Week 2 - Wednesday

COMP 3400

Last time

- What did we talk about last time?
- Finished system architectures
- State models
- Implementing state models in C
- Sequence models

Questions?

Assignment 1

Assignment 2

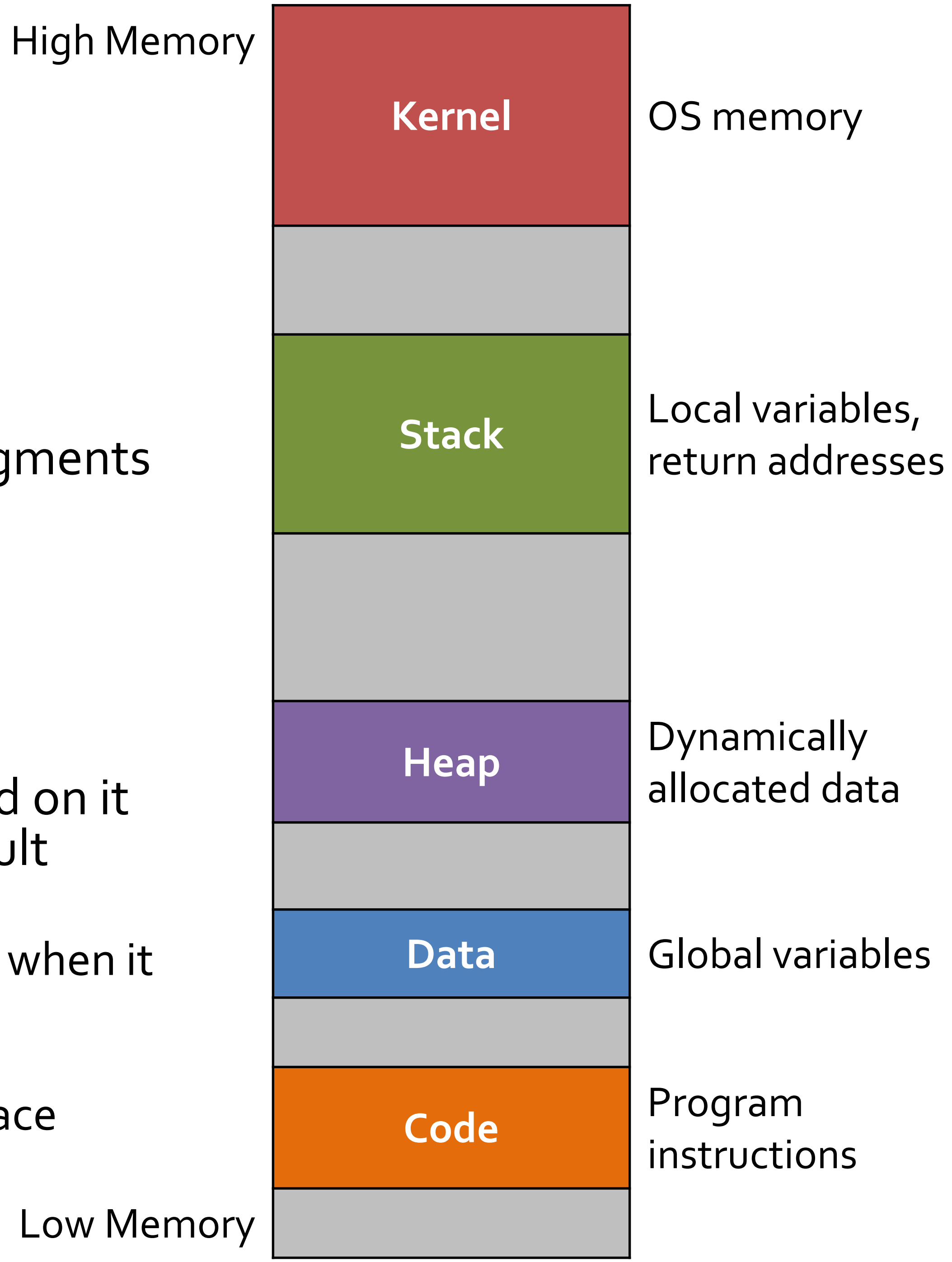
Processes

Processes

- A **program** is an implementation of an algorithm in a programming language
  - A list of instructions for the computer
- A **process** is program being executed
  - Usually, processes are different programs
  - But it's not unusual to have several processes running at the same time that are the same program
- Running a program creates a new process

Virtual memory

- Every process has its own virtual memory
  - Addresses from 0 up to 2^{32} or 2^{64} bytes
- Each instance of virtual memory is organized into segments
  - Code
  - Data
  - Heap
  - Stack
  - Kernel
- Each segment has certain kinds of operations allowed on it
- Do illegal operations, and you get a segmentation fault
- As functions get called, the stack grows downward
  - Call too many functions, and you'll get a stack overflow when it gets too big
- Depending on the system, the heap can grow too
  - **`malloc()`** returns NULL when you run out of heap space



64-bit Linux memory layout

- Different operating systems have different layouts, but the fundamental ideas are the same
- The Linux machines in this lab use 64-bit processors with 64-bit versions of Ubuntu
- But 64-bit stuff is confusing
  - They're still working out where the eventual standard will be
  - 64-bit addressing allows 16,777,216 terabytes of memory to be addressed (which is far beyond what anyone needs)
- Current implementations only use 48 bits
  - User space (text up through stack) gets low 128 terabytes
  - Kernel space gets the high 128 terabytes
- The starting points for many segments used to be fixed, but Linux now randomizes them a little for security

Why is it *virtual* memory?

- Addresses in one process have nothing to do with addresses in another
- The OS maps the virtual addresses to physical addresses
  - Transparently!
  - Each process has no idea what the location of, for example, its virtual address **`0x0432A8F8`** is in physical memory
- Benefits:
  - **Security:** One process cannot (normally) interfere with the memory inside another process
  - **Bookkeeping:** The OS only gives each process what it needs and can temporarily store parts of a process's memory on disk to make more space

Operating systems

- OS sometimes means the entire operating system, including utilities, window managers, and lots of other stuff
- Sometimes OS means just the kernel
- The kernel is the part of the OS that does deep stuff:
  - Scheduling processes
  - Accessing devices
  - Managing memory
- Some operations can only be done in **kernel mode**, the mode that the kernel runs in
- Normal programs run in **user mode**

More on the kernel

- It's a special program
- Part of every process's virtual memory is used by the kernel
- The kernel is reactive
  - When a normal program needs something that only the kernel can do, it asks for it
  - Then, control switches to the kernel
  - It runs as if it's part of the currently running program
  - Then, it gives control back

Multiprogramming

Multiprogramming

- When you're using your computer, you might be doing a lot of things at the same time:
  - Browsing the Internet
  - Chatting on Discord
  - Coding
  - Streaming video and audio
- It *feels* like all of these programs are running at the same time
  - This "running at the same time" feeling is called **concurrency**
- Because your computer has multiple cores, several of them can be running at the same time
- But your computer has a small number of cores and **a lot** of programs to run

Kickin' it old school

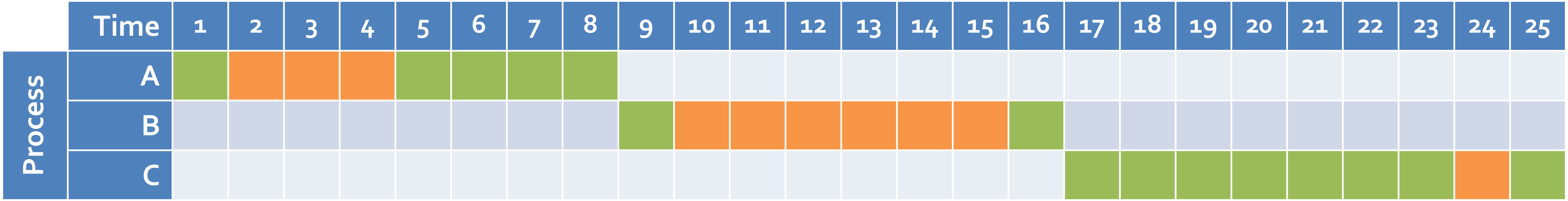
- Let's go back to 2000 (and earlier) when virtually all desktop and laptop computers were single core machines
- In order for it to *seem* like all of those programs were running at the same time, the OS actually gives each process some time to run before switching to another process
- Because computers were fast (even in 2000), it seemed like all those programs were running at the same time

Kickin' it even older school

- Back in the dim, dark past, programmers submitted their programs to be run
  - Computers were large and expensive
  - Programs were submitted on cards
  - Jobs were run in batches
- If you had a syntax error in your program, you had to resubmit your deck of cards the next day

Problems with naïve batch processing

- One approach to batch processing is running Process A until it's done, then Process B, then Process C
- The problem is that programs do I/O
  - I/O is slow
  - The CPU isn't in use while waiting for I/O
- Consider the following example:
  - Green is computation
  - Orange is I/O
- Nothing is getting done during I/O!



CPU utilization

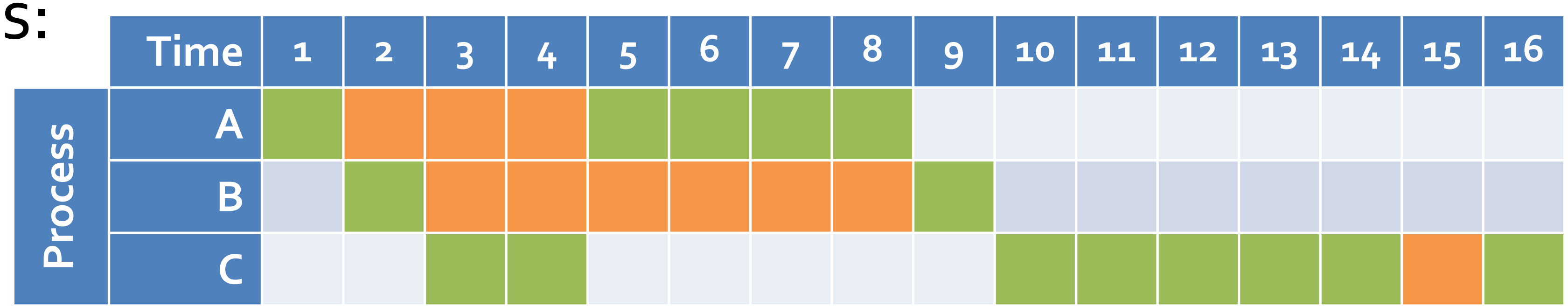
- **CPU utilization** is the percentage of time that the CPU is being used
- On the previous slide, process time broke down like this:

Process	CPU Time	I/O Time
A	5	3
B	2	6
C	8	1
Total	15	10

- Consequently, we have a CPU utilization of 15/25 = 60%
- It took 25 time units to finish all jobs

Multiprogramming

- With true multiprogramming, you have more than one process loaded into memory
- Then, when one process is waiting on I/O, we can start running another
- Using multiprogramming, we could run Processes A, B, and C as follows:



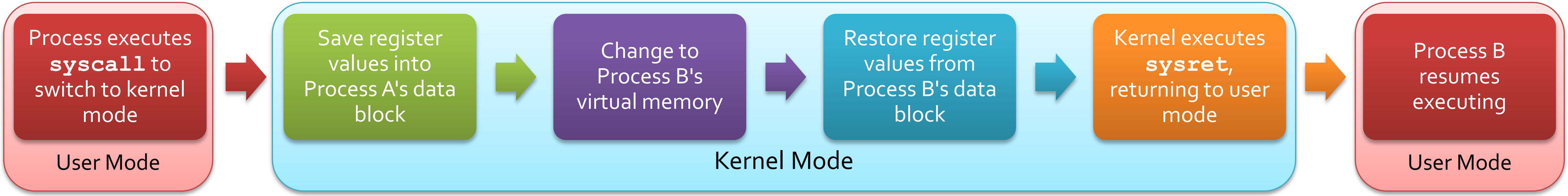
- Doing so gives us a CPU utilization of 15/16 = 93.75% and only 16 time units to finish the work

Types of multiprogramming

- **Preemptive multitasking:**
  - Processes get a maximum amount of time to run called a **quantum**
  - If the process starts doing I/O, the OS switches to another process
  - Otherwise, the OS switches when the process runs out of time
  - There's research about the ideal length of a quantum
- **Cooperative multitasking:**
  - Processes run until they do some I/O or voluntarily give up control
- Cooperative is good because it's simple and can have lower overhead
- Unfortunately, the problem of processes that don't give up control means that most modern systems use preemptive multitasking

Context switches

- A context switch happens when the running process changes
  - The virtual memory of one process changes to another
  - The kernel memory stays the same
- The **scheduler** in the OS decided which process runs next



- Because memory has to get saved and restored, cache is invalidated, and there's a switch from user mode to kernel mode and back, context switches have **overhead** that slows things down

Ticket Out the Door

Upcoming

Next time…

- Kernel mechanics
- System calls
- Processes

Reminders

- **Finish Assignment 1**
  - Due Friday by midnight!
- Look over Assignment 2
- Read sections 2.3, 2.4, and 2.5