CS 3650 Computer Systems – Summer 2025

Processes

Unit 5

Northeastern University

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Ji-Yong Shin, and Alden Jackson’s lecture slides for the same course.

Processes

Diving into the Operating Systems

- So far, we have been preparing for our further exploration:
  - Assembly
  - C

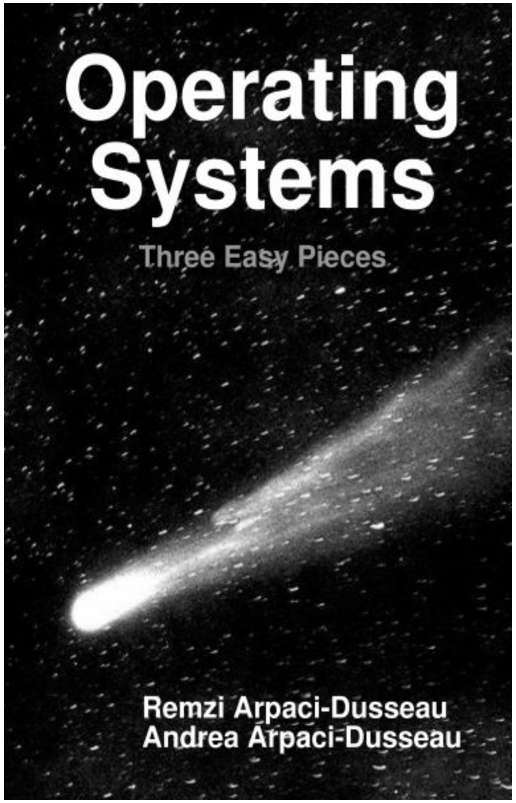
- Today we will dive into the OS itself. What we learned will be useful
  - Registers and instruction concepts
  - Memory as a linear array and ways to work with memory addresses
  - C is at the core of many common OSes

OS: Virtualization + Abstraction

- The OS is a land of magic and <u>illusions</u>
- OS makes a computer “easy” to use
- OS hides overwhelming complexities of hardware behind an API
  - This is abstraction
- OS creates the illusion of an ideal, general, and powerful machine
  - This is virtualization
- We will start by looking at how the processor virtualizes the CPU
- And then process and other abstractions the OS uses

Recommended Reading

- The OSTEP book: up to Ch. 3-6
- Online: https://pages.cs.wisc.edu/~remzi/OSTEP/
- Hard copy: Lulu or Amazon



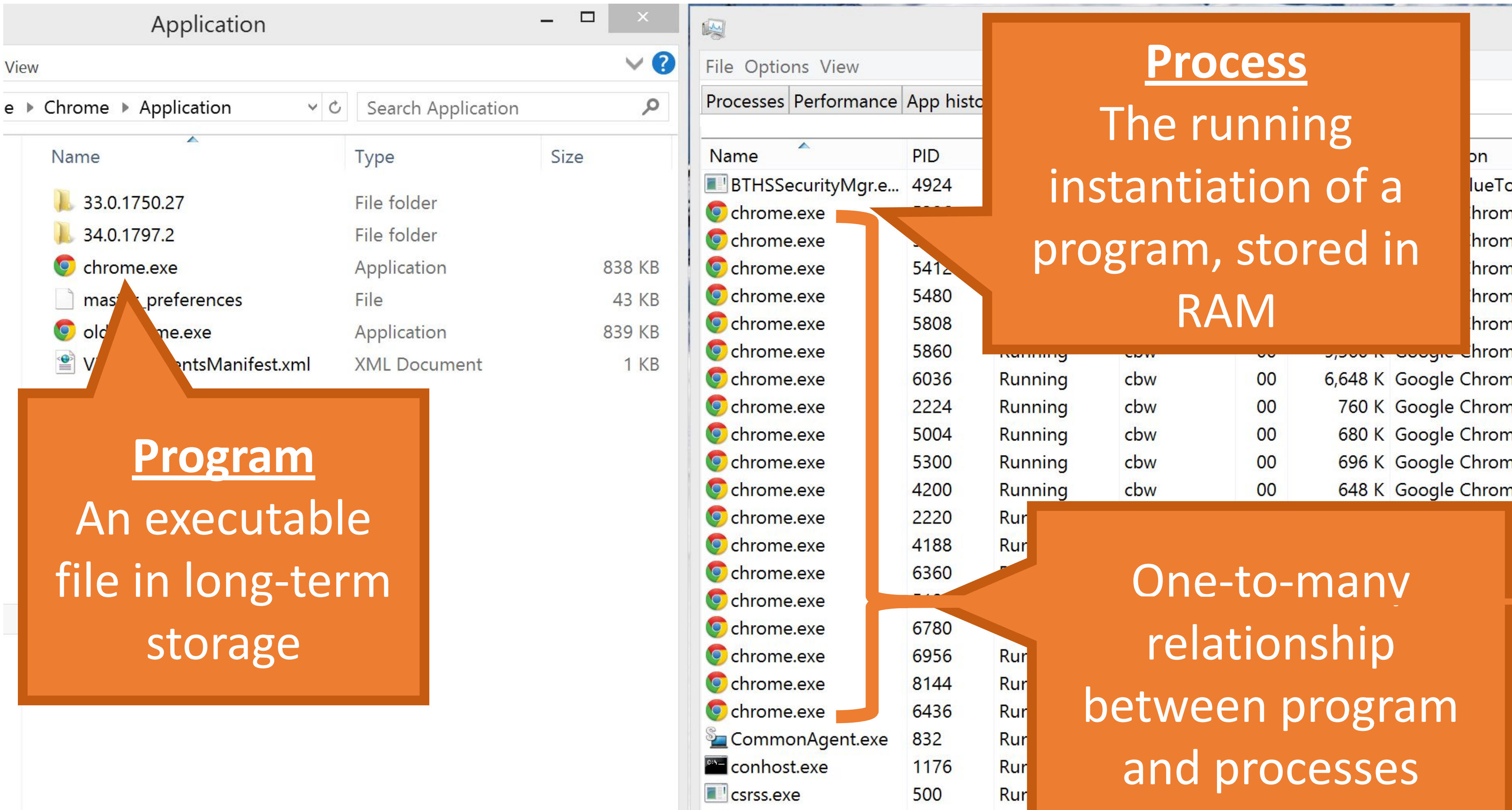
Running Dynamic Code

- Basic function of an OS is to execute and manage code dynamically:

  for example,
  - A command issued at a command line terminal
  - An icon double clicked from the desktop
  - Jobs/tasks run as part of a batch system

- A process is the basic unit of a program in execution

Programs and Processes



Process
The running instantiation of a program, stored in RAM

Program
An executable file in long-term storage

One-to-many relationship between program and processes

How to Run a Program?

- How does the OS turn the a double-clicked .exe file into a process?

- What information must the .exe file contain to run as a program?

Program Formats

- Programs obey specific file formats
  - **CP/M** (control program monitor) and **DOS** (disk operating system): COM executables (*.com)
  - **DOS**: MZ executables (*.exe)
    - Named after Mark Zbikowski, a DOS developer
  - **Windows**: Windows Portable Executable (PE, PE32+) (*.exe)
    - Modified version of Unix COFF executable format
    - PE files start with an MZ header.
  - **Unix/Linux**: Executable and Linkable Format (ELF)
  - **Mac OSX**: Mach object file format (Mach-O)

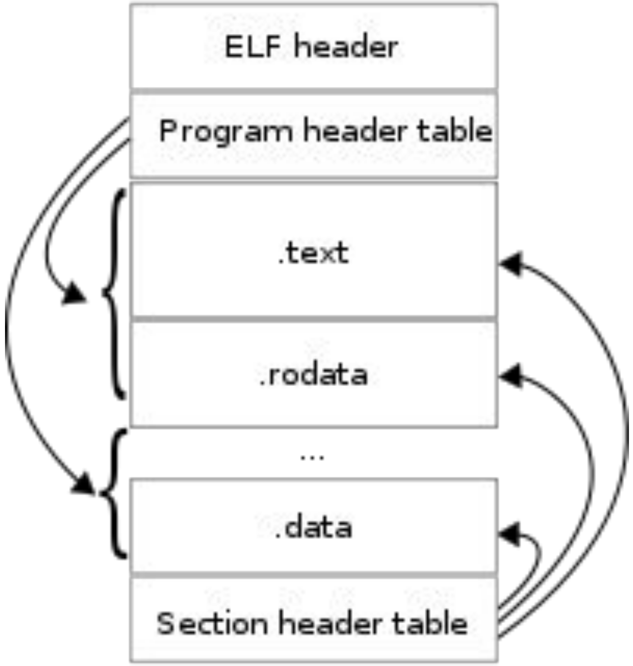
ELF File Format

- Spec: https://refspecs.linuxfoundation.org/elf/elf.pdf

- ELF Header
  - Contains compatibility info
  - Entry point of the executable code

- Program header table
  - Lists all the segments in the file
  - Used to load and execute the program
  - How to layout memory

- Section header table
  - Used by the linker

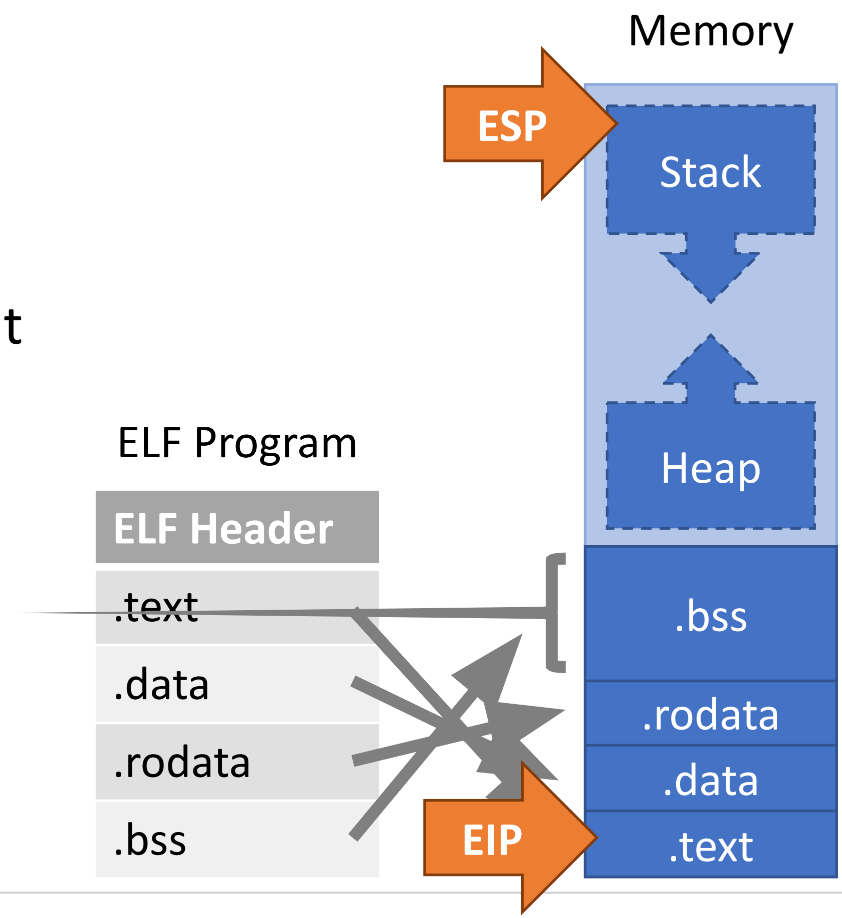


ELF Header Example

```
$ gcc –g –o test test.c
$ readelf --header test
ELF Header:
 Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400460
  Start of program headers:          64 (bytes into file)
  Start of section headers:          5216 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         36
  Section header string table index: 33
```

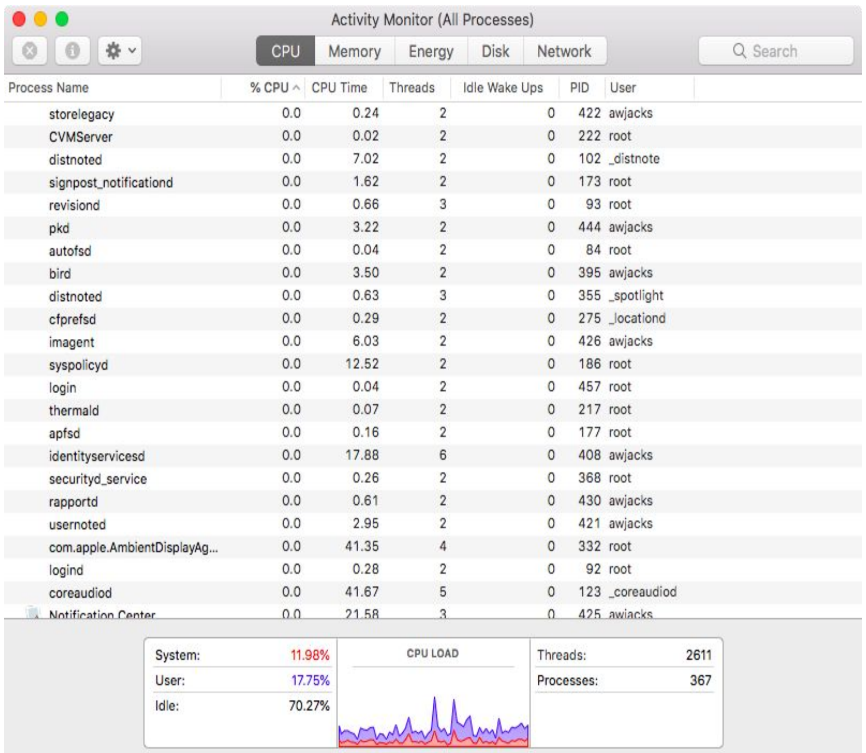
The Program Loader

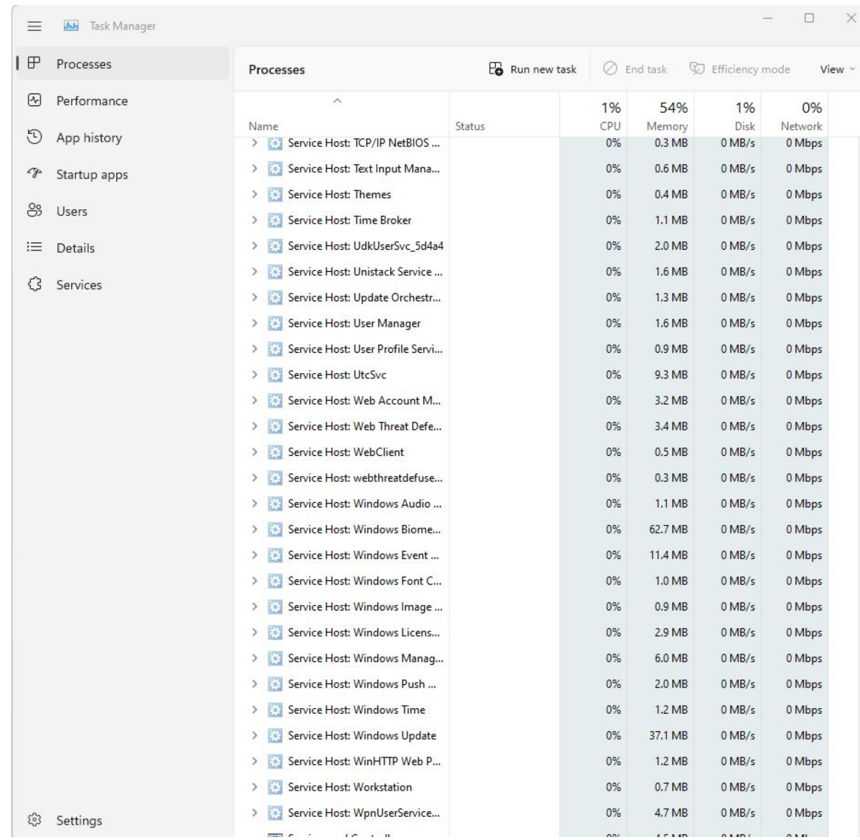
- OS functionality that loads programs into memory, creates processes
  - Places segments into memory
  - Loads necessary dynamic libraries
  - Performs relocation
  - Allocated the initial stack frame
  - Sets EIP to the programs entry point
- Process is a live program execution context or basic unit of execution



Warmup

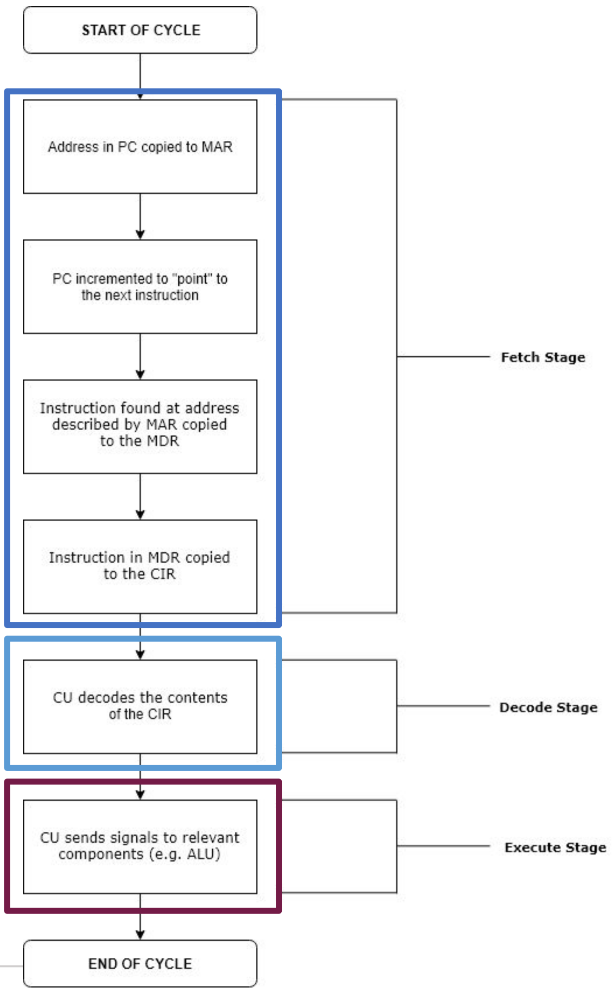
- How many processes do you have open at any given time?
  - 10s, 100s? More!? :)





First: Instruction Execution

- Code in an executable is a sequence of instructions
- CPU runs an instruction at a time
- This is done in a fetch-decode-execute cycle
- If you have **4 cores**, your processor can do **4 FDE cycles** at a time
- But how do we see ~100s of programs running on 4 cores?
- What about a single core CPU?

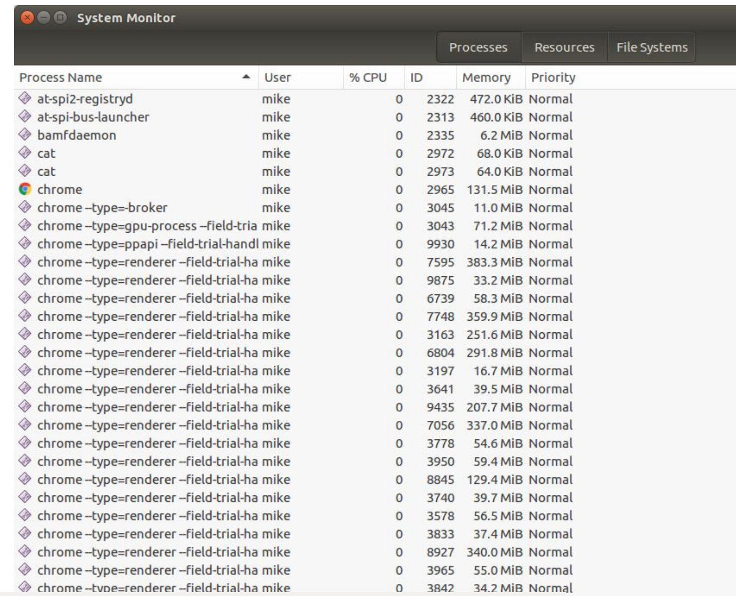


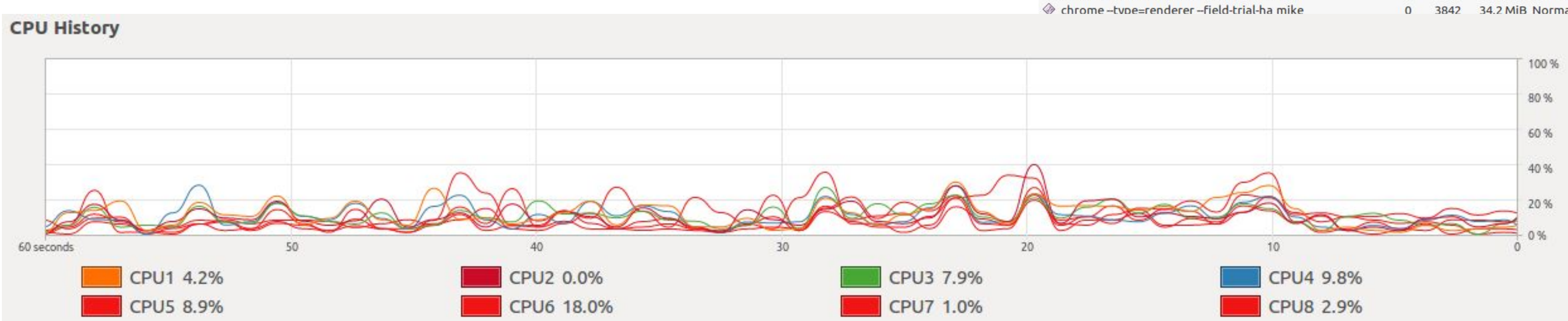
MAR: holds address of current instruction, MDR: holds contents of address in MAR
CIR: stores current instruction, so not overwritten by additional fetches to MAR/MDR

From the warm up

- Many programs are running, but only 8 CPUs that do the work

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





Virtualization with time sharing

- 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
    - Repeat

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

- This concept is known as time sharing
- Are the two states, **Running** and **Ready**, enough?

Process States

- What if the process needs to read/write to disk or perform a network request? Any problems?
  - These operations take (comparatively) long to complete
  - Keeping process state to **Running?**
    - Hogs the CPU just waiting for disk/network access to complete
  - Keeping process state to **Ready?**
    - Might not be ready to run when its turn comes
    - Asking it to run may be waste of time
- Solution?
  - Introduce a 3rd state, **Blocked**
    - Meaning: the process requested some I/O operation and cannot run until that operation is completed

Process States

- Each process can be in one of several states
- The 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 issued some blocking operation
        - I/O is a common blocking operation

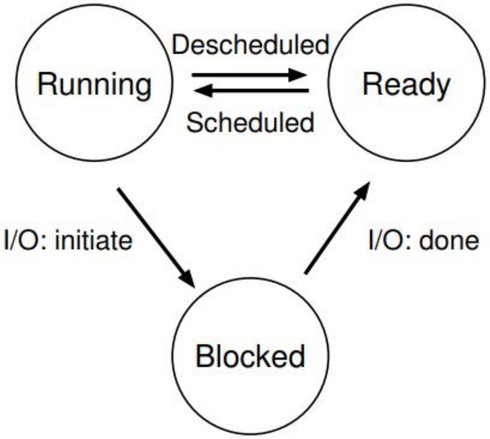


Figure 4.2: **Process: State Transitions**

Then how does OS switch processes?

OS Challenges to Virtualization

- Performance
  - How to implement virtualization without excessive overhead

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

Switching between processes

- 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?
  - Think about how you would design the OS!

When Do You Switch Processes?

- To share CPU between multiple processes, control must eventually return to the OS
  - When should this happen?
  - What mechanisms implements the switch from user process back to the OS?

- Four approaches:
  1. Voluntary yielding
  2. Switch during API calls to the OS
  3. Switch on I/O
  4. Switch based on a timer interrupt

Voluntary Yielding

- Idea: processes must voluntary give up control by calling an OS API, e.g. thread_yield()

- Problems?
  - Misbehaving or buggy apps may never yield
    e.g., while (1) { //do something without yielding }

  - No guarantee that apps will yield in a reasonable amount of time

  - Waste of CPU resources, i.e. what if a process is idle-waiting on I/O?

Interjection on OS APIs

- Idea: whenever a process calls an OS API, this gives the OS an opportunity to context switch
  - E.g. printf(), fopen(), socket(), etc…
- The original Apple Macintosh used this approach
  - Cooperative multi-tasking
- Problems?
  - Misbehaving or buggy apps may never yield
  - Some normal apps don’t use OS APIs for long periods of time
    - E.g. a long, CPU intensive matrix calculation

Switching on I/O

- Idea: when one process is waiting on I/O, switch to another process
  - I/O APIs already go through the OS, so context switching is easy

- Problems?
  - Some apps don't have any I/O for long periods of time

Preemptive Switching

- So far, processes will not switch to another until an action is taken
  - e.g. an API call or an I/O interrupt

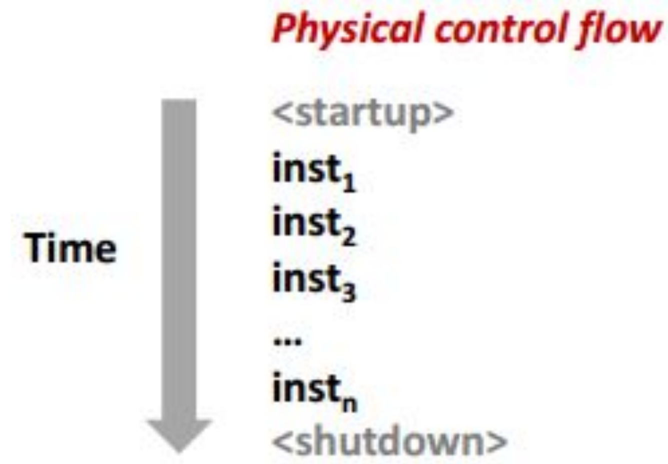
- Idea: use a **timer** to force context switching at set intervals
  - Timer is running at a fixed frequency to measure how long a process has been running
  - If it's been running for some max duration (scheduling quantum), the handler switches to the next process

- Problems? Who will trigger the timer
  - Requires hardware support (a programmable timer)
    - Thankfully, this is built-in to most modern CPUs

Mechanisms for switching: 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, sometimes we want to react based on the system state
    - E.g., you hit Ctrl+C on the keyboard in your terminal and execution stops.

Exceptional Control Flow Mechanisms

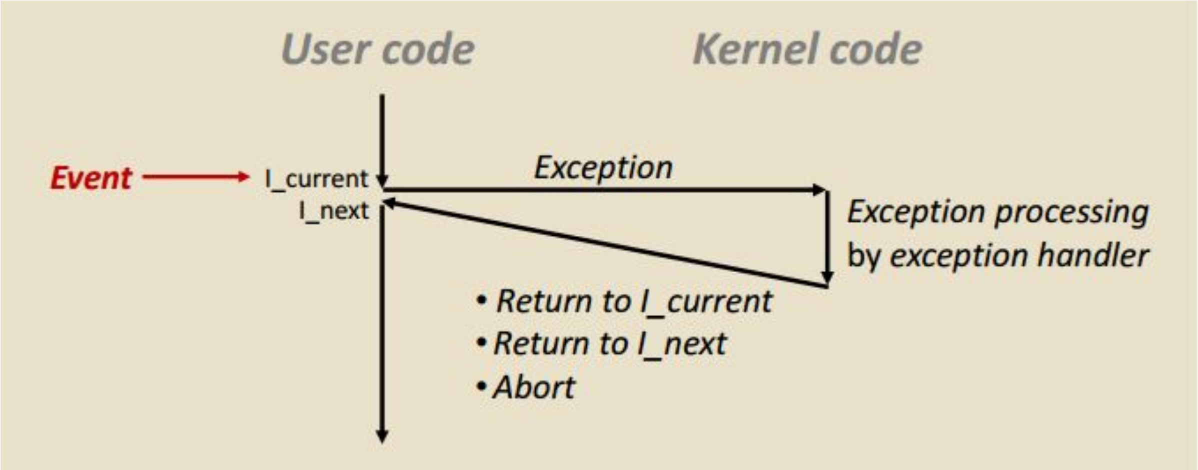
- Low level mechanism
  - Exceptions
    - Change in control flow in response to a system event.
    - This is implemented in hardware and OS software

Exceptional Control Flow Mechanisms

- High level mechanisms
  - Process context switch
    - 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 and CPU hardware

Exceptions

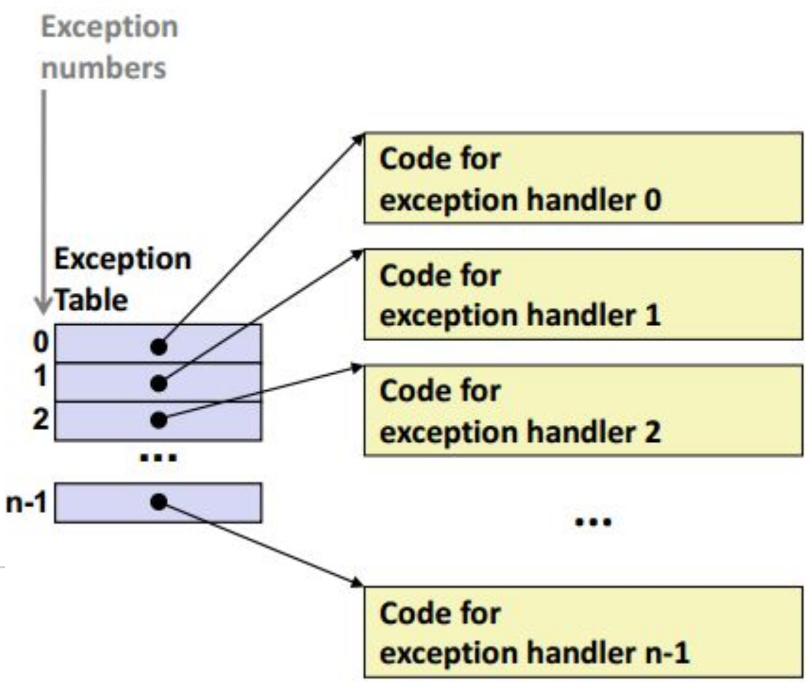
- An exception is a transfer of control to the OS kernel
    - The kernel is the memory-resident part of the OS
        - Meaning OS lives in memory forever: we do not modify this!
- Examples of exceptions we may be familiar with:
    - Divide by 0, arithmetic overflow, or typing Ctrl+C



- How does the OS know how to handle the exception?

Exception Tables

- Somewhere in the OS, a table exists with different exceptions.
  - Think of it like a giant switch or many if else-if statements.
- This is part of a kernel that you cannot modify.
  - This code is in a **"protected region"** of memory
- For each exception, there is one way to handle it
  - (We call these **"exception handlers"**)

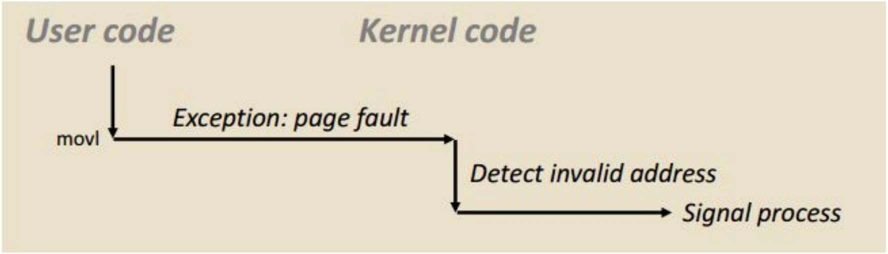


Our favorite: Invalid Memory Reference

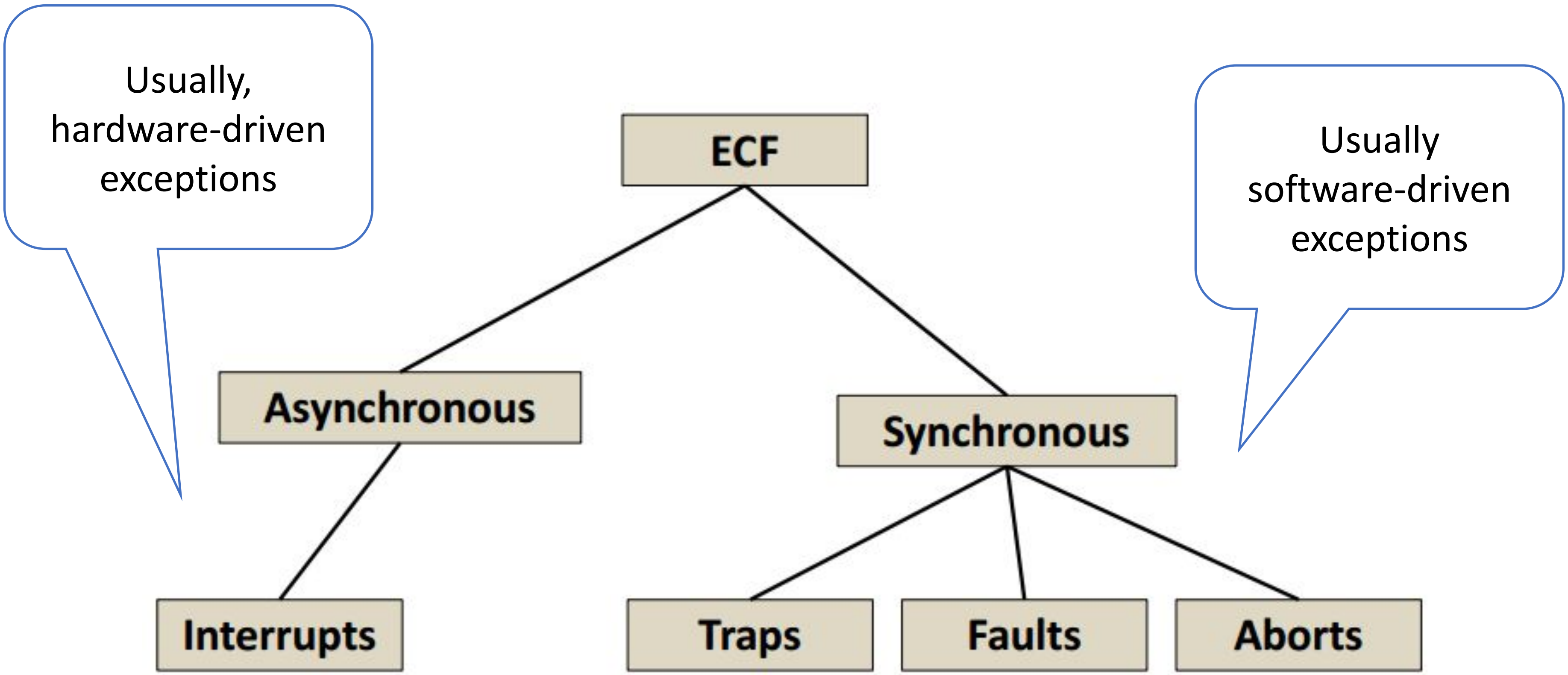
- That is, the segmentation fault
  - OS sends signal SIGSEGV to our user process
  - This time the program gets terminated.

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

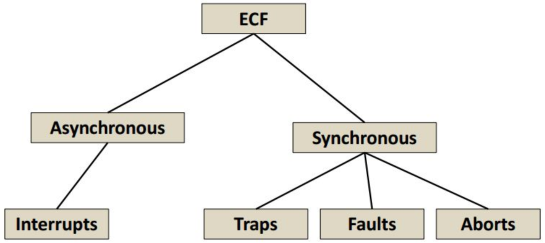
```
80483b7:       c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```



Exceptional Control Flow Taxonomy

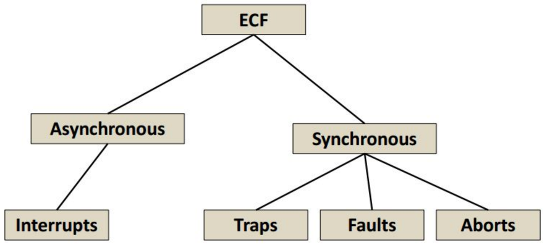


Asynchronous Exceptions (Interrupts)



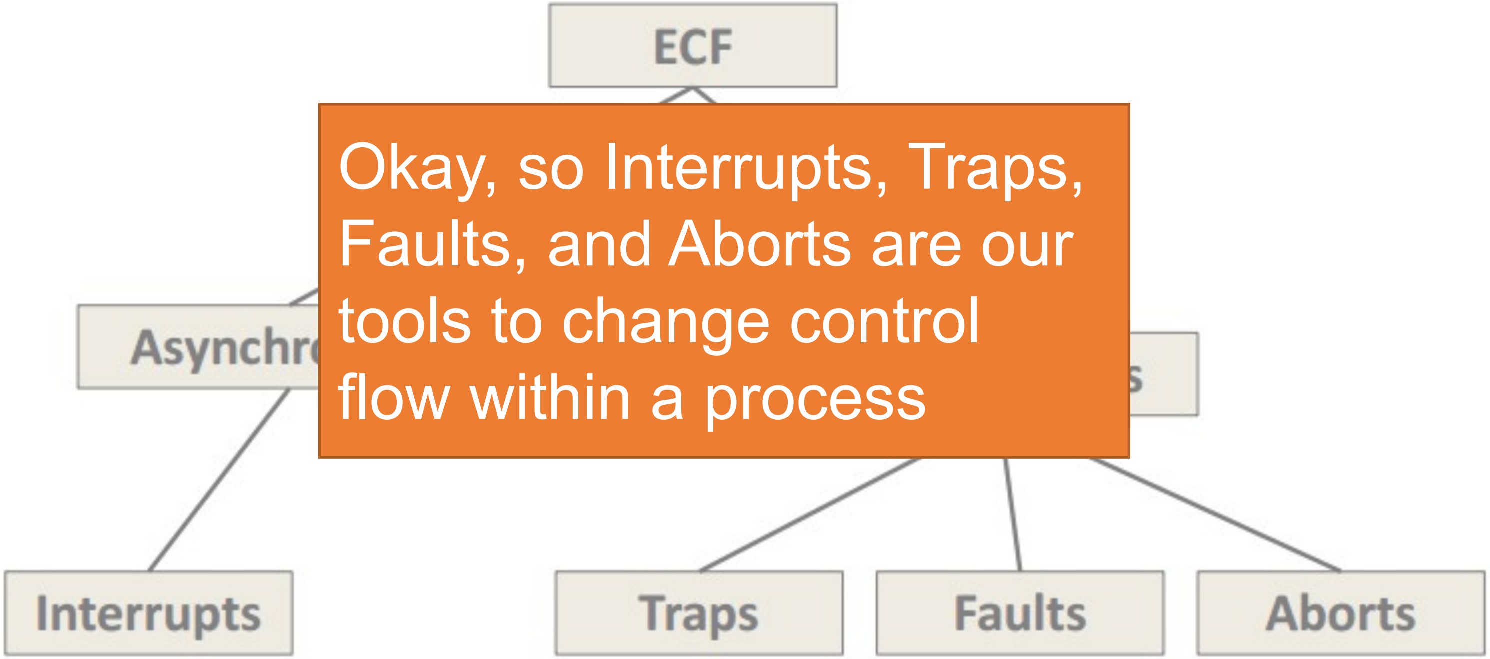
- Caused by events external to processor
  - I.e., not from the result of an instruction the user wrote
  - E.g.
    - Timer interrupts scheduled to happen every few milliseconds
      - A kernel can use this to take back control from a program/user
    - Some network data arrives (I/O)
    - A nice 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

Exceptional Control Flow Taxonomy



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

CS 3650 Computer Systems – Spring 2023

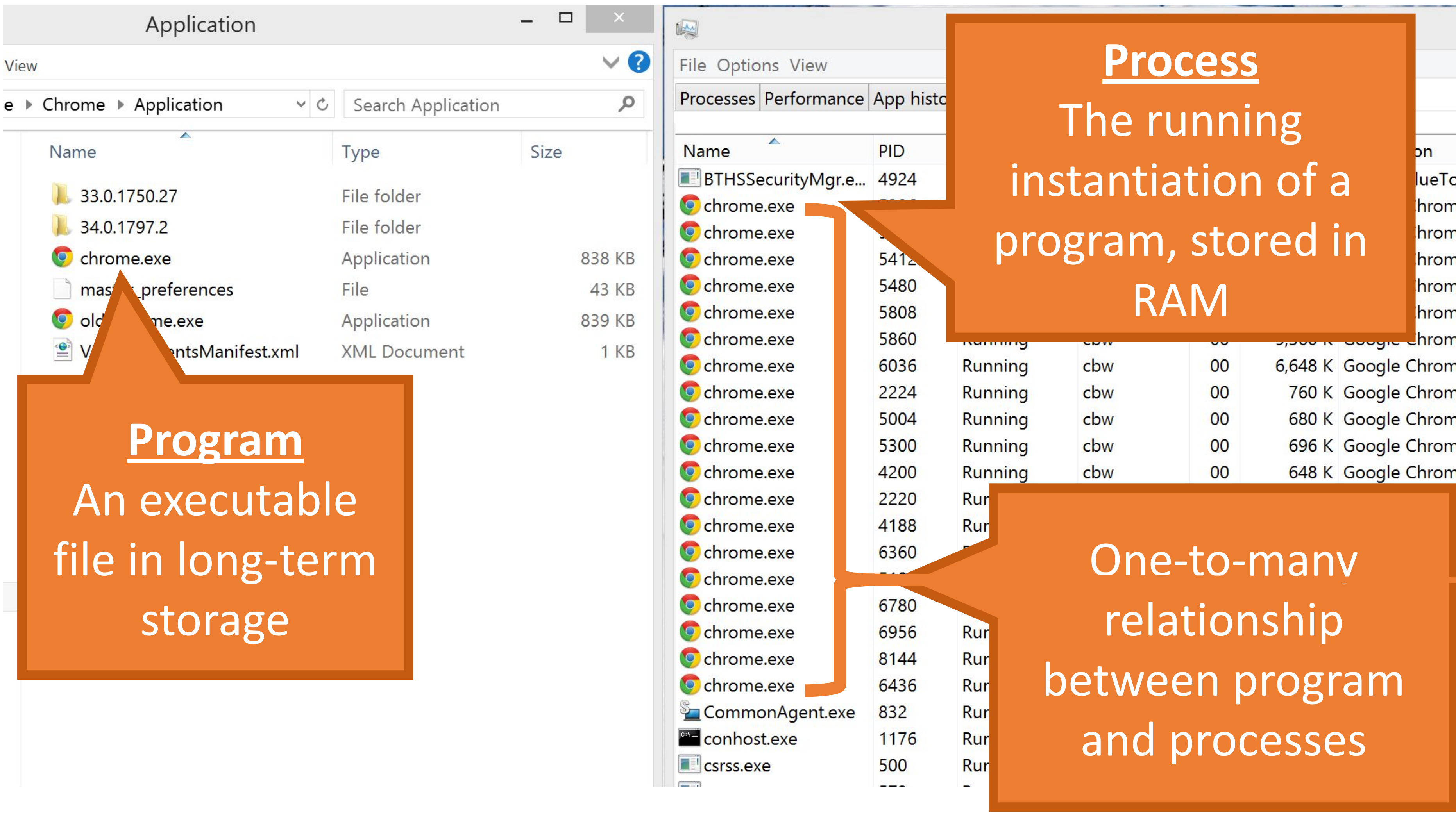
Processes

Week 5

Northeastern University

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, and Alden Jackson's lecture slides for the same course.

Programs and Processes



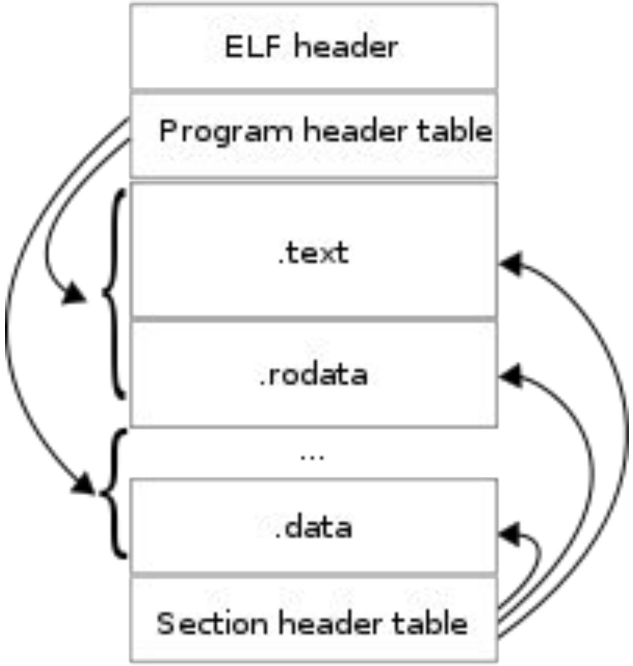
ELF File Format

- Spec: https://refspecs.linuxfoundation.org/elf/elf.pdf

- ELF Header
  - Contains compatibility info
  - Entry point of the executable code

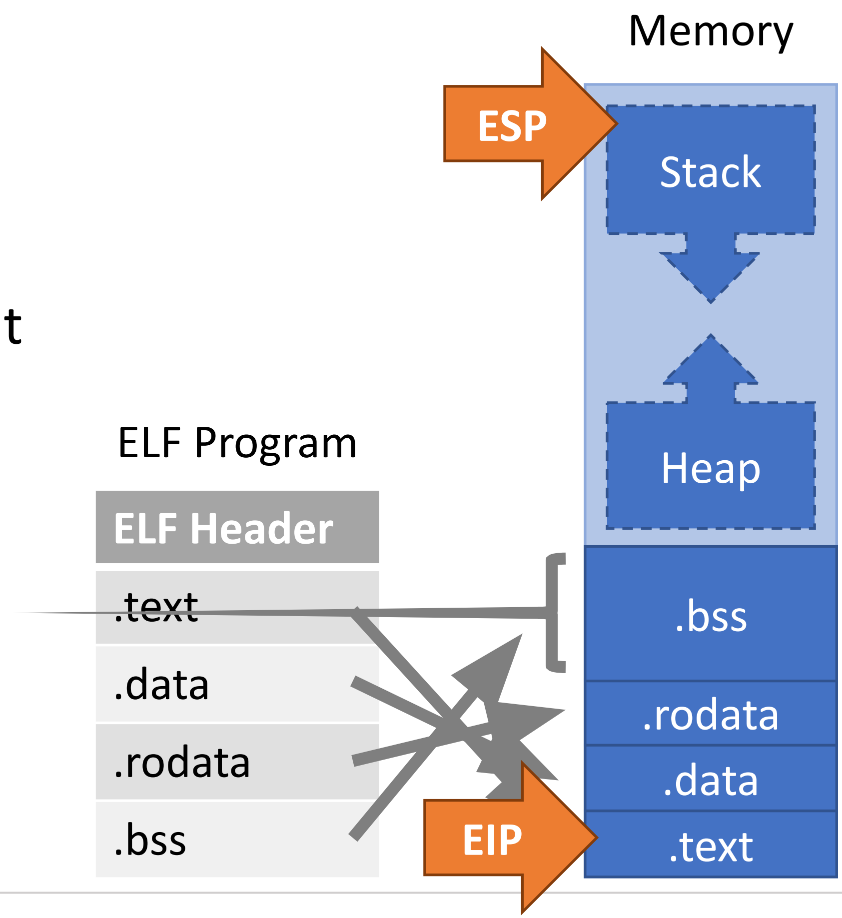
- Program header table
  - Lists all the segments in the file
  - Used to load and execute the program
  - How to layout memory

- Section header table
  - Used by the linker



The Program Loader

- OS functionality that loads programs into memory, creates processes
  - Places segments into memory
  - Loads necessary dynamic libraries
  - Performs relocation
  - Allocated the initial stack frame
  - Sets EIP to the programs entry point
- Process is a live program execution context or basic unit of execution



Process States

- Each process can be in one of several states
- The 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 issued some blocking operation
    - I/O is a common blocking operation

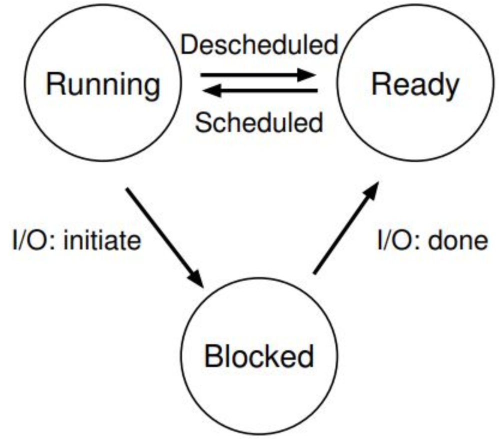


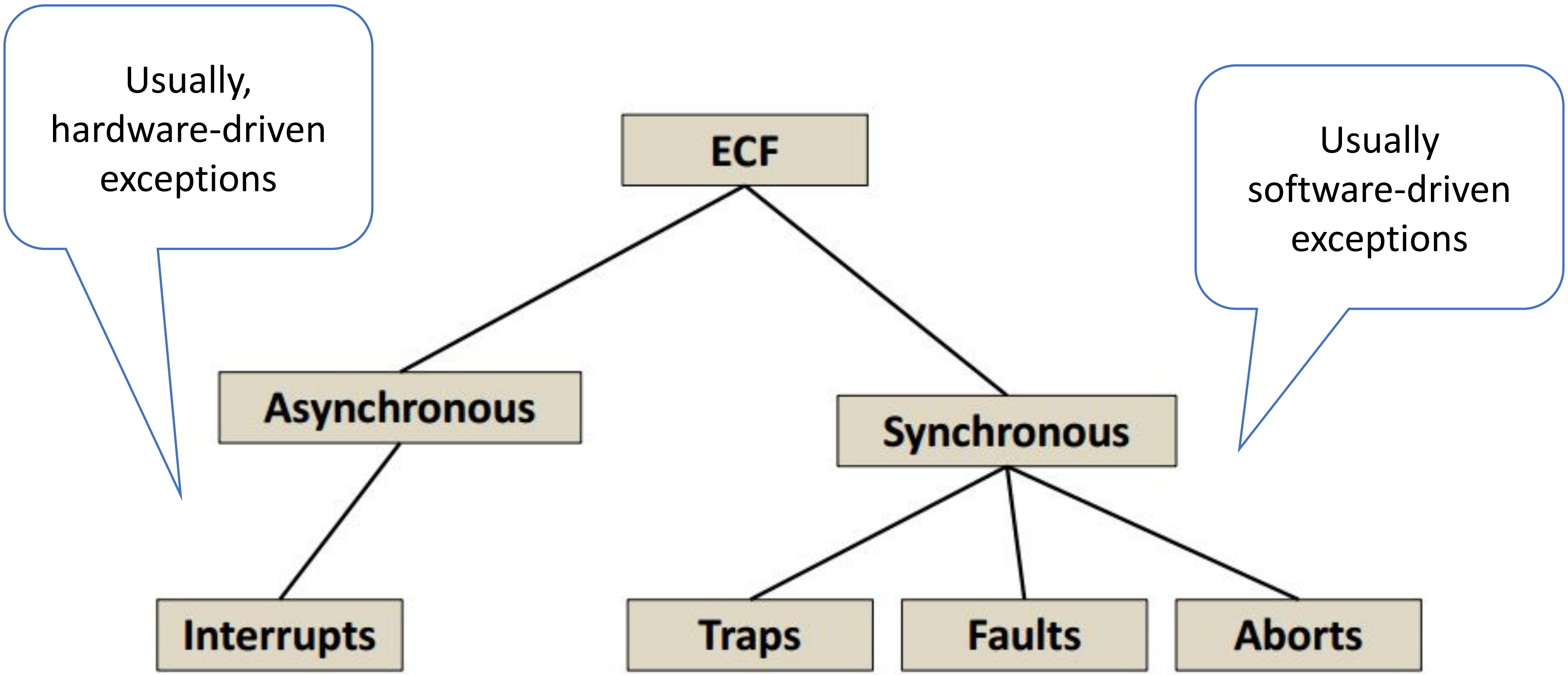
Figure 4.2: **Process: State Transitions**

When Do You Switch Processes?

- To share CPU between multiple processes, control must eventually return to the OS
  - When should this happen?
  - What mechanisms implements the switch from user process back to the OS?

- Four approaches:
  1. Voluntary yielding
  2. Switch during API calls to the OS
  3. Switch on I/O
  4. Switch based on a timer interrupt

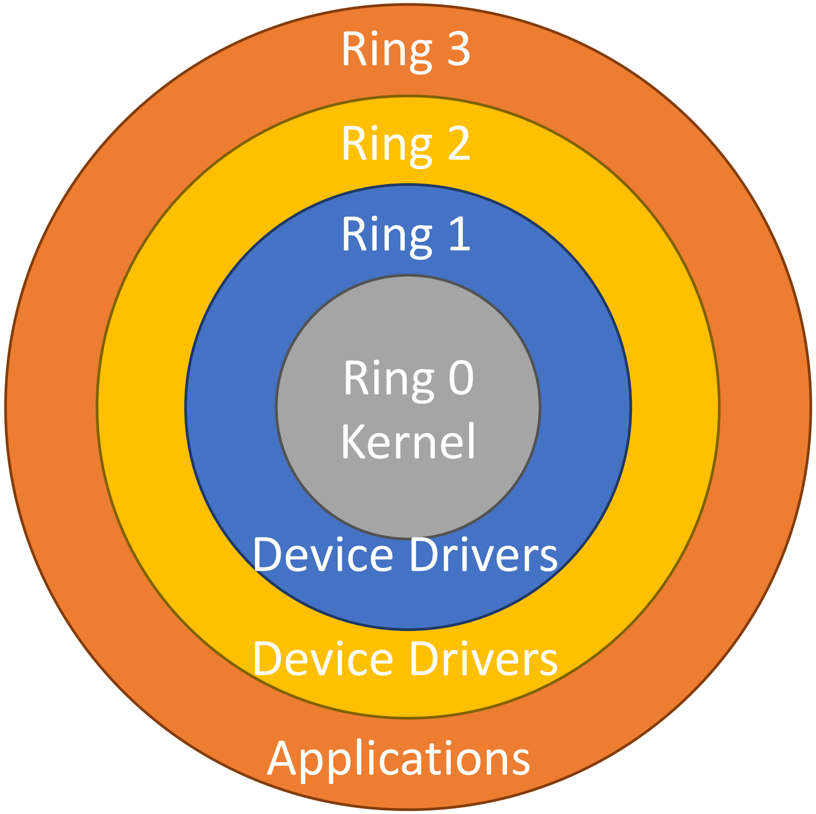
Exceptional Control Flow Taxonomy



System calls

Different privilege levels

- Most modern CPUs support protected mode
- x86 CPUs support three rings with different privileges
  - Ring 0: OS kernel
  - Ring 1, 2: device drivers
  - Ring 3: userland
- Most OSes only use rings 0 and 3



Dual-Mode Operation

- Ring 0: kernel/supervisor mode
  - Execution with the <u>full privileges</u> of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet

- Ring 3: user mode or "userland"
  - Limited privileges
  - Only those granted by the operating system kernel

Protected Features

- What system features are impacted by protection?
  - Privileged instructions
    - Only available to the kernel
  - Limits on memory accesses
    - Prevents user code from overwriting the kernel
  - Access to hardware
    - Only the kernel may directly interact with peripherals
  - Programmable Timer Interrupt
    - May only be set by the kernel
    - Used to force context switches between processes

System Calls

- Syscall is the lowest level of interaction with an operating system from a C programmer
- A user program can ask the OS for services that the OS manages

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

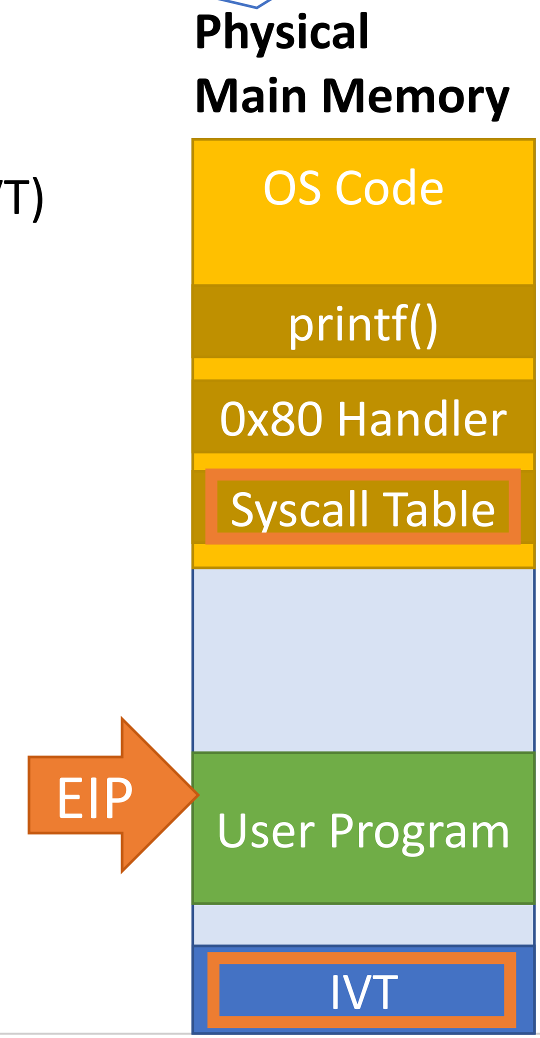
Changing Modes

- Applications often need to access the OS
  - i.e. system calls
  - Writing files, displaying on the screen, receiving data from the network, etc…
- But the OS is ring 0, and apps are ring 3
- How do apps get access to the OS?
  - Apps invoke system calls with an interrupt
    - E.g. int 0x80
  - **int** causes a mode transfer from ring 3 to ring 0

System Call Example

Note: this shows a physical memory layout. The user program thinks it owns the entire memory space (the diagram that we saw in previous lectures).

1. Software executes int 0x80
   - Pushes EIP, CS, and EFLAGS
2. CPU transfers execution to the OS handler
   - Look up the handler in the Interrupt Vector Table (IVT)
   - Switch from ring 3 to 0
3. OS executes the system call
   - Save the processes state
   - Use EAX to locate the system call
   - Execute the system call
   - Restore the processes state
   - Put the return value in EAX
4. Return to the process with iret
   - Pops EIP, CS, and EFLAGS
   - Switches from ring 0 to 3



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

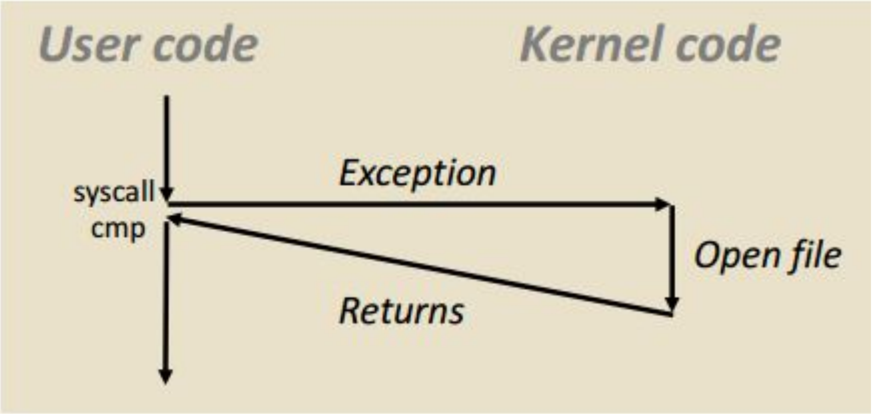
- rax holds the system call # that we want to pass.
  - Other arguments accessed as follows

%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

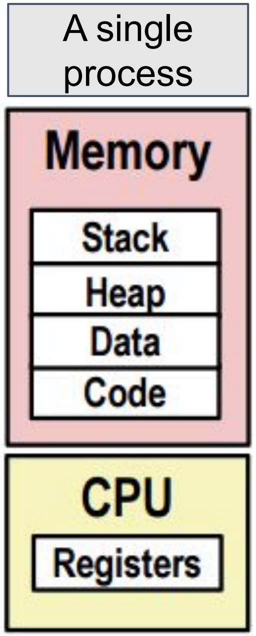
Opening a File | Illustration

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



Processes

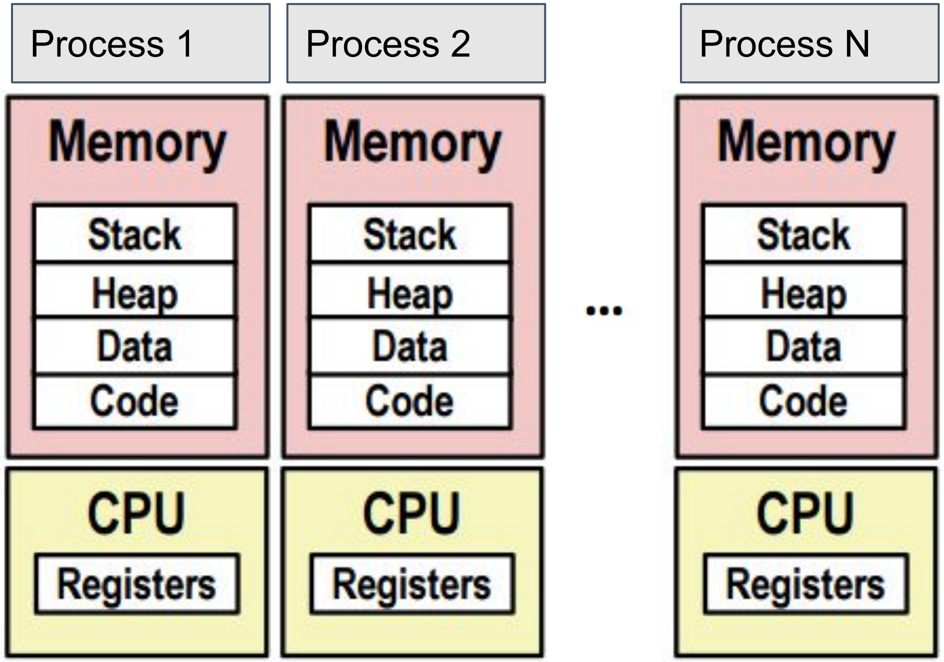
The Process



- 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 <u>control over the CPU</u>
    - Done by "context switching"
  - Private Address Space
    - Each program "appears" to have exclusive <u>use of main memory</u>
    - Provided by mechanism called virtual memory

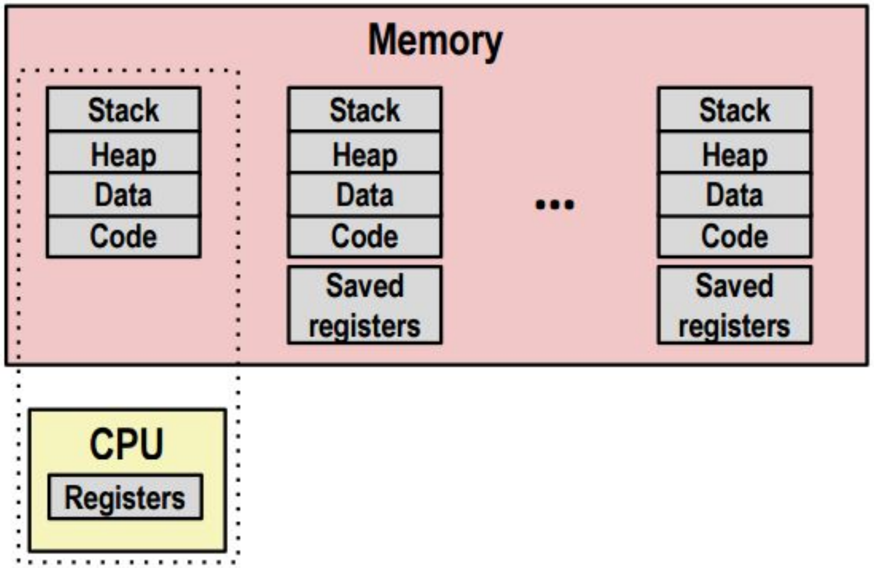
Multiprocessing: Illusion

- When running processes, it appears that we are running many different tasks at the same time
- It also appears that our memory is neatly organized.
  - Note from this diagram we see every process has its own
    - stack
    - heap
    - data
    - code
    - registers

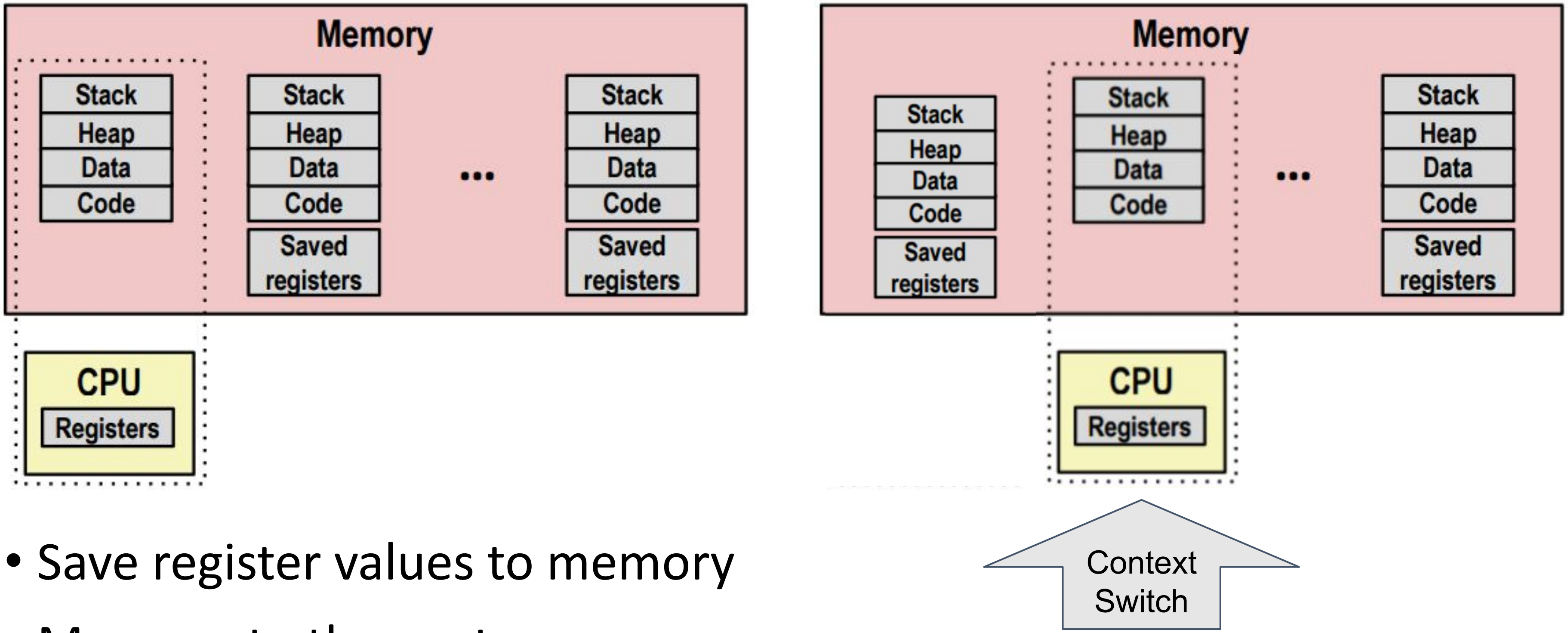


Multiprocessing: Reality

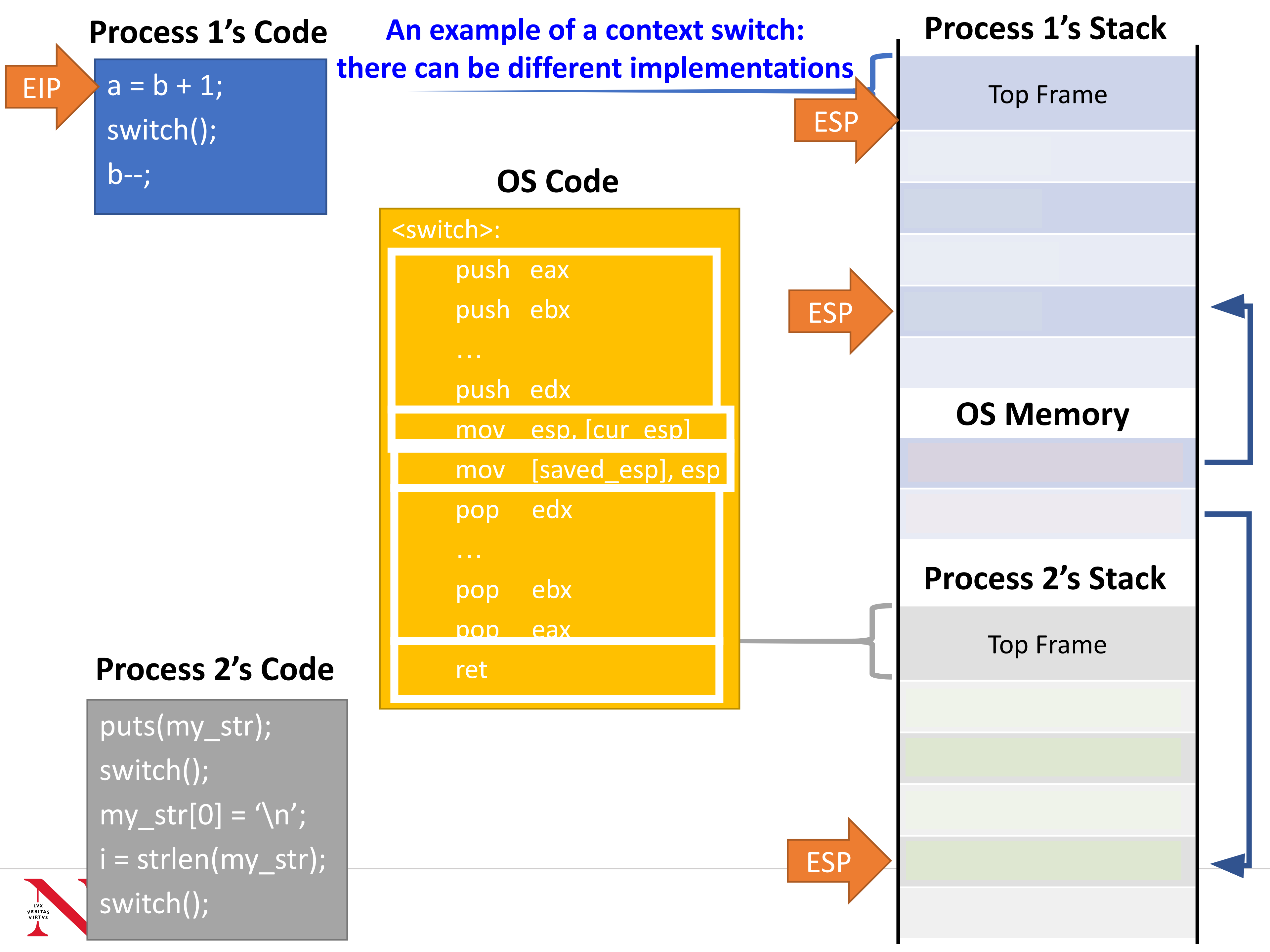
- Remember, at any time, only one processor is really running code
- Program execution is interleaved
- OS manages memory addresses in virtual memory
- OS stores the saved registers for different programs.
  - (At some point in this class, you probably figured 16 registers is not enough for all of the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



Context switch: a high-level view



- Save register values to memory
- Move on to the next process
  - Point to the stack of the next process
  - Restore saved register values
- Start running executing the next process



An example of a context switch:
there can be different implementations
Process 1's Code
EIP
a = b + 1;
switch();
b--;
OS Code
<switch>:
push eax
push ebx
…
push edx
mov esp, [cur_esp]
mov [saved_esp], esp
pop edx
…
pop ebx
pop eax
ret
Process 2's Code
puts(my_str);
switch();
my_str[0] = '\n';
i = strlen(my_str);
switch();
Process 1's Stack
Top Frame
ESP
ESP
OS Memory
Process 2's Stack
Top Frame
ESP

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

Process state

Process id

Registers that we saw earlier

- Ac … uch as th … d by the OS.
- … re which … about each process (linux uses task_struct)
- … uct … 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
};
```

man proc

2. ssh

bash ⌘1 ssh ⌘2

```
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
```

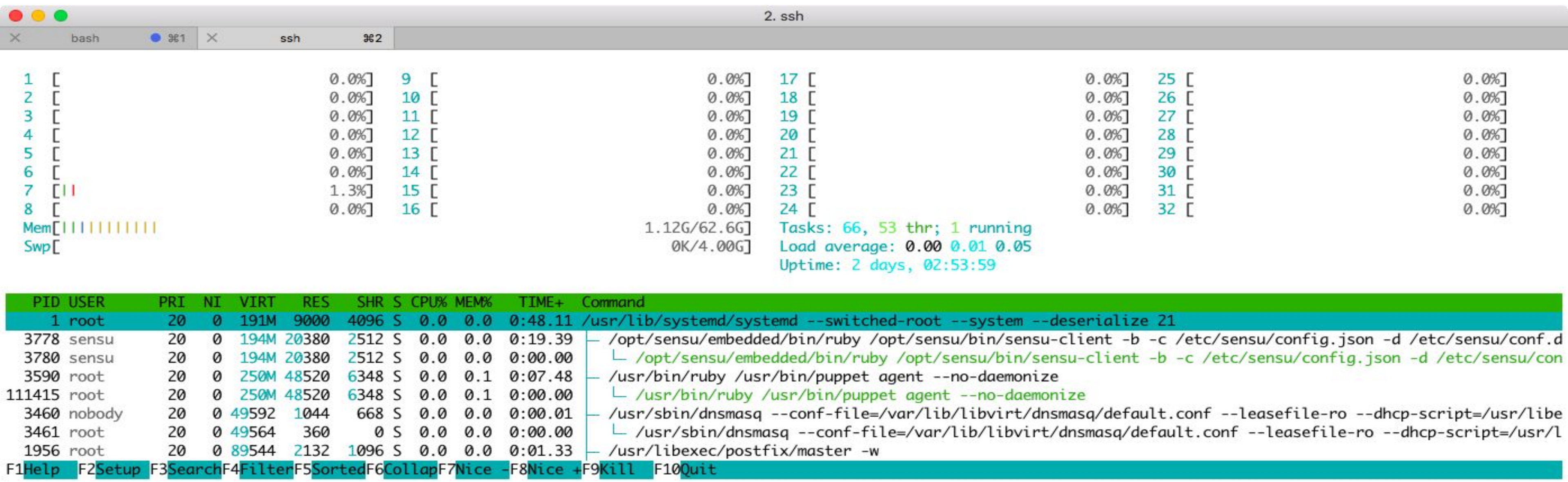
- top is a program that will show linux processes that are running
  - Top shows all of the processes running on a system
  - Intuitively, it must be possible for a machine to host multiple processes, we do so when we ssh.

```
2. ssh
bash  ⌘1    ssh  ⌘2
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



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 *ps*
    - *-ef* means view all of the processes

```
2. ssh
bash  ⌘1    ssh  ⌘2

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)
```

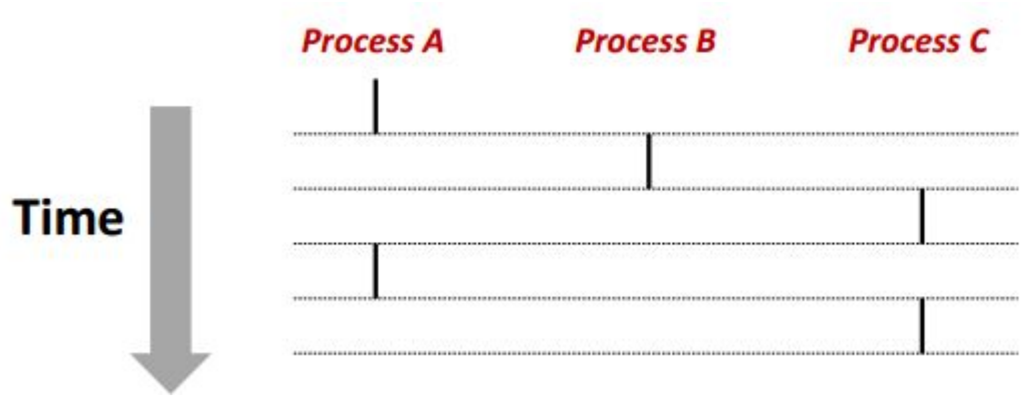
Gathering more information from proc

- 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
- Demo

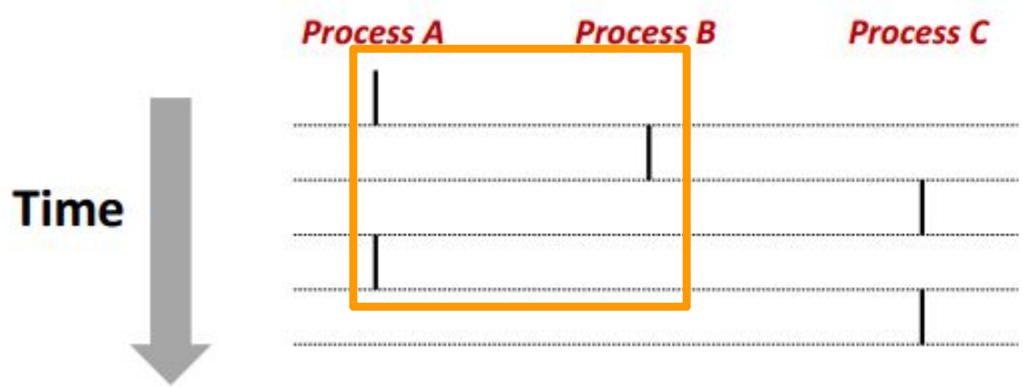
Concurrent Processing

- 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 to each other?
    - Concurrent:
  - Which are sequential?
    - Sequential:



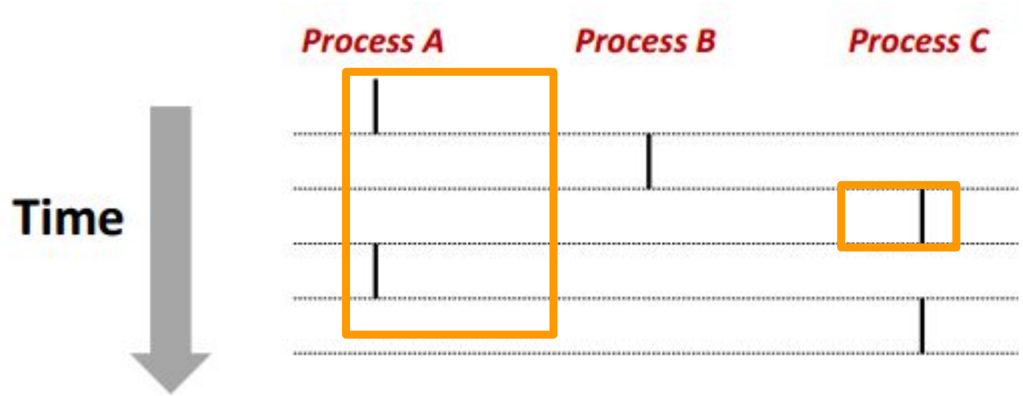
Concurrent Processing

- 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 to each other?
    - Concurrent: A&B
  - Which are sequential?
    - Sequential:



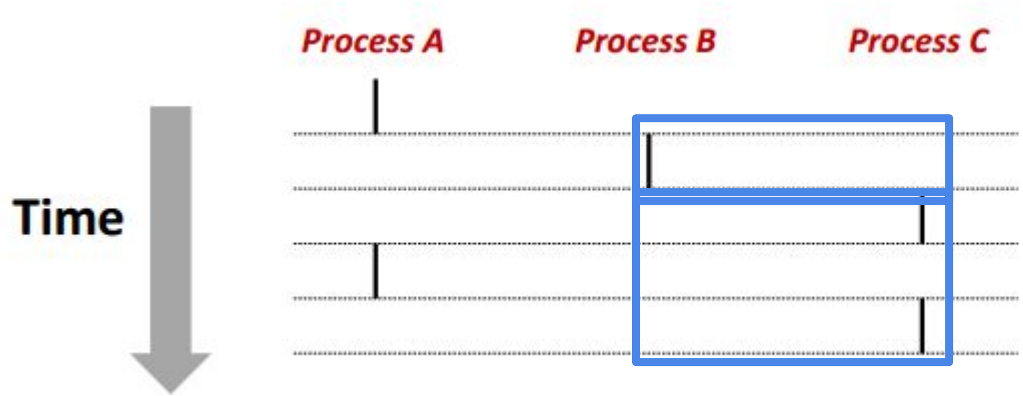
Concurrent Processing

- 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 to each other?
    - Concurrent: A&B, A&C
  - Which are sequential?
    - Sequential:



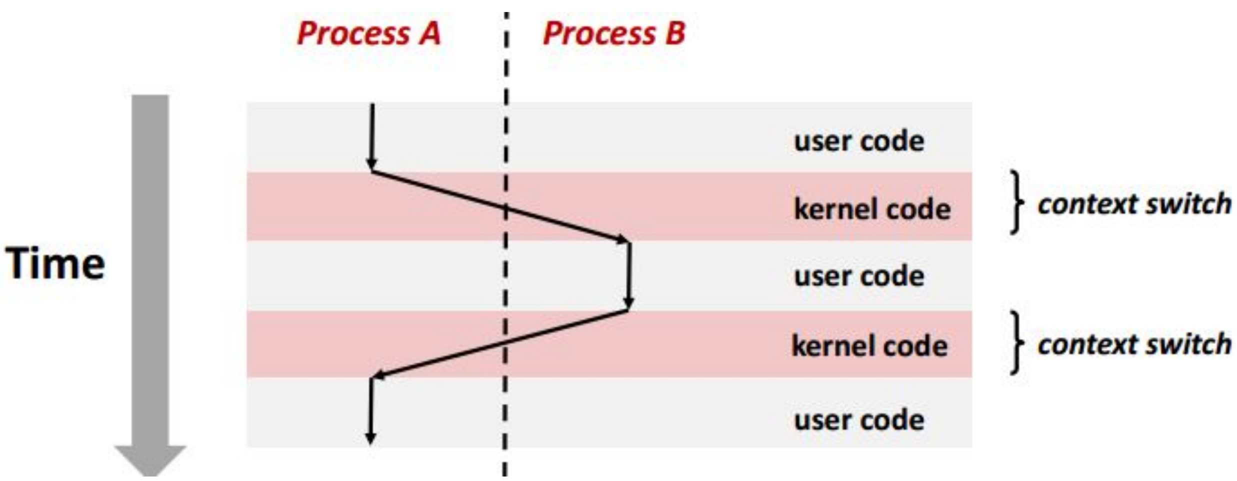
Concurrent Processing

- 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 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 **kernel**
  - 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

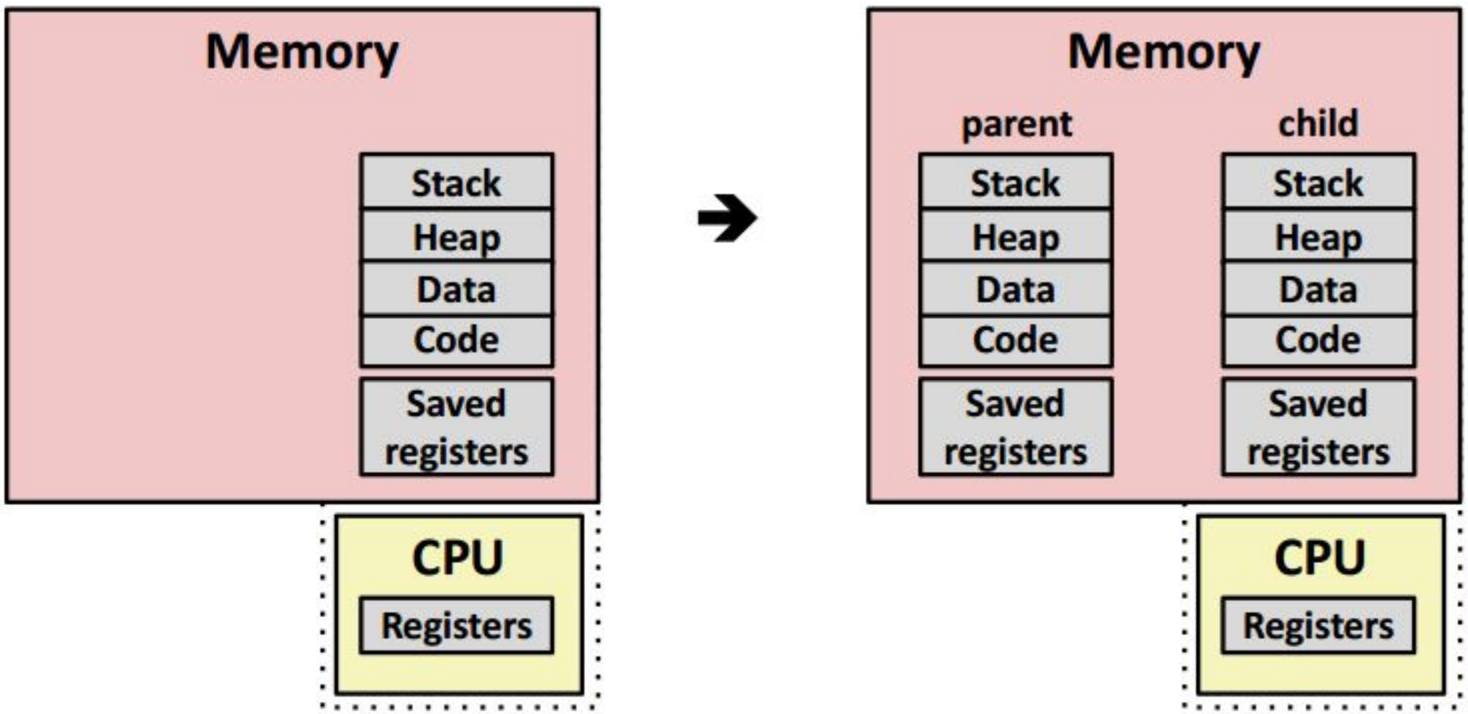
Creating a Process

- 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,**
    Returns child's PID 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.

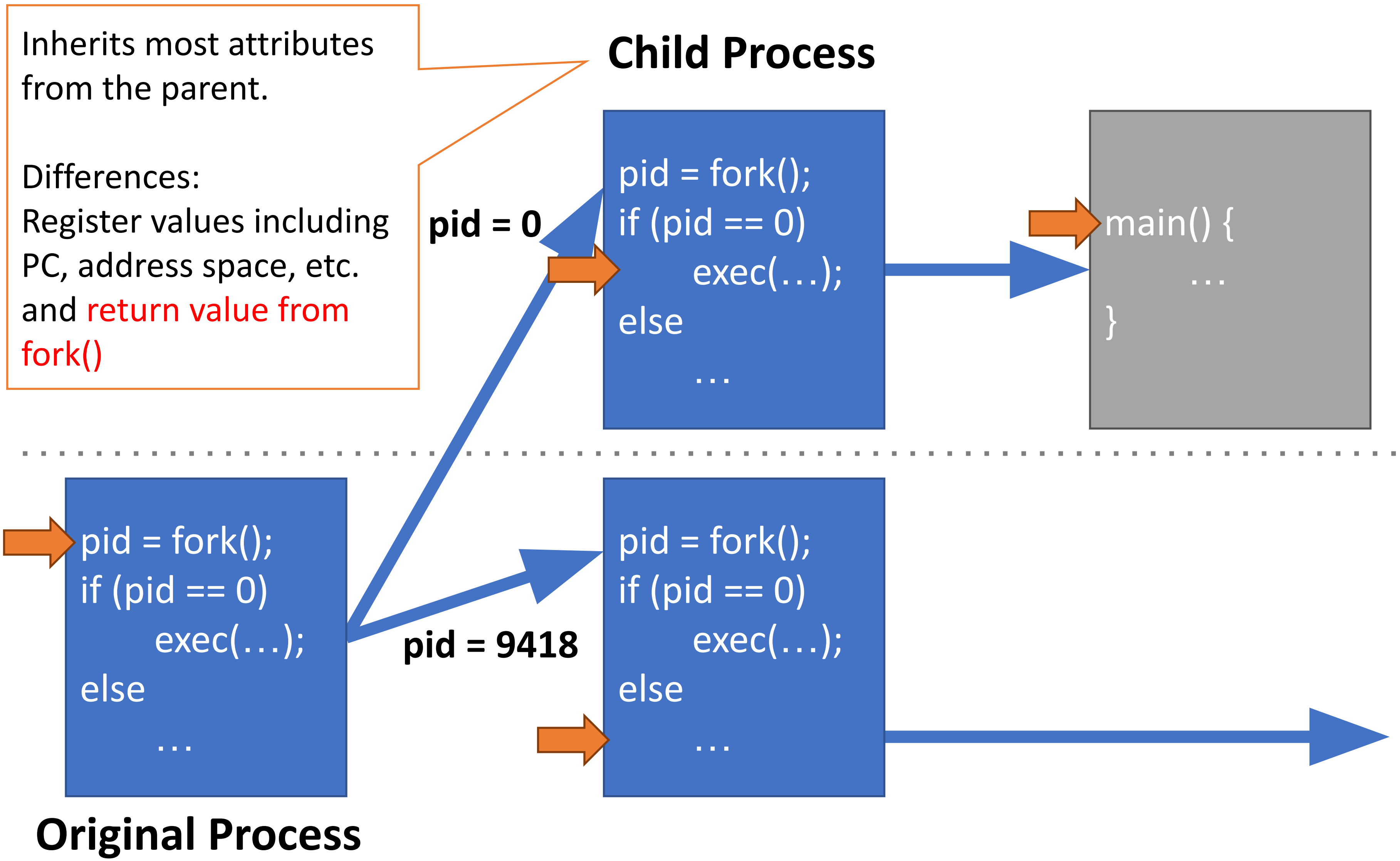
man fork

```
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)
```

Conceptual View of fork() | The before and after



UNIX Process Management



Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {          // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                       // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Fork demo

Additional Process commands

- int exec(const char *pathname, char *argv[], ...)
  - System call to change the program being run by the current process
- wait() – system call to wait for a process to finish
- signal() – system call to send a notification to another process

- 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)

Process State

- When our process is running, it may be in one of the states below
  - Running
  - Ready
  - Blocked

- What if it’s stopped permanently?
  - Terminated

Process Termination

- Process may be terminated for 3 reasons
  - Receives a signal to terminate
  - Returns from main routine
    (what we have normally been doing in the class)
  - Calling the exit function
    - 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)

Process Termination

- Typically, a process will wait(pid) until its child process(es) complete
  - You will learn about zombie and orphaned processes in the lab

- abort(pid) can be used to immediately end a child process