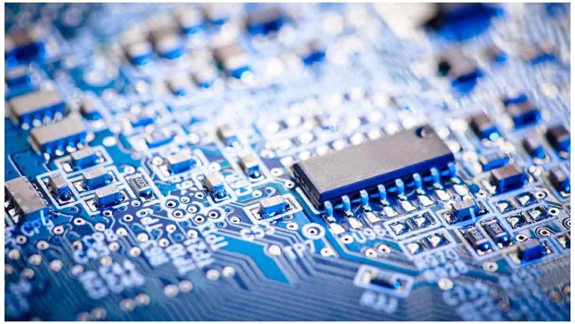
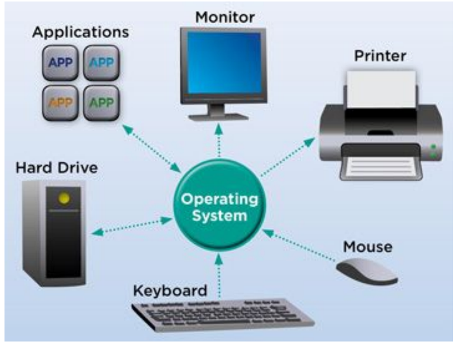
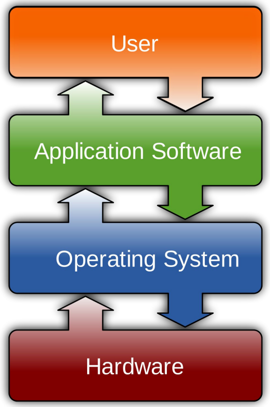
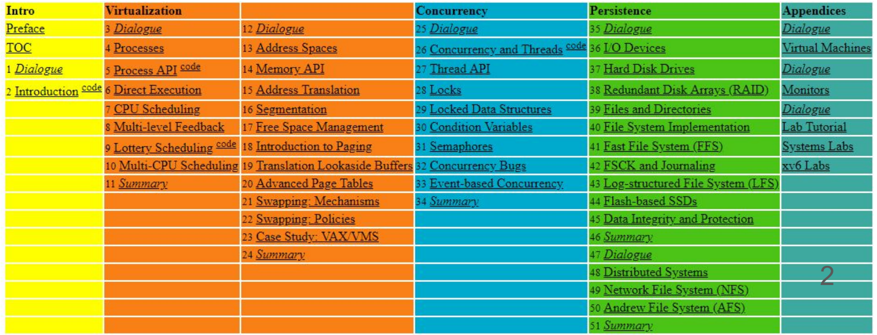




CS 3650 Computer Systems

Ferdinand Vesely / Alden Jackson

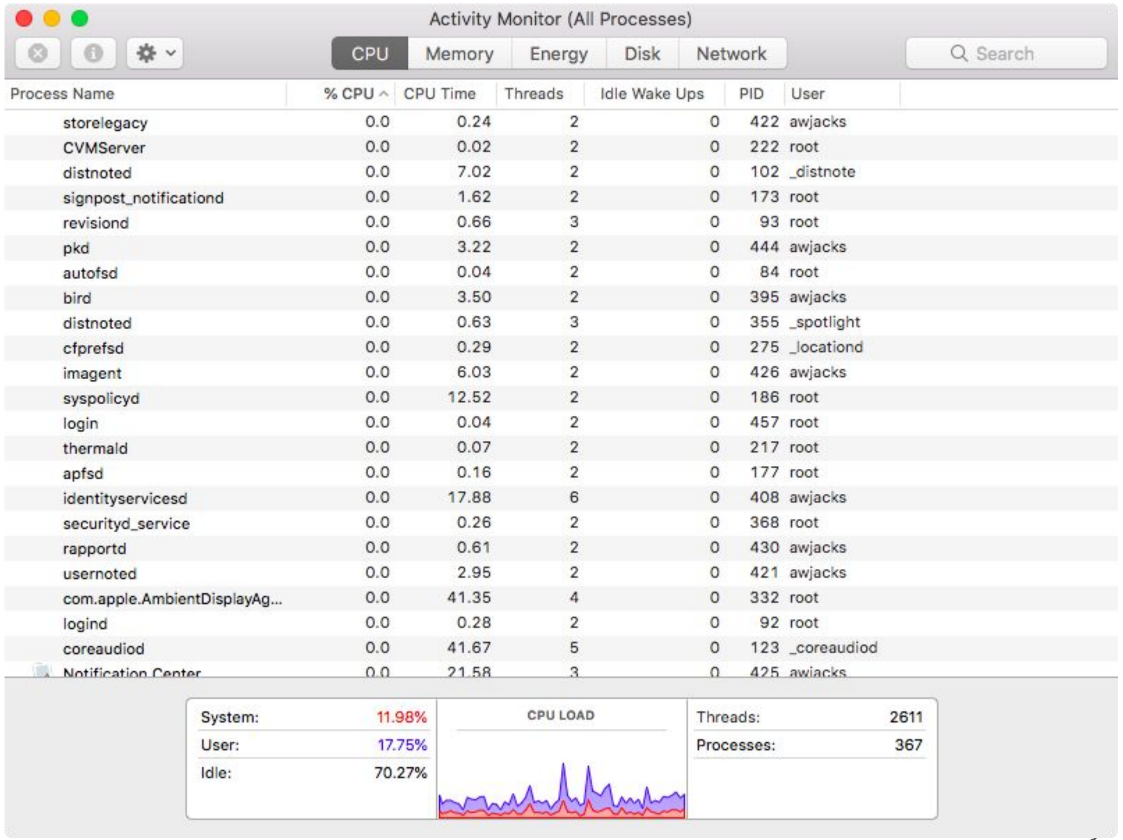




Intro	Virtualization		Concurrency	Persistence	Appendices
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	
		21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs	
		22 Swapping: Policies		45 Data Integrity and Protection	
		23 Case Study: VAX/VMS		46 *Summary*	
		24 *Summary*		47 *Dialogue*	
				48 Distributed Systems	
				49 Network File System (NFS)	
				50 Andrew File System (AFS)	
				51 *Summary*	

Pre-Class Warmup

- How many processes do you have open at any given time?
  - 10s, 100s? More!? :)



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
storelegacy	0.0	0.24	2	0	422	awjacks
CVMServer	0.0	0.02	2	0	222	root
distnoted	0.0	7.02	2	0	102	_distnote
signpost_notificationd	0.0	1.62	2	0	173	root
revisiond	0.0	0.66	3	0	93	root
pkd	0.0	3.22	2	0	444	awjacks
autofsd	0.0	0.04	2	0	84	root
bird	0.0	3.50	2	0	395	awjacks
distnoted	0.0	0.63	3	0	355	_spotlight
cfprefsd	0.0	0.29	2	0	275	_locationd
imagent	0.0	6.03	2	0	426	awjacks
syspolicyd	0.0	12.52	2	0	186	root
login	0.0	0.04	2	0	457	root
thermald	0.0	0.07	2	0	217	root
apfsd	0.0	0.16	2	0	177	root
identityservicesd	0.0	17.88	6	0	408	awjacks
securityd_service	0.0	0.26	2	0	368	root
rapportd	0.0	0.61	2	0	430	awjacks
usernoted	0.0	2.95	2	0	421	awjacks
com.apple.AmbientDisplayAg...	0.0	41.35	4	0	332	root
logind	0.0	0.28	2	0	92	root
coreaudiod	0.0	41.67	5	0	123	_coreaudiod
Notification Center	0.0	21.58	3	0	425	awjacks

System:	11.98%
User:	17.75%
Idle:	70.27%

CPU LOAD

Threads:	2611
Processes:	367

Upcoming Labs and Assignments

- Assignment 4 is due Thursday - How’s it going?
- Lab 5 will be on the Unix Process API: `fork()` and `exec()`
- Assignment 5 will be on writing a simple shell program

Lecture 5 - Processes

Ferdinand Vesely - Alden Jackson

Diving into the Operating Systems

- So far, we’ve been building some tools and understanding for our further exploration:
  - Assembly (fun?)
  - C

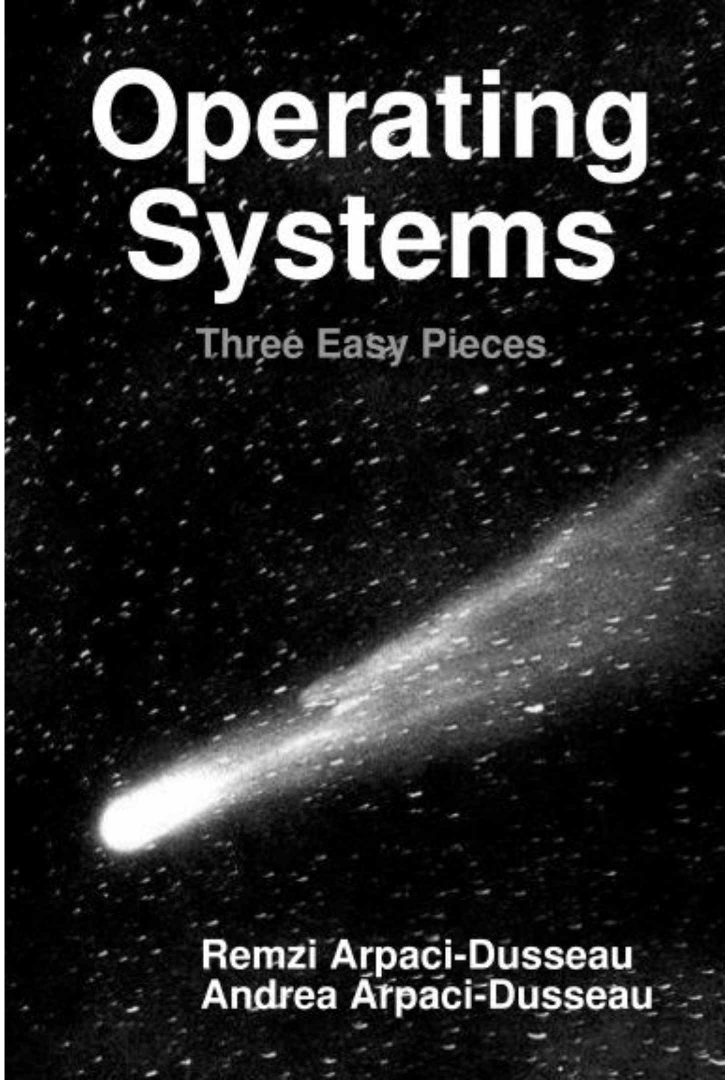
- Today we will dive into the OS itself
  - Knowing about registers and the concept of instructions will be useful
  - Knowing about memory as a linear array and addressing: also useful
  - Knowing C: well, it’s the language at the core of many commonly use OSs

OS: Virtualization + Abstraction

- The OS is a (software) land of magic and illusions
- Essentially, the purpose of an OS is to make a computer “easy” to use
- It does this by hiding the overwhelming complexities of underlying hardware behind an API
  - ➢ This is **abstraction**
- It also creates the illusion of an ideal, more general and powerful, machine
  - ➢ This is **virtualization**
- We will start by looking at how the processor virtualizes the CPU and the first abstraction: process

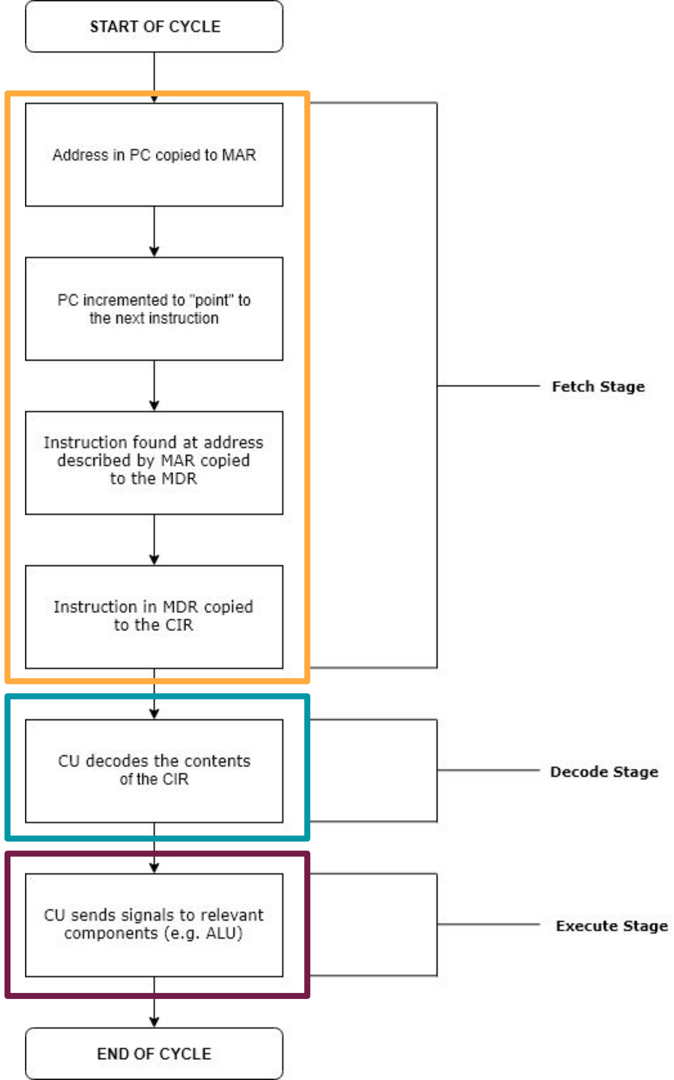
Recommended Reading

- The OSTEP book: up to Ch. 5
- Online: https://pages.cs.wisc.edu/~remzi/OSTEP/
- Hard copy: Lulu or Amazon



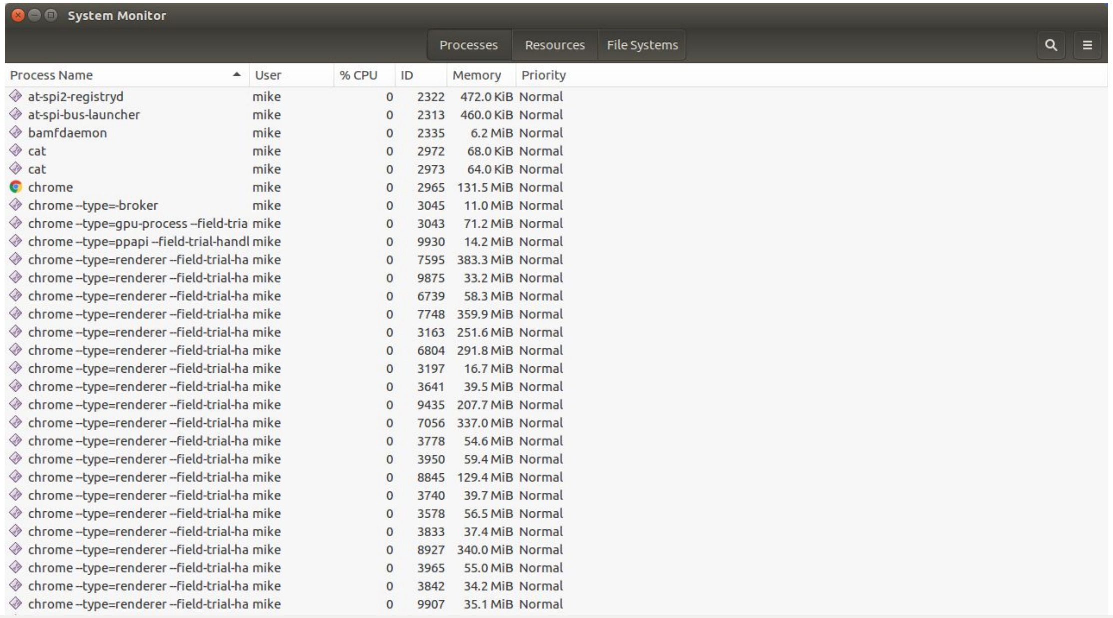
First: Instruction Execution

- Remember: code in an executable is a sequence of instructions
- A processor (core) performs an instruction at a time
- This is done in a fetch-decode-execute cycle
- If you have 4 cores, your processor can do 4 instances of this cycle at a time
- But … bottlenecks


START OF CYCLE
Address in PC copied to MAR
PC incremented to "point" to the next instruction
Instruction found at address described by MAR copied to the MDR
Instruction in MDR copied to the CIR
Fetch Stage
CU decodes the contents of the CIR
Decode Stage
CU sends signals to relevant components (e.g. ALU)
Execute Stage
END OF CYCLE

From the warm up

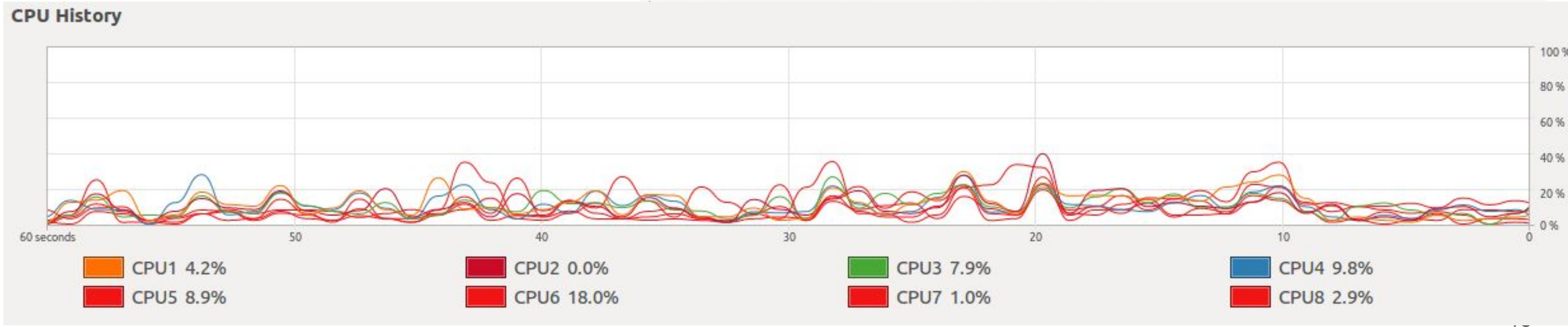
- I have lots of programs running, but I only have 8 CPUs that can do work



System Monitor

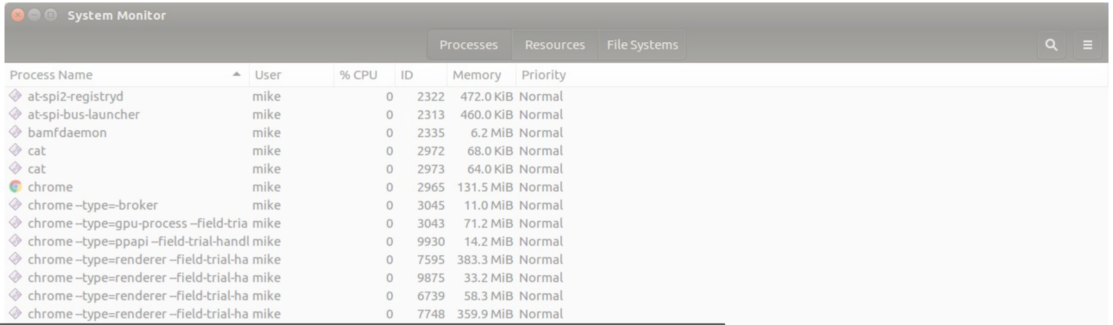
Processes | Resources | File Systems

Process Name	User	% CPU	ID	Memory	Priority
at-spi2-registryd	mike	0	2322	472.0 KiB	Normal
at-spi-bus-launcher	mike	0	2313	460.0 KiB	Normal
bamfdaemon	mike	0	2335	6.2 MiB	Normal
cat	mike	0	2972	68.0 KiB	Normal
cat	mike	0	2973	64.0 KiB	Normal
chrome	mike	0	2965	131.5 MiB	Normal
chrome –type=-broker	mike	0	3045	11.0 MiB	Normal
chrome –type=gpu-process –field-tria	mike	0	3043	71.2 MiB	Normal
chrome –type=ppapi –field-trial-handl	mike	0	9930	14.2 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	7595	383.3 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	9875	33.2 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	6739	58.3 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	7748	359.9 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3163	251.6 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	6804	291.8 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3197	16.7 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3641	39.5 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	9435	207.7 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	7056	337.0 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3778	54.6 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3950	59.4 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	8845	129.4 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3740	39.7 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3578	56.5 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3833	37.4 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	8927	340.0 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3965	55.0 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	3842	34.2 MiB	Normal
chrome –type=renderer –field-trial-ha	mike	0	9907	35.1 MiB	Normal



From the warm up

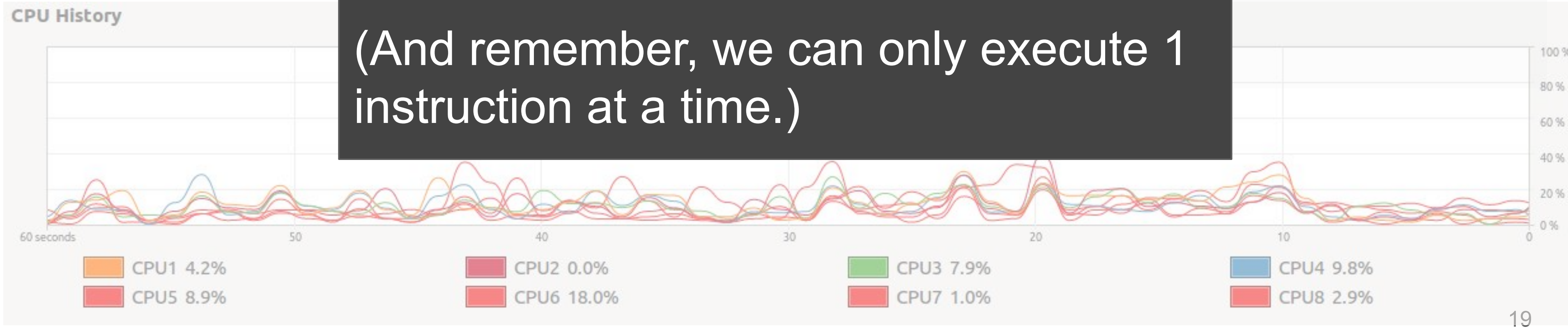
- I have lots of programs running, bu CPUs that c



Process Name	User	% CPU	ID	Memory	Priority
at-spi2-registryd	mike	0	2322	472.0 KiB	Normal
at-spi-bus-launcher	mike	0	2313	460.0 KiB	Normal
bamfdaemon	mike	0	2335	6.2 MiB	Normal
cat	mike	0	2972	68.0 KiB	Normal
cat	mike	0	2973	64.0 KiB	Normal
chrome	mike	0	2965	131.5 MiB	Normal
chrome --type=-broker	mike	0	3045	11.0 MiB	Normal
chrome --type=gpu-process --field-tria	mike	0	3043	71.2 MiB	Normal
chrome --type=ppapi --field-trial-handl	mike	0	9930	14.2 MiB	Normal
chrome --type=renderer --field-trial-ha	mike	0	7595	383.3 MiB	Normal
chrome --type=renderer --field-trial-ha	mike	0	9875	33.2 MiB	Normal
chrome --type=renderer --field-trial-ha	mike	0	6739	58.3 MiB	Normal
chrome --type=renderer --field-trial-ha	mike	0	7748	359.9 MiB	Normal

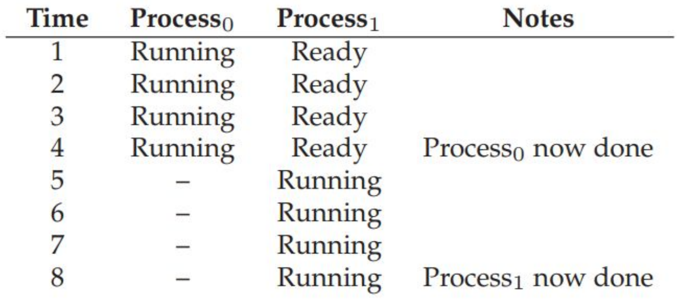
<u>The Problem:</u> So how does our Operating System provide the illusion of 100s of processes running at once?

(And remember, we can only execute 1 instruction at a time.)



Virtualization

- The Operating System(OS) runs one process at a time,
  - That executes one instruction a time
    - After some amount of time the process stops or finishes
    - Then the OS starts another process
    - Eventually the same process will run again and continue where it left off
    - and on and on.
    - This concept is known as **time sharing**



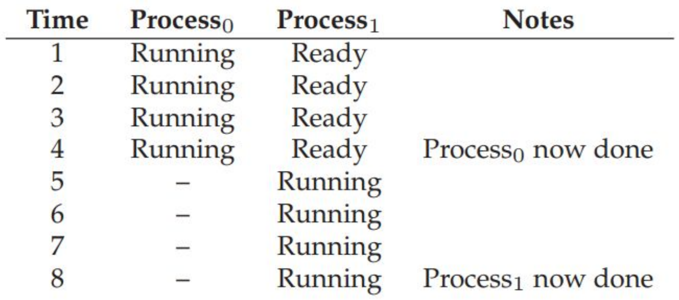
Time	Process$_0$	Process$_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process$_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process$_1$ now done

Figure 4.3: **Tracing Process State: CPU Only**

Process States

Each process can be in one of several states

- The Operating System (OS) schedules the state the process is in
- Typically these are:
  - Running - The process is executing on the CPU
  - Ready - The process is ready to execute, but the OS did not choose to run it
  - Blocked - The process has performed some kind of operation that blocks it from running.
    - In the figure below, an I/O operation has started that blocks other processors
    - I/O is a common bottleneck.



Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process_1 now done

Figure 4.3: **Tracing Process State: CPU Only**

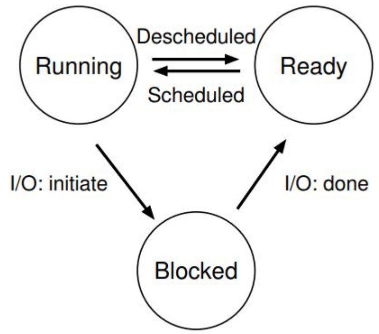


Figure 4.2: **Process: State Transitions**

Process States

- Each process can be in one of several states
- The Operating System (OS) schedules the state the proc
- Typically thes
  - Running -
  - Ready - Th did not cho
  - Blocked - operation t
    - In the figure below, an I/O operation has started that blocks other processors
    - I/O is a common bottleneck.

Next question, how does the OS switch states for a processor?

(What is the mechanism)

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 now done
		Running	
		Running	
		Running	
		Running	Process_1 now done

ng Process State: CPU Only

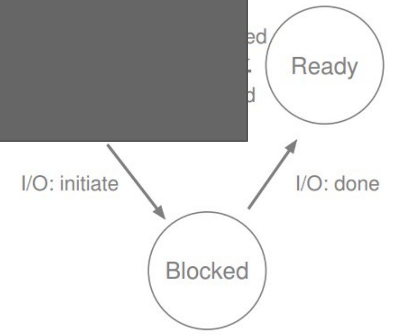


Figure 4.2: **Process: State Transitions**

OS Challenges to Virtualization

- Performance
  - How to implement virtualization without excessive overhead

- Control
  - How to run multiple processes efficiently without losing control over the CPU?
  - Without OS control, a process
    - could run forever
    - access memory it does not have access impacting system safety and security

Switching between processes: Cooperative

- Switching between processes is a challenge, because if the CPU is running a program, then the OS is not running
- If OS is not running, then how can it switch out/in processes?

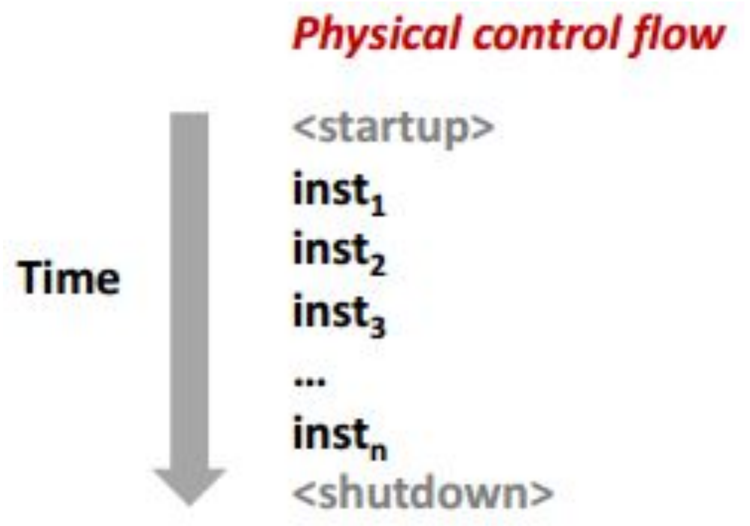
- Cooperative: Programs periodically give up CPU so OS can run
  - How: When a syscall is made or access is needed to something the OS manages, like i/o or creating a new process
  - OS assumes programs are trustworthy

- But what if a program doesn’t make syscalls or is NOT trustworthy

Mechanism:
Exceptional Control Flow

Remember

- Computers only really do one thing, they execute one instruction one after another
  - This is based on the execution in your program.
  - Your programs follow some control flow based on jumps and branches (and calls and returns)
    - This is based on your **programs state**.
  - However, sometime we want to react based on the **system state**
    - e.g. you hit Ctrl+C on the keyboard in your terminal and execution stops.



Two categories of Exceptional Control Flow Mechanisms

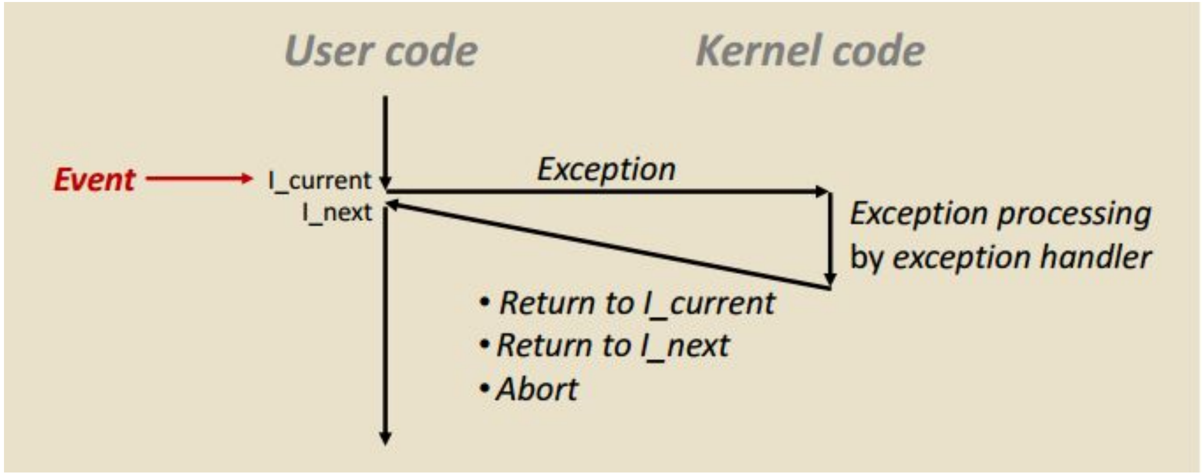
- Low level mechanism
    - Exceptions
        - Change in control flow in response to a system event.
        - This is implemented in hardware and OS software
- High level mechanisms
    - Process context switch
        - Implemented by OS software and hardware timer
            - e.g. It appears that multiple programs are running at once on your OS, but remember only one instruction at a time.
            - Context switches provide this illusion
    - Signals
        - Implemented by OS software
    - Nonlocal jumps: setjmp() or longjmp()
        - Implemented in C runtime library.

Two categories of Exceptional Control Flow Mechanisms

- Low level mechanism
  - Exceptions
    - Change in control flow in response to a system event.
    - This is implemented in hardware and OS software
- High level mechanisms
  - Process context switch
    - Implemented by OS software and hardware timer
      - e.g. It appears that multiple programs are running at once on your OS, but remember only one instruction at a time.
      - Context switches provide this illusion
  - Signals
    - Implemented by OS software
  - Nonlocal jumps: setjmp() or longjmp()
    - Implemented in C runtime library.

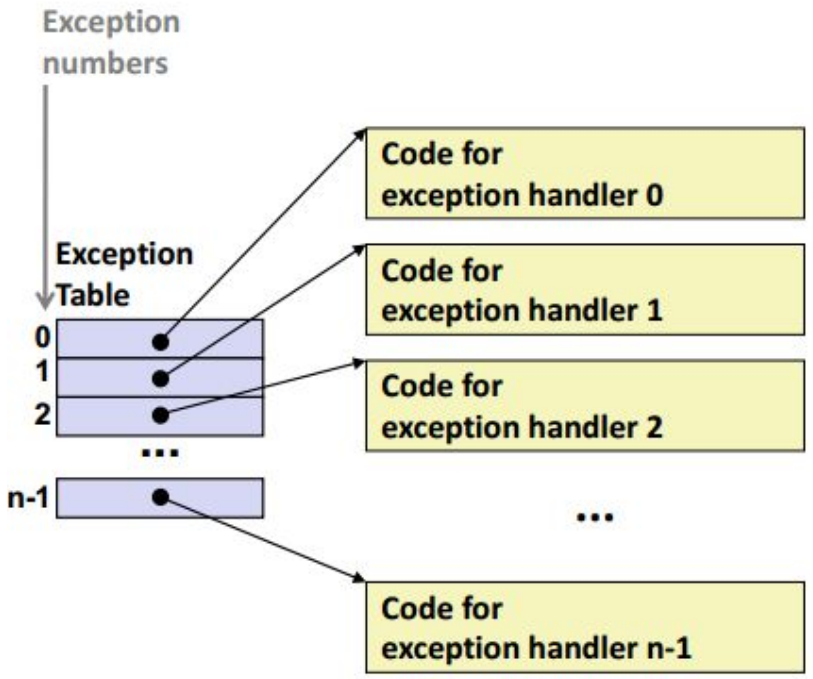
Exceptions

- An exception is a transfer of control to the OS kernel
  - The kernel is the memory-resident part of the OS
    - memory-resident meaning lives in memory forever--we do not modify this!
- Examples of exceptions we may be familiar with:
  - Divide by 0, arithmetic overflow, or typing Ctrl+C

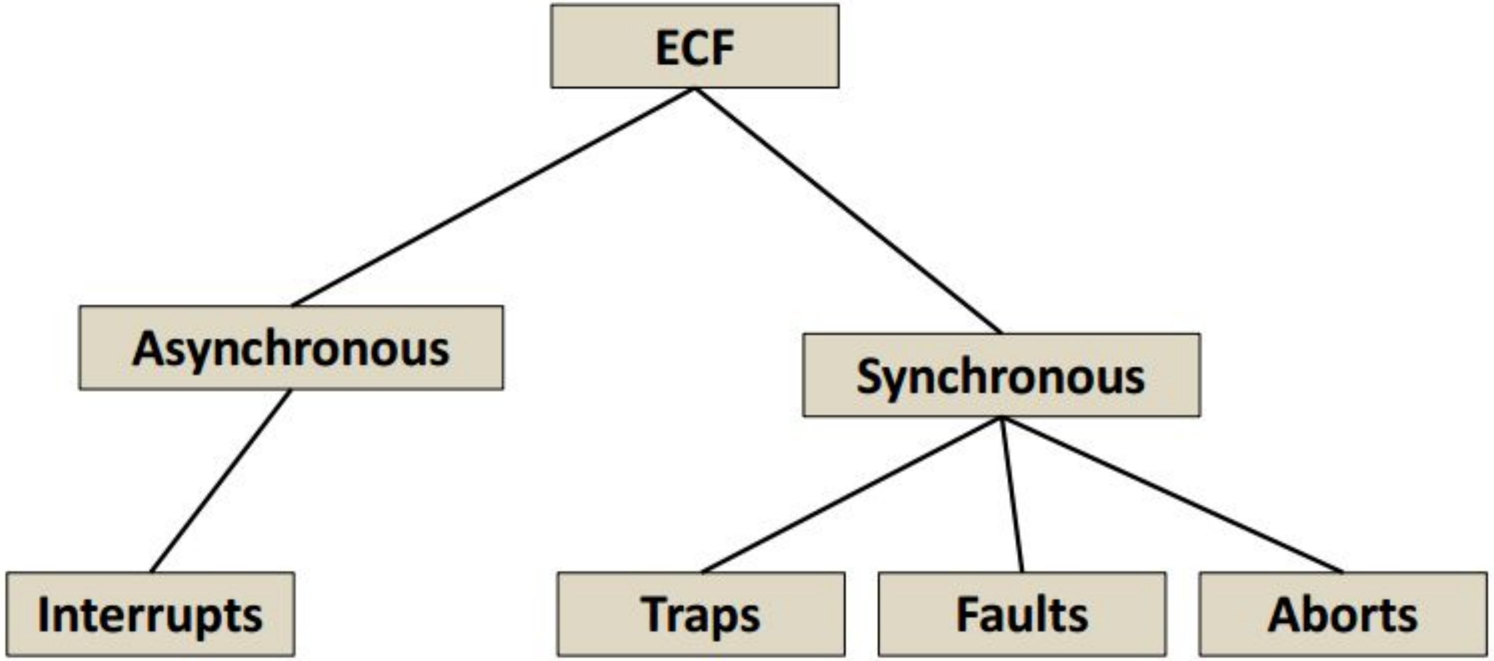


Exception Tables

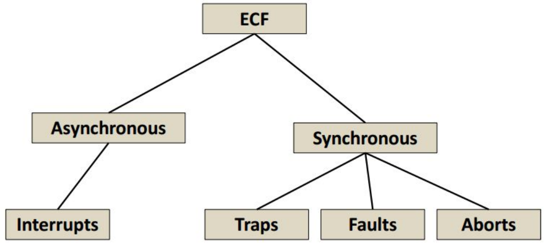
- Somewhere in the operating system, a table exists with different exceptions.
  - Think of it like a giant switch or many if else-if statements.
- Again, this part of a kernel, you cannot modify.
  - This code is in a “protected region” of memory
- For each exception, there is one way to <u>handle</u> it
  - (We call these “handlers”)


Exception numbers
Exception Table
0
1
2
...
n-1
Code for exception handler 0
Code for exception handler 1
Code for exception handler 2
...
Code for exception handler n-1

Exceptional Control Flow Taxonomy

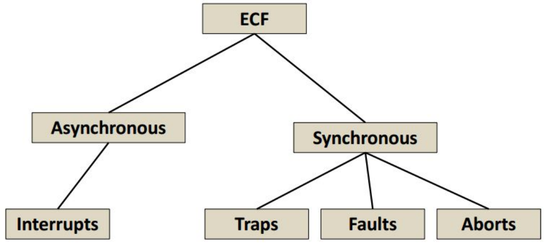


Asynchronous Exceptions (Interrupts)



- Caused by events external to processor
  - i.e. not from the result of an instruction the user wrote
  - e.g.
    - Timer interrupts scheduled to happen every few seconds
      - A kernel might use this to take back control from a user and do OS related tasks
    - Hitting Ctrl+C - Sends a signal (SIGINT) to end a program
    - Some network data arrives (I/O)
    - A nice example is while reading from disk
      - The processor can start reading, then hop over and perform some other tasks until memory is actually fetched.

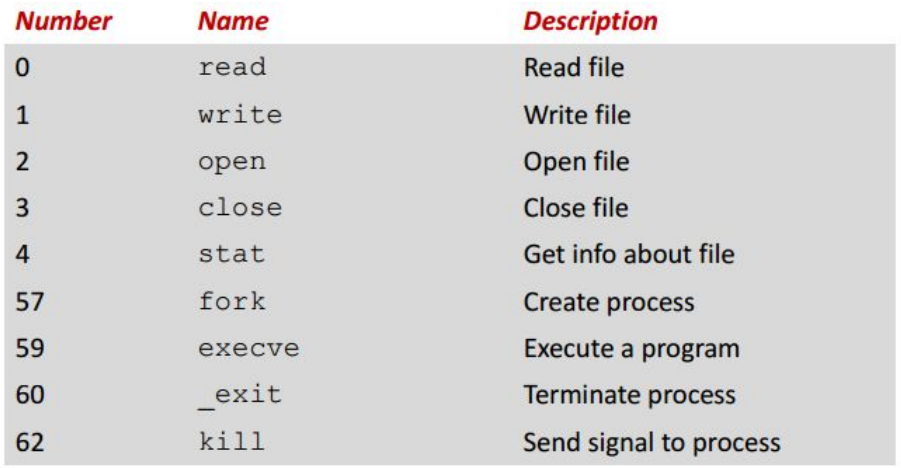
Synchronous Exceptions



- Events caused by executing an instruction
  - Traps
    - Intentionally done by the user
      - e.g. system calls, breakpoints (like in gdb)
    - Returns control to the next instruction
  - Faults
    - Unintentional, but possibly recoverable
      - e.g. page faults (we'll learn more about soon), floating point exceptions
    - Handled by re-executing current instruction or aborting execution
  - Aborts
    - Unintentional and unrecoverable
      - e.g. illegal instruction executed, parity error
- If you are using C++, typically you can only handle synchronous exceptions

System Calls

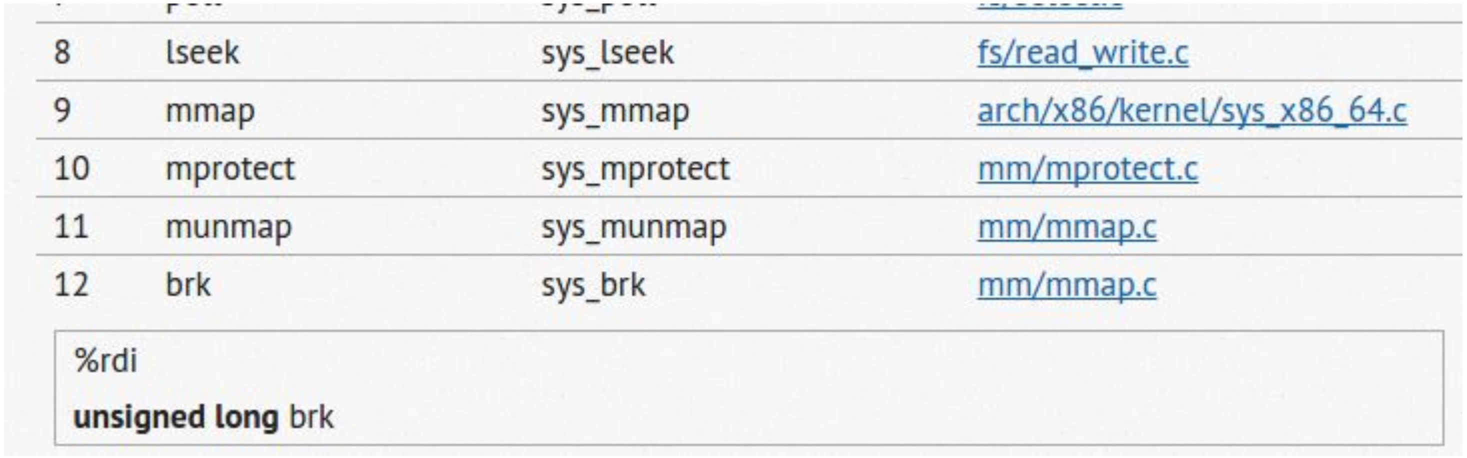
- syscall is the lowest level of interaction with an operating system from a C programmer
    - You may have used '_exit' in your assignment



Number	Name	Description
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

System Calls and arguments

- Helpful webpage with syscalls and arguments
  - https://filippo.io/linux-syscall-table/



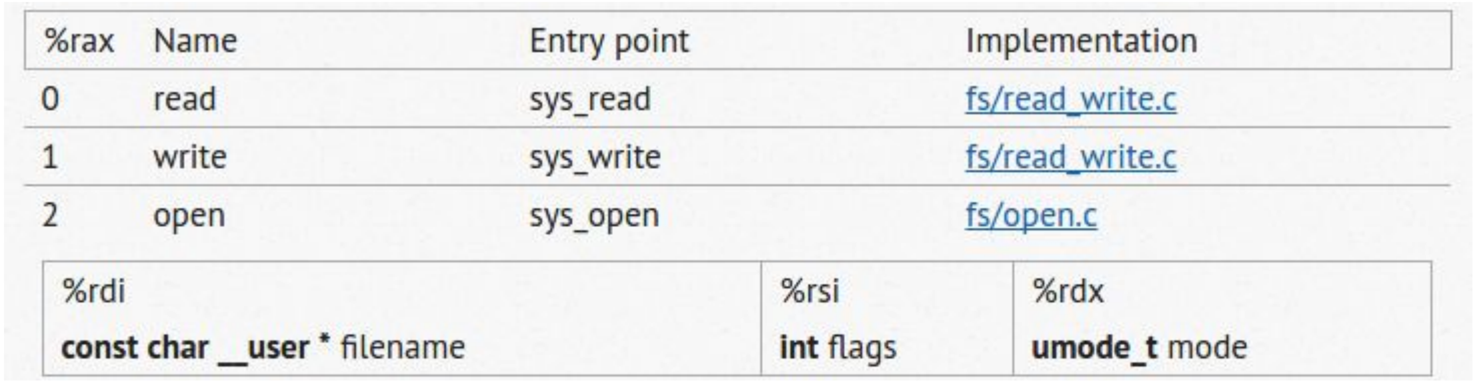
8	lseek	sys_lseek	fs/read_write.c
9	mmap	sys_mmap	arch/x86/kernel/sys_x86_64.c
10	mprotect	sys_mprotect	mm/mprotect.c
11	munmap	sys_munmap	mm/mmap.c
12	brk	sys_brk	mm/mmap.c

%rdi

unsigned long brk

Opening a File

- rax holds the system call # that we want to pass.
  - Other arguments accessed as follows

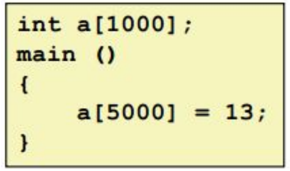


%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c

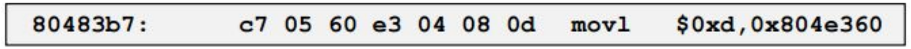
%rdi	%rsi	%rdx
const char __user * filename	**int** flags	**umode_t** mode

Our favorite: Invalid Memory Reference

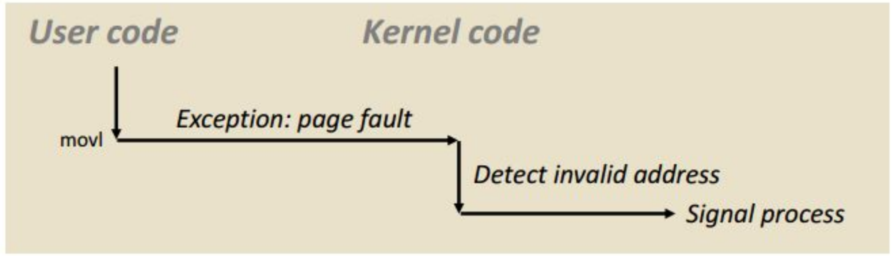
- That is, the segmentation fault
  - OS sends signal SIGSEGV to our user process
  - This time the program gets terminated.



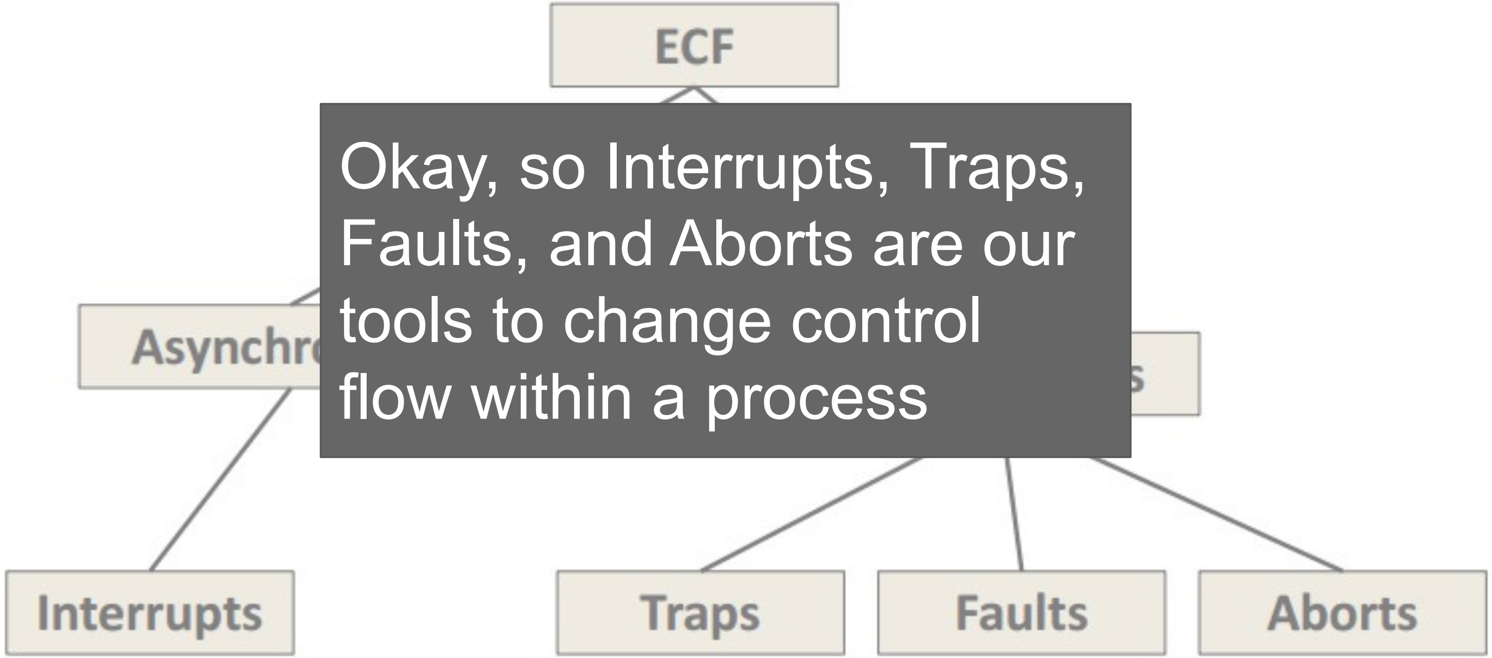
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```



```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Exceptional Control Flow Taxonomy

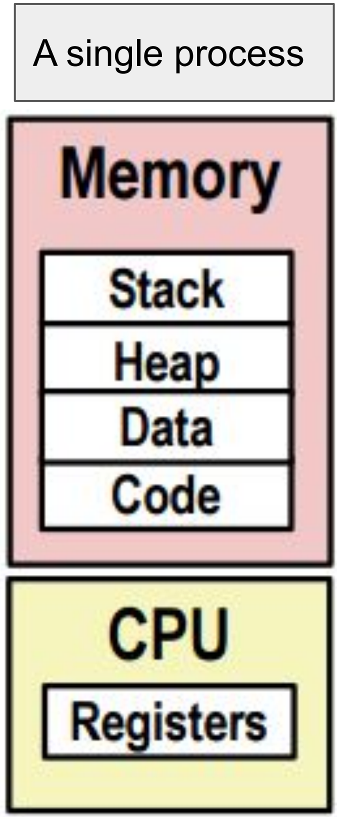


Okay, so Interrupts, Traps, Faults, and Aborts are our tools to change control flow within a process

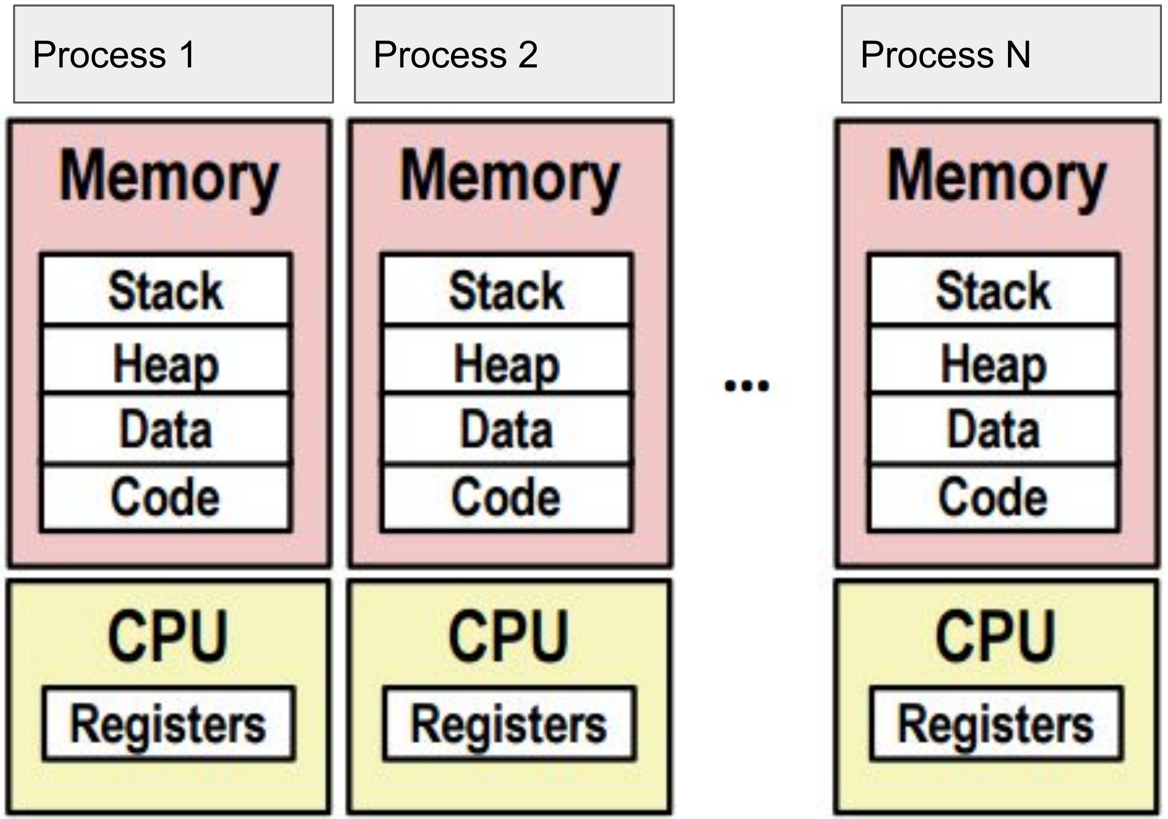
Processes

The Process

- A process is alive, a program is dead. Long live the process!
    - (A program is just the code.)
- Processes are organized by the OS using two key abstractions
    - Logical Control Flow
        - Programs “appear” to have exclusive control over the CPU
        - Done by “context switching”
    - Private Address Space
        - Each program “appears” to have exclusive use of main memory
        - Provided by mechanism called virtual memory

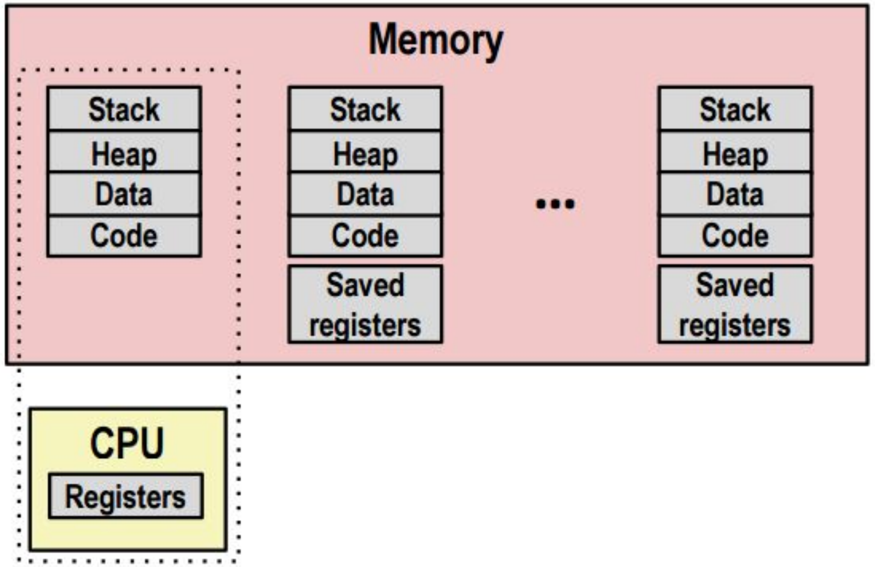


Multiprocessing: Illusion



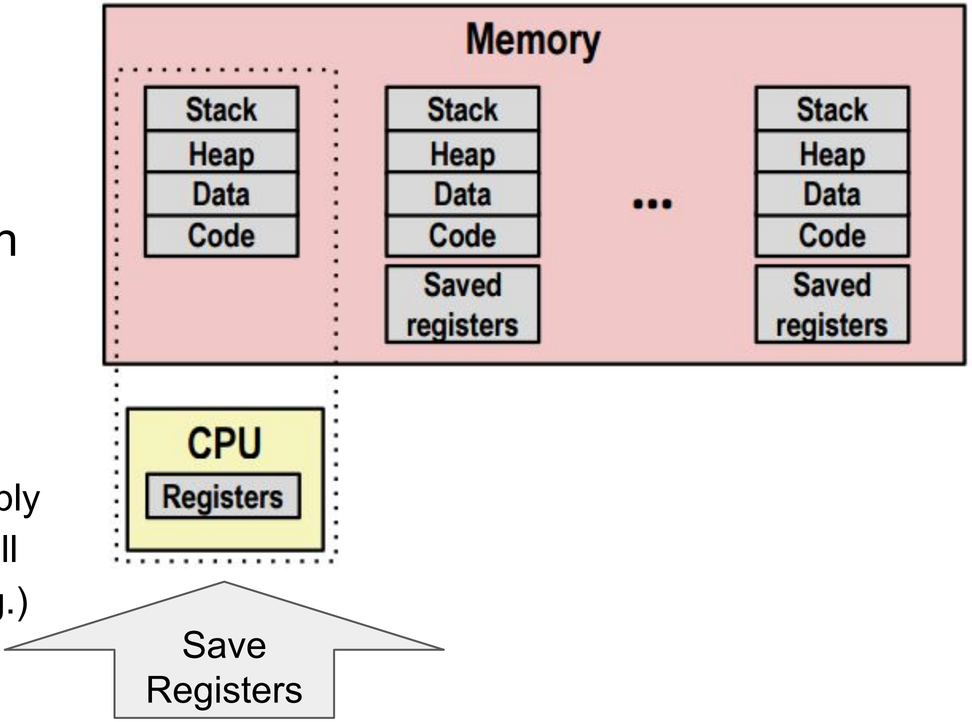
Multiprocessing: Reality

- Remember, at any time, only one processor is really running code
- Program execution is interleaved
- OS manages memory addresses in virtual memory
- OS stores the saved registers for different programs.
  - (At some point in this class, you probably figured 16 registers is not enough for all of the processes that you were running.)
- When we switch which process is executing this is a **context switch**



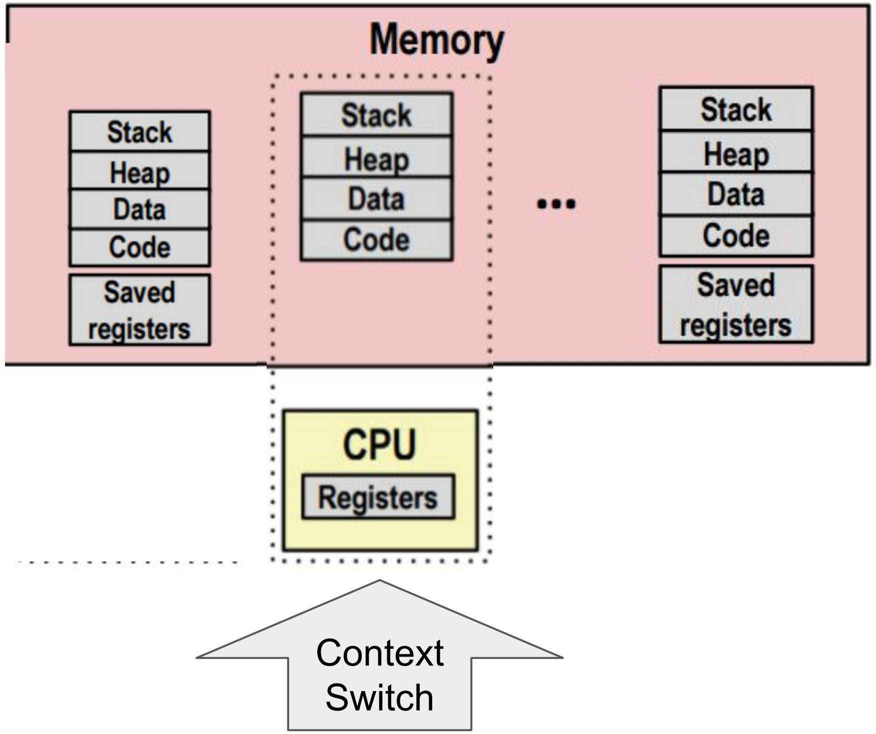
Multiprocessing: Reality

- Remember, at any time, only one processor is really running code
- Program execution is interleaved
- OS manages memory addresses in virtual memory
- OS stores the saved registers for different programs.
  - (At some point in this class, you probably figured 16 registers is not enough for all of the processes that you were running.)
- When we switch which process is executing this is a **context switch**



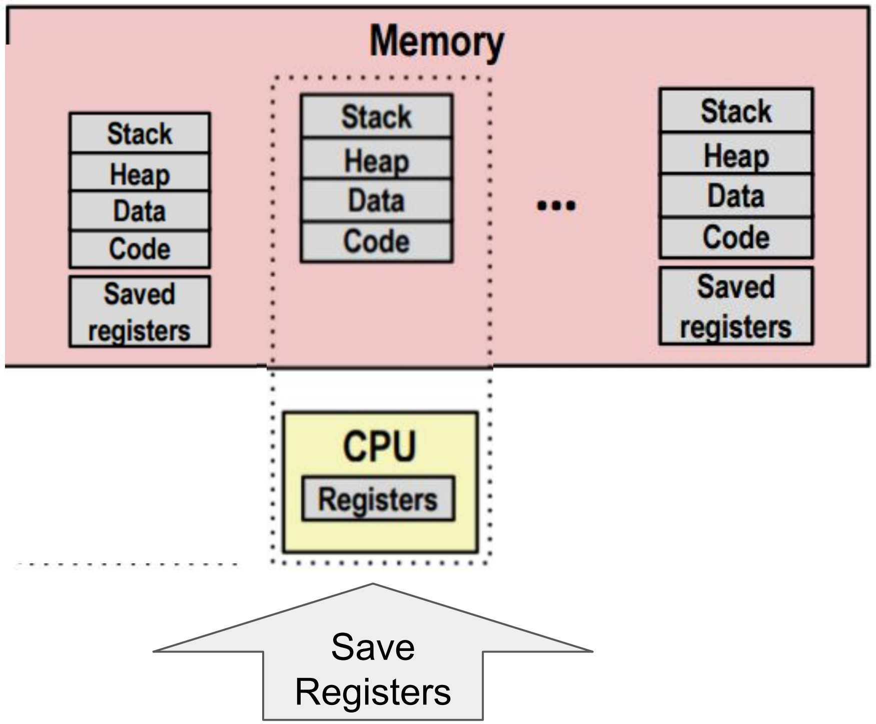
Multiprocessing: Reality

- Remember, at any time, only one processor is really running code
- Program execution is interleaved
- OS manages memory addresses in virtual memory
- OS stores the saved registers for different programs.
  - (At some point in this class, you probably figured 16 registers is not enough for all of the processes that you were running.)
- When we switch which process is executing this is a **context switch**



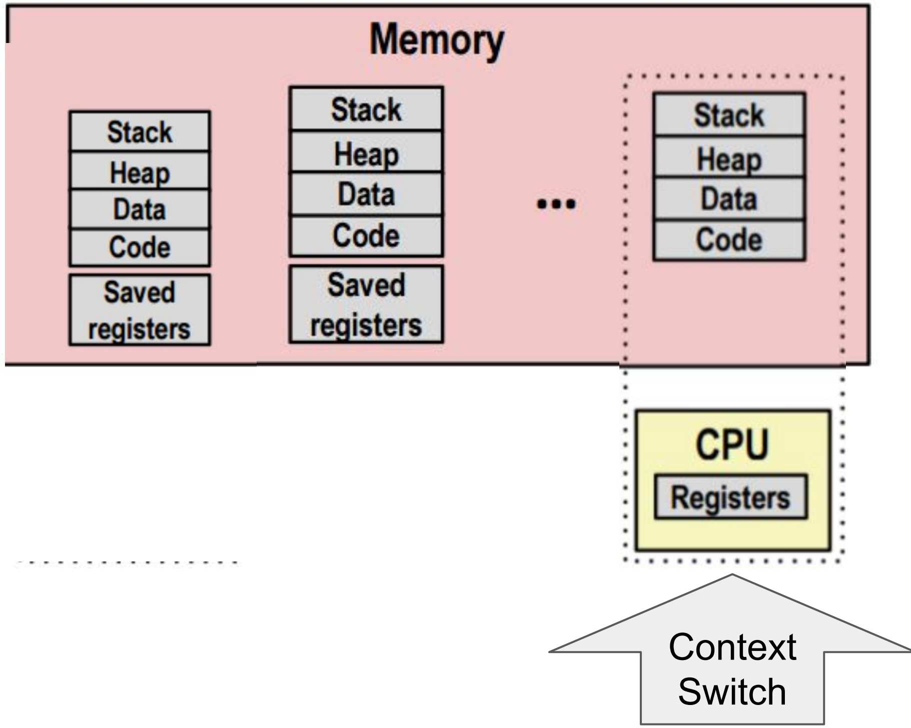
Multiprocessing: Reality

- Remember, at any time, only one processor is really running code
- Program execution is interleaved
- OS manages memory addresses in virtual memory
- OS stores the saved registers for different programs.
    - (At some point in this class, you probably figured 16 registers is not enough for all of the processes that you were running.)
- When we switch which process is executing this is a **context switch**



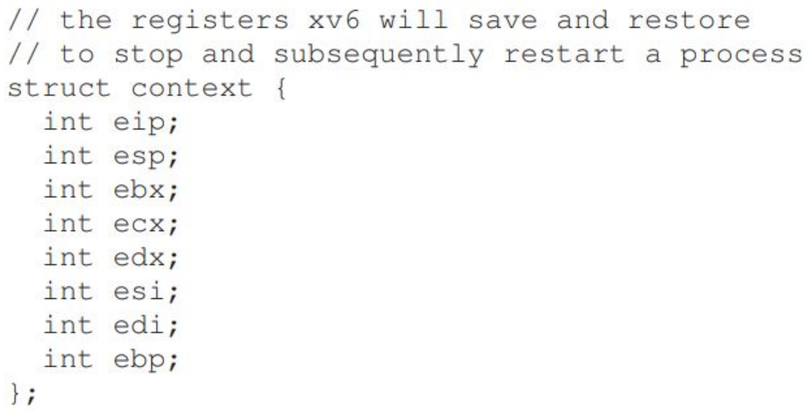
Multiprocessing: Reality

- Remember, at any time, only one processor is really running code
- Program execution is interleaved
- OS manages memory addresses in virtual memory
- OS stores the saved registers for different programs.
    - (At some point in this class, you probably figured 16 registers is not enough for all of the processes that you were running.)
- When we switch which process is executing this is a **context switch**



Storing Register Context | Data Structures

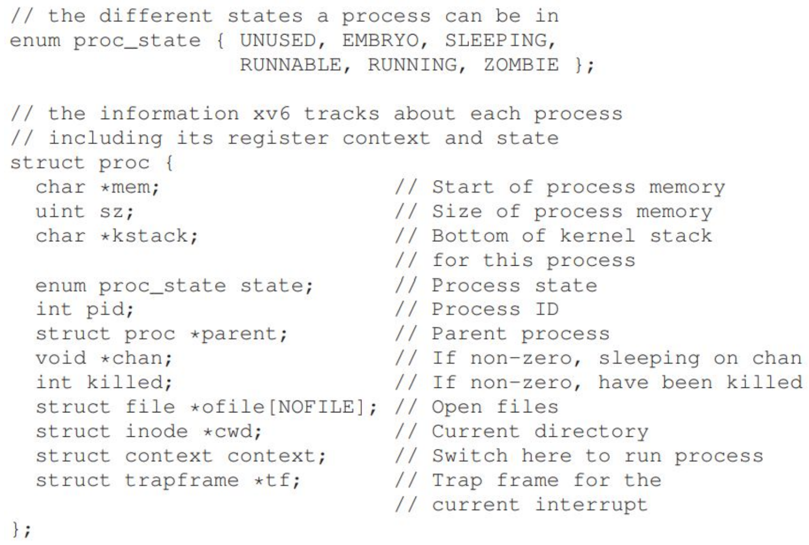
- In order to store the state of the registers, your OS will keep track of this information
- Typically there is a process list, and the list contains information like the registers.
- To the right is a *struct* for the xv6 operating system storing 32-bit registers. *We will use xv6 later in the semester.*



```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};
```

Storing Process Information | Data Structures

- Additional information such as the process state is stored by the OS.
- **<u>proc</u>** is the data structure which stores information about each process
- To the right is the `struct proc` for the xv6 operating system



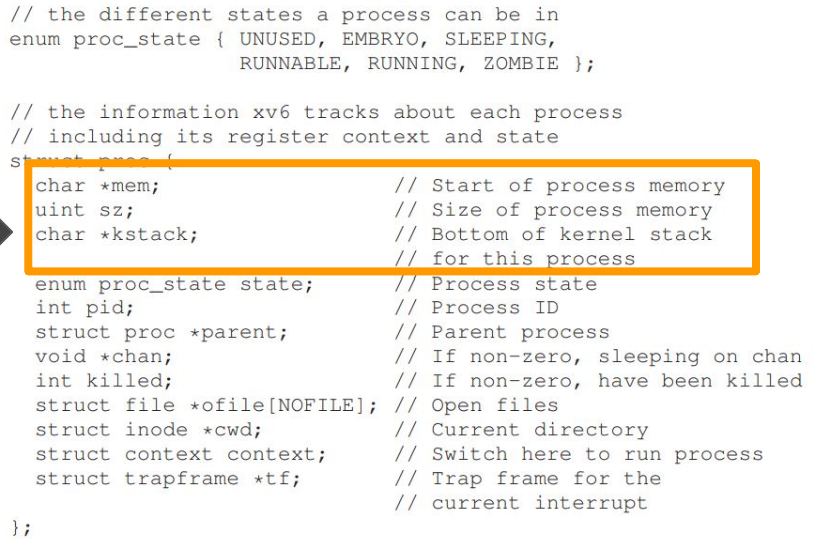
```
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Storing Process Information | Data Structures

- Additional information such as the process state is stored by the OS
- **proc** is the

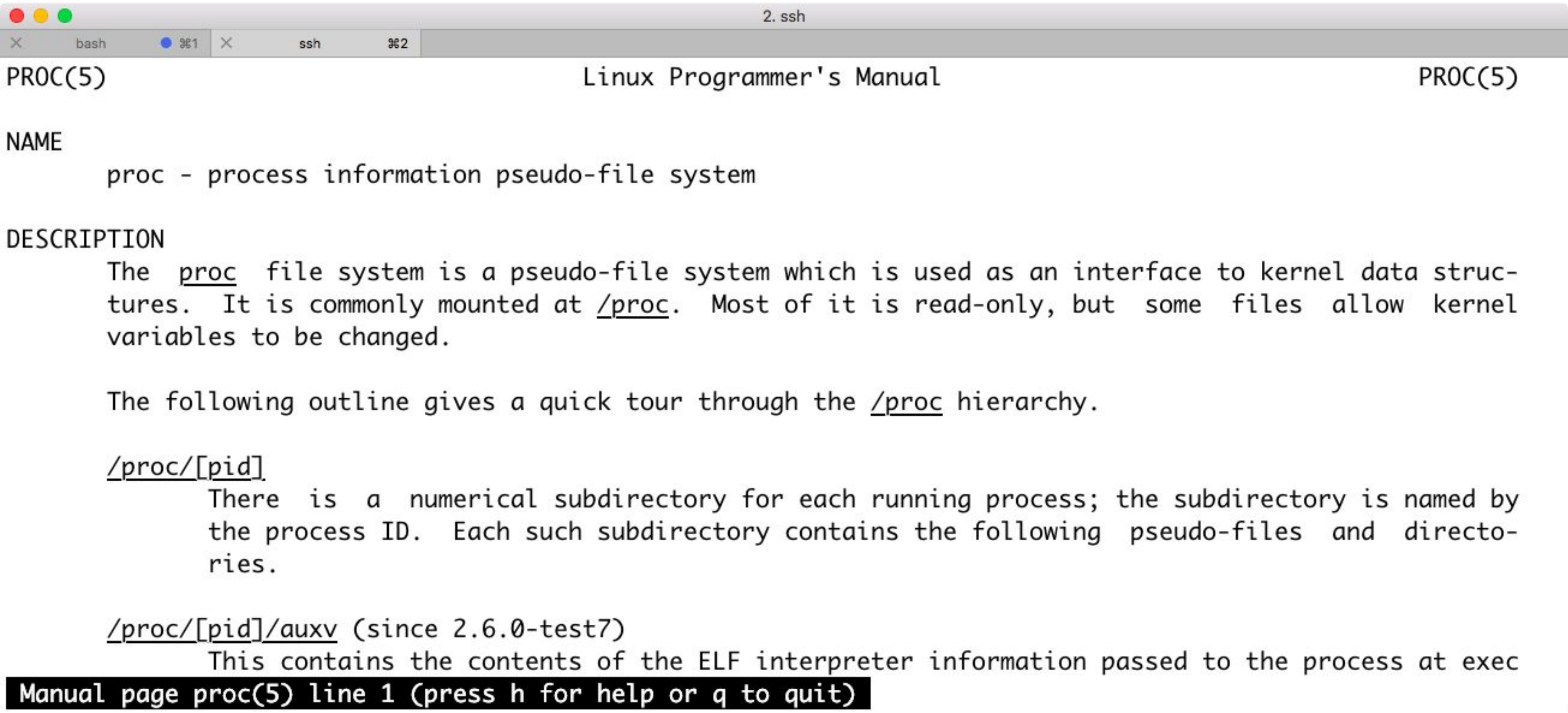
information

We are also familiar with some of these concepts



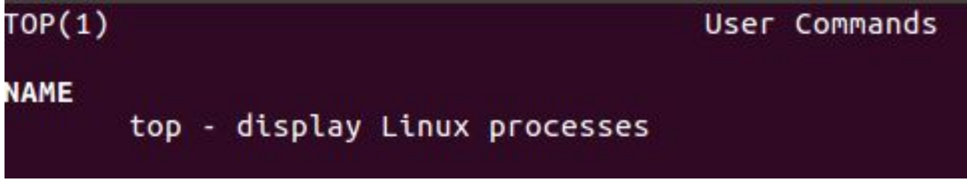
```
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

man proc



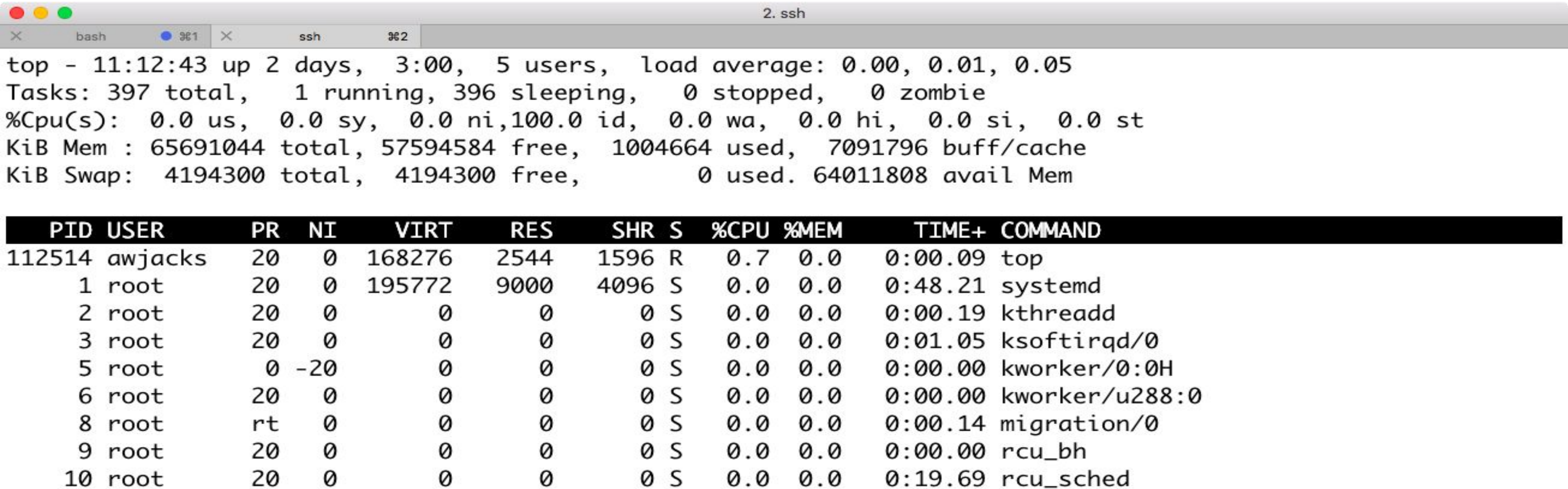
```
PROC(5)                    Linux Programmer's Manual                    PROC(5)

NAME
       proc - process information pseudo-file system

DESCRIPTION
       The  proc  file system is a pseudo-file system which is used as an interface to kernel data struc-
       tures.  It is commonly mounted at /proc.  Most of it is read-only, but  some  files  allow  kernel
       variables to be changed.

       The following outline gives a quick tour through the /proc hierarchy.

       /proc/[pid]
              There  is  a  numerical subdirectory for each running process; the subdirectory is named by
              the process ID.  Each such subdirectory contains the following  pseudo-files  and  directo-
              ries.

       /proc/[pid]/auxv (since 2.6.0-test7)
              This contains the contents of the ELF interpreter information passed to the process at exec
 Manual page proc(5) line 1 (press h for help or q to quit)
```

top



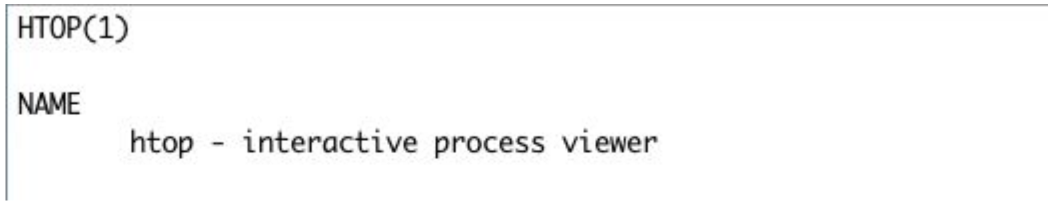
```
TOP(1)                    User Commands

NAME
       top - display Linux processes
```

- top is a program that will show linux processes that are running
  - Top shows all of the processes running on a system
  - Intuitively, it must be possible for a machine to host multiple processes, we do so when we ssh.



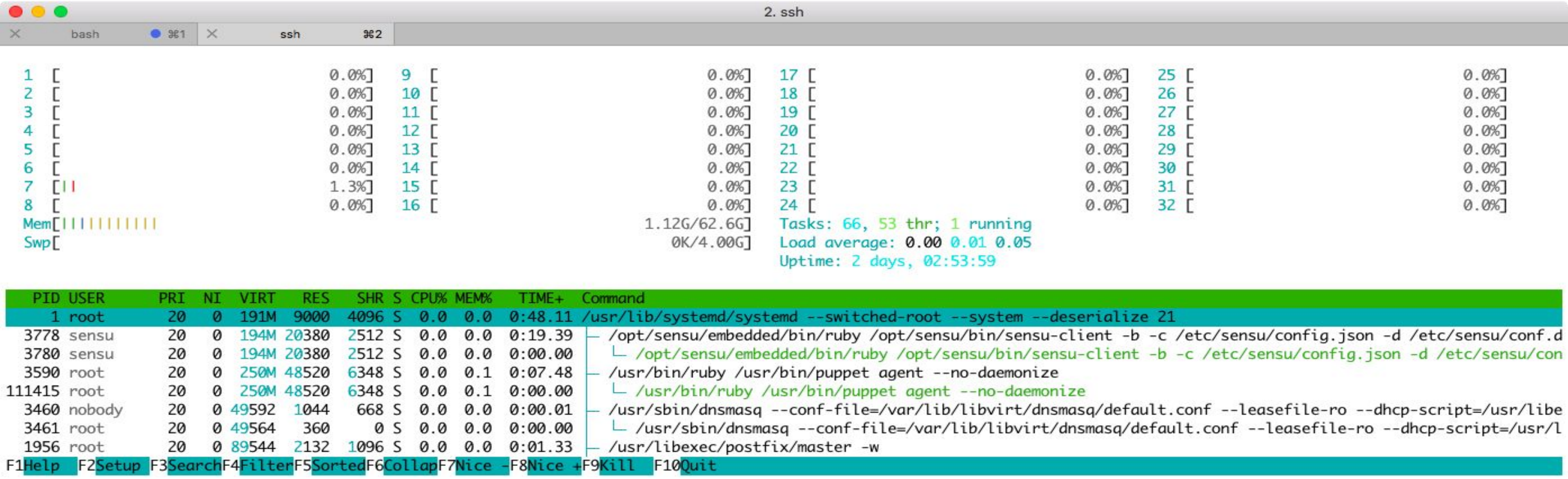
```
top - 11:12:43 up 2 days,  3:00,  5 users,  load average: 0.00, 0.01, 0.05
Tasks: 397 total,   1 running, 396 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 65691044 total, 57594584 free,  1004664 used,  7091796 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used. 64011808 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
112514 awjacks   20   0  168276   2544   1596 R   0.7  0.0   0:00.09 top
     1 root      20   0  195772   9000   4096 S   0.0  0.0   0:48.21 systemd
     2 root      20   0       0      0      0 S   0.0  0.0   0:00.19 kthreadd
     3 root      20   0       0      0      0 S   0.0  0.0   0:01.05 ksoftirqd/0
     5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
     6 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kworker/u288:0
     8 root      rt   0       0      0      0 S   0.0  0.0   0:00.14 migration/0
     9 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    10 root      20   0       0      0      0 S   0.0  0.0   0:19.69 rcu_sched
```

htop



```
HTOP(1)

NAME
       htop - interactive process viewer
```

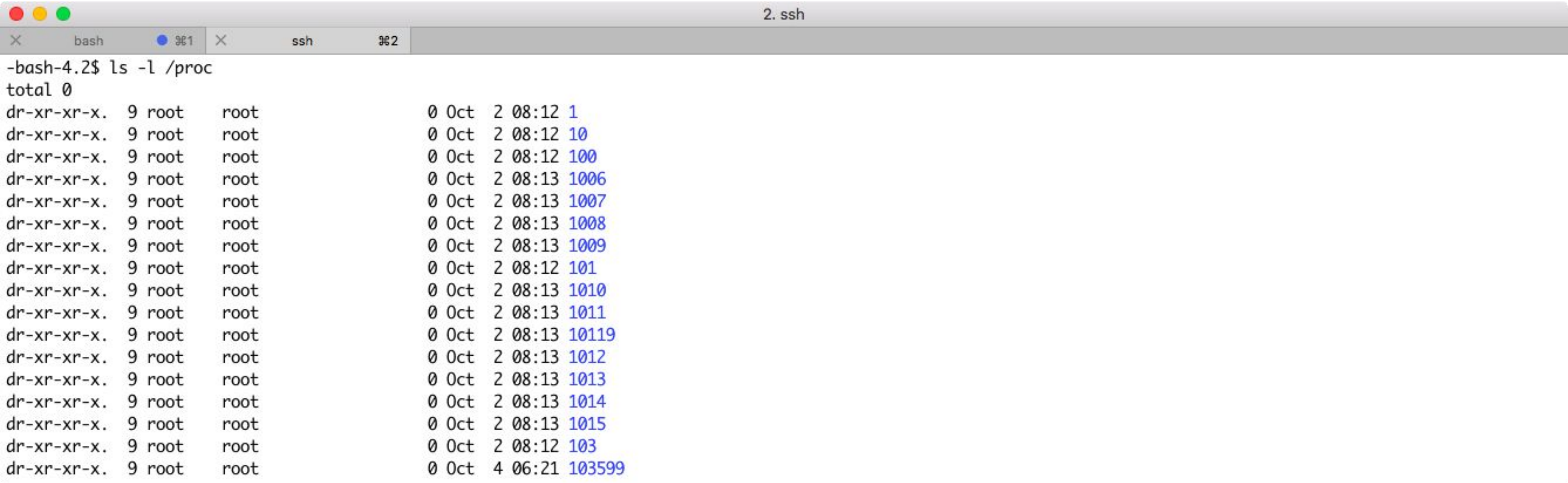
- htop is another program to show running processes
  - It shows cores and their load
  - It also shows the process tree (process / subprocess relationships)
  - It can be scrolled left/right and up/down



```
  1  [                                 0.0%]   9  [                                 0.0%]   17 [                                 0.0%]   25 [                                 0.0%]
  2  [                                 0.0%]   10 [                                 0.0%]   18 [                                 0.0%]   26 [                                 0.0%]
  3  [                                 0.0%]   11 [                                 0.0%]   19 [                                 0.0%]   27 [                                 0.0%]
  4  [                                 0.0%]   12 [                                 0.0%]   20 [                                 0.0%]   28 [                                 0.0%]
  5  [                                 0.0%]   13 [                                 0.0%]   21 [                                 0.0%]   29 [                                 0.0%]
  6  [                                 0.0%]   14 [                                 0.0%]   22 [                                 0.0%]   30 [                                 0.0%]
  7  [||                               1.3%]   15 [                                 0.0%]   23 [                                 0.0%]   31 [                                 0.0%]
  8  [                                 0.0%]   16 [                                 0.0%]   24 [                                 0.0%]   32 [                                 0.0%]
  Mem[|||||||||||                                                          1.12G/62.6G]   Tasks: 66, 53 thr; 1 running
  Swp[                                                                         0K/4.00G]   Load average: 0.00 0.01 0.05
                                                                                            Uptime: 2 days, 02:53:59

   PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
     1 root       20   0  191M  9000  4096 S  0.0  0.0  0:48.11 /usr/lib/systemd/systemd --switched-root --system --deserialize 21
  3778 sensu      20   0  194M 20380  2512 S  0.0  0.0  0:19.39 ├─ /opt/sensu/embedded/bin/ruby /opt/sensu/bin/sensu-client -b -c /etc/sensu/config.json -d /etc/sensu/conf.d
  3780 sensu      20   0  194M 20380  2512 S  0.0  0.0  0:00.00 │  └─ /opt/sensu/embedded/bin/ruby /opt/sensu/bin/sensu-client -b -c /etc/sensu/config.json -d /etc/sensu/con
  3590 root       20   0  250M 48520  6348 S  0.0  0.1  0:07.48 ├─ /usr/bin/ruby /usr/bin/puppet agent --no-daemonize
111415 root       20   0  250M 48520  6348 S  0.0  0.1  0:00.00 │  └─ /usr/bin/ruby /usr/bin/puppet agent --no-daemonize
  3460 nobody     20   0 49592  1044   668 S  0.0  0.0  0:00.01 ├─ /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf --leasefile-ro --dhcp-script=/usr/libe
  3461 root       20   0 49564   360     0 S  0.0  0.0  0:00.00 │  └─ /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf --leasefile-ro --dhcp-script=/usr/l
  1956 root       20   0 89544  2132  1096 S  0.0  0.0  0:01.33 ├─ /usr/libexec/postfix/master -w
F1Help  F2Setup F3SearchF4FilterF5SortedF6CollapF7Nice -F8Nice +F9Kill  F10Quit
```

Viewing processes (Like we did with *top* or system monitor)

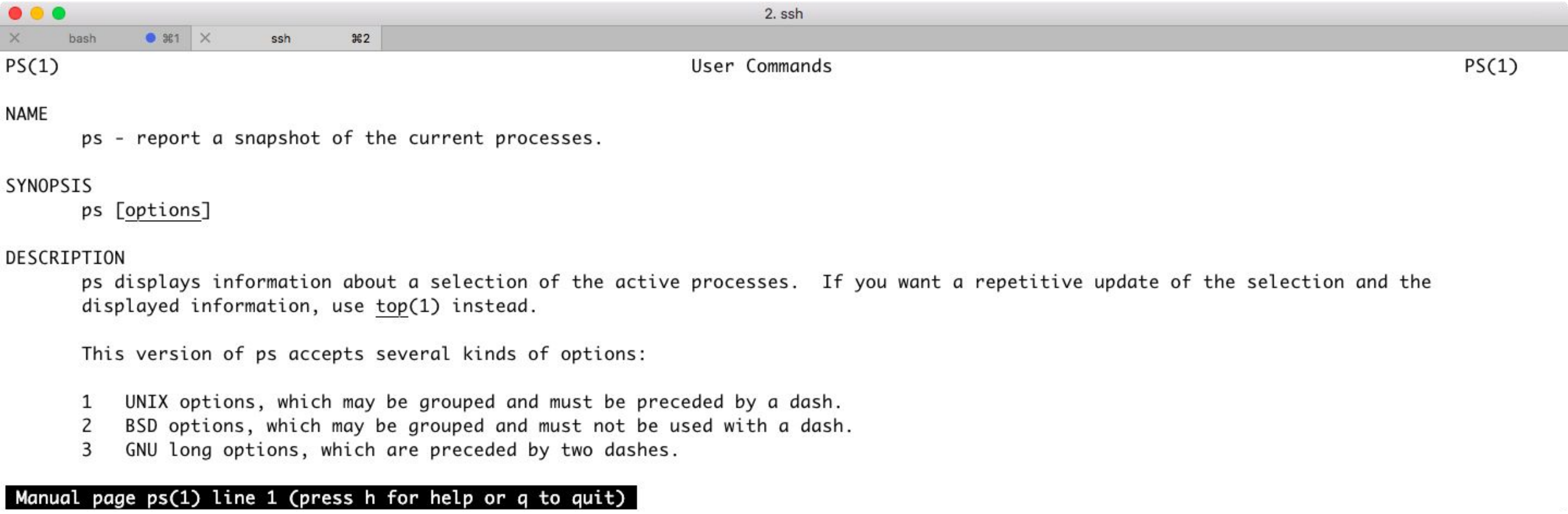
- proc itself is like a filesystem
  - (We'll talk more about everything in Unix being viewed as a file).
- We can navigate to it with cd /proc then list all of the processes.



```
-bash-4.2$ ls -l /proc
total 0
dr-xr-xr-x.  9 root    root                0 Oct  2 08:12 1
dr-xr-xr-x.  9 root    root                0 Oct  2 08:12 10
dr-xr-xr-x.  9 root    root                0 Oct  2 08:12 100
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1006
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1007
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1008
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1009
dr-xr-xr-x.  9 root    root                0 Oct  2 08:12 101
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1010
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1011
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 10119
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1012
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1013
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1014
dr-xr-xr-x.  9 root    root                0 Oct  2 08:13 1015
dr-xr-xr-x.  9 root    root                0 Oct  2 08:12 103
dr-xr-xr-x.  9 root    root                0 Oct  4 06:21 103599
```

man ps | Run *ps -ef*

- (Another way to view actively running processes is through the *ps* program.
    - *-ef* means view all of the processes



```
PS(1)                                   User Commands                                   PS(1)

NAME
       ps - report a snapshot of the current processes.

SYNOPSIS
       ps [options]

DESCRIPTION
       ps displays information about a selection of the active processes.  If you want a repetitive update of the selection and the
       displayed information, use top(1) instead.

       This version of ps accepts several kinds of options:

       1   UNIX options, which may be grouped and must be preceded by a dash.
       2   BSD options, which may be grouped and must not be used with a dash.
       3   GNU long options, which are preceded by two dashes.

 Manual page ps(1) line 1 (press h for help or q to quit)
```

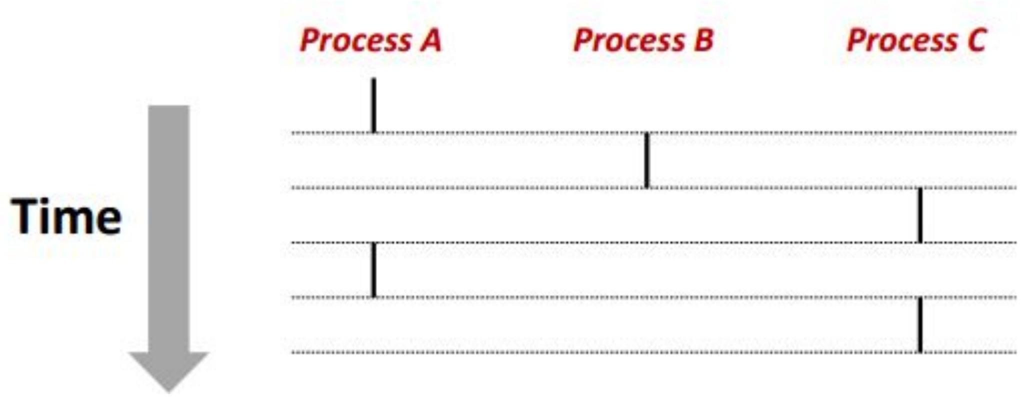
Gathering more information from proc

- We can run *cat stat* to output status information from proc

- Try some of the examples below in your VM: https://www.networkworld.com/article/2693548/unix-viewing-your-processes-through-the-eyes-of-proc.html

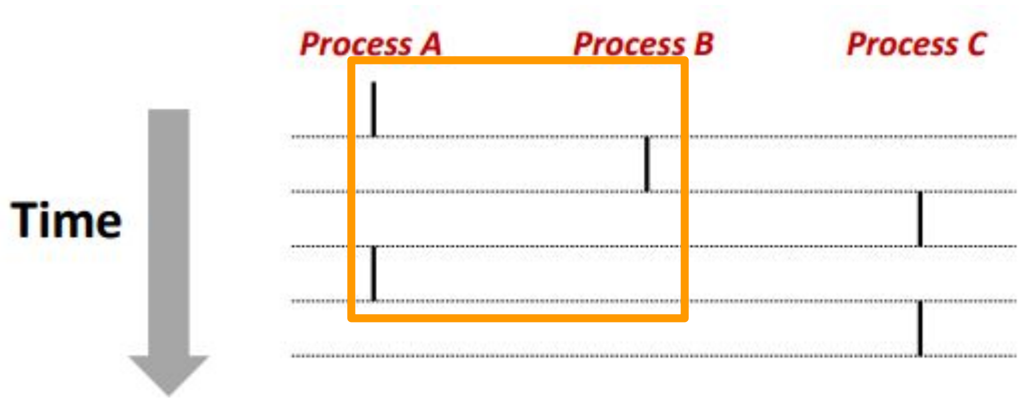
Concurrent Processing

- Each process running has its own control flow
- If they overlap in their lifetime, then they are running concurrently
  - otherwise they are sequential
- Remember only 1 process at a time can execute
  - On a single core, which processes here are concurrent relative to each other?
    - Concurrent:
  - Which are sequential?
    - Sequential:



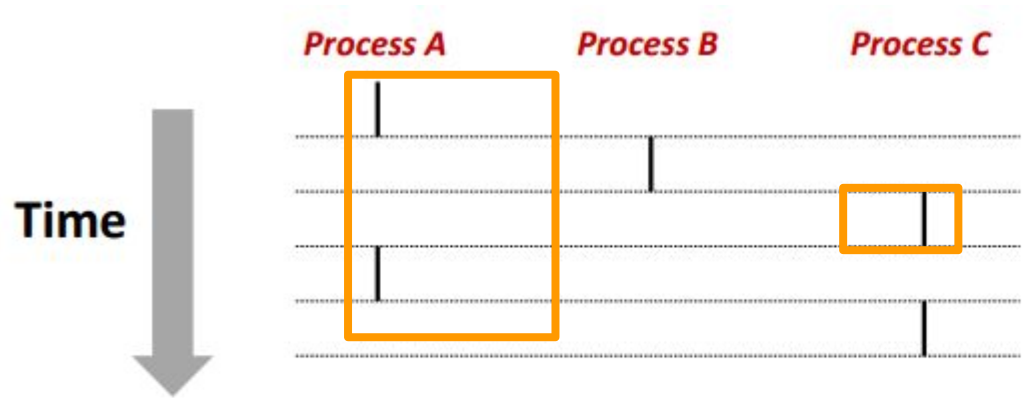
Concurrent Processing

- Each process running has its own control flow
- If they overlap in their lifetime, then they are running concurrently
  - otherwise they are sequential
- Remember only 1 process at a time can execute
  - On a single core, which processes here are concurrent relative to each other?
    - Concurrent: A&B
  - Which are sequential?
    - Sequential:



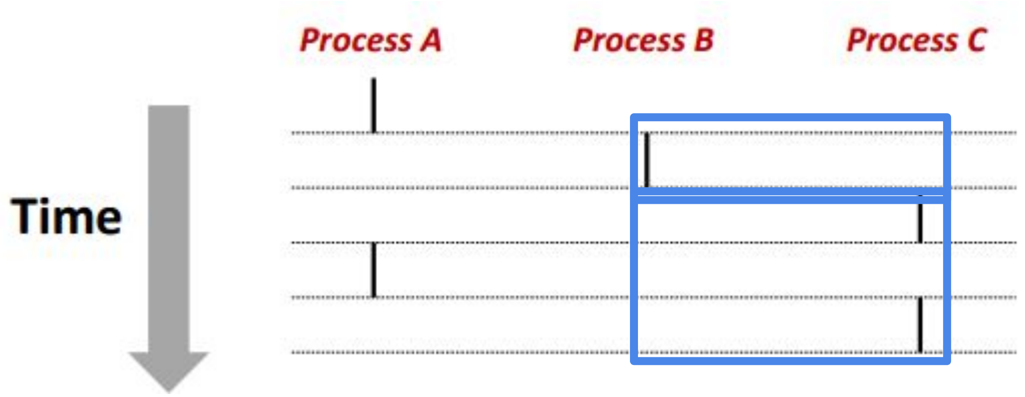
Concurrent Processing

- Each process running has its own control flow
- If they overlap in their lifetime, then they are running concurrently
  - otherwise they are sequential
- Remember only 1 process at a time can execute
  - On a single core, which processes here are concurrent relative to each other?
    - Concurrent: A&B, A&C
  - Which are sequential?
    - Sequential:



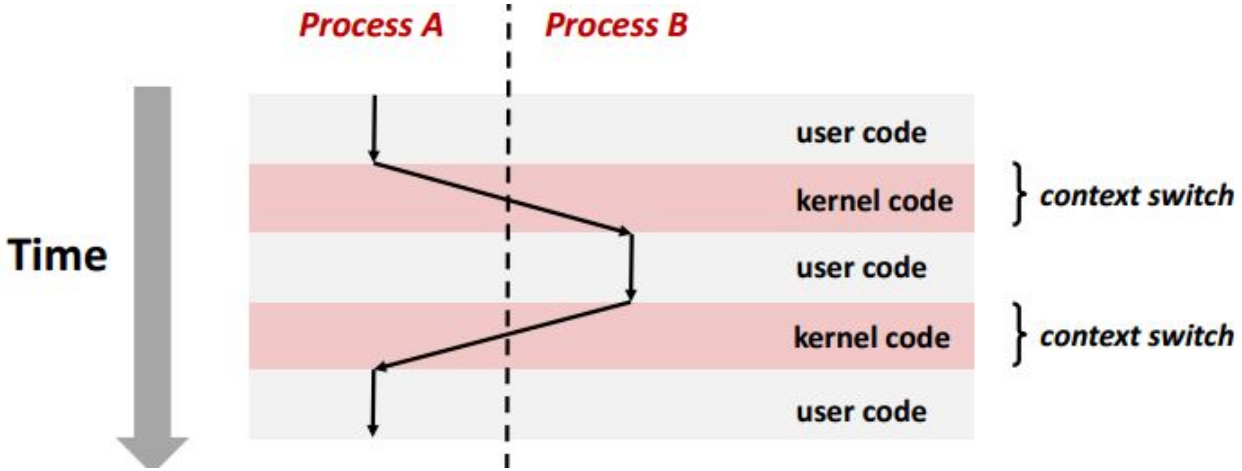
Concurrent Processing

- Each process running has its own control flow
- If they overlap in their lifetime, then they are running concurrently
    - otherwise they are sequential
- Remember only 1 process at a time can execute
    - On a single core, which processes here are concurrent relative to each other?
        - Concurrent: A&B, A&C
    - Which are sequential?
        - Sequential: B &C



Context Switching Illustration

- Processes are managed by a shared chunk of memory-resident OS code called the **<u>kernel</u>**
  - The kernel is not a separate process itself, but runs as part of other existing processes
- Context Switches pass the control flow from one process to another
  - Note how going from A to B (and B to A) requires some kernel code to be executed

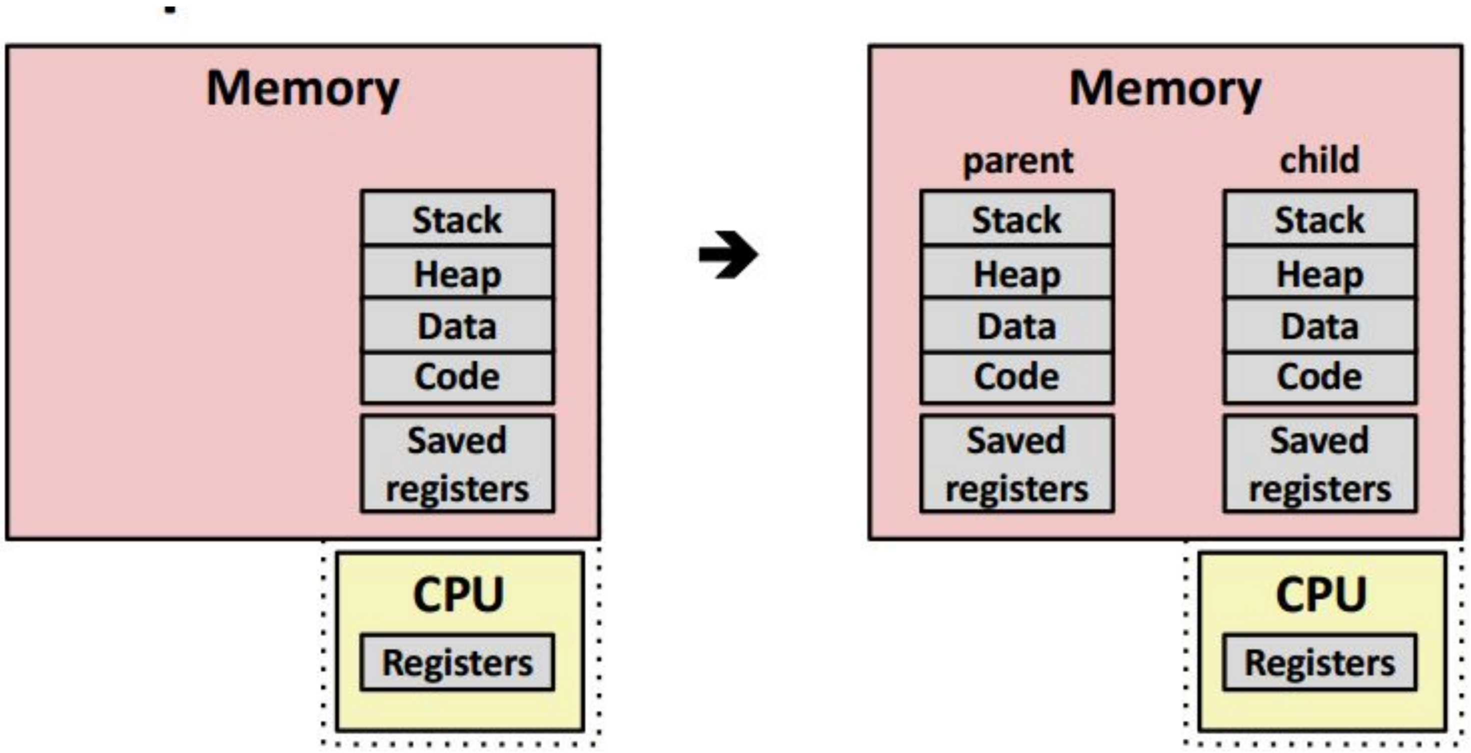


Process Control

Creating a Process

- When we want to create a new process, we can do so from our parent process using the **fork**() command.
    - This creates a new child process that runs.
        - Conceptually, this new child is a clone of itself
- int fork(void)
    - Returns 0 to the child process, child's PID returned to the parent process
    - PID = process ID
        - Child is almost identical to parent
        - Child gets a copy (that is separate) to the parent's virtual address space
        - Child gets a copy of open file descriptors
        - Child has a different PID than parent.
    - Note: Fork actually returns twice (once to the parent, and once to the child), even though it is called once.

Conceptual View of fork() | The before and after



Process State

- When our process is running, it may be in one of the following states
    - Running
        - Executing or waiting to be executed (i.e. scheduled to execute by the kernel)
    - Stopped
        - Process is suspended and will not be scheduled until further notice
            - e.g. out of main memory, process is blocked from executing by another, etc.
    - Terminated
        - Process is stopped permanently

Terminating Process

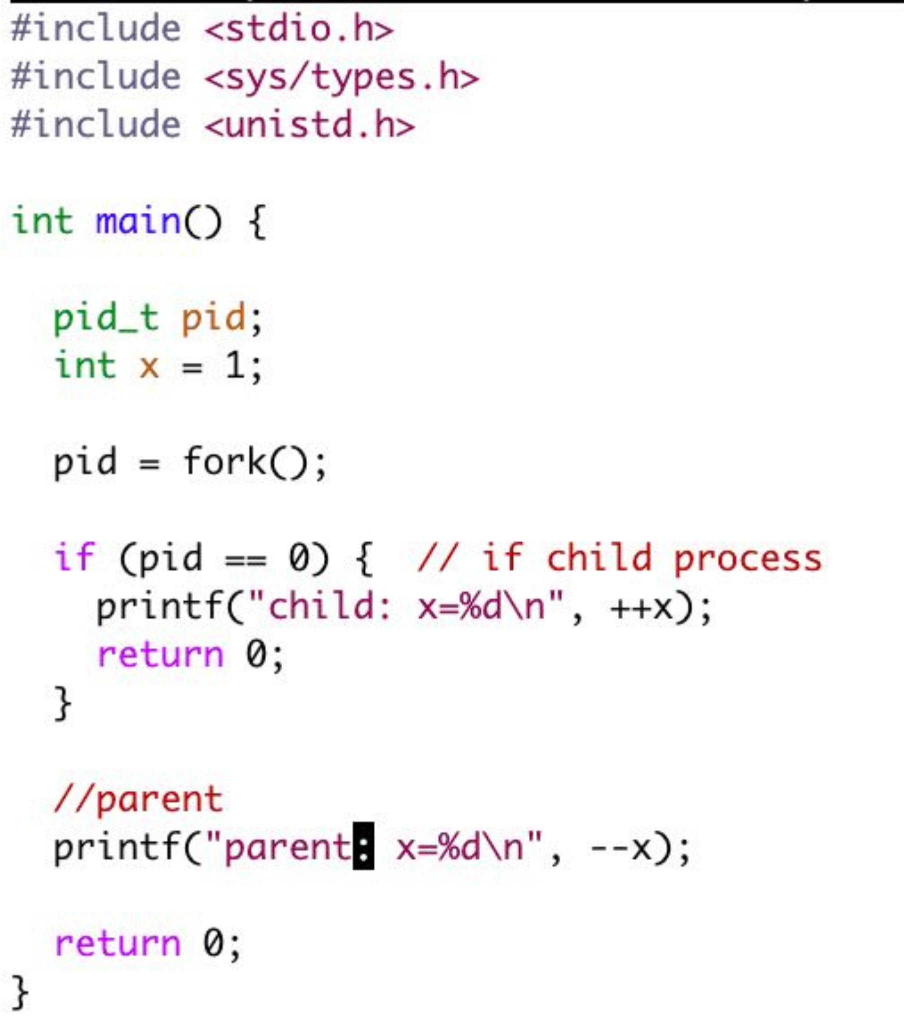
- Process may be terminated for 3 reasons
  - 1. Receives a signal to terminate
  - 2. Returns from *main* routine (what we have normally been doing in the class)
  - 3. Calling the *exit* function
    - void exit(int status)
      - Terminates with a given status
      - Returning 0 means no error
      - When exit is called, this only happens once, and it does not return
        - Note that if we have an error in our system, sometimes we do not want to exit right away (e.g. safety critical system)

Additional Process commands

- pid_t getpid(void)
    - Return PID of the current process
- pid_t getppid(void)
    - Returns PID of parent process
- Note that when we create a process with fork
    - The parent child relationship, makes a tree.
- (Note pid_t is a signed integer)

Fork Example

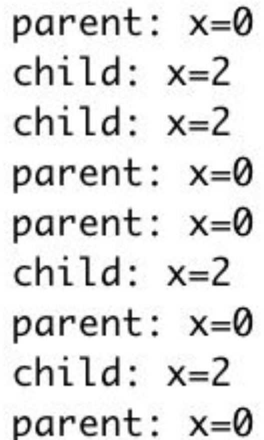
- Code walkthrough
  - Store a pid
  - fork our parent process and create a child
  - printf from our parent and/or printf from our child
- What will the following print out?



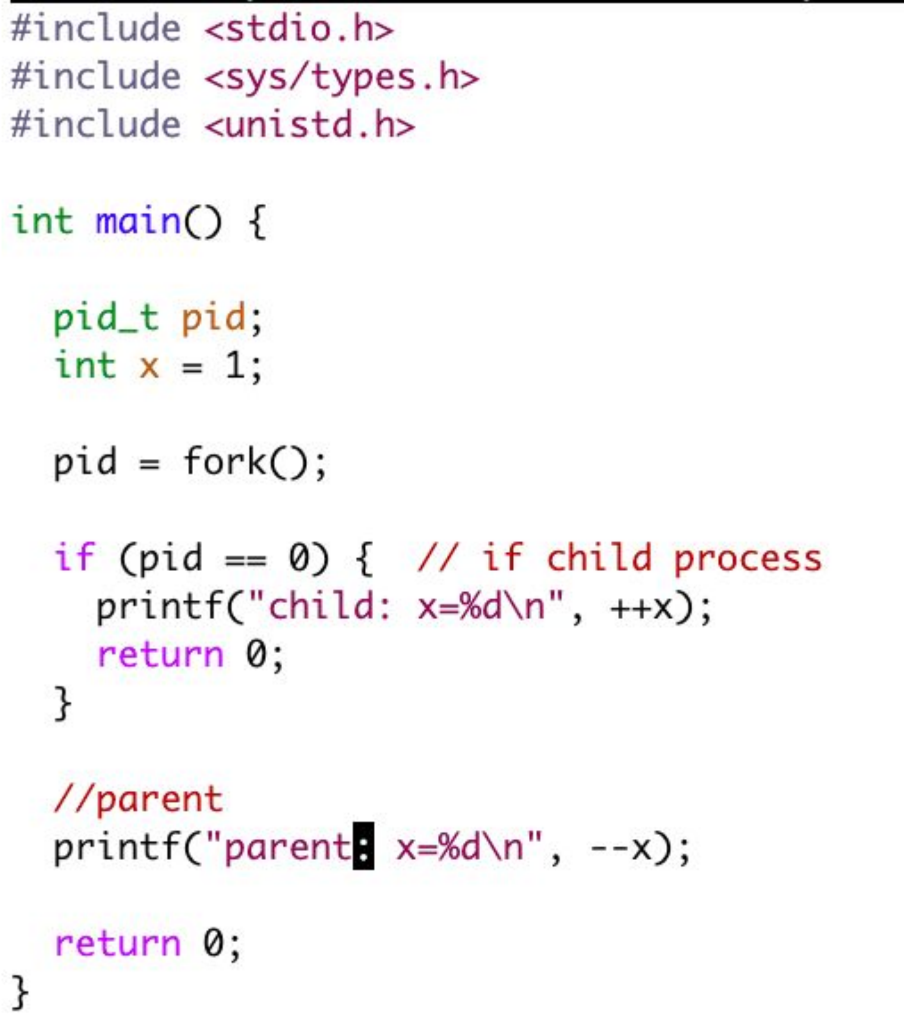
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {

  pid_t pid;
  int x = 1;

  pid = fork();

  if (pid == 0) {  // if child process
    printf("child: x=%d\n", ++x);
    return 0;
  }

  //parent
  printf("parent: x=%d\n", --x);

  return 0;
}
```

Fork Example

- Code walkthrough
  - Store a pid
  - fork our parent process and create a child
  - printf from our parent and/or printf from our child
- What will the following print out?



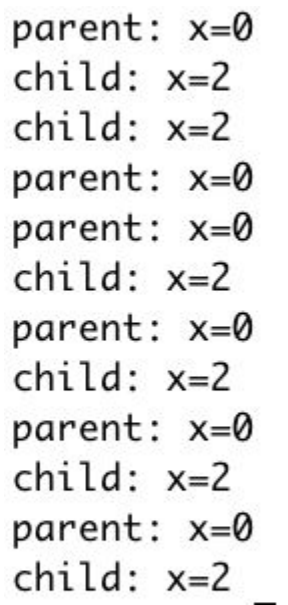
```
parent: x=0
child: x=2
child: x=2
parent: x=0
parent: x=0
child: x=2
parent: x=0
child: x=2
parent: x=0
```



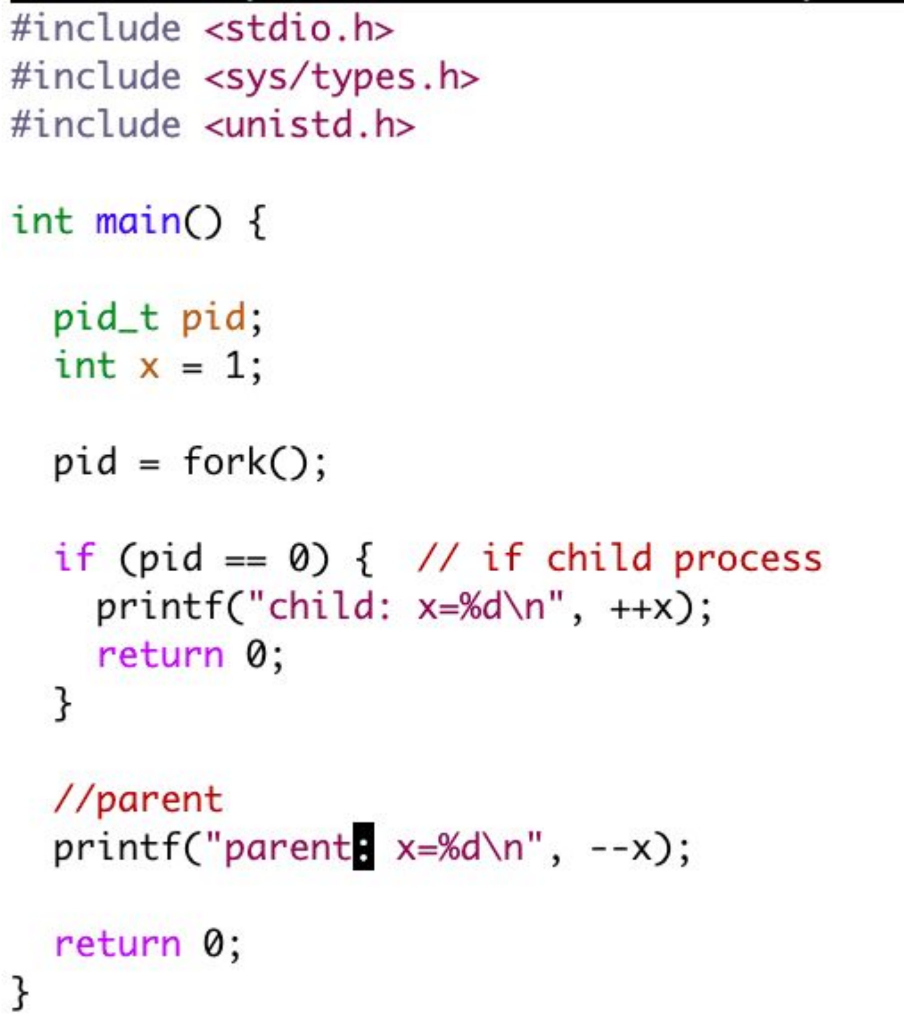
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {

  pid_t pid;
  int x = 1;

  pid = fork();

  if (pid == 0) {  // if child process
    printf("child: x=%d\n", ++x);
    return 0;
  }

  //parent
  printf("parent: x=%d\n", --x);

  return 0;
}
```

Fork Example

- After the fork, remember that the x's are completely different between the parent and child

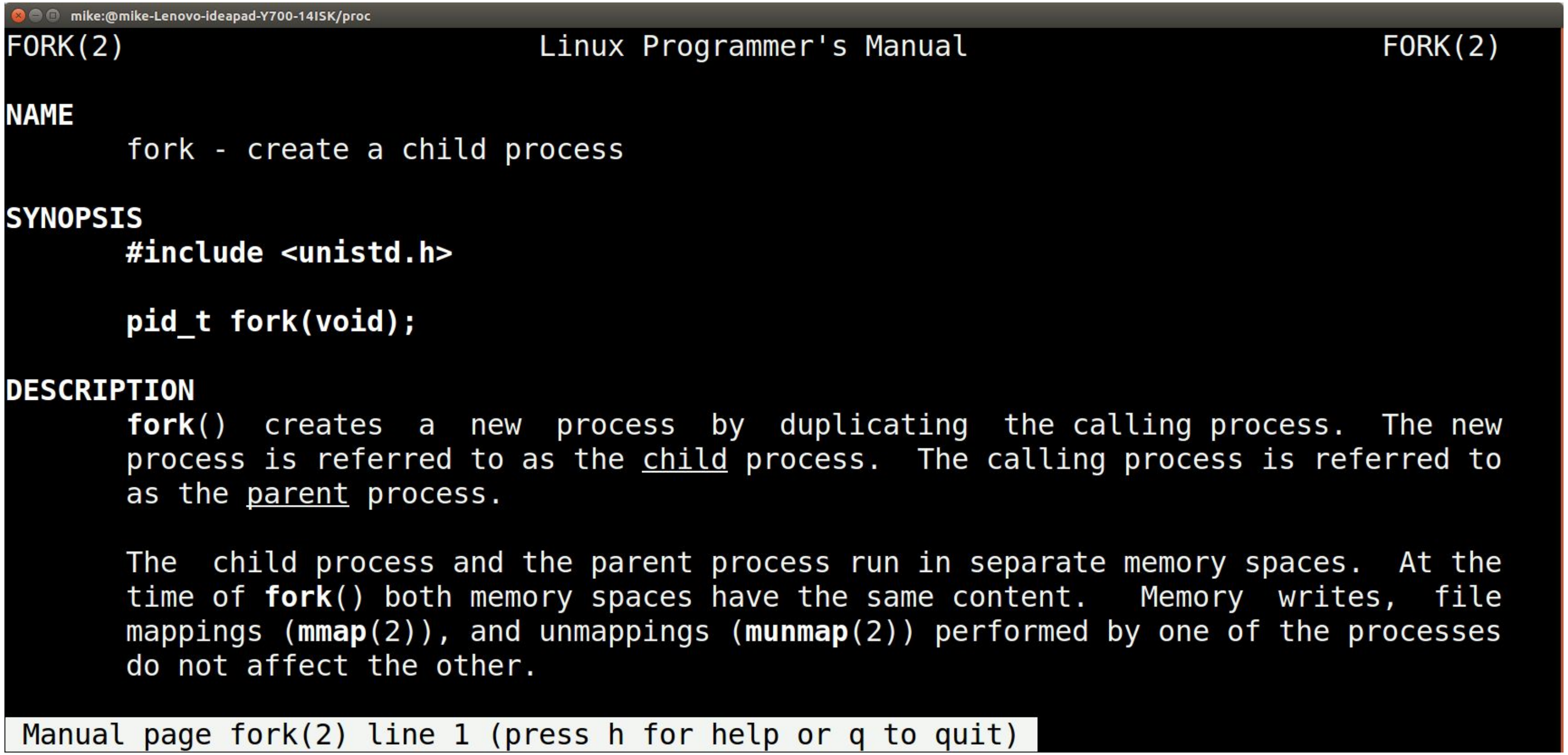


```
parent: x=0
child: x=2
child: x=2
parent: x=0
parent: x=0
child: x=2
parent: x=0
child: x=2
parent: x=0
child: x=2
parent: x=0
child: x=2
```



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {

  pid_t pid;
  int x = 1;

  pid = fork();

  if (pid == 0) {  // if child process
    printf("child: x=%d\n", ++x);
    return 0;
  }

  //parent
  printf("parent: x=%d\n", --x);

  return 0;
}
```

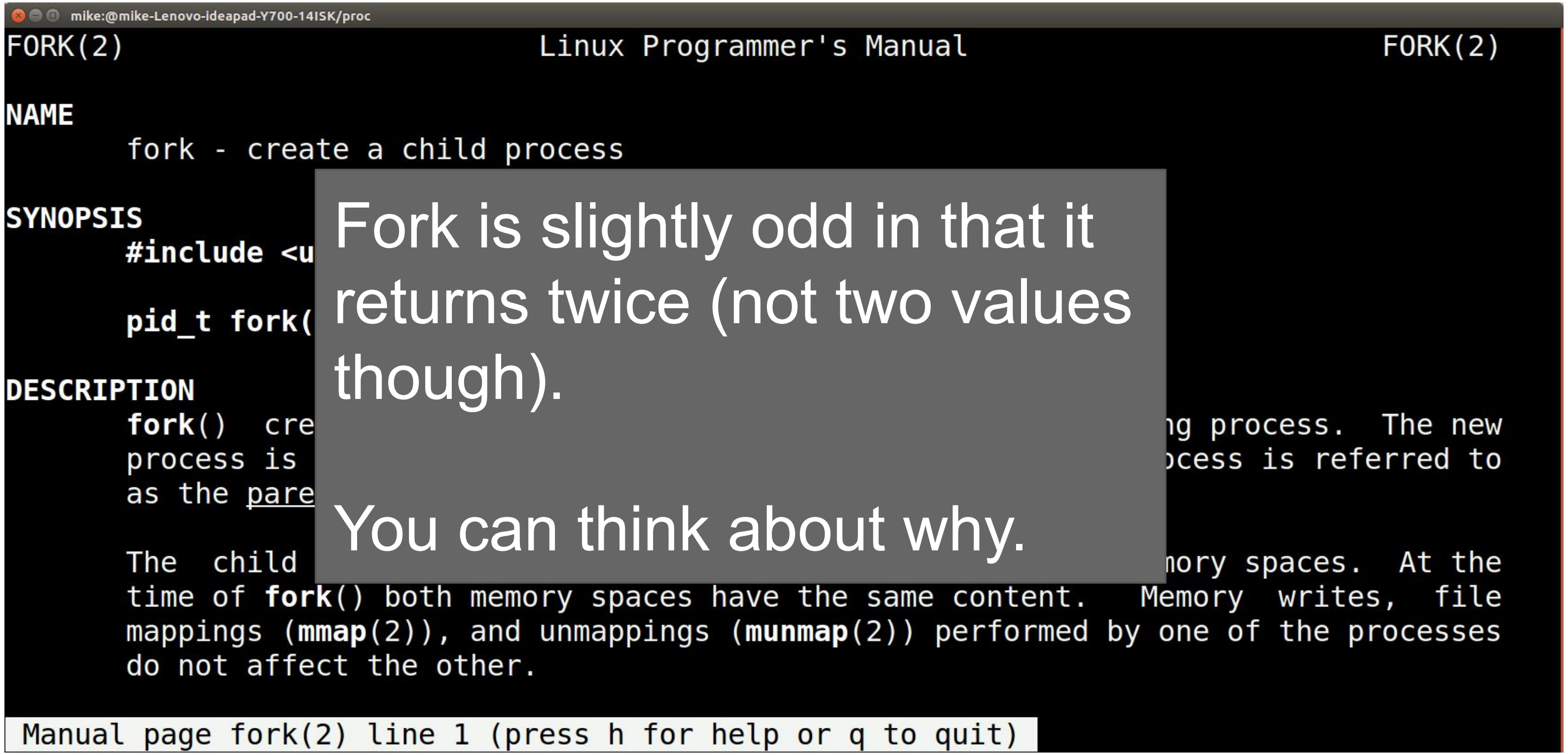
man fork



mike:@mike-Lenovo-ideapad-Y700-14ISK/proc

```
FORK(2)                     Linux Programmer's Manual                    FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a  new  process  by  duplicating  the calling process.  The new
       process is referred to as the child process.  The calling process is referred to
       as the parent process.

       The  child process and the parent process run in separate memory spaces.  At the
       time of fork() both memory spaces have the same content.   Memory  writes,  file
       mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes
       do not affect the other.

 Manual page fork(2) line 1 (press h for help or q to quit)
```

man fork



```
FORK(2)                     Linux Programmer's Manual                    FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <u

       pid_t fork(

DESCRIPTION
       fork()  cre                                       ng process.  The new
       process is                                        ocess is referred to
       as the pare

       The  child                                        mory spaces.  At the
       time of fork() both memory spaces have the same content.   Memory  writes,  file
       mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes
       do not affect the other.
 Manual page fork(2) line 1 (press h for help or q to quit)
```

Fork is slightly odd in that it returns twice (not two values though).

You can think about why.

End of Lecture