Lecture 17
Processes

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2026

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administrivia

- Homework 3 due tonight
  - No late submissions
- SETI Lab due Thursday
  - You can still apply slip days / late penalties to it if you need to
  - Be aware: testing takes 5-20 minutes. Come Thursday evening there may be a queue for testing
- Midterm Exam 3 details posted
  - Covers Memory Security through Networks

Today's Goals

- Introduce the role of an operating system

- Explore how processes work, particularly how they interact with the OS.
  - Describe mechanisms: Exceptions, System Calls, and Signals
  - Introduce control rules: scheduling policies

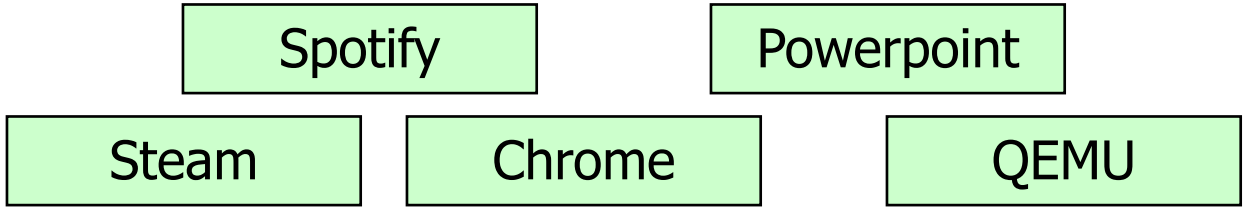
Outline

- **Intro to Operating Systems**

- Processes
  - Process Contents
  - Control Flow
  - Scheduling
  - System Calls
  - Signals

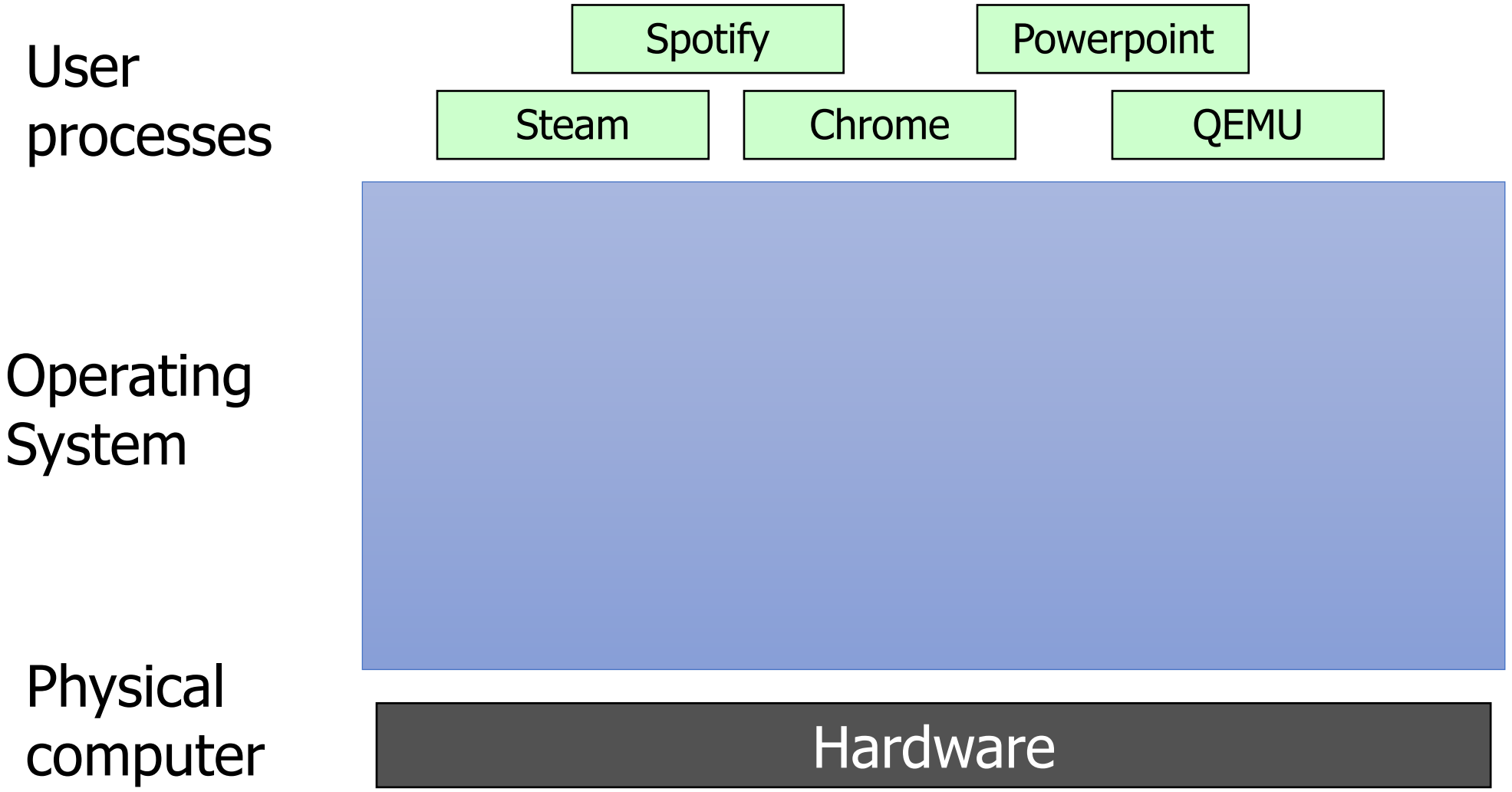
What is an operating system?

User processes

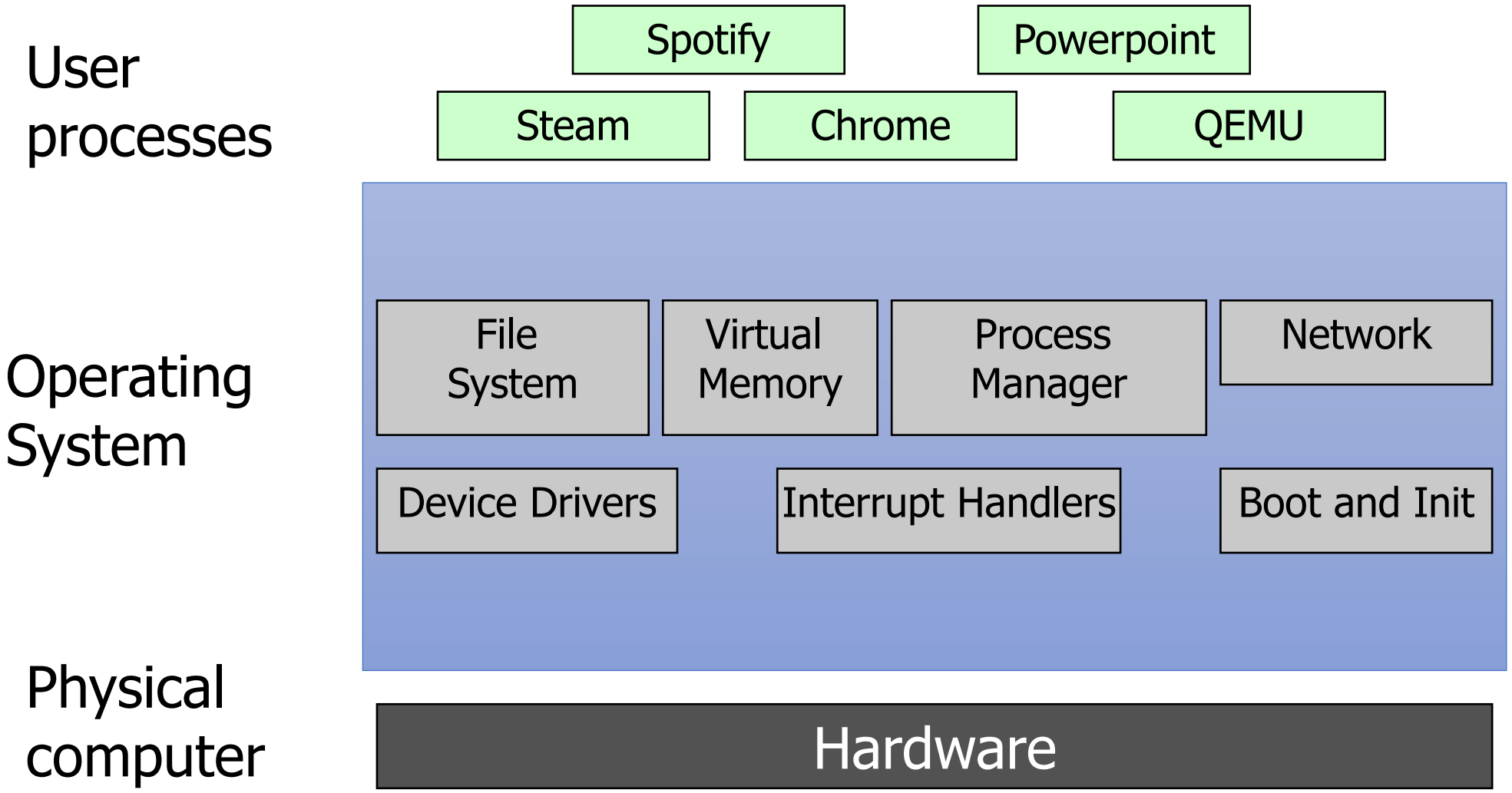


Physical computer

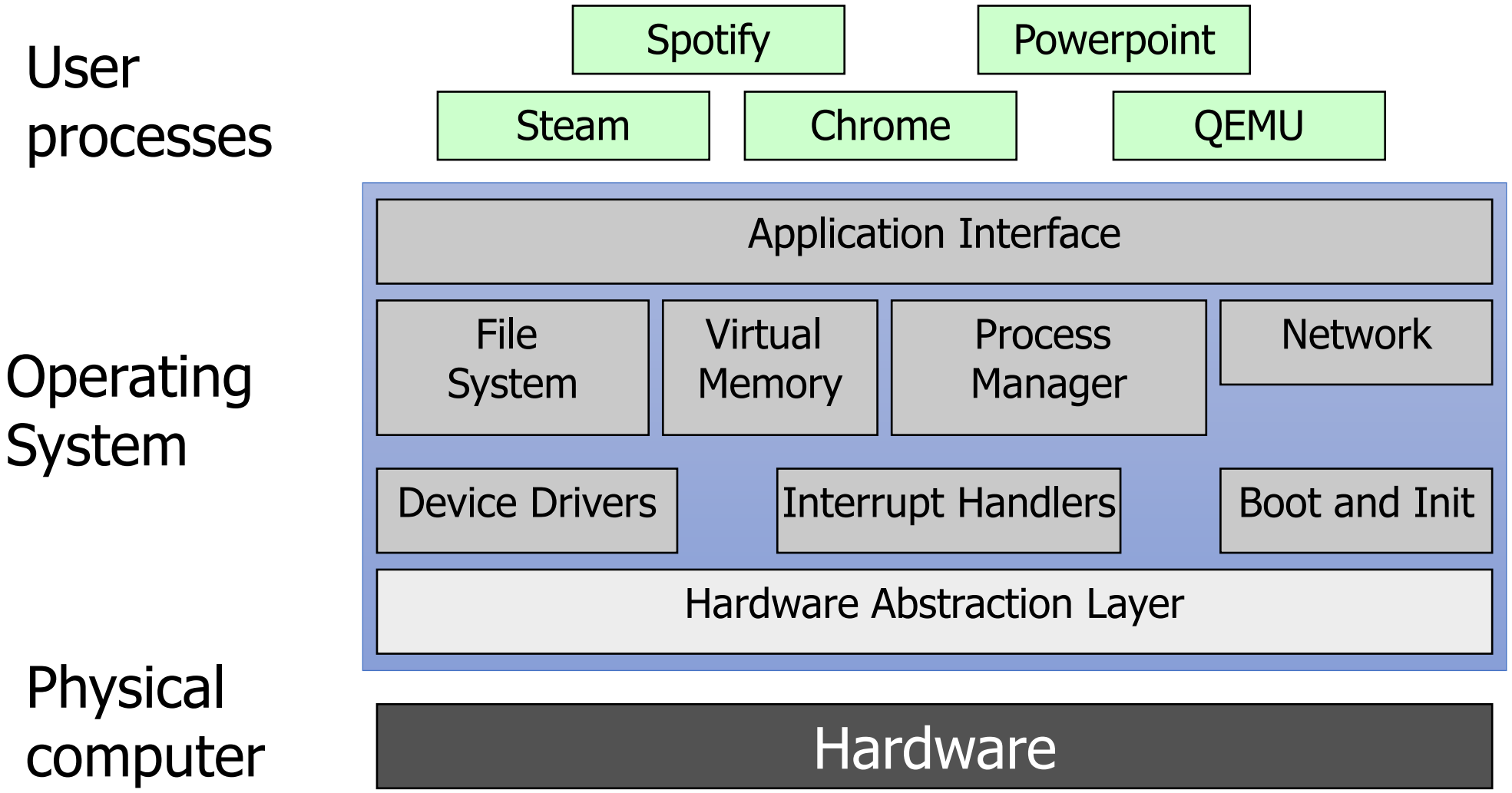
What is an operating system?



What is an operating system?



What is an operating system?



What's part of the OS?

- **OS kernel** – the part of the OS code that has full control of the system

- Process scheduling (who uses CPU)
- Memory allocation (who uses RAM)
- Accesses hardware devices
  - Outputs graphics
  - Reads/writes to network
  - Read/write to disks
  - Handles boot-up and power-down

- **OS distribution** – the kernel + lots of other useful stuff

- GUI / Window manager
- Command shell
- Software package manager
  - "app store", yum, apt, brew
- Common software libraries
- Useful apps:
  - Text editor, compilers, web browser, web server, SSH, anti-virus, file-sharing, media libraries,

Major goals of an Operating System

- OSes make advancing technology available to rapidly evolving applications. They do so with two major goals:

  1. Provide **abstractions** to applications to enable hardware compatibility
     - Why: allow reuse of common features, avoid low-level details
     - Challenges: What are the correct abstractions?

  2. Manage **sharing of resources** across many applications
     - Why: protect applications, enforce fair access
     - Challenges: What are the mechanisms and what are the policies?

- Good operating systems do these quickly, efficiently, and securely

Three features we'll consider today

- Memory
- Files
- Processes

OS Abstraction: Memory Management

- Memory management
  - OS is in charge of allocating memory
  - Abstraction: Address Space via Virtual Memory

- OS allows user processes to ignore real messiness of memory

Another example OS abstraction: File Systems

- OS manages file systems to restrict and abstract them

- Restrictions
  - Prevent users from accessing other's files without permission

- Abstractions
  - Files can grow infinitely large
  - Where a file exists in memory or disk isn't important!
  - Default file system types, named directories, file explorer

Files

- Collections of data
  - Usually in permanent storage on your computer (on disk)
  - Interactions managed by the OS

- Types of files
  - Regular files
    - Arbitrary data
    - Think of as a big array of bytes
  - Directories
    - Collections of regular files
  - Special files
    - Links, pipes, devices

File permissions

- Files have owners and permissions associated with them

```
[brghena@ubuntu code] $ ls -la
total 32
drwxrwxr-x 2 brghena brghena  4096 Nov 18 22:07 .
drwxr-xr-x 4 brghena brghena  4096 Nov 18 18:38 ..
-rwxrwxr-x 1 brghena brghena 16704 Nov 18 21:42 arguments
-rw-rw-r-- 1 brghena brghena   235 Nov 18 21:42 arguments.c
```

File permissions

- Files have owners and permissions associated with them

```
[brghena@ubuntu code] $ ls -la
total 32
drwxrwxr-x 2 brghena brghena  4096 Nov 18 22:07 .
drwxr-xr-x 4 brghena brghena  4096 Nov 18 18:38 ..
-rwxrwxr-x 1 brghena brghena 16704 Nov 18 21:42 arguments
-rw-rw-r-- 1 brghena brghena   235 Nov 18 21:42 arguments.c
```

- Permissions for the owner and name of the owner
  - Read, Write, eXecute
    - Cannot execute `arguments.c`
  - For directories: Read contents, Write new contents, Traverse directory

File permissions

- Files have owners and permissions associated with them

```
[brghena@ubuntu code] $ ls -la
total 32
drwxrwxr-x 2 brghena brghena  4096 Nov 18 22:07 .
drwxr-xr-x 4 brghena brghena  4096 Nov 18 18:38 ..
-rwxrwxr-x 1 brghena brghena 16704 Nov 18 21:42 arguments
-rw-rw-r-- 1 brghena brghena   235 Nov 18 21:42 arguments.c
```

- Permissions for the group and name of the group
  - Example: I could make a CS213 group, add you all to it, and only give that group access to some folder or file

File permissions

- Files have owners and permissions associated with them

```
[brghena@ubuntu code] $ ls -la
total 32
drwxrwxr-x 2 brghena brghena  4096 Nov 18 22:07 .
drwxr-xr-x 4 brghena brghena  4096 Nov 18 18:38 ..
-rwxrwxr-x 1 brghena brghena 16704 Nov 18 21:42 arguments
-rw-rw-r-- 1 brghena brghena   235 Nov 18 21:42 arguments.c
```

- Permissions for everyone else on the computer
  - Not the owner and not in the group
  - For my personal machine, not particularly relevant
  - For Moore, probably don’t want to let others read your files…

Three features we'll consider today

- Memory
- Files
- Processes

Processes are an abstraction provided by the OS

- The machine itself usually doesn’t support processes
  - Just has a processor and a set of registers
  - Memory is just arbitrary memory

- OS provides the abstraction
  - Multiple processes can run at the “same time”
  - Each has its own registers
  - Each has its own isolated memory

- Processes enable
  - Multiple functionalities on a computer
  - Multiprogramming of a system

Deep-dive into Processes

- Three major questions to consider:
  1. What makes up a process?
  2. How are processes controlled?
  3. How do processes and the OS interact?

Break + Open Question

- What makes Windows and MacOS different?

Break + Open Question

- What makes Windows and MacOS different?
  - Distribution: look and feel of desktop, programs they come packaged with
  - Kernel internals
    - Specific implementations of managing processor and memory
    - Security and permissions rules
    - Interfaces for interacting with them
  - A particular difference: hardware device support
    - Windows: supports lots of hardware, with drivers from companies
    - MacOS: supports minimal hardware, with their own driver software

Outline

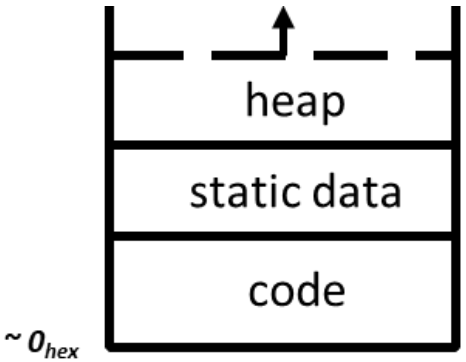
- Intro to Operating Systems

- **Processes**
  - **Process Contents**
  - Control Flow
  - Scheduling
  - System Calls
  - Signals

Reminder: view of a process

- Process: program that is being executed
- Contains code, data, and a thread
  - Thread contains registers, instruction pointer, and stack

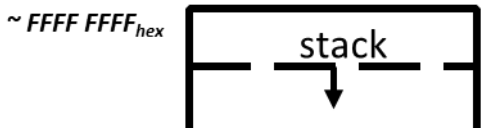
- Code and Data



- Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Instruction Pointer
- Condition Codes
- Stack



Process memory

- The OS creates a Page Table for each process
  - Allocates memory in RAM
  - Page Table connects chunks of RAM to meaningful process parts
    - I.e., this part is the stack
- Example memory map for ctarget from Attack Lab

```
[branden@moore ~]$ pmap -x 3379171
3379171:   ./ctarget -q
Address           Kbytes     RSS   Dirty Mode  Mapping
0000000000400000      16      16       0 r-x-- ctarget
0000000000603000       4       4       4 r---- ctarget
0000000000604000       4       4       4 rw--- ctarget
0000000000605000       4       4       4 rw---   [ anon ]
00000000022ab000     132       4       4 rw---   [ anon ]
0000000055586000    1024     264     264 rwx--   [ anon ]
00007f8724857000    1844    1252       0 r-x-- libc-2.28.so
00007f8724a24000    2044       0       0 ----- libc-2.28.so
00007f8724c23000      16      16      16 r---- libc-2.28.so
00007f8724c27000       8       8       8 rw--- libc-2.28.so
00007f8724c29000      16      12      12 rw---   [ anon ]
00007f8724c2d000     188     188       0 r-x-- ld-2.28.so
00007f8724e40000       8       4       4 rw---   [ anon ]
00007f8724e5a000       8       8       8 rw---   [ anon ]
00007f8724e5c000       4       4       4 r---- ld-2.28.so
00007f8724e5d000       8       8       8 rw--- ld-2.28.so
00007fff58b75000     132      20      20 rw---   [ stack ]
00007fff58bf8000      16       0       0 r----   [ anon ]
00007fff58bfc000       8       4       0 r-x--   [ anon ]
ffffffffff600000       4       0       0 r-x--   [ anon ]
---------------- ------- ------- -------
total kB            5488    1820     360
```

POSIX processes also have file descriptors

- Integers specifying a file the process is interacting with
  - Process contains a table linking integers to files (and permissions)

- Default file descriptors
  - 0 - Standard input (stdin)
  - 1 - Standard output (stdout)
  - 2 - Standard error (stderr)

- Function calls to interact with files
  - `int open (const char *path, int oflag, ... );`
  - `ssize_t read (int fildes, void *buf, size_t nbyte);`
  - `ssize_t write (int fildes, const void *buf, size_t nbyte);`

Example file descriptors

```
[brghena@ubuntu northwesternos.github.io] [master] $ lsof -p 6447
COMMAND  PID    USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
vim     6447 brghena  cwd    DIR    8,5     4096  524310 /home/brghena/Dropbox/class/cs343/northwesternos.github.io
vim     6447 brghena  rtd    DIR    8,5     4096       2 /
vim     6447 brghena  txt    REG    8,5  2906824 3418729 /usr/bin/vim.basic
vim     6447 brghena  mem    REG    8,5    51832 3415904 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
vim     6447 brghena  mem    REG    8,5 14537584 3414469 /usr/lib/locale/locale-archive
vim     6447 brghena  mem    REG    8,5    47064 3415927 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.4
vim     6447 brghena  mem    REG    8,5   182344 3416338 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.8
vim     6447 brghena  mem    REG    8,5    14848 3416317 /usr/lib/x86_64-linux-gnu/libutil-2.31.so
vim     6447 brghena  mem    REG    8,5   108936 3416470 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
vim     6447 brghena  mem    REG    8,5   182560 3415356 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.11
vim     6447 brghena  mem    REG    8,5    39368 3415768 /usr/lib/x86_64-linux-gnu/libltdl.so.7.3.1
vim     6447 brghena  mem    REG    8,5   100520 3416225 /usr/lib/x86_64-linux-gnu/libtdb.so.1.4.2
vim     6447 brghena  mem    REG    8,5    38904 3416342 /usr/lib/x86_64-linux-gnu/libvorbisfile.so.3.3.7
vim     6447 brghena  mem    REG    8,5   584392 3415988 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.9.0
vim     6447 brghena  mem    REG    8,5  2029224 3415140 /usr/lib/x86_64-linux-gnu/libc-2.31.so
vim     6447 brghena  mem    REG    8,5   157224 3416045 /usr/lib/x86_64-linux-gnu/libpthread-2.31.so
vim     6447 brghena  mem    REG    8,5  5416192 3416058 /usr/lib/x86_64-linux-gnu/libpython3.8.so.1.0
vim     6447 brghena  mem    REG    8,5    18816 3415275 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
vim     6447 brghena  mem    REG    8,5    22456 3415526 /usr/lib/x86_64-linux-gnu/libgpm.so.2
vim     6447 brghena  mem    REG    8,5    39088 3415026 /usr/lib/x86_64-linux-gnu/libacl.so.1.1.2253
vim     6447 brghena  mem    REG    8,5    71680 3415157 /usr/lib/x86_64-linux-gnu/libcanberra.so.0.2.5
vim     6447 brghena  mem    REG    8,5   163200 3416142 /usr/lib/x86_64-linux-gnu/libselinux.so.1
vim     6447 brghena  mem    REG    8,5   192032 3416251 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
vim     6447 brghena  mem    REG    8,5  1369352 3415780 /usr/lib/x86_64-linux-gnu/libm-2.31.so
vim     6447 brghena  mem    REG    8,5   191472 3414925 /usr/lib/x86_64-linux-gnu/ld-2.31.so
vim     6447 brghena    0u   CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    1u   CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    2u   CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    4u   REG    8,5    16384  524588 /home/brghena/Dropbox/class/cs343/northwesternos.github.io/.index.html.swp
```

Also the code files mapped to the address space

```
[brghena@ubuntu northwesternos.github.io] [master] $ lsof -p 6447
COMMAND  PID    USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
vim     6447 brghena  cwd    DIR    8,5     4096  524310 /home/brghena/Dropbox/class/cs343/northwesternos.github.io
vim     6447 brghena  rtd    DIR    8,5     4096       2 /
vim     6447 brghena  txt    REG    8,5  2906824 3418729 /usr/bin/vim.basic
vim     6447 brghena  mem    REG    8,5    51832 3415904 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
vim     6447 brghena  mem    REG    8,5 14537584 3414469 /usr/lib/locale/locale-archive
vim     6447 brghena  mem    REG    8,5    47064 3415927 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.4
vim     6447 brghena  mem    REG    8,5   182344 3416338 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.8
vim     6447 brghena  mem    REG    8,5    14848 3416317 /usr/lib/x86_64-linux-gnu/libutil-2.31.so
vim     6447 brghena  mem    REG    8,5   108936 3416470 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
vim     6447 brghena  mem    REG    8,5   182560 3415356 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.11
vim     6447 brghena  mem    REG    8,5    39368 3415768 /usr/lib/x86_64-linux-gnu/libltdl.so.7.3.1
vim     6447 brghena  mem    REG    8,5   100520 3416225 /usr/lib/x86_64-linux-gnu/libtdb.so.1.4.2
vim     6447 brghena  mem    REG    8,5    38904 3416342 /usr/lib/x86_64-linux-gnu/libvorbisfile.so.3.3.7
vim     6447 brghena  mem    REG    8,5   584392 3415988 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.9.0
vim     6447 brghena  mem    REG    8,5  2029224 3415140 /usr/lib/x86_64-linux-gnu/libc-2.31.so
vim     6447 brghena  mem    REG    8,5   157224 3416045 /usr/lib/x86_64-linux-gnu/libpthread-2.31.so
vim     6447 brghena  mem    REG    8,5  5416192 3416058 /usr/lib/x86_64-linux-gnu/libpython3.8.so.1.0
vim     6447 brghena  mem    REG    8,5    18816 3415275 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
vim     6447 brghena  mem    REG    8,5    22456 3415526 /usr/lib/x86_64-linux-gnu/libgpm.so.2
vim     6447 brghena  mem    REG    8,5    39088 3415026 /usr/lib/x86_64-linux-gnu/libacl.so.1.1.2253
vim     6447 brghena  mem    REG    8,5    71680 3415157 /usr/lib/x86_64-linux-gnu/libcanberra.so.0.2.5
vim     6447 brghena  mem    REG    8,5   163200 3416142 /usr/lib/x86_64-linux-gnu/libselinux.so.1
vim     6447 brghena  mem    REG    8,5   192032 3416251 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
vim     6447 brghena  mem    REG    8,5  1369352 3415780 /usr/lib/x86_64-linux-gnu/libm-2.31.so
vim     6447 brghena  mem    REG    8,5   191472 3414925 /usr/lib/x86_64-linux-gnu/ld-2.31.so
vim     6447 brghena    0u   CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    1u   CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    2u   CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    4u   REG    8,5    16384  524588 /home/brghena/Dropbox/class/cs343/northwesternos.github.io/.index.html.swp
```

Standard I/O is a process thing, not a C thing

- You can access them in Python, for instance
  - https://docs.python.org/3/library/sys.html#sys.stdin

`sys.`**`stdin`**
`sys.`**`stdout`**
`sys.`**`stderr`**

File objects used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to `input()`);
- `stdout` is used for the output of `print()` and expression statements and for the prompts of `input()`;
- The interpreter's own prompts and its error messages go to `stderr`.

These streams are regular text files like those returned by the `open()` function. Their parameters are chosen as follows:

Additional process contents

- Whatever else the OS thinks is useful
  - Process ID
  - Priority
  - Time Used
  - Process State

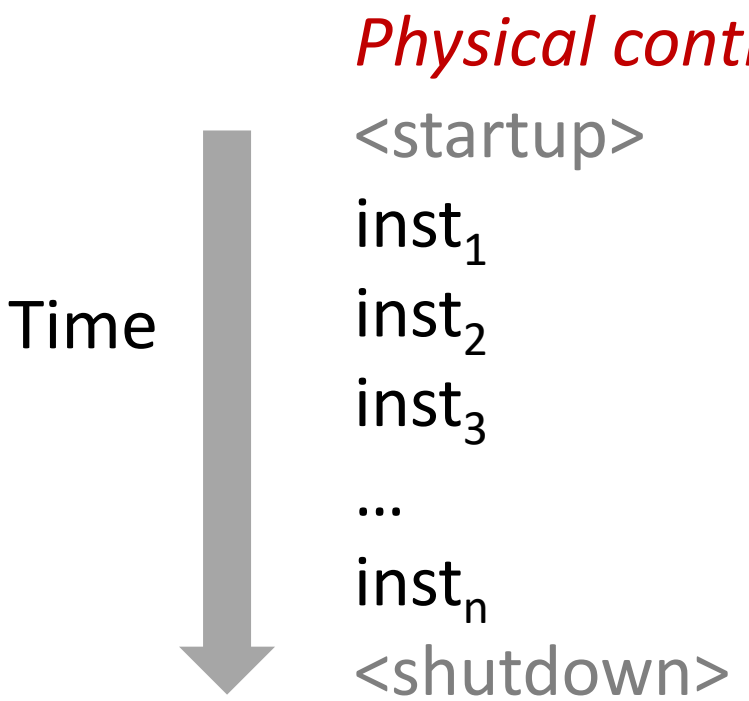
- Different OSes will attach different things to the "process abstraction"
  - Literally one or more data structures per process

Outline

- Intro to Operating Systems
- **Processes**
  - Process Contents
  - **Control Flow**
  - Scheduling
  - System Calls
  - Signals

Control flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)



Altering control flow

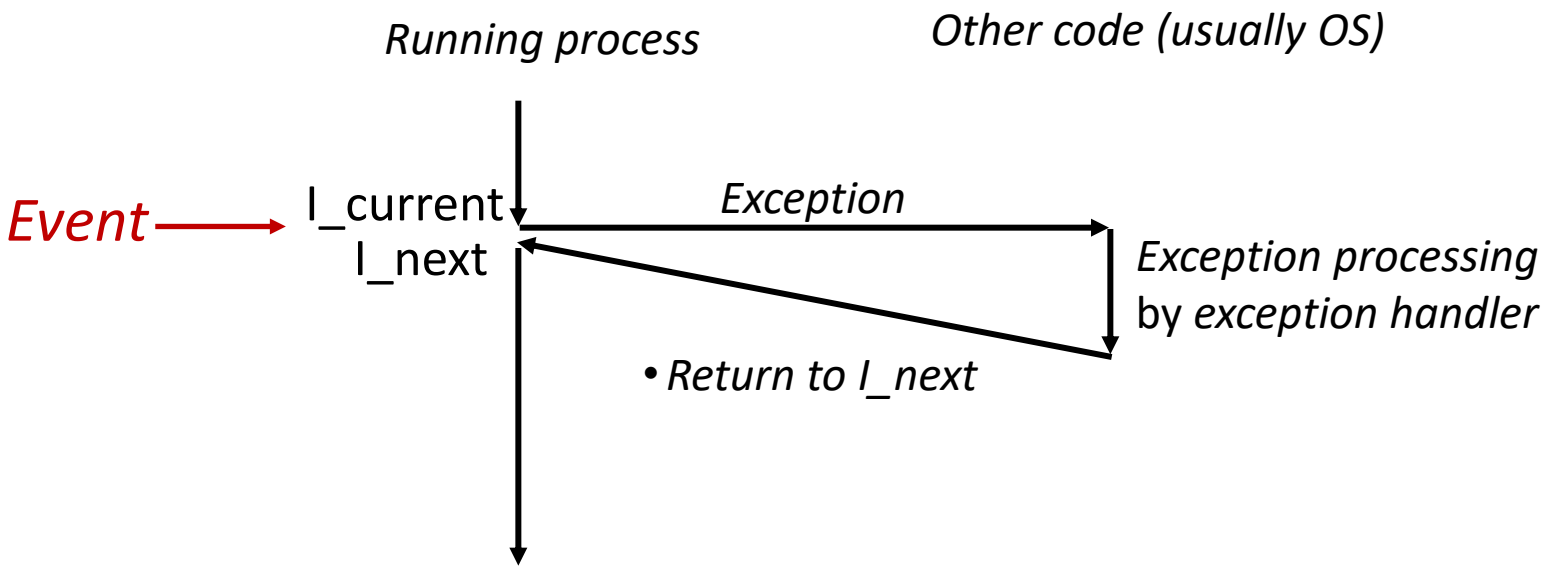
- Instructions that change control flow allow software to react changes in program state
  - Jumps/branches
  - Call/return

- But computers also need to react to changes in system state
  - Data arrives at network adapter
  - Instruction divides by zero
  - User hits Ctrl-C on the keyboard
  - System timer expires

- These mechanisms are known as “exceptional control flow”

Exceptional control flow

- Context switch: execution changes from process to OS
  - Typically triggered by a hardware or software event: exceptions



Exceptions

- Hardware detects an event that OS software needs to resolve immediately
- Could be an error
  - Invalid memory access
  - Invalid instruction
- Could just be something the OS should handle
  - Page fault
  - USB device detected
- OS has a table of “exception handlers”, which are functions that handle each exception class (also known as interrupt handlers)
  - Hardware jumps execution to the proper handler

Perspective of the process on exceptions

- The process doesn’t know they happened at all
  - First it was running an instruction
  - Them some time passed
  - Then it was running the next instruction
- And there’s no guarantee on how long an instruction takes anyways
  - Could be a really time-intensive instruction
  - Could be a cache miss
  - Could require loading data from disk
- So the program runs and works like normal
  - But the OS briefly takes over to handle something first

Outline

- Intro to Operating Systems
- **Processes**
  - Process Contents
  - Control Flow
  - **Scheduling**
  - System Calls
  - Signals

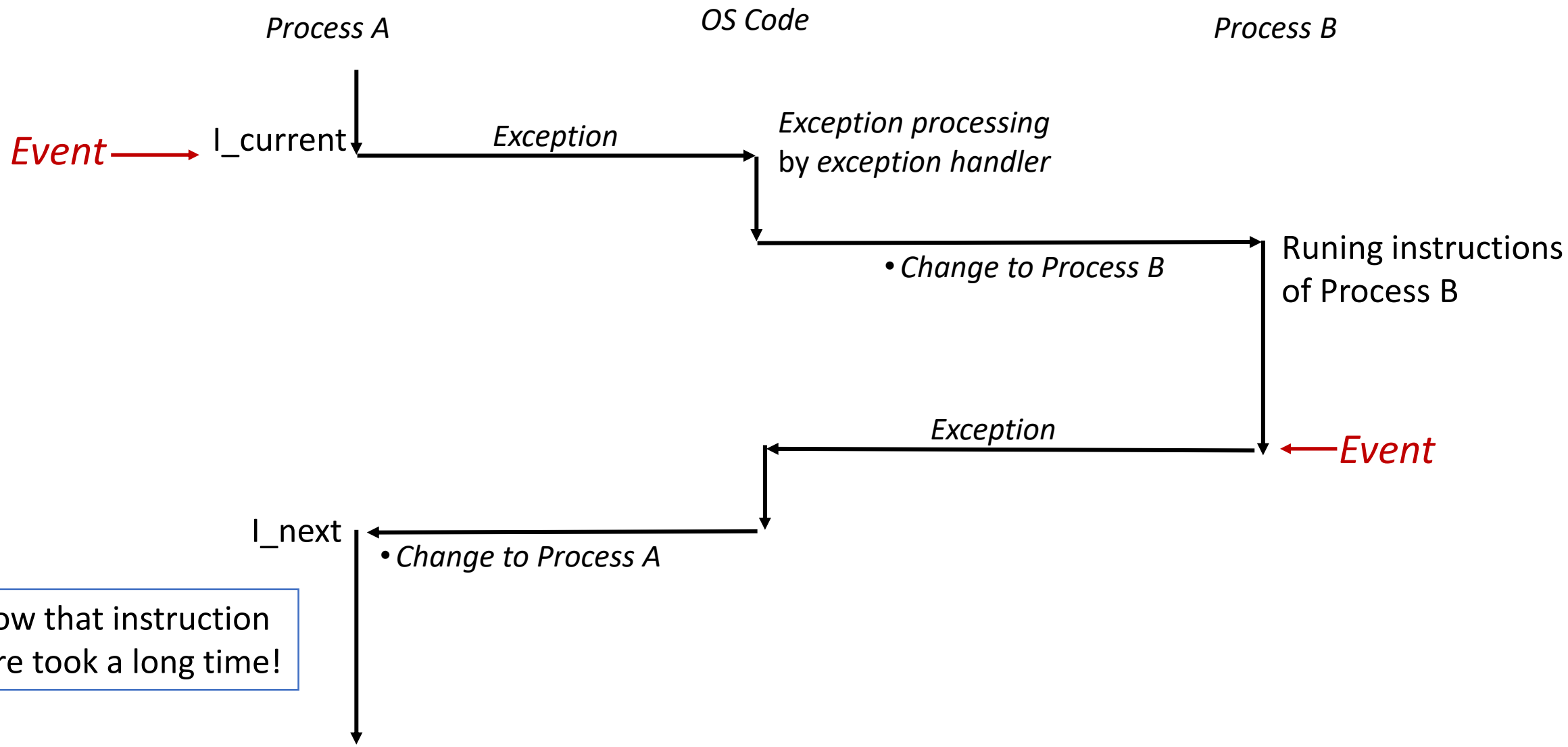
Lies your operating system always told you

- "Every process on your computer gets to run at the same time!"
  - This is an *illusion*

- My desktop at home (running Windows)
  - Current load: 250 processes with 2987 threads

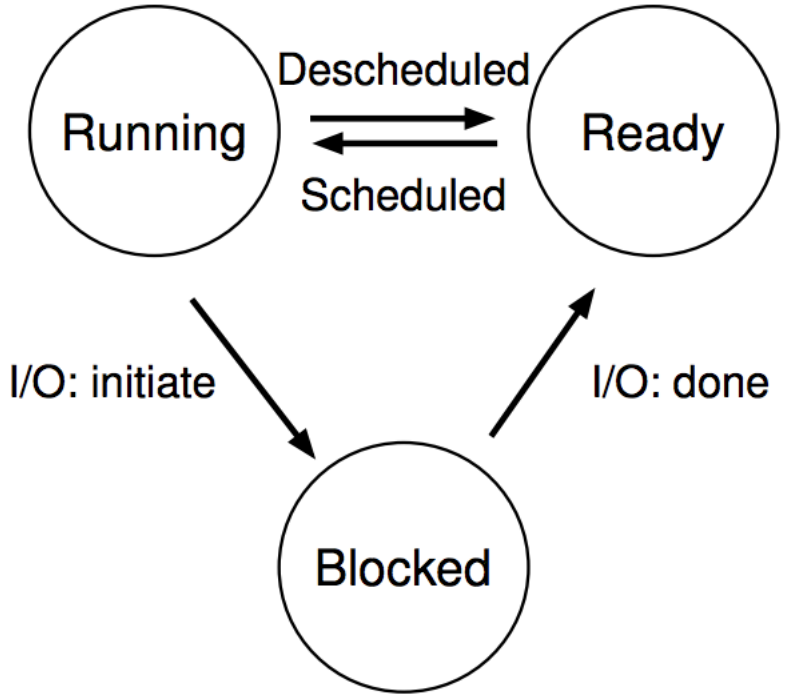
- So how does the magic work?

Exceptional control flow can change the running process



Processes don’t run all the time

The three basic process states:

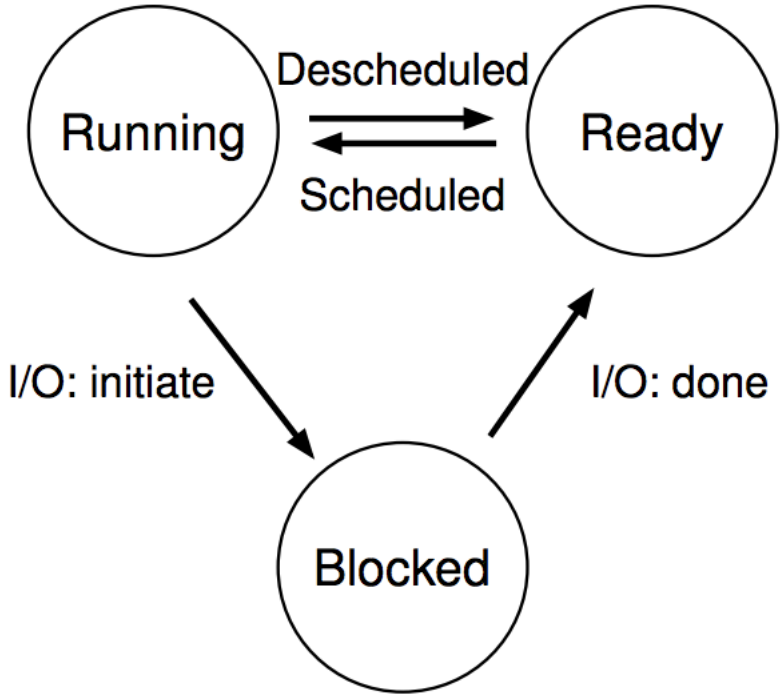


- OS ***schedules*** processes
  - Decides which of many competing processes to run.
- A ***blocked*** process is not ready to run and is waiting on I/O
- I/O means input/output – anything other than computing.
  - For example, reading/writing disk, sending network packet, waiting for keystroke
  - While waiting for results, the OS **blocks** the process, waiting to do more computation until the result is ready

Multiprogramming processes

- Even with a single processor, the OS can provide the illusion of many processes running simultaneously
  - And also use this opportunity to get more useful work done
- When one process is Blocked, OS can schedule a different process that is Ready
- OS can also swap between various Ready processes so they all make progress

The three basic process states:



Scheduling

- We know that multiple processes will be sharing the CPU
  - Possibly multiple threads in each process
  - Possibly multiple cores in the CPU

- Scheduling is creating a *policy* for sharing the CPU
  - Which process/thread is chosen to run, and when?
  - When (if ever) does the OS change which process is running?

When can the OS make scheduling decisions?

- Whenever the OS is actually running
  - i.e. after a context switch

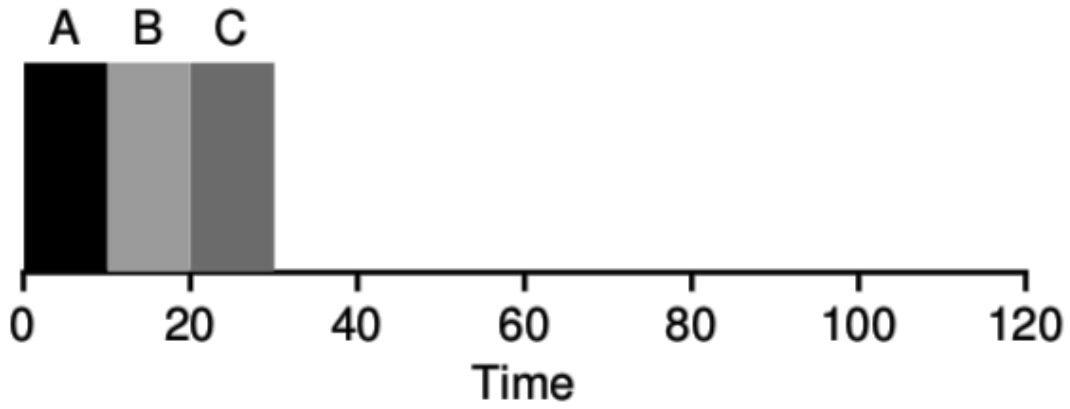
- Possible triggers
  - Hardware events (interrupts)
    - I/O complete
    - Timer triggers – this is the default option

  - Software events
    - Faults in processes
    - System calls (requests from processes)

Example scheduler: FIFO Scheduling

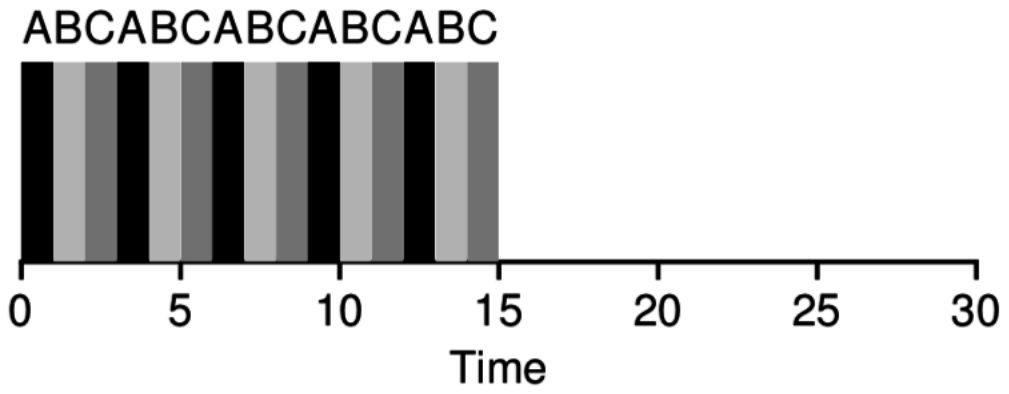
- First In, First Out (FIFO)
  - also known as First Come First Served (FCFS)

- Policy
  - First job to arrive gets scheduled first
  - Let a job continue until it is complete
  - Then schedule next remaining job with earliest arrival



Example scheduler: Round Robin Scheduling

- Round Robin scheduling runs a job for a small *timeslice* (quanta), then schedules the next job



- Over time each job makes growing amounts of progress
- If we switch fast enough, it seems to the user that all jobs are running in parallel

Break + Open Question

- Does it always make sense to use Round Robin scheduling? Can you think of examples where we would want something different?

Break + Open Question

- Does it always make sense to use Round Robin scheduling? Can you think of examples where we would want something different?
  - Server requests
    - Probably want to prioritize *finishing* requests rather than getting some work done on many of them
  - High-reliability computing
    - Might have a literal priority of some tasks as more important than others

Outline

- Intro to Operating Systems
- **Processes**
  - Process Contents
  - Control Flow
  - Scheduling
  - **System Calls**
  - Signals

Things a program cannot do itself

- Print “hello world”
  - *because the display is a shared resource.*
- Download a web page
  - *because the network card is a shared resource.*
- Save or read a file
  - *because the filesystem is a shared resource, and the OS wants to check file permissions first.*
- Launch another program
  - *because processes are managed by the OS*
- Send data to another program
  - *because each program runs in isolation, one at a time*

The OS can do things Processes cannot

- **Bad Idea:** let processes manage memory or devices directly
  - We do NOT trust processes
  - Instead the OS should play referee and manager here
  - Hardware supports this limitation with “kernel mode”
    - Some operations cannot be performed unless in kernel mode

- Alternative: processes can *request* something from the OS
  - The OS can consider whether it should do that thing
  - Then the OS has the power to actually do it if necessary

- Requirements
  1. Context switch into kernel
  2. Inform the kernel what you want it to do

Trap instructions

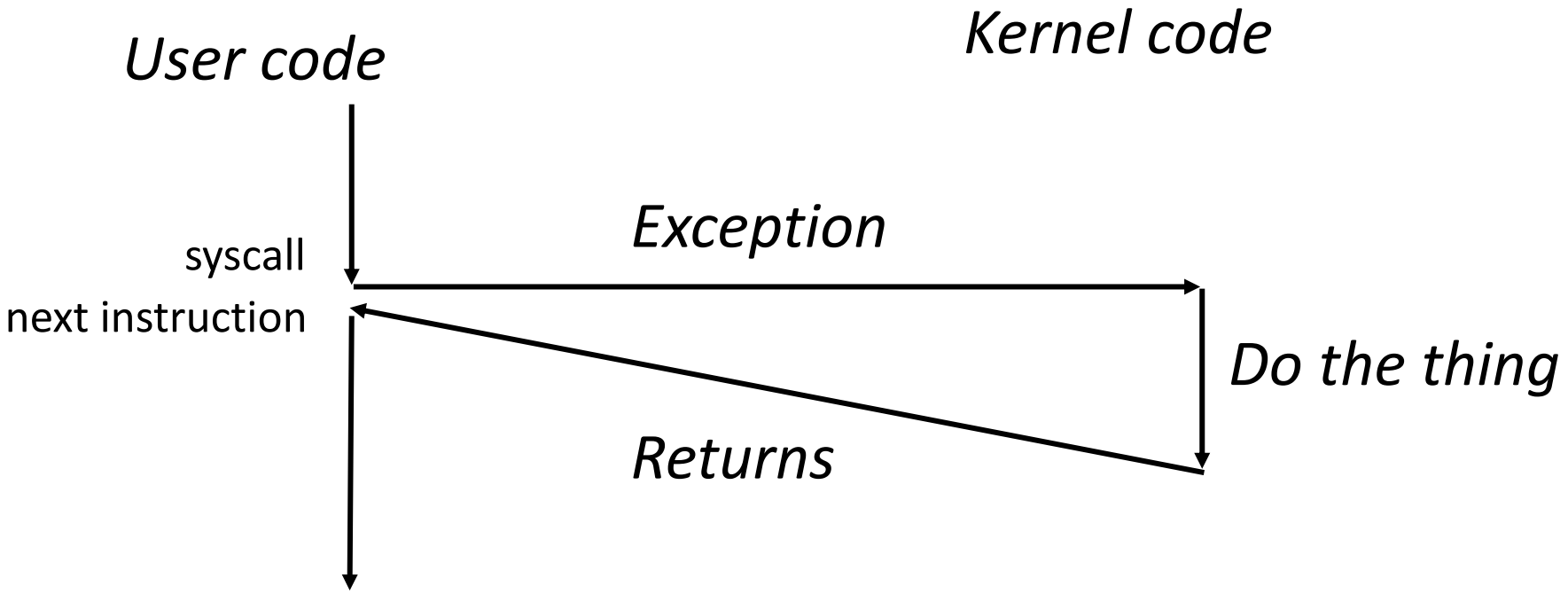
- On almost all architectures, there is a special instruction that causes an exception to occur
  - Triggers a context switch into the OS

- x86-64: `syscall`
- Other systems
  - x86 (32-bit and older): `int` (short for interrupt)
  - ARM: `svc`
  - RISC-V: `ecall`

- Hardware triggers a context switch, running the OS handler

System call example

- System call: making a request of the OS from a process
  - Uses exceptional control flow to enter OS kernel
  - Returns back to process when complete
    - Returns to instruction *after* the system call



System call steps (simplification)

1. Process loads parameters into registers (just like a function call)
2. Process executes trap instruction (`int`, `syscall`, `svc`, etc.)
3. Hardware moves `%rip` to "handler" and switches to kernel mode
4. OS checks what the process wants to do from registers
5. OS decides *whether* the process is allowed to do so

Returning from a system call (simplification)

- After OS finishes whatever operation it was asked to do
  - And when the process is scheduled to run again

1. OS places return result in a register (just like a function call)
2. OS changes mode to user mode (and sets virtual memory stuff)
3. OS sets `%rip` to instruction after the system call

- Process continues and can use results of system call

Linux system calls

- Example system calls
  - https://man7.org/linux/man-pages/man2/syscalls.2.html

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

Example using system calls

- Let’s create new processes with system calls

- From process view:
  - Just look like regular C functions
  - Take arguments, return values

- Underneath:
  - Function uses special assembly instruction to trigger exception

The C function for syscalls just performs the correct assembly

- Example system call: fork

Number	*Name*	*Description*
57	`fork`	Create process

  - C code:

```
fork();
```

  - x86-64 assembly implementation:

```
fork:
   movq $57, %rdi
   syscall
   retq
```

Process management system calls

`pid_t fork(void);`

- Create a new process that is a copy of the current one
- Returns either PID of child process (parent) or 0 (child)

`void _exit(int status);`

- Exit the current process (exit(), the library call cleans things up first)

`pid_t waitpid(pid_t pid, int *status, int options);`

- Suspends the current process until a child (*pid*) terminates

`int execve(const char *filename, char *const argv[], char *const envp[]);`

- Execute a new program, replacing the existing one
- Replaces code and data, clears registers, sets `%rip` to start again

Creating a new process

```
#include <stdio.h>
#include <unistd.h>

int main(){
  if(fork() == 0) {
    printf("Child!\n");
  } else {
    printf("Parent!\n");
  }

  printf("Both!\n");
  return 0;
}
```

Creating a new process

```
#include <stdio.h>
#include <unistd.h>

int main(){
  if(fork() == 0) {
    printf("Child!\n");
  } else {
    printf("Parent!\n");
  }

  printf("Both!\n");
  return 0;
}
```

Existential crisis

Executing a new program

```
#include <stdio.h>
#include <unistd.h>

int main(){
  if(fork() == 0) {
    execve("/bin/python3", ...);
  } else {
    printf("Parent!\n");
  }

  printf("Only parent!\n");
  return 0;
}
```

Break + Question

- What does the following code do?

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

int main() {
  while(1){
    fork();
  }
  return 0;
}
```

Break + Question

- What does the following code do?

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

int main() {
  while(1){
    fork();
  }
  return 0;
}
```

- Creates a new process
  - Then each process creates a new process
  - Then each of those creates a new process…
- Known as a Fork bomb!
  - Machine eventually runs out of memory and processing power and will stop working
- Defense: limit number of processes per user

Fork bombs in various languages

- Python fork bomb

```
import os
while 1:
   os.fork()
```

- Rust fork bomb

```
#[allow(unconditional_recursion)]
fn main() {
    std::thread::spawn(main);
    main();
}
```

- Bash fork bomb

```
:(){ :|:& };:
```

- Bash with spacing and a clearer function name

```
fork() {
    fork | fork &
}
fork
```

Other Syscalls – File interactions

- open() – open file
- read() – read contents from current offset
- write() – write data to file (usually at the end)
- lseek() – modify current offset in the file
- close() – close file

- These system calls work with process file descriptors

Higher-level methods of file interaction

- Here, we're talking about system calls to the OS

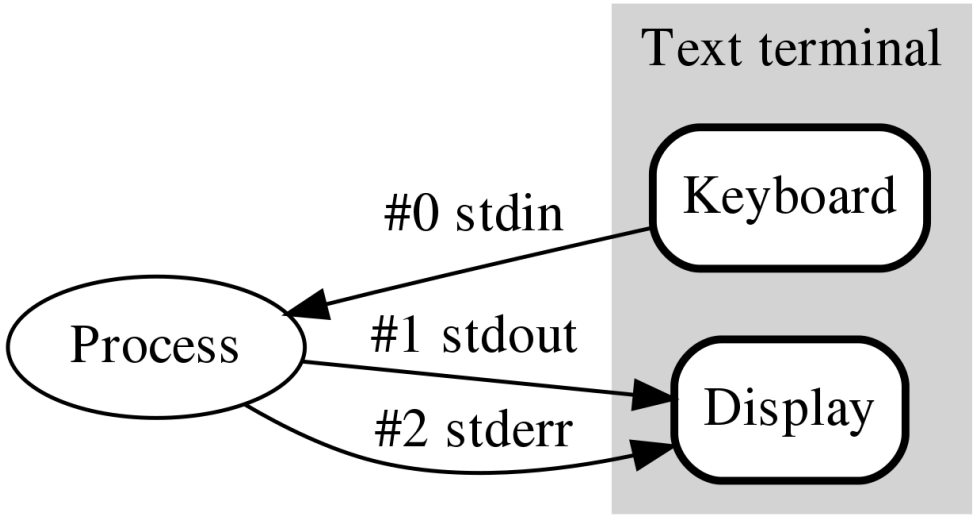
- C standard library also defines file interactions
  - fopen, fread, fwrite, fseek, fclose

  - All are wrappers on top of the actual syscalls

  - Buffers your interactions to make them more efficient
    - Reads/Writes large chunks of data at a time
    - Might collect multiple fwrite's before doing a single real write
    - fflush() guarantees that the buffer is written *now*

How do programs talk to users?

- We often gloss over this in CS211
  - printf()
  - gets()



- Work through the same file mechanism
  - Using standard I/O files set up when process starts
    - stdin – standard input (file descriptor 0)
    - stdout – standard output (file descriptor 1)
    - stderr – standard error (file descriptor 2)

- printf(…) -> fprintf(1, …) -> handle arguments then write(1, …)

Sidebar: how do you figure out how these calls work?

- Manual pages
- Online: https://man7.org/linux/man-pages/man2/close.2.html

close(2) — Linux manual page

NAME | SYNOPSIS | DESCRIPTION | RETURN VALUE | ERRORS | CONFORMING TO | NOTES | SEE ALSO | COLOPHON

CLOSE(2) **Linux Programmer's Manual** **CLOSE(2)**

NAME top

close - close a file descriptor

SYNOPSIS top

```
#include <unistd.h>

int close(int fd);
```

Outline

- Intro to Operating Systems
- **Processes**
  - Process Contents
  - Control Flow
  - Scheduling
  - System Calls
  - **Signals**

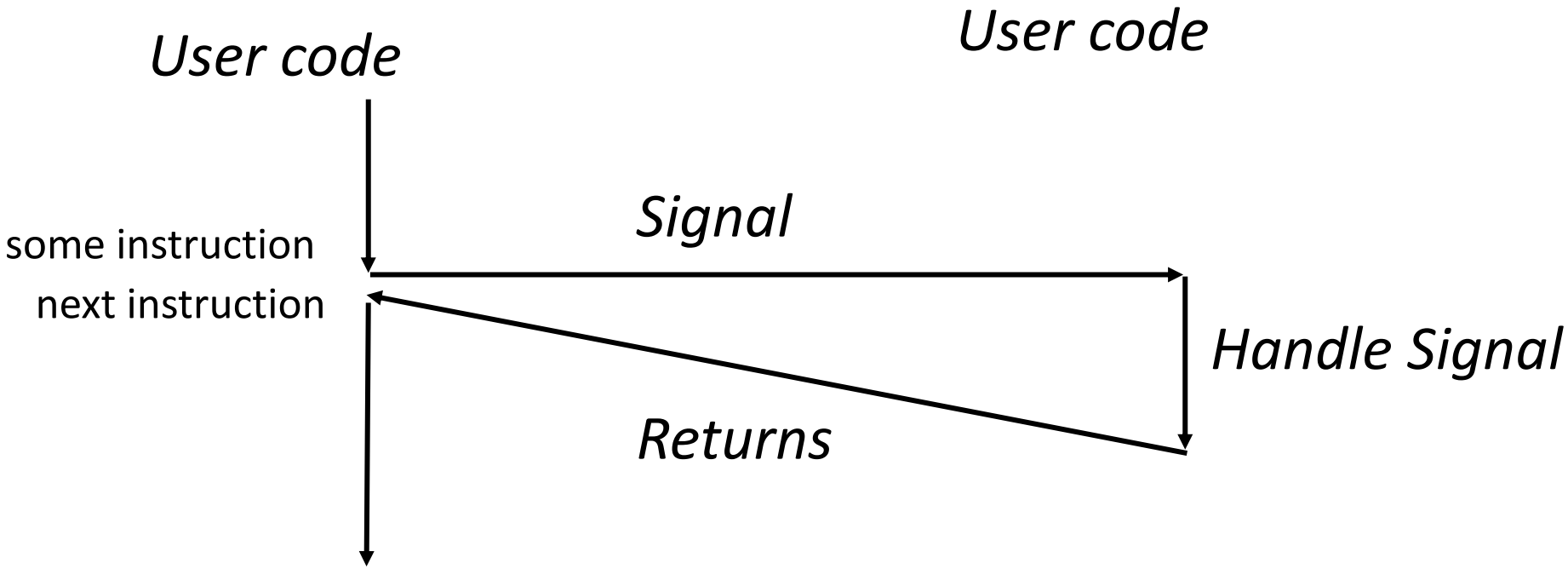
Alerting processes of events

- How do we let a process know there was an event?
  - Errors
  - Termination
  - User commands (like CTRL-C or CTRL-\)

- Events could happen whenever
  - Need to interrupt process control flow and run an event handler
  - Linux mechanism to do so is called “signals”

Signals are a different version of exceptional control flow

- Signal is generated by the OS
- Interrupts user code and jumps to a signal handler
  - Then returns back to user code afterwards
  - Unless the signal handler ends the program (this is the default handler)



Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called “Signals”

```
 1) SIGHUP     2) SIGINT   3) SIGQUIT   4) SIGILL  5) SIGTRAP
 6) SIGABRT    7) SIGBUS   8) SIGFPE    9) SIGKILL 10) SIGUSR1
11) SIGSEGV   12) SIGUSR2 13) SIGPIPE  14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT  19) SIGSTOP 20) SIGTSTP
21) SIGTTIN   22) SIGTTOU 23) SIGURG   24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO   30) SIGPWR
31) SIGSYS          ...
```

Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called “Signals”

```
 1) SIGHUP     2) SIGINT     3) SIGQUIT    4) SIGILL     5) SIGTRAP
 6) SIGABRT    7) SIGBUS     8) SIGFPE     9) SIGKILL   10) SIGUSR1
11) SIGSEGV   12) SIGUSR2   13) SIGPIPE   14) SIGALRM   15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD   18) SIGCONT   19) SIGSTOP   20) SIGTSTP
21) SIGTTIN   22) SIGTTOU   23) SIGURG    24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF   28) SIGWINCH  29) SIGIO     30) SIGPWR
31) SIGSYS           ...
```

Process Errors

Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called "Signals"

```
 1) SIGHUP     2) SIGINT    3) SIGQUIT    4) SIGILL     5) SIGTRAP
 6) SIGABRT    7) SIGBUS    8) SIGFPE     9) SIGKILL   10) SIGUSR1
11) SIGSEGV   12) SIGUSR2  13) SIGPIPE   14) SIGALRM   15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD  18) SIGCONT   19) SIGSTOP   20) SIGTSTP
21) SIGTTIN   22) SIGTTOU  23) SIGURG    24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF  28) SIGWINCH  29) SIGIO     30) SIGPWR
31) SIGSYS             ...
```

Process Termination

Sending signals

- OS sends signals when it needs to

- Processes can ask the OS send signals with a system call
  - `int kill(pid_t pid, int sig);`

- Users send signals through OS from command line or keyboard
  - Shell command: kill -9 *pid* (SIGKILL)
  - CTRL-C (SIGINT)

Handling signals

- Programs can register a function to handle individual signals
  - `signal(int sig, sighandler_t handler);`

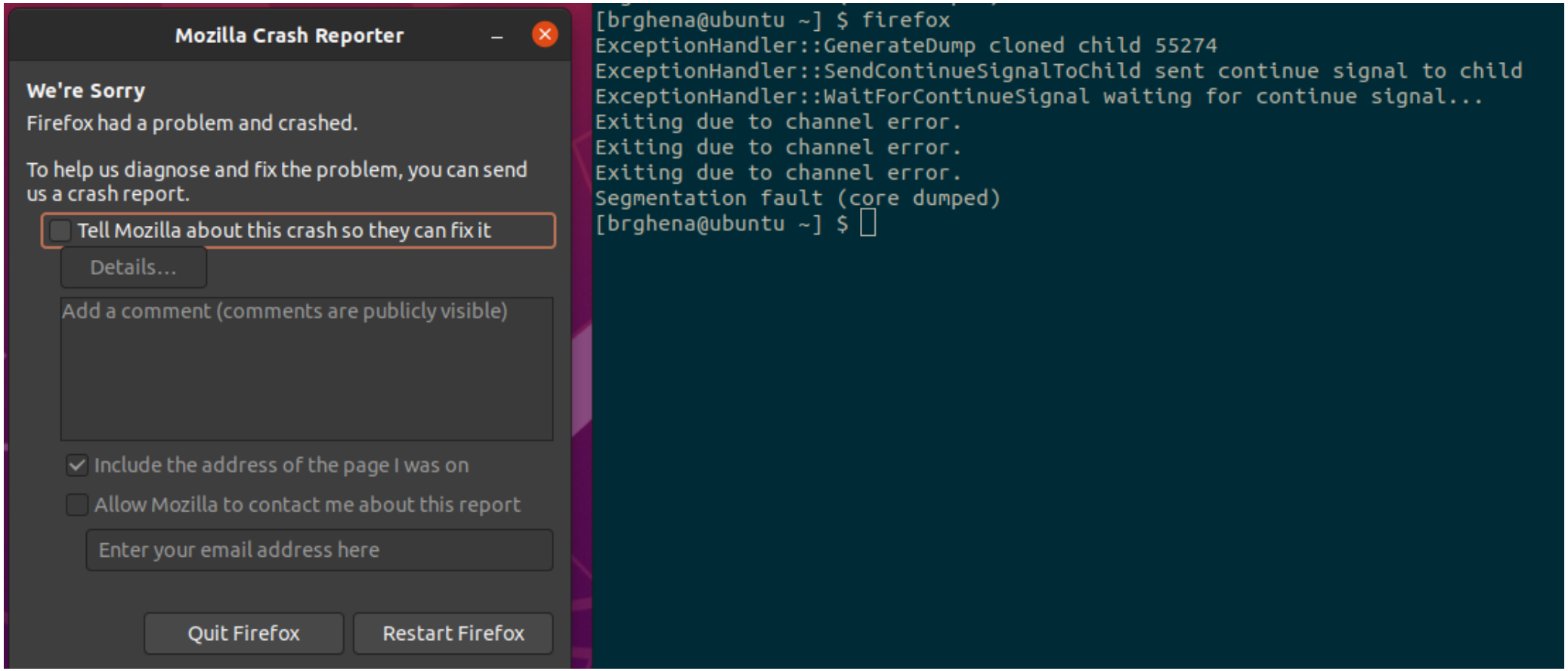
- What are you supposed to do about it?
  - Do some *quick* processing to handle it
    - That needs to be "reentrant" safe
  - Reset the process and try again
  - Quit the process (default handler)
  - Not much: signals are not supported by all OSes

Examples: sending a signal

> kill -11 *pid* (11 is SIGSEGV – a.k.a segfault)

Examples: sending a signal

> kill -11 *pid* (11 is SIGSEGV – a.k.a segfault)



CS343 – Operating Systems

- 213 topics in more depth: Concurrency, Processes, Virtual Memory
  - Deal with data races directly
  - Implement virtual memory system

- Also, totally new topics: File Systems and Devices

- Focus: “how does the Operating System make the computer work”?

Outline

- Intro to Operating Systems
- Processes
  - Process Contents
  - Control Flow
  - Scheduling
  - System Calls
  - Signals