How Programs Are Managed

11

CS/COE 0449
Introduction to Systems Software

wilkie
(with content borrowed from Vinicius Petrucci and Jarrett Billingsley)

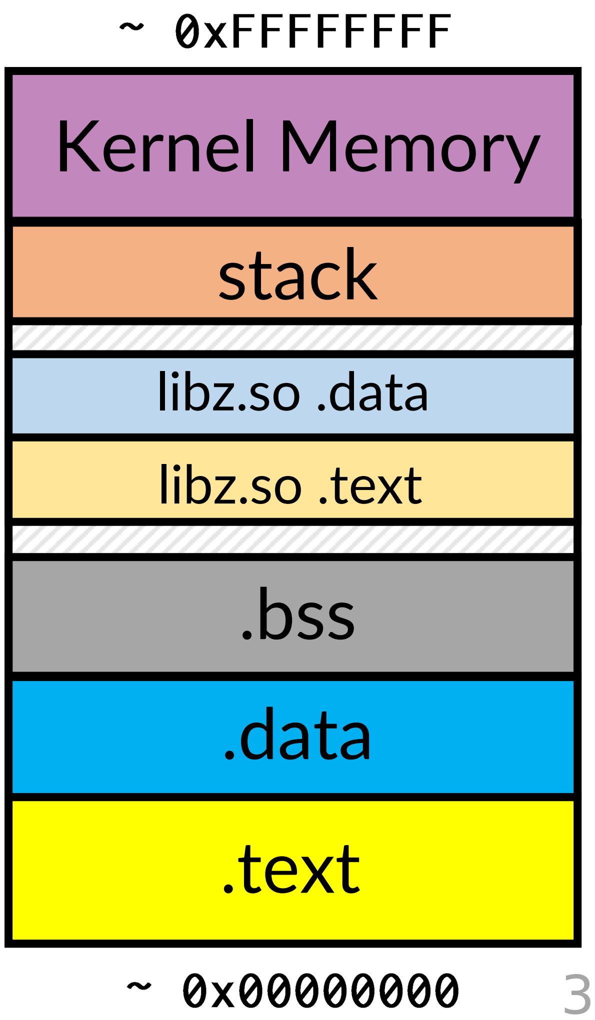
Spring 2019/2020

Where's the Lie?

And other operating systems questions.

On the last episode...

- Programs are loaded into memory by the operating system.
- They have to exist in memory before they can be executed.
- Programs go through a lot of trouble to have all their data/code in memory.



The Lie

- Programs are told that they are the only things running…
- The only things in memory…
- We know that this is not true!
- Operating Systems are big liars crafting illusions.

The Truth

- In reality, many programs can be running at the same time.

- Each program, when running, is typically called a **process**.
  - A **multitasking** OS is (a rather common) one that supports concurrent processes.

- The OS must handle switching from one process to another.
  - Which processes get to run?
  - What if you have more processes than CPUs?
  - When do you switch from one to another?
  - What if one is more urgent??

My process is one of method…

- A **process** is an abstraction representing a *single instance* of a program.
  - An executable represents the initial state of a program and thus the process.
  - A program can be instantiated multiple times, if needed.
  - Each one would be a separate process… of the same program.
  - Note: A *processor* is the hardware unit that executes a process. (makes sense!!)

- The Operating System defines what a process and its abstraction is.
  - There is an OS representation and metadata associated with a process.
  - The OS maintains two key lies:
    - The control flow (exclusive use of CPU): as defined by the code (this lecture)
    - The memory layout (exclusive use of memory): defined by executable/code (next lecture)

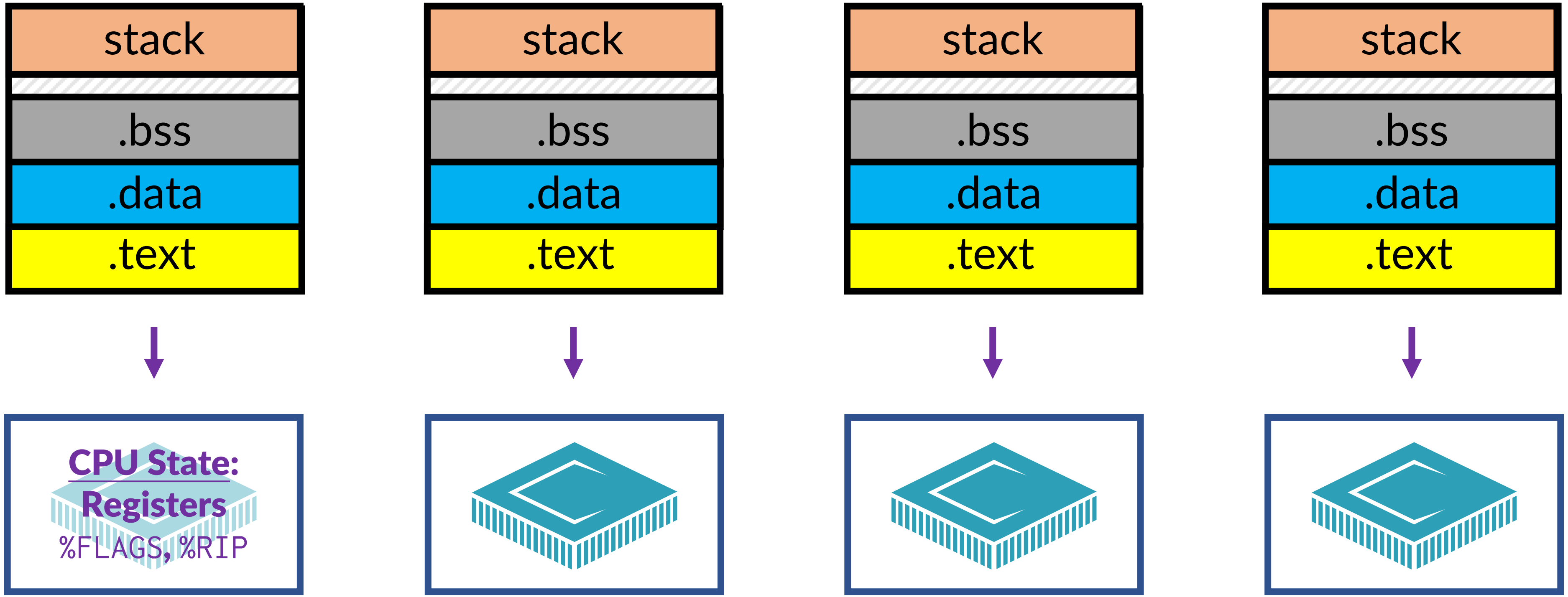
- We are focusing on the control flow, here.
  - How do we determine when a program runs? When does the lie… break down?

CPU Scheduling

Eeny Meeny Miney Moe

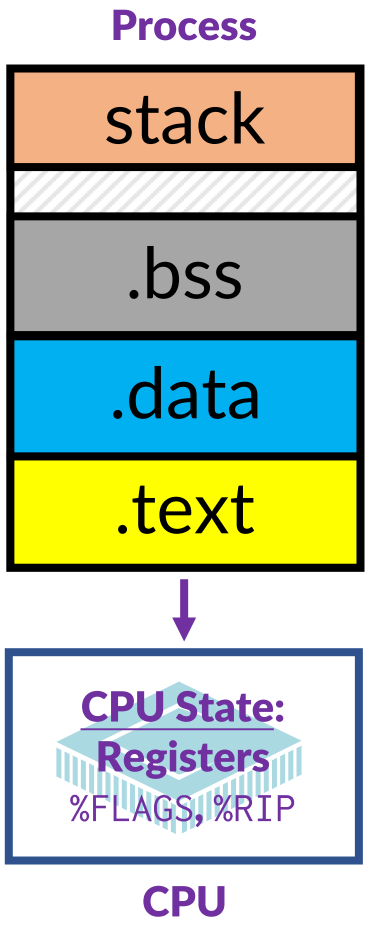
The Reality

- **Let us say that we have a machine with four separate CPUs.**
  - You could run four processes concurrently (at the same time) relatively easily.
  - What about the fifth?



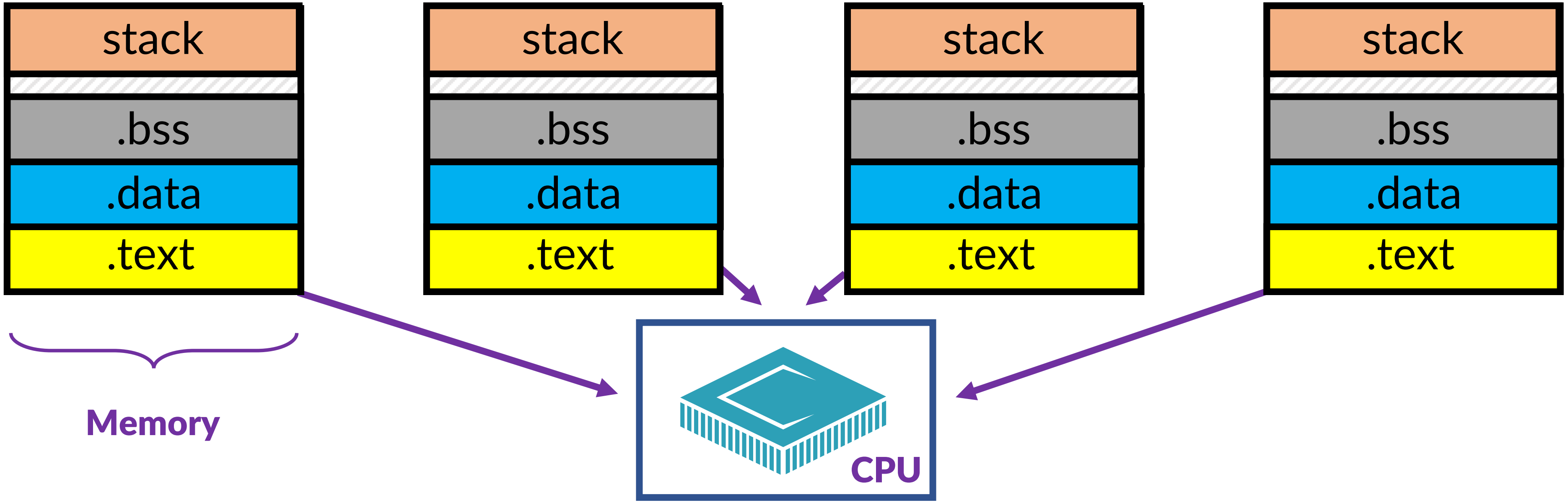
Multiplexing the CPU

- Truth be told, we often have fewer resources than needed.
  - Sharing a common resource is called **multiplexing**.
- Now, consider a machine with a single CPU.
- We often want to run something in the foreground.
  - Word processor, web browser, minesweeper… whatever.
- We still want some things running the background…
  - Music player, virus scanner, chat client.
- We need to switch from one process to another at particular times.
  - Yet… we have to keep the illusion that the program is uninterrupted…



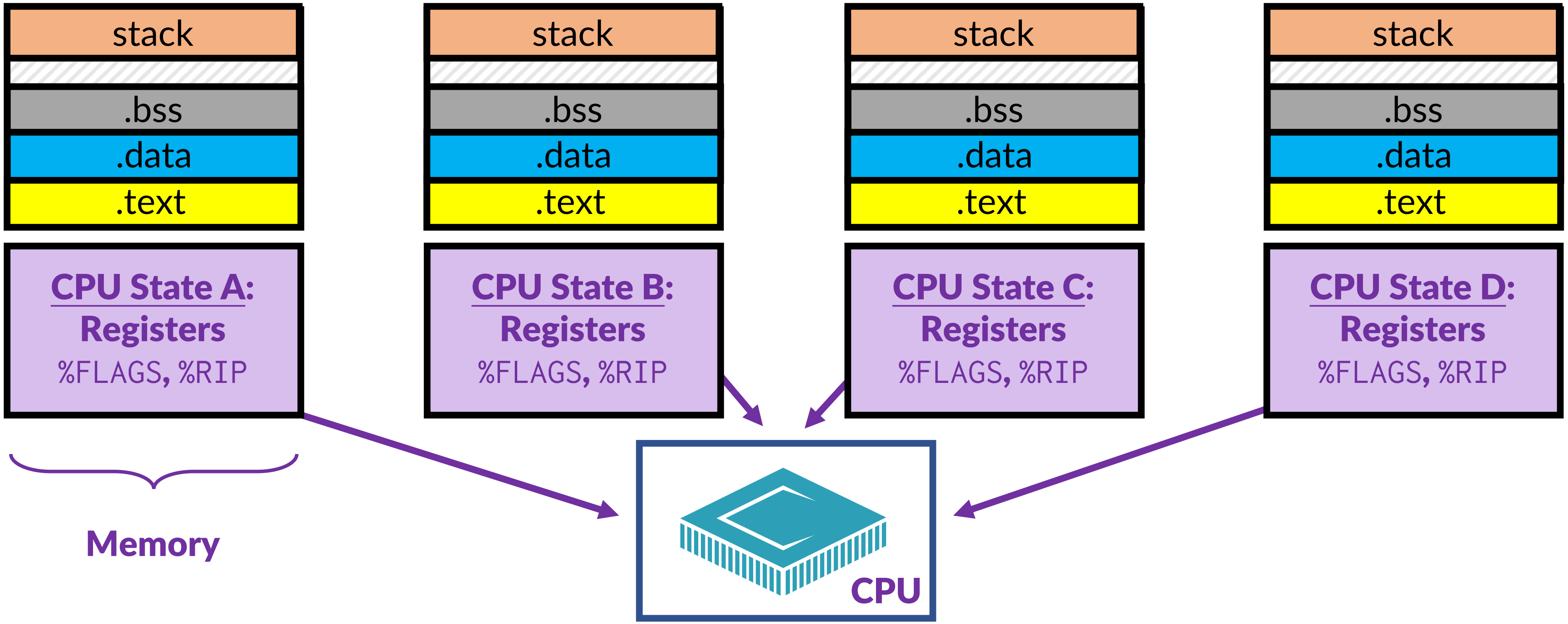
Naïve Campbell was great in The Craft (1996)

- One way is to run processes sequentially (the naïve solution)
  - When one process ends… run the next.
  - Yet that's not very *flexible*. (Stop your music player to open a PDF)
    - Humans are in the mix! We need computers to be useful to us.



The cruel passage of time

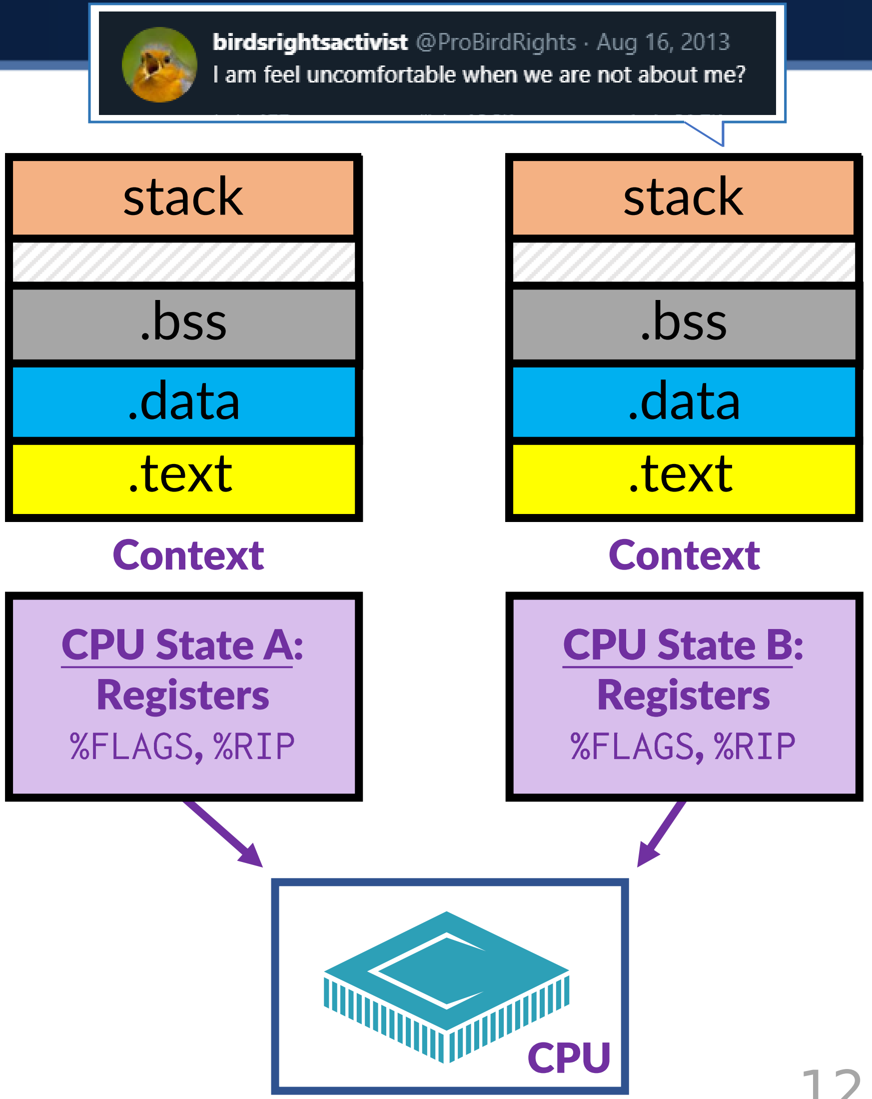
- To multiplex the CPU, we quickly switch from process to process.
- The OS retains/restores the state (**context**) of the process.
  - The OS must store this as a form of process metadata in memory.



The Context Switch

- When an Operating System goes from one process to another, it performs a **context switch**.
- This swaps out the CPU state of one process for the next one to run.

1. Store registers (including stack pointer and program counter) to memory.
2. Determine next process to run.
3. Load those registers from memory. Switch memory space. *(see next lecture: virtual memory)*
4. Jump to old program counter. Go!



A deeper dive

- When we pause a process… we store the state of registers (context)

x86-64 (gas / AT&T syntax) – Process A

```
        .data
0x1008 db: .asciz  "Hello, world!\n"

        .text

        .global _start

        _start:
          # write(2, db, 14)
0x0e33    mov      $1, %rax     # syscall 1: write
0x0e34    mov      $2, %rdi     # file 2 is stderr
0x0e35    lea      (db), %rsi   # address of 'db'
0x0e40    mov      $14, %rdx    # number of bytes
0x0e44    syscall               # invoke OS

          # exit(0)
0x0e48    mov      $60, %rax    # syscall 60: exit
0x0e49    xor      %rdi, %rdi   # return code 0
0x0e50    syscall               # invoke OS
```

Context (A)

```
%rax 0x0001
%rdi 0x0002
%rsi 0x1008
%rdx 0x0000
%rip 0x0e40
......
```

CPU State

```
%rax 0x003c
%rdi 0x0000
%rsi 0x1008
%rdx 0x000e
%rip 0x0e50
......
```

x86-64 (gas / AT&T syntax) – Process B

```
        .data
0x1008   arr: .int 1, -2, 6, -1, 11

        .global _start
        .text

        _start:
0x0e33   lea (arr), %rbx
0x0e38   mov $3, %rdi
0x0e39   lea (%rbx, %rdi, 4), %rax
0x0e44   movl (%rax), %eax

0x0e46   mov %rax, %rdi

0x0e47   mov     $60, %rax    # syscall 60: exit
0x0e48   syscall              # invoke OS
```

When is a good time to call you?

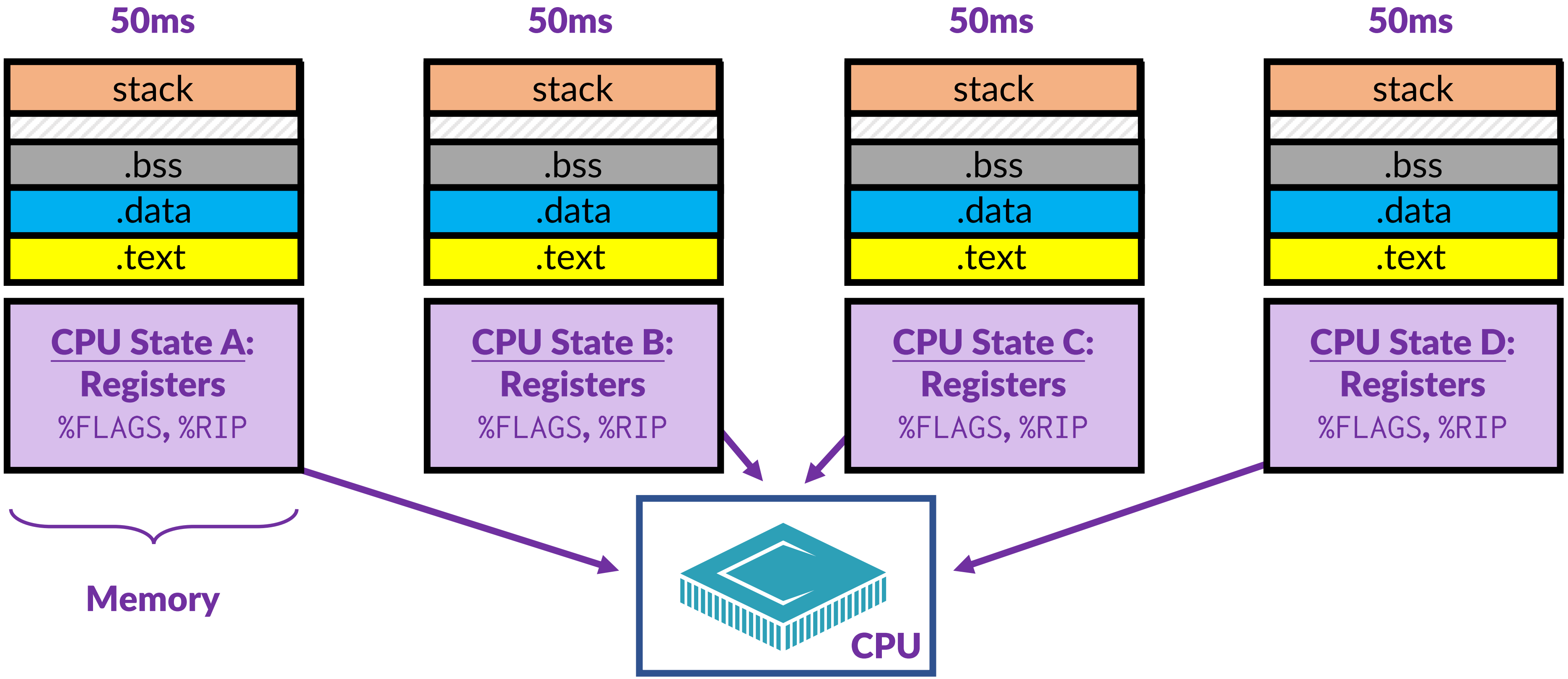
- When should a program pause and let another one go?

- When programs voluntarily pause, this is called **cooperative scheduling**.
  - They may give up control at convenient points such as system calls.

- We often do not expect this, so modern Operating Systems forcibly pause programs from time to time. Called **preemptive scheduling**.
  - Processes give up control when hardware interjects via an "interrupt"
  - How does this work?

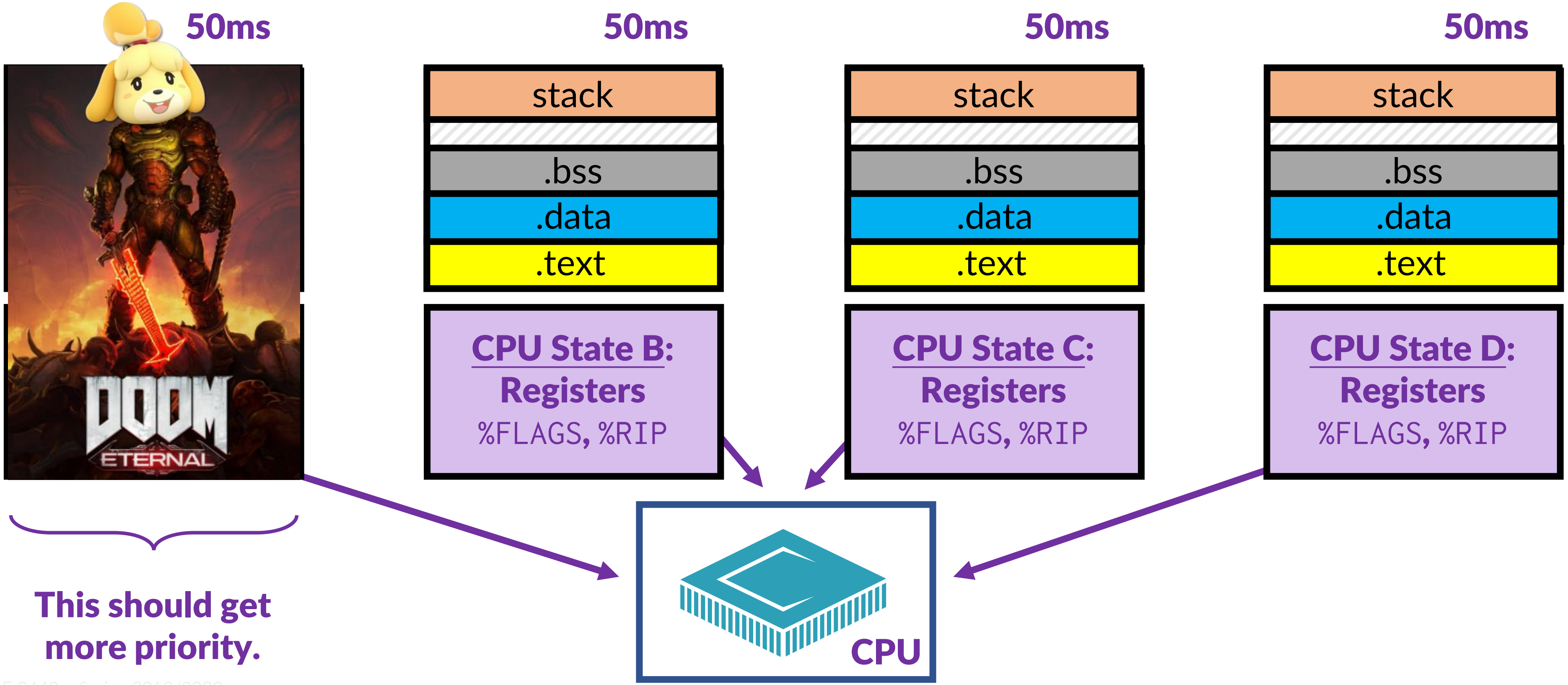
Round Robin Scheduling

- One method is to just cycle through each process each for equal time.
  - This is an element of "fairness" … each gets equal stake.



Problems with “fairness”

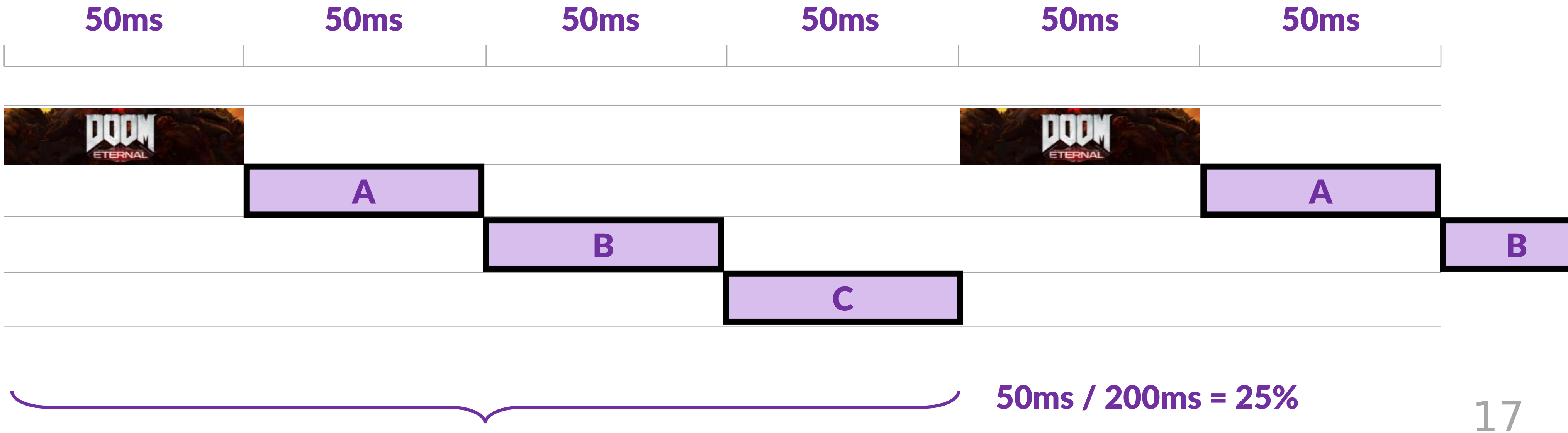
- Let’s say I want to play DOOM Eternal
  - In round-robin, I give the video game 25% of my resources.



I have priorities!

- Round-Robin schedulers are fair; then we tweak to meet expectations.
  - How might we add a sense of "priority" to the scheduler?

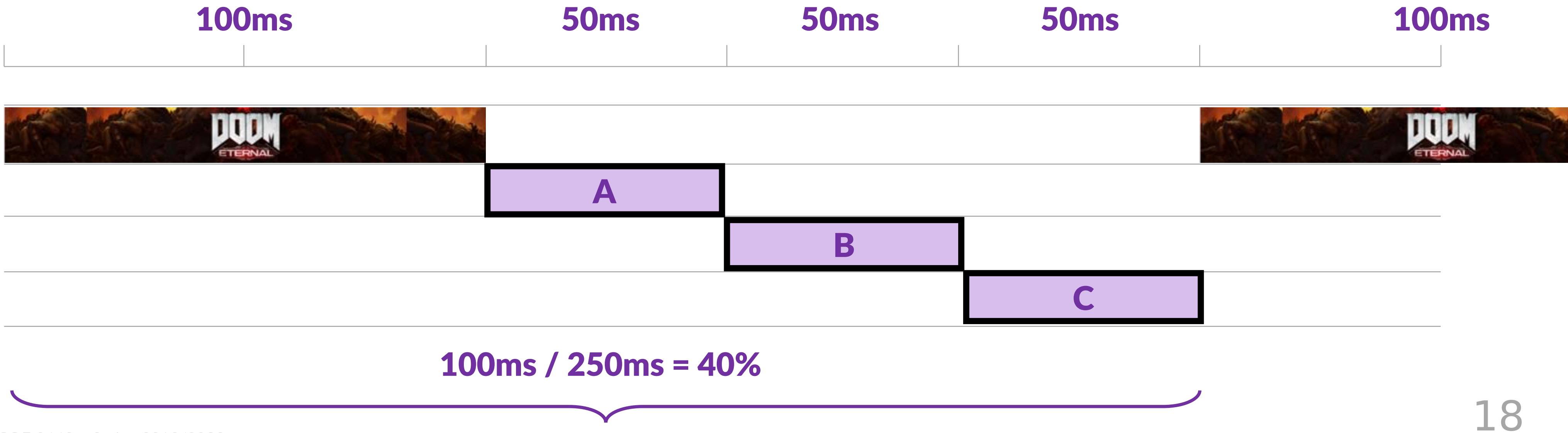
- Let's look at a visualization of how processes are currently scheduled with a round-robin scheme: (Doom gets only 25% of resources!) ☹



I have priorities!

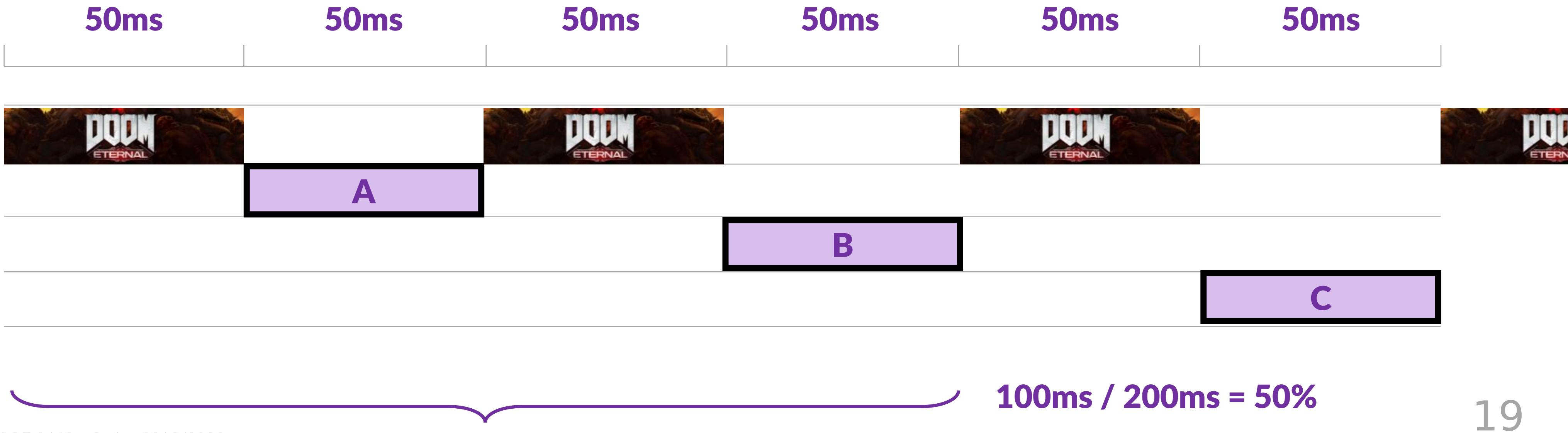
- Round-Robin schedulers are fair; then we tweak to meet expectations.
  - How might we add a sense of “priority” to the scheduler?

- **We could give some tasks a longer quantum.**
  - A **quantum** is the amount of time a task is guaranteed to run.



I have priorities!

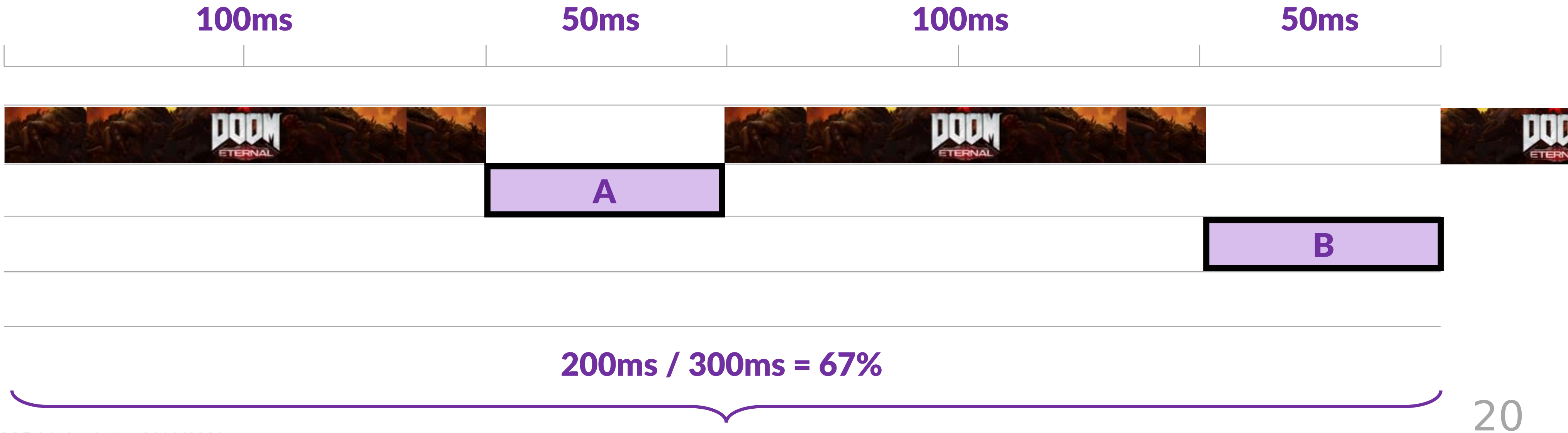
- Round-Robin schedulers are fair; then we tweak to meet expectations.
  - How might we add a sense of "priority" to the scheduler?
- We could increase the chance a specific task is scheduled.
  - Round-robin + priority: two queues, switch back and forth and round-robin within them.



I have priorities!

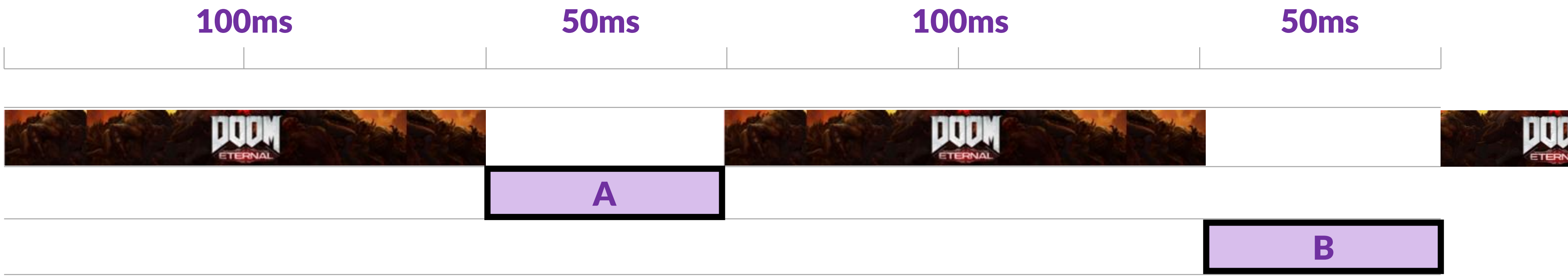
- Round-Robin schedulers are fair; then we tweak to meet expectations.
  - How might we add a sense of "priority" to the scheduler?

- **We can then always do some sort of combination.**
  - Hybrid approaches do seem very alluring. Hmm. The power of trade-offs.

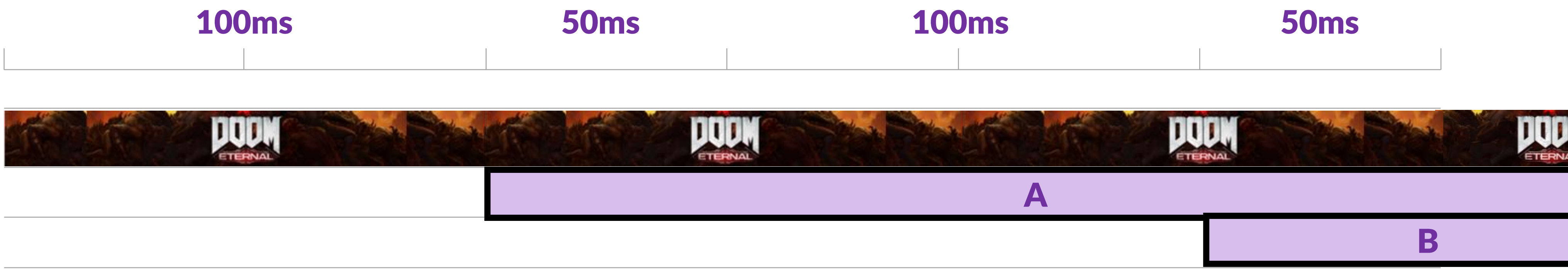


Ideal circumstances: Human perception

- **The reality:** (very quickly switching)



- **The illusion:** (an ideal: perceived concurrency… no delay noticed)



There is no optimal.

- **Like many of the topics in this course, there is no possible "best".**
  - That is, there is *no way to perfectly schedule general processes*.
- **Consider: It would be very lovely to schedule a process that handles some user input, like a button press or a network request.**
  - Perfect situation: the OS schedules the task that handles the button immediately before the button is pressed. What luck!
- **However: You do not know when that button will be pressed.**
  - Maybe it is a sensor, like for detecting a fire!
    - FIRE SEEMS IMPORTANT!! ... and yet.
- **Moral of the story: humans being users make things very hard.**

Again, it is not magic.

- But… wait… how does hardware *stop* a program?
  - For instance, when the quantum is up, how does the OS get control and perform the context switch?

- Ah, the hardware has support for “being rude” which is called an interrupt.
  - A programmable mechanism for asynchronously calling a function when a particular type of error or signal is noticed.

- Let’s take a look.

Interrupts

It's rude... but necessary.

How rude

- An **interrupt** is an exceptional state that diverts execution from its normal flow.
  - When issued by hardware, sometimes referred to as a **hardware exception**
    - For instance, a hardware timer or external event caused by a sensor.
  - When caused by a user process, sometimes referred to as a **software trap**
    - Divide-by-zero error, some floating-point exceptions, system calls.

- We have seen these before!
  - System calls are a type of interrupt (software trap).
  - This is an intentional interrupt caused by a specific program instruction.
    - The program is “interrupted” while the OS performs a task.

- We have also encountered them in our failures.
  - Segmentation / Protection / Page Faults are also interrupts. (trap? exception?) 😨
  - These are (usually) unintentional interrupts caused by a generic instruction.

Here are some typical UNIX/Linux system calls:

Number	Name	Description
`0x00`	read	Reads bytes from an open file.
`0x01`	write	Writes bytes to an open file.
`0x02`	open	Opens a file and returns the file handle.
`0x03`	close	Closes an open file.
`0x04`	stat	Returns metadata about a file.
`0x57`	fork	Spawns a copy of the current process.
`0x59`	execve	Loads and then executes a program.

System calls

- System calls: predictable, intentional interrupts at specific instructions.
  - Interrupts occurring at specific instructions are **synchronous interrupts**.

- In x86-64, the program pauses at a `syscall` instruction, then resumes at the following instruction when the OS finishes the task
  - (… and the OS calls the `sysret` instruction)

- Let's take a deeper look.

Hello, Hello World

x86-64 (gas / AT&T syntax) - Application

```
.data
db: .asciz  "Hello, world!\n"

.text

.global _start

_start:
  # write(2, db, 14)
  mov     $1, %rax    # system call 1 is write
  mov     $2, %rdi    # file handle 2 is stderr
  lea     (db), %rsi  # address of string
  mov     $14, %rdx   # number of bytes
  syscall             # invoke OS to print

  # exit(0)
  mov     $60, %rax   # system call 60 is exit
  xor     %rdi, %rdi  # we want return code 0
  syscall             # invoke OS to exit
```

A jump to the kernel

x86-64 (gas / AT&T syntax) - Kernel (main OS program)

```
open: # (syscall 0)
  # Open implementation
  # ... use %rdi, rsi, etc
  retq

write: # (syscall 1)
  # Write implementation
  # ... use %rdi, rsi, etc
  retq

# system call jump table (array of function pointers)
syscalls: .word open, write # ... etc

syscall_enter:
  # Preserve CPU context
  call save_cpu
  lea  (syscalls), %rbx
  lea  (%rbx, %rax, 8), %rax
  call *(%rax) # call syscalls[old rax]
syscall_exit:
  # Restore CPU context
  call restore_cpu
  sysret
```

Pre-registered to be called on syscall

Saves CPU state

Performs action

Restores state

Returns to process

Tick tock tick tock merrily sings the clock

- A hardware timer can preempt (forcibly pause) a program at any time.
  - Interrupts that occur at any instruction are **asynchronous interrupts**.

- In a preemptive operating system, a hardware timer is used to give a maximum bound to how long a process runs.
  - Your operating system programs the timer such that it sends a signal at a regular interval.
  - Your operating system has a function that is called when such a signal is read.
  - That function will respond by invoking the scheduler and pausing the current task and resuming or starting another.

- Let's look at the basic procedure an OS uses to program an interrupt.

Programming interruption

- On most hardware, there is a programmable table somewhere in memory that, when written to, defines where code exist to handle each interrupt.
- Every possible interrupt is given a number. Segmentation faults might be interrupt 10. Timers might be interrupt 0. Et cetera.
- When an interrupt occurs, based on its interrupt number, the corresponding entry in a lookup table called an **interrupt vector table** or an **interrupt descriptor table** would be used to determine where in the kernel to jump.

The Interrupt Table

#	Value	Description
00	`0xffff8010`	Divide by zero
01	`0xffff8014`	Overflow
02	`0xffff8020`	Double Fault
03	`0xffff8040`	General Protection Fault
04	`0xffff8066`	Page Fault
07	`0xffff8080`	Stack Fault
06	`0xffff80a8`	Alignment Error
...	...	...
32	`0xffff81e8`	Timer Signal
33	`0xffff81fc`	Network Device Signal
34	`0xffff8218`	Audio Device Signal

- The interrupt table is a simple table.
- Fun Fact: It is often located at address 0x0 in memory!
  - So, operating system kernels can't exactly always treat zero as an invalid address…
- When a process triggers a listed interrupt or external hardware sends a signal to the interrupt controller…
  - the CPU jumps to the given address.

Ah! There art thee ol' interrupt!

- Let's take a look at interrupt handling…

x86-64 (gas / AT&T syntax) – Process B

```
        .data
0x1008    arr: .int 1, -2, 6, -1, 11

        .global _start
        .text

        _start:
0x0e33    lea (arr), %rbx
0x0e38    mov $3, %rdi
0x0e39    lea (%rbx, %rdi, 4), %rax
0x0e44    movl (%rax), %eax

0x0e46    mov %rax, %rdi

0x0e47    mov     $60, %rax    # syscall 60: exit
0x0e48    syscall              # invoke OS
```

Context (A)

```
%rax
%rdi
%rsi
%rdx
%rip
......
```

CPU State

```
%rax 0x0001
%rdi 0x0002
%rsi 0x1008
%rdx 0x0000
%rip 0x0e40
......
```

x86-64 (gas / AT&T syntax) – Kernel

```
              _timer_handler:
0xffff81e8    call save_cpu
0xffff81ed    call schedule
0xffff81ee    call restore_cpu
0xffff81f3    iret
```

- An interrupt is the likely cause of our prior interruption.
- The interrupt handler is the code that handles context switching and scheduling

Overview

- Interrupts can be categorized in several ways:
  - They can occur outside of our program: **hardware exceptions**
  - They can occur on an instruction in our program: **software trap**
  - They can occur at any time: **asynchronous interrupts**
  - They can occur at specific times: **synchronous interrupts**
- Interrupts are what allow operating systems to function!
  - When you press a key on your keyboard.
  - When you receive a packet on the network.
  - When your sound card wants the next second of audio.
  - When you divide by zero…
    - To then mercilessly murder your process.