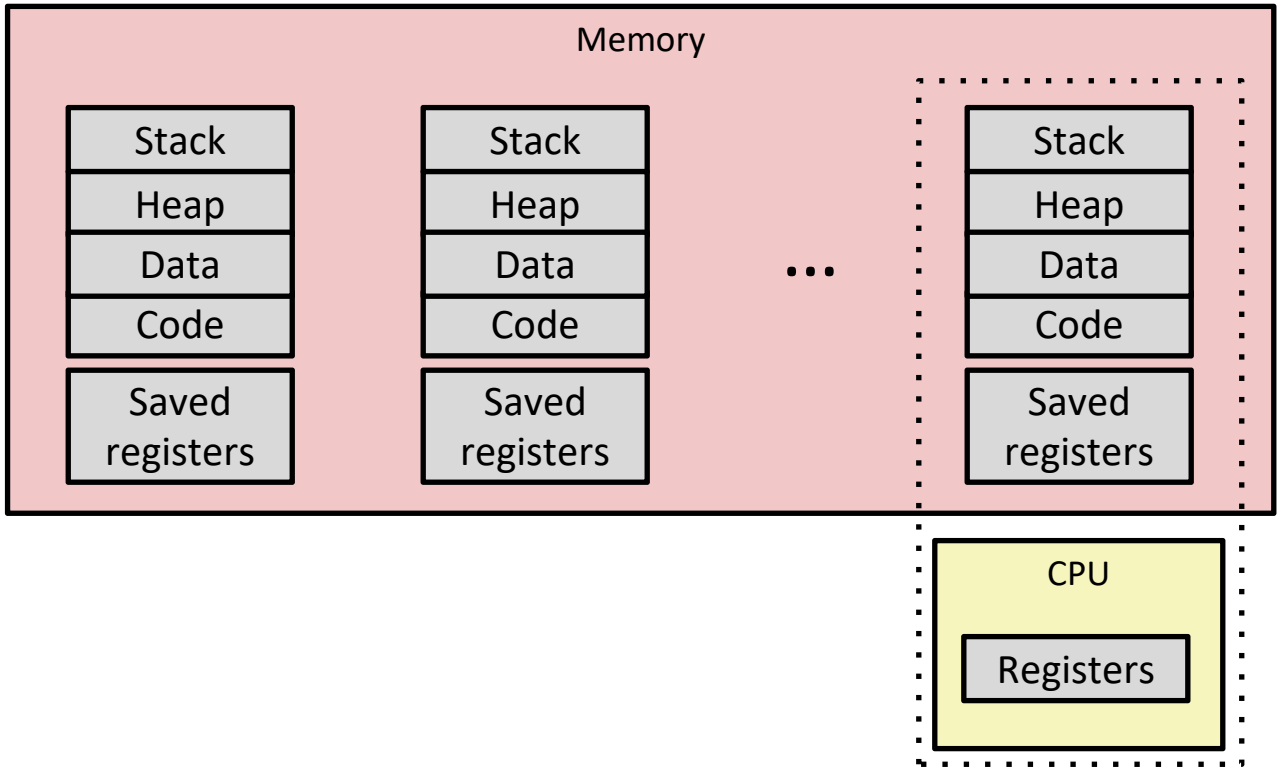
- Single processor executes multiple processes *concurrently*
  - Process executions are interleaved
  - Each CPU runs *one process at a time*

Multiprocessing: The Reality



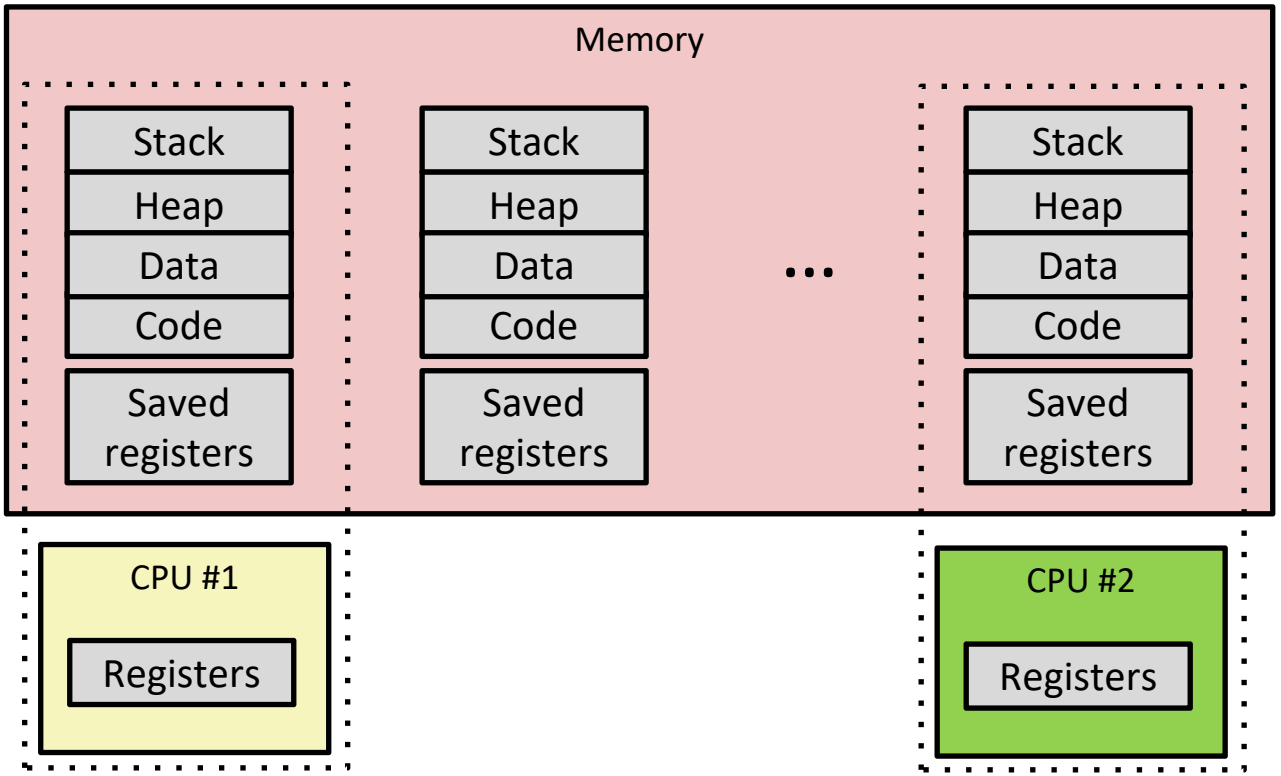
- Single processor executes multiple processes *concurrently*
  - Process executions are interleaved
  - Each CPU runs *one process at a time*

Multiprocessing: The Reality



- Single processor executes multiple processes *concurrently*
  - Process executions are interleaved
  - Each CPU runs *one process at a time*

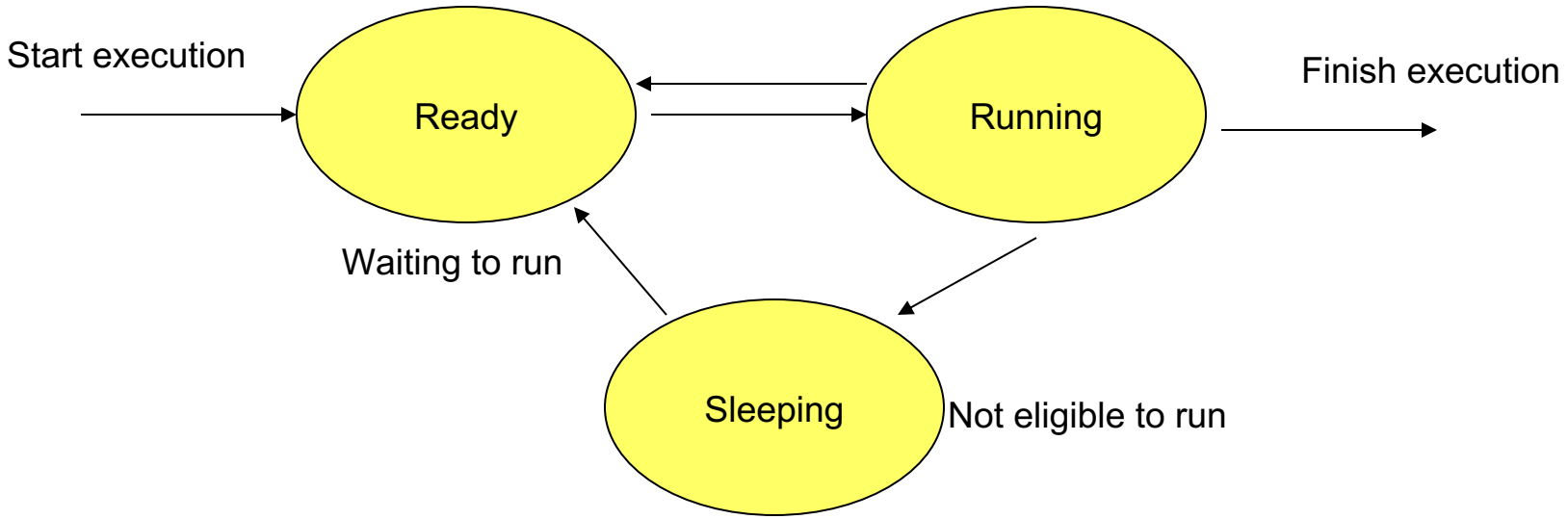
Multiprocessing: The Reality



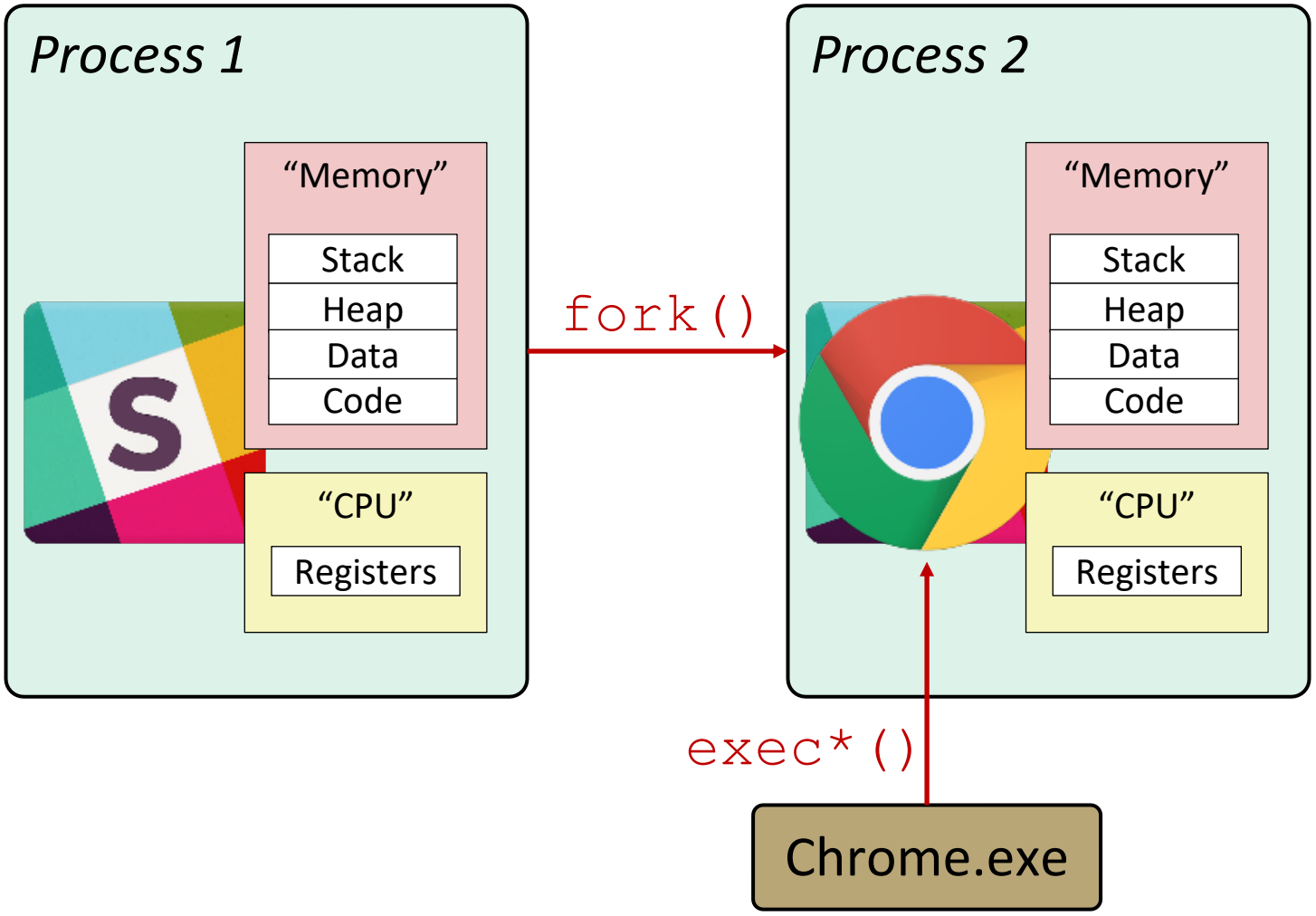
- Multicore processors
  - Multiple CPUs ("cores") on single chip
  - Shares main memory and I/O devices
  - Each CPU will execute a separate process
    - Kernel schedules processes to cpu cores

Process States

- A program that is claimed to be executing is called a process
- A process is said to be in at least in one of the following three states:



Creating New Processes & Programs



Creating New Processes & Programs

- fork-exec model (Linux):
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process' code and address space with the code for a different program
    - Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  - `fork()` and `execve()` are *system calls*
- Parent process has the responsibility to check if the child is done and collect the exit return value
  - `wait()`, `waitpid()`

The Process ID (PID)

What is the PID?

- PID: is the Process Identifier Number
- In Linux, uniquely identifies every process
- May be used as input argument to various commands that manipulate processes (**`kill`**)

Creating a Child Process

- **Child Process**:
  - **`fork()`**: In Unix, a child process is created when the parent process invoke the **`fork()`** system call.
  - *Child processes are initially identical copies of their parent until they exec() a different program*
  - Both parent and child continue execution of the same code after the fork() completes
  - *The return value of fork() system call is used by a process to determine if it is the parent or the child*
    - *child process – fork returns 0*
    - *parent process – fork returns PID of child*
  - When a child process changes "state" (exits, stops execution, etc.), a "SIGNAL" is sent to the parent (SIGCHLD) the parent can then act

- **Orphan process**:
  - When a parent process dies, the child becomes an orphan process.
  - Child process are immediately adopted by the **`init`** process!

Daemons and Zombies

- **Daemon process**:
  - A program run as a background process without being controlled by an interactive user. Sometimes, daemon processes are adopted by **`init`**.

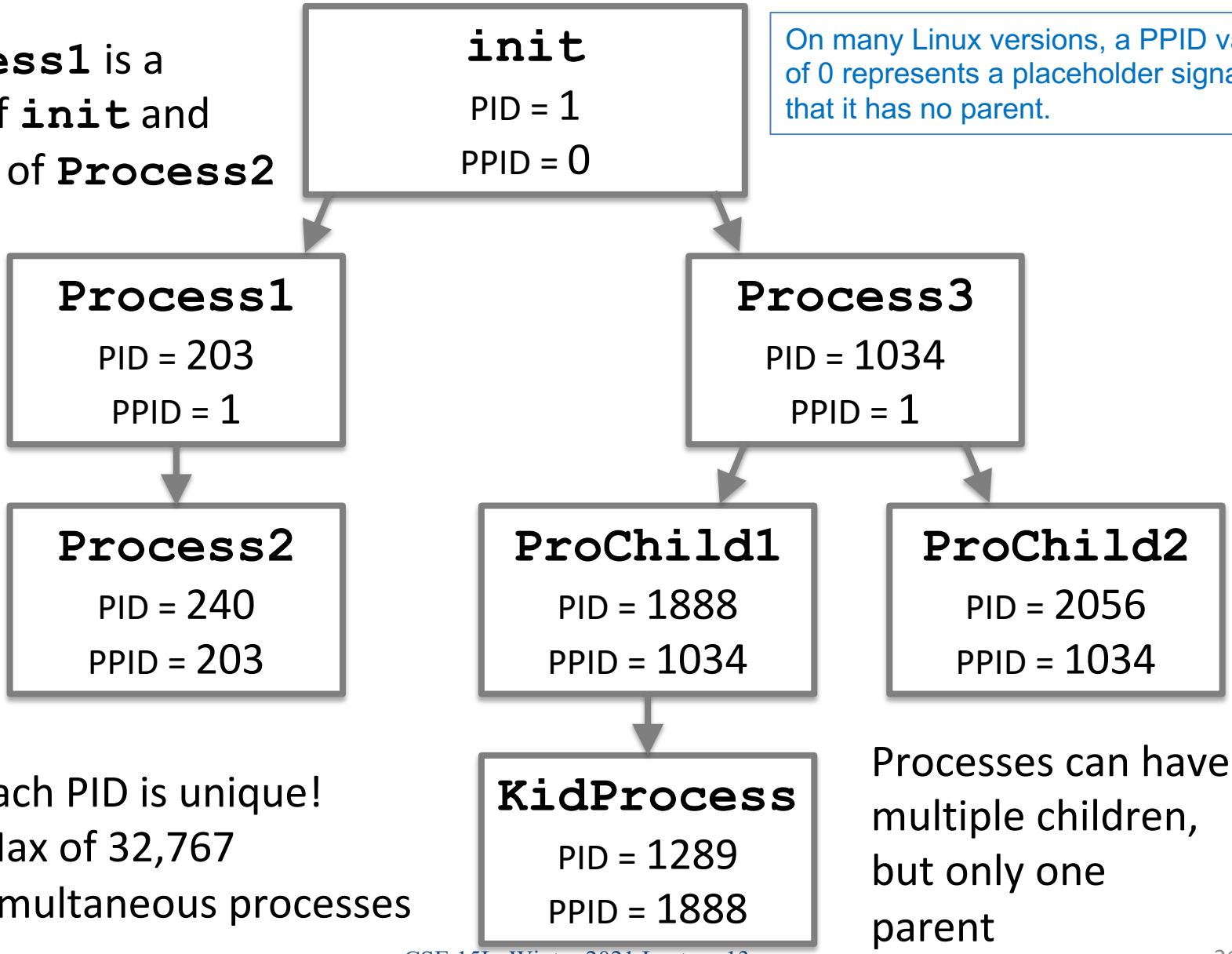
- **Zombie process**
  - A process that has completed execution, memory and resources are deallocated, but still has an entry in the process table (**`z`**)
  - Why? it is waiting for the parent to collect it's exit status (using the **`wait()or waitpid()`** system call)
  - A zombie is not an orphan!

Child, parent, and grandparent of all processes…

- Every process has a parent that started it, tracked using PPID (Parent Process ID)
- **`init`** (Linux) or **`launchd`** (macOS) is the grandparent of all processes, started when system starts up
  - Always PID 1
- Processes are hierarchical! Think of **`init`** or **`launchd`** as the root of the process tree
- **`pstree`**: this command shows the relationship of all processes in a tree-like structure

Example: Process Hierarchy

`Process1` is a child of **`init`** and parent of **`Process2`**



On many Linux versions, a PPID value of 0 represents a placeholder signaling that it has no parent.

Each PID is unique!
Max of 32,767 simultaneous processes

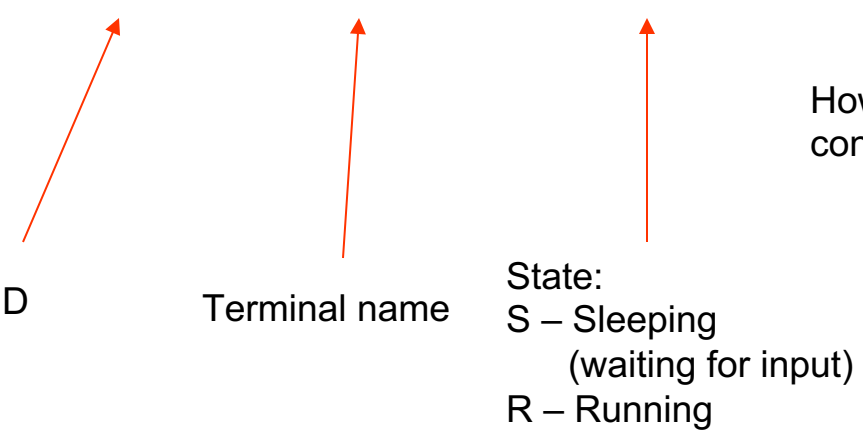
Processes can have multiple children, but only one parent

Examining Processes In Linux

- *ps command*
  - Standard process attributes
- /proc directory
  - More interesting information.
  - Try “man proc”
- *Top, vmstat command*
  - *Examining CPU and memory usage statistics.*

Simple PS Command

```
  PID   TTY      STAT   TIME COMMAND
14748   pts/1    S      0:00 -bash
14795   pts/0    S      0:00 -bash
14974   pts/0    S      0:00 vi test1.txt
14876   pts/1    R      0:00 ps …
```



Example `ps` output

```
$ ps -aef | head
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 12:40 ?        00:00:01 /sbin/init splash
root           2     0  0 12:40 ?        00:00:00 [kthreadd]
root           4     2  0 12:40 ?        00:00:00 [kworker/0:0H]
root           6     2  0 12:40 ?        00:00:00 [mm_percpu_wq]

$ ps -aef | tail
UID          PID  PPID  C STIME TTY          TIME CMD
mostert+  2853  2844  0 12:44 pts/0    00:00:00 bash
root      3002     1  0 12:45 ?        00:00:00 /usr/sbin/cupsd -l
root      3066     2  0 12:45 ?        00:00:00 [kworker/1:0]
root      3077     2  0 12:46 ?        00:00:00 [kworker/1:3]
mostert+  3176  2853  0 12:50 pts/0    00:00:00 ps -aef
mostert+  3177  2853  0 12:50 pts/0    00:00:00 tail
```

Example `ps` output

```
$ ps -aef | head
UID          PID  PPID  C
root           1     0  0
root           2     0  0
root           4     2  0
root           6     2  0
```

UID: User ID that this process belongs to (the person running it)
PID: Process ID
PPID: Parent process ID (the ID of the process that started it)
C: CPU utilization of processor
STIME: Process start time
TTY: Terminal type associated with the process
TIME: CPU time taken by the process
CMD: The command that started this process

```
$ ps -aef | tail
UID          PID  PPID  C STIME TTY          TIME CMD
mostert+    2853  2844  0 12:44 pts/0    00:00:00 bash
root        3002     1  0 12:45 ?        00:00:00 /usr/sbin/cupsd -l
root        3066     2  0 12:45 ?        00:00:00 [kworker/1:0]
root        3077     2  0 12:46 ?        00:00:00 [kworker/1:3]
mostert+    3176  2853  0 12:50 pts/0    00:00:00 ps -aef
mostert+    3177  2853  0 12:50 pts/0    00:00:00 tail
```

Shell: Foreground & Background

- Foreground:
  - bash forks
  - child bash "execs" into command to execute
  - parent bash sleeps waiting for child to die before prompting
  - Processes executed on terminal run in foreground by default (Receive input from keyboard, Send output to screen)
  - The shell will not run another command until previous command finishes
- Background:
  - Invoke a command with an & at the end
  - bash forks
  - child bash "execs" into command to execute
  - parent bash doesn't wait for child to die before prompting for the next command
  - Unless stdin is redirected, a background task will pause execution if it tries to read the keyboard
- nohup: (no hangup)
  - Used as prefix when starting a background process so you can log off and the process keeps executing

Listing Current Background Processes (Jobs)

- **`jobs`**
  - Lists all background processes/jobs in current shell
  - **`+`** indicates default for **`fg`**/**`bg`** command

```
$ gnome-calculator &
[1] 1246
$ date &
[2] 1247
$ Mon Nov 18 14:01:00 PST 2019
[2]+ Done    date
$ ./myscript &
[2] 1269
$ jobs
[1]- Running    gnome-calculator &
[2]+ Running    myscript &
```

Managing Foreground/Background

`fg <#>`

- Brings job number **<#>** to the foreground for keyboard input

Ctrl-Z

- Suspends current foreground job temporarily, placing it in the background in a suspended (stopped) condition

`bg <#>`

- Resumes running suspended job **<#>** in the background

Optional: Sobell, "A practical Guide to Linux," pg 150-152, 304-306.

Using cntrl-z

```
$ vi a
(typed a cntrl-z)
zsh: suspended  vi a
$ vi b
(typed a cntrl-z)
zsh: suspended  vi b
$ jobs -l
[1]  - 2817 suspended  vi a
[2]  + 2818 suspended  vi b
$ ps 2817
  PID   TT  STAT      TIME COMMAND
 2817 s000  T      0:00.02 vi a
$ ps 2818
  PID   TT  STAT      TIME COMMAND
 2818 s000  T      0:00.02 vi b
$ %1
(resume execution of vi a)
```

Example: Background tasks

```
$ (sleep 10; cat > note.txt) &
[1] 985
$ jobs -l
[1]+   985 Running      ( sleep 10; cat > note.txt ) &
$
[1]+  Stopped           ( sleep 10; cat > note.txt )
$ fg
( sleep 10; cat > note.txt )
Hi there kitty
$ cat note.txt
Hi there kitty
$
```

Reminder: Ctrl-D indicates end-of-file

Using NOHUP

```
$ jobs -l
[1]-  1125 Stopped      vi a
[2]+  1126 Stopped      vi b
[3]   1163 Running      ( sleep 10; nohup cat > note.txt ) &
$ nohup: ignoring input and redirecting stderr to stdout
[3]   Exit 1            ( sleep 10; nohup cat > note.txt )
$ cat note.txt
cat: -: Bad file descriptor
$ %+
(resume execution of vi b)
```

How to kill a process?

Step 1:

First look at all the processes that are running and filter the ones you want to terminate

```
ps -aef | grep ggillespie
```

This shows the running processes limited to those belonging to **`ggillespie`**.

Step 2:

Then kill the process you selected. This command will kill the process with PID number of 125

```
kill 125
```

Step 3:

Then make sure that the process was killed by looking at the process table again and confirming 125 is not listed

```
ps -aef
```

`kill` command

Syntax:

```
kill [-signal] PID
kill [-signal] %JOBID
```

- Can send termination and non-termination signals! But will send **`TERM`** by default
- Catching a signal allows a process to clean up (removing temp file, etc) before exiting
- If signal isn't trapped, process will die.
- Signal **`-9`** can't be ignored.
- Only works on processes you own (or when run by **`root`** user).

Errors and Signals and Traps

- Consider the following program:

```
echo "this script will endlessly loop until you stop it"
while true; do
        : # Do nothing
done
```

- After you launch this script, it will hang (it is stuck inside a loop!)
- Once started, the script will continue until bash receives a signal to stop it.
- You can send such a signal by typing Ctrl-C which is the signal called **`SIGINT`** (short for SIGnal INTerupt)

Common signals…

Signal name	Signal number	Signal description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl-C)
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl-D)
SIGFPE	8	Issued if an illegal mathematical operation is attempted (division by 0)
SIGKILL	9	If a process gets this signal, it must quit immediately, and will not perform any cleanup operations
SIGALRM	14	Alarm clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default)

`kill -l` : this command will display all the signals supported by the system.

`trap` command

Syntax:

`trap cmd signals`

`signals` is a list of signals to intercept

`cmd` is command to execute when one of signals is received

- **`trap`** command allows to execute a command when a signal is received by your script

Optional: Sobell, "A practical Guide to Linux," pg 496-499.

Example: **`trap`** command

```
$ cat talkingloop.sh
#!/bin/bash
trap 'echo PROGRAM INTERRUPTED; exit 1' INT
while true
do
    echo "Program running"
    sleep 1
done

$ ./talkingloop.sh
Program running
Program running
Program running
<Ctrl-C>
PROGRAM INTERRUPTED
```

Note: Hard quotes with **`trap`** *cause variables to be evaluated when signal is received. Double quotes cause variables to be evaluated when command is encountered in the script!*

Next Lecture

- Diagnostic Output
- Java Logging Framework