

CS 31: Intro to Systems C Programming

L18: Operating Systems

Vasanta Chaganti & Kevin Webb

Swarthmore College

Nov 14, 2023

Reading Quiz

OS Big Picture Goals

- OS is a layer of code between user programs and hardware.
- Goal: Make life easier for users and programmers.
- How can the OS do that?

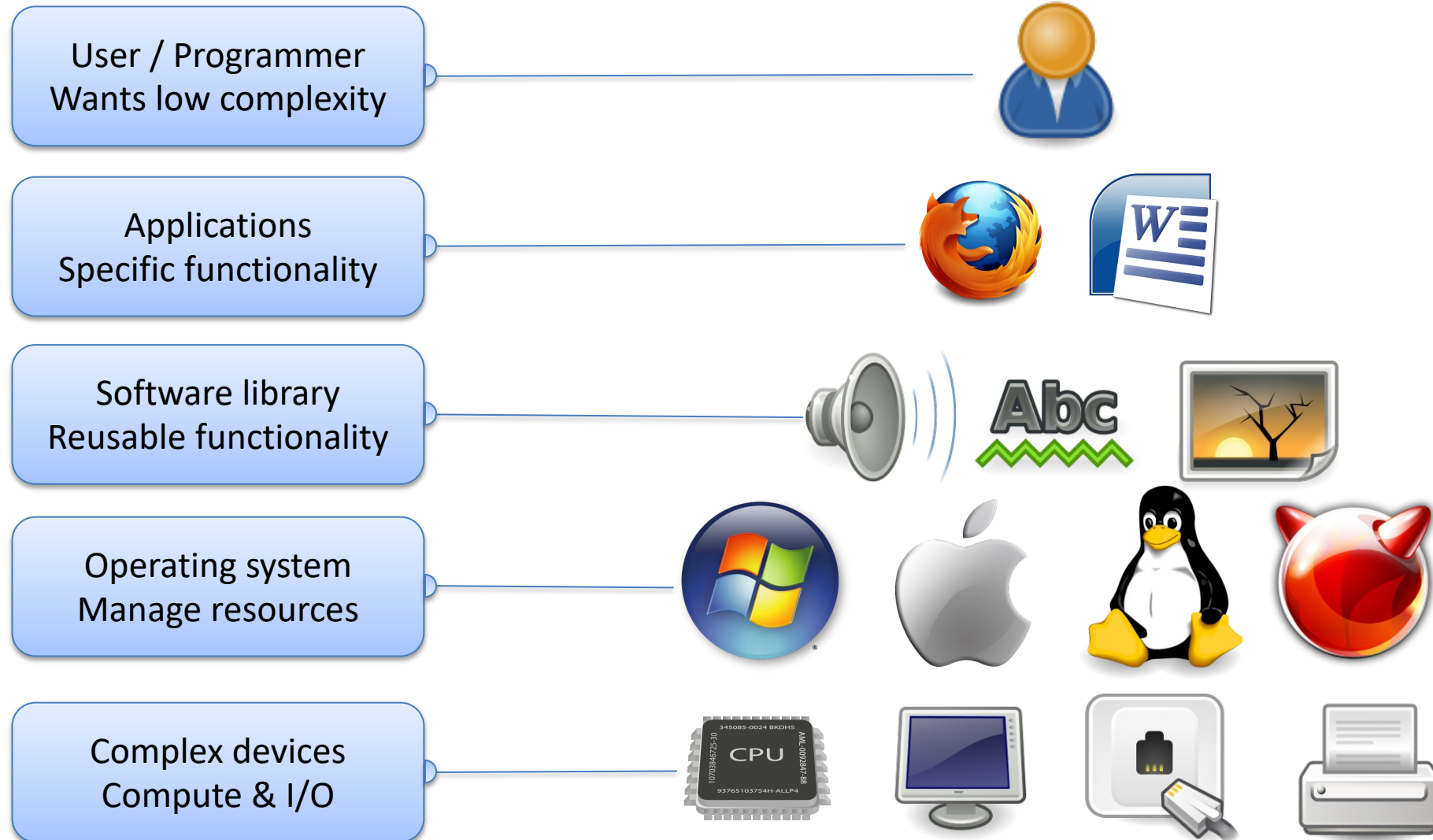
Key OS Responsibilities

1. Simplifying abstractions for programs
2. Resource allocation and/or sharing
3. Hardware gatekeeping and protection

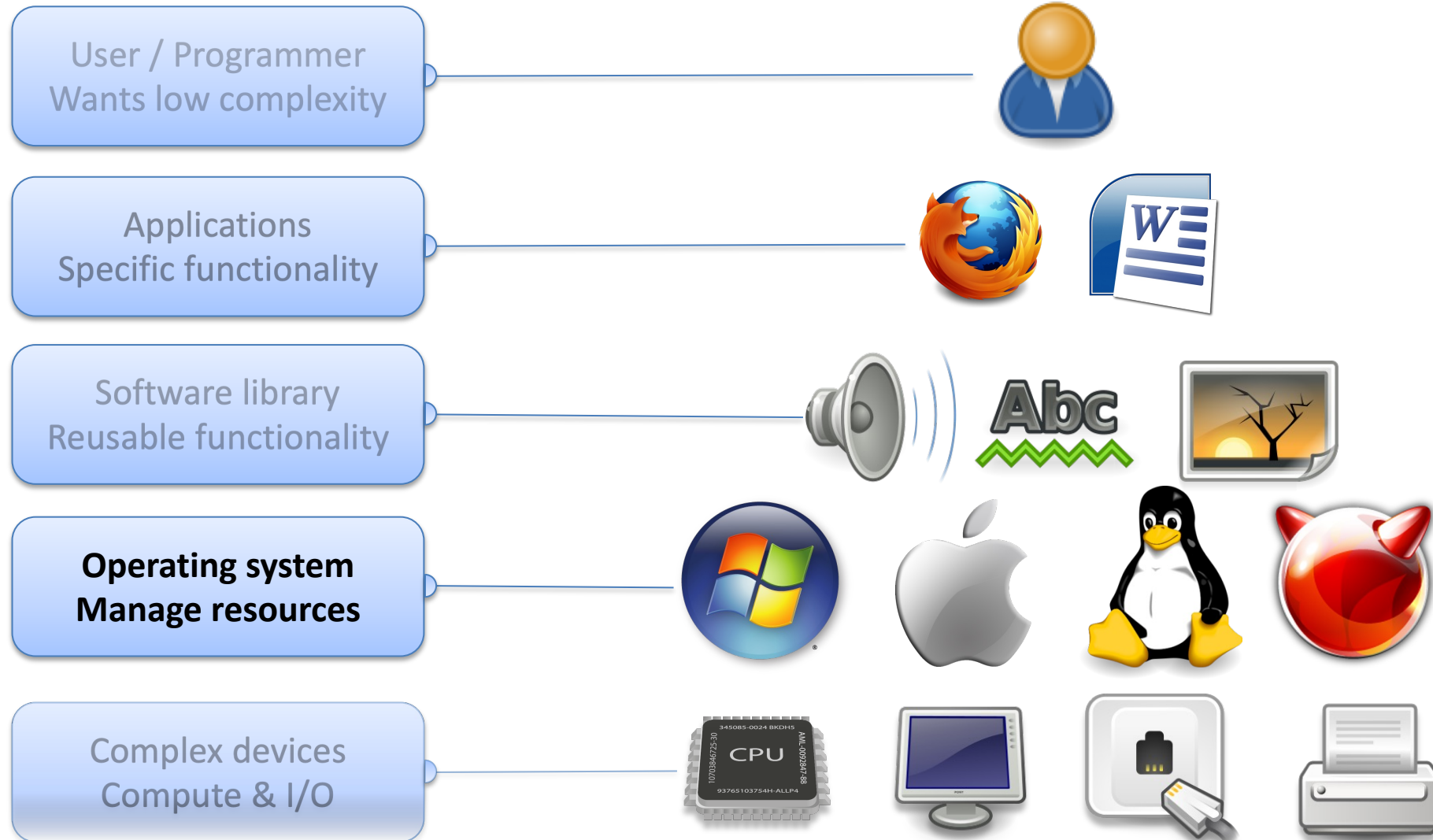
OS Big Picture Goals

- OS is a layer of code between user programs and hardware.
- Goal: Make life easier for users and programmers.
- How can the OS do that?

Abstraction



Abstraction



If you were asked to design a layer between user programs and the hardware, what might your layer provide?

- What sort of services might the programs you've written need?
- (Discuss with your neighbors.)

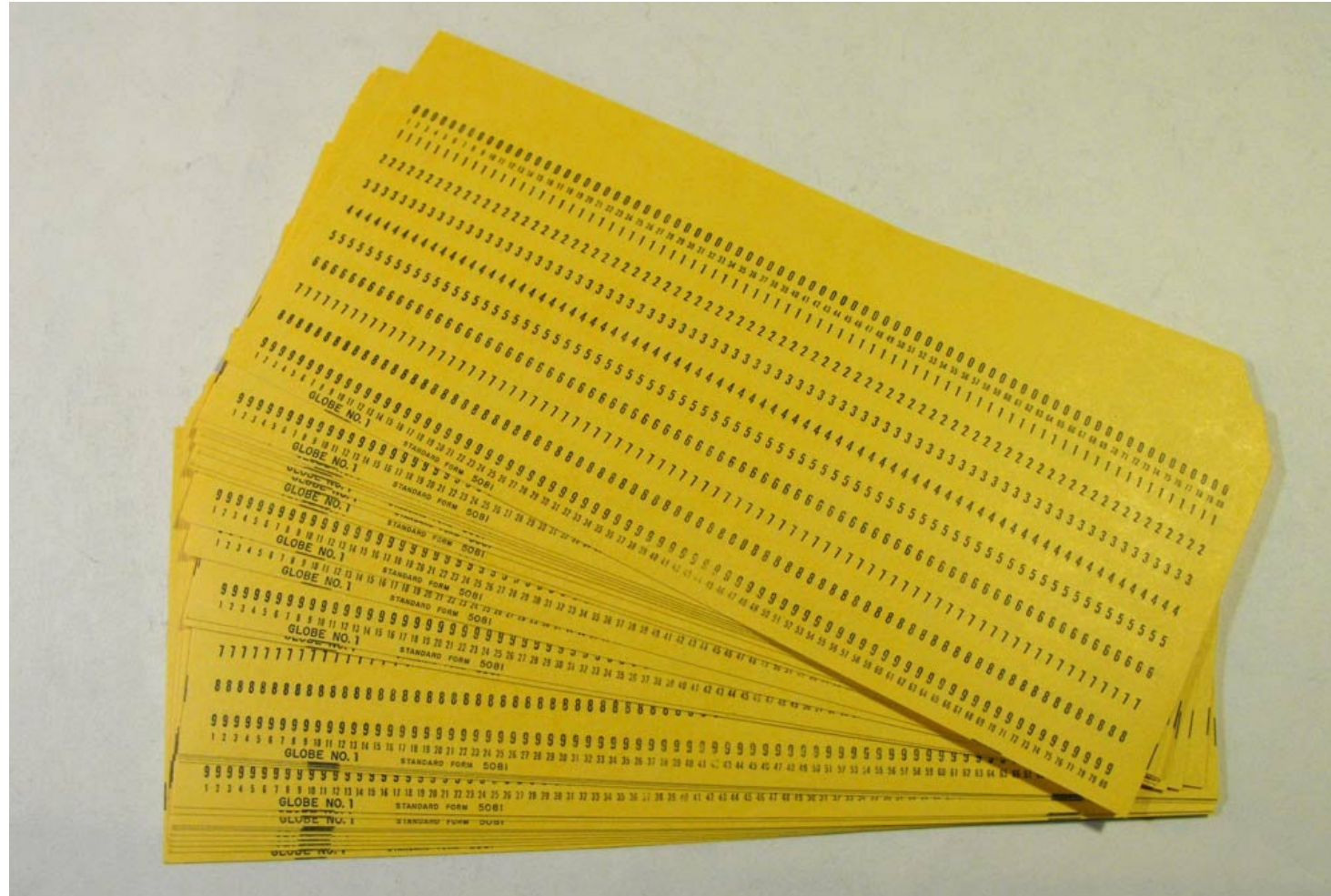
Key OS Responsibilities

1. Simplifying abstractions for programs
2. Resource allocation and/or sharing
3. Hardware gatekeeping and protection

OS: Turn undesirable into desirable

- Turn undesirable inconveniences: reality
 - Complexity of hardware
 - Single processor
 - Limited memory

Before Operating Systems



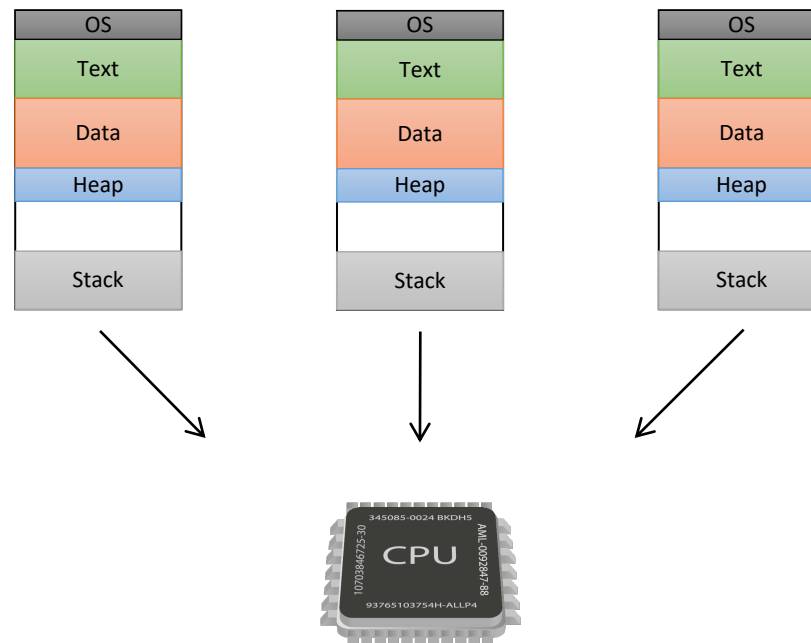
- One program executed at a time...

Why is it not ideal to have only a single program available to the hardware?

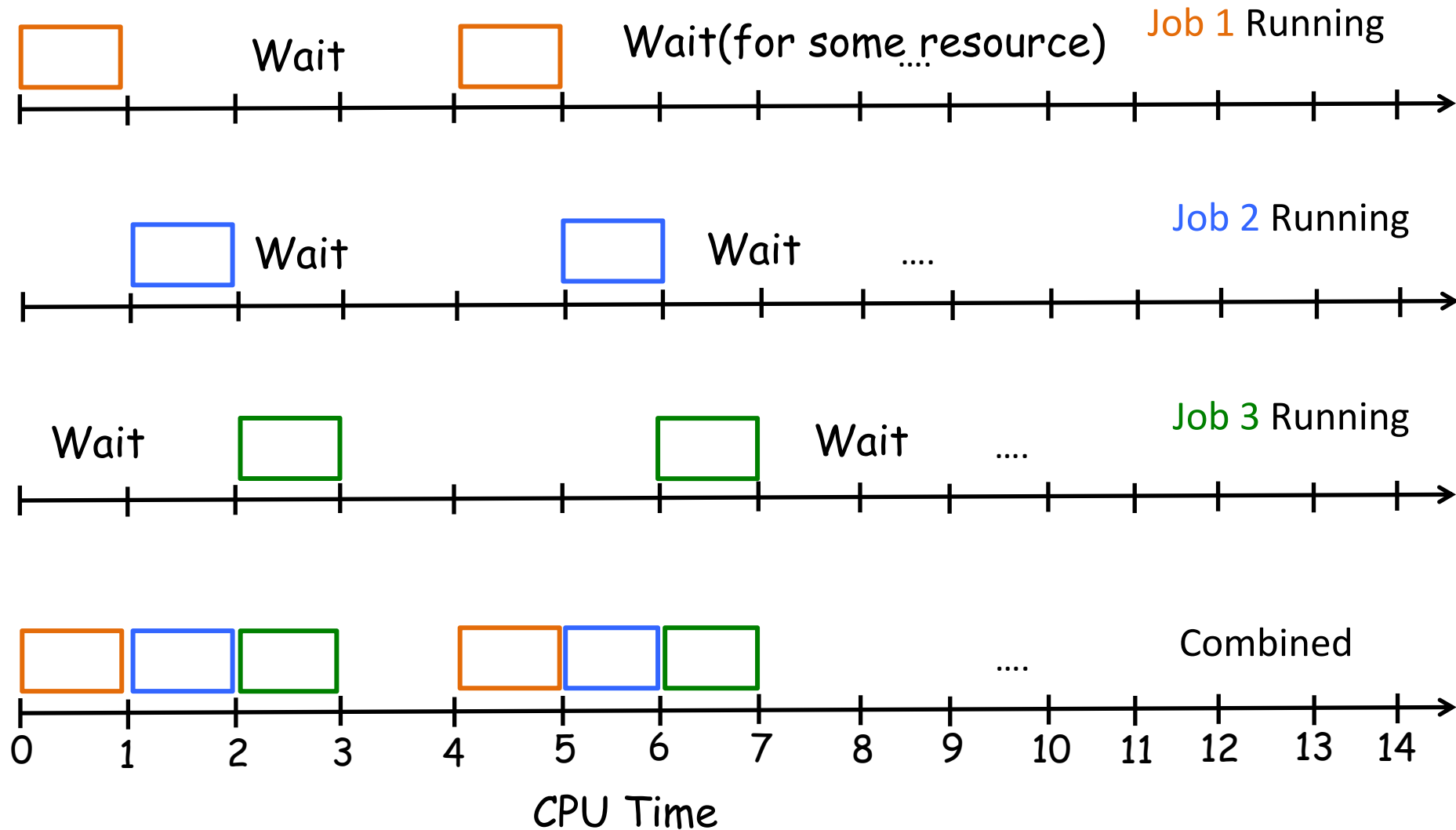
- A. The hardware might run out of work to do.
- B. The hardware won't execute as quickly.
- C. The hardware's resources won't be used as efficiently.
- D. Some other reason(s). (What?)

Today: Multiprogramming

- Multiprogramming: have multiple programs available to the machine, even if you only have one CPU core that can execute them.



Multiprogramming on one core



How many programs do you think are running on a typical desktop computer?

A. 1-10

B. 20-40

C. 40-80

D. 80-160

E. 160+

Running multiple programs

More than 200 processes running on a typical desktop!

- **Benefits:** when I/O issued, CPU not needed
 - Allow another program to run
 - Requires yielding and sharing memory
- **Challenges:** what if one running program...
 - Monopolizes CPU, memory?
 - Reads/writes another's memory?
 - Uses I/O device being used by another?

OS: Turn undesirable into desirable

- Turn undesirable inconveniences: reality
 - Complexity of hardware
 - Single processor
 - Limited memory
- Into **desirable conveniences**: illusions
 - Simple, easy-to-use resources
 - Multiple/unlimited number of processors
 - Large/unlimited amount of memory

Virtualization

- Rather than exposing real hardware, introduce a “virtual”, abstract notion of the resource
- Multiple virtual processors
 - By rapidly switching CPU use
- Multiple virtual memories
 - By memory partitioning and re-addressing
- Virtualized devices
 - By simplifying interfaces, and using other resources to enhance function

We'll focus on the OS 'kernel'

- “Operating system” has many interpretations
 - E.g., all software on machine minus applications (user or even limited to 3rd party)
- Our focus is the *kernel*
 - What's necessary for everything else to work
 - Low-level resource control
 - Originally called the nucleus in the 60's

The Kernel

- All programs depend on it
 - Loads and runs them
 - Exports system calls to programs
- Works closely with hardware
 - Accesses devices
 - Responds to interrupts (hardware events)
- Allocates basic resources
 - CPU time, memory space
 - Controls I/O devices: display, keyboard, disk, network



Tron, 1982

Kernel provides common functions

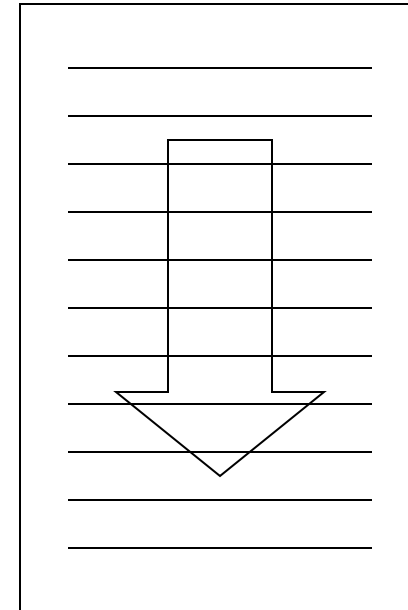
- Some functions useful to many programs
 - I/O device control
 - Memory allocation
- Place these functions in central place (kernel)
 - Called by programs ("system calls")
 - Or accessed in response to hardware events
- What should functions be?
 - How many programs should benefit?
 - Might kernel get too big?

OS Kernel

- **Big Design Issue:** How do we make the OS efficient, reliable, and extensible?
- General OS Philosophy: **The design and implementation of an OS involves a constant tradeoff between simplicity and performance.**
- As a general rule, strive for simplicity.
 - except when you have a strong reason to believe that you need to make a particular component complicated to achieve acceptable performance
 - (strong reason = simulation or evaluation study)

Main Abstraction: The Process

- Abstraction of a running program
 - “a program in execution”
- Dynamic
 - Has state, changes over time
 - Whereas a program is static
- Basic operations
 - Start/end
 - Suspend/resume



Basic Resources for Processes

- To run, process needs some basic resources:
 - CPU
 - Processing cycles (time)
 - To execute instructions
 - Memory
 - Bytes or words (space)
 - To maintain state
 - Other resources (e.g., I/O)
 - Network, disk, terminal, printer, etc.

What sort of information might the OS need to store to keep track of a running process?

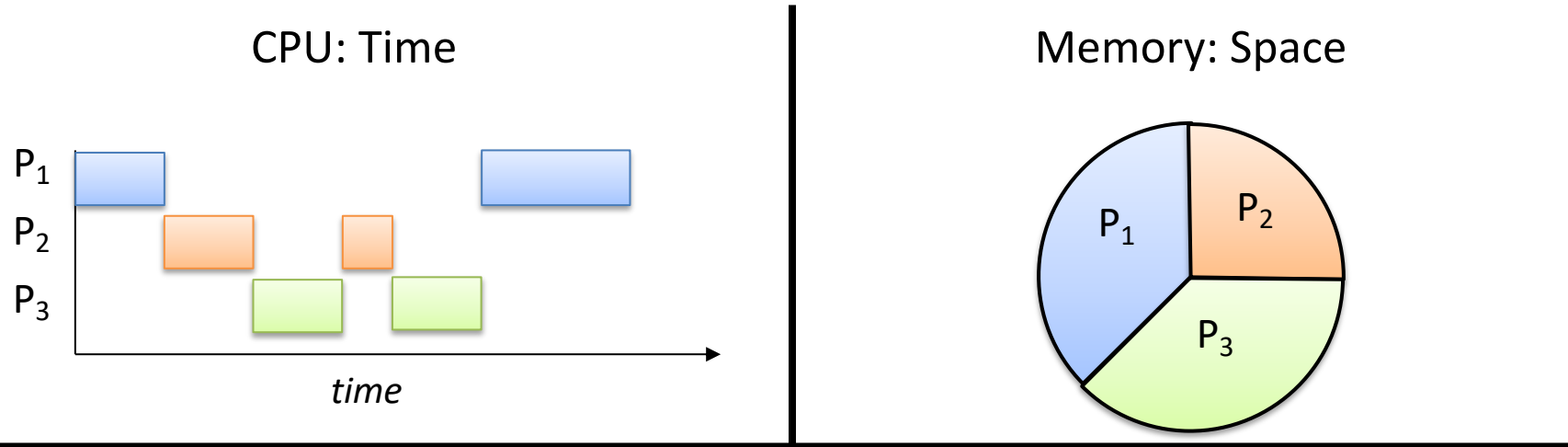
- That is, what **MUST** an OS know about a process?
- (Discuss with your neighbors.)

Machine State of a Process

- CPU or processor context
 - PC (program counter)
 - SP (stack pointer)
 - General purpose registers
- Memory
 - Code
 - Global Variables
 - Stack of activation records / frames
 - Other (registers, memory, kernel-related state)

Must keep track of these
for every running process !

Resource Sharing



Reality

- Multiple processes
- Small number of CPUs
- Finite memory

Abstraction

- Process is all alone
- Process is always running
- Process has all the memory

Resource: CPU

- Many processes, limited number of CPUs.
- Each process needs to make progress over time. Insight: processes don't know how quickly they *should* be making progress.
- **Illusion**: every process is making progress in parallel.

Timesharing: Sharing the CPUs

- **Abstraction goal**: make every process think it's running on the CPU all the time.
 - Alternatively: If a process was removed from the CPU and then given it back, it shouldn't be able to tell
- **Reality**: put a process on CPU, let it run for a short time (~10 ms), switch to another, ... ("context switching")

How is Timesharing Implemented?

- Kernel keeps track of progress of each process
- Characterizes state of process's progress
 - **Running**: actually making progress, using CPU
 - **Ready**: able to make progress, but not using CPU
 - **Blocked**: not able to make progress, can't use CPU
- Kernel selects a ready process, lets it run
 - Eventually, the kernel gets back control
 - Selects another ready process to run, ...

Multiprogramming

- Given a running process
 - At some point, it needs a resource, e.g., I/O device
 - If resource is busy (or slow), process can't proceed
 - “Voluntarily” gives up CPU to another process
- Mechanism: Context switching

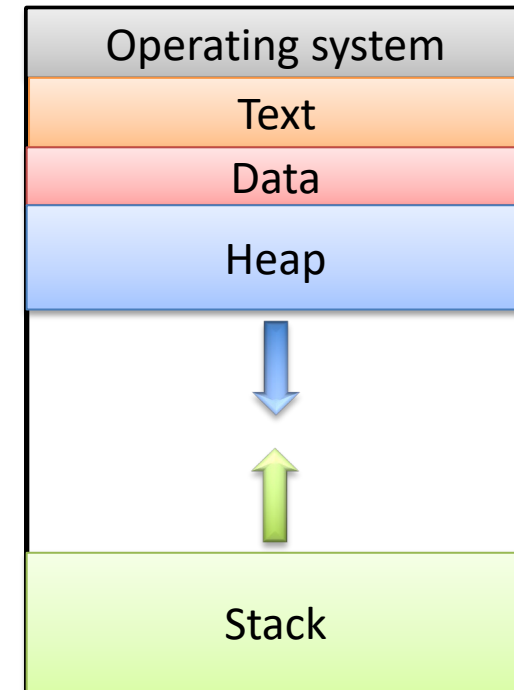
Time Sharing / Multiprogramming

- Given a running process
 - At some point, it needs a resource, e.g., I/O device
 - If resource is busy (or slow), process can't proceed
 - “Voluntarily” gives up CPU to another process
- Mechanism: Context switching
- Policy: CPU scheduling

Resource: Memory

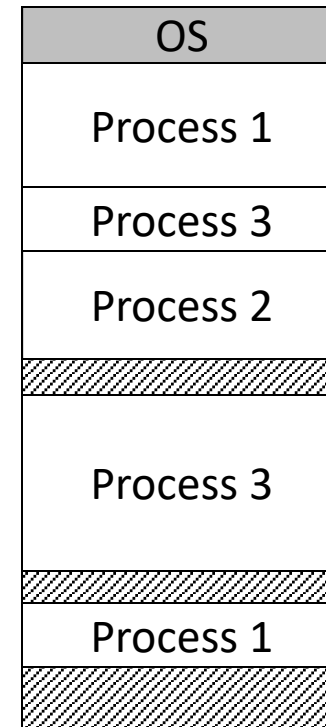
Abstraction goal: make every process think it has the same memory layout.

- MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.



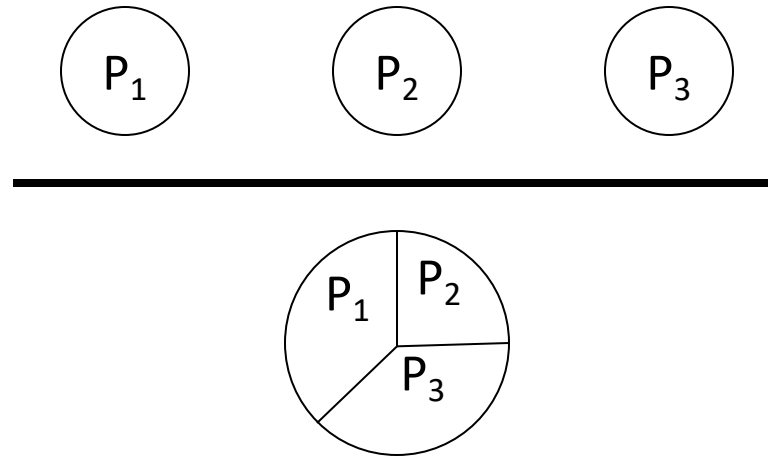
Memory

- **Abstraction goal:** make every process think it has the same memory layout.
 - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.
- **Reality:** there's only so much memory to go around
 - no two processes should use the same (physical) memory addresses (unless they're sharing).



OS (with help from hardware) will keep track of who's using each memory region.

Virtual Memory: Sharing Storage



- Like CPU cache, memory is a cache for disk.
- Processes never need to know where their memory truly is, OS translates virtual addresses into physical addresses for them.

Kernel Execution

- Great, the OS is going to somehow give us these nice abstractions.
- So...how / when should the kernel execute to make all this stuff happen?

The operating system kernel...

- A. Executes as a process.
- B. Is always executing, in support of other processes.
- C. Should execute as little as possible.
- D. More than one of the above. (Which ones?)
- E. None of the above.

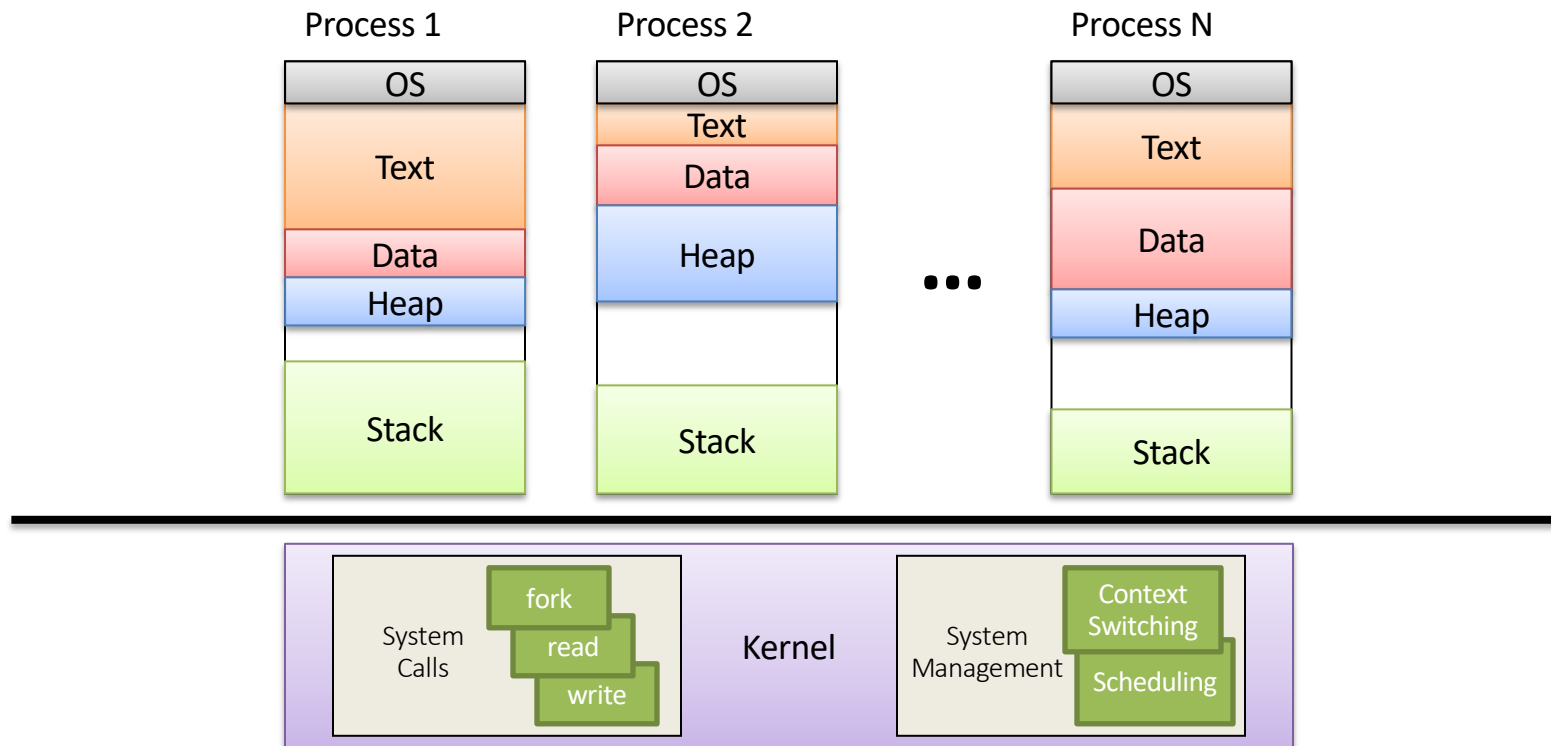
Process vs. Kernel

- Is the kernel itself a process?
 - No, it supports processes and devices
- OS only runs when necessary...
 - as an extension of a process making system call
 - in response to a device issuing an interrupt

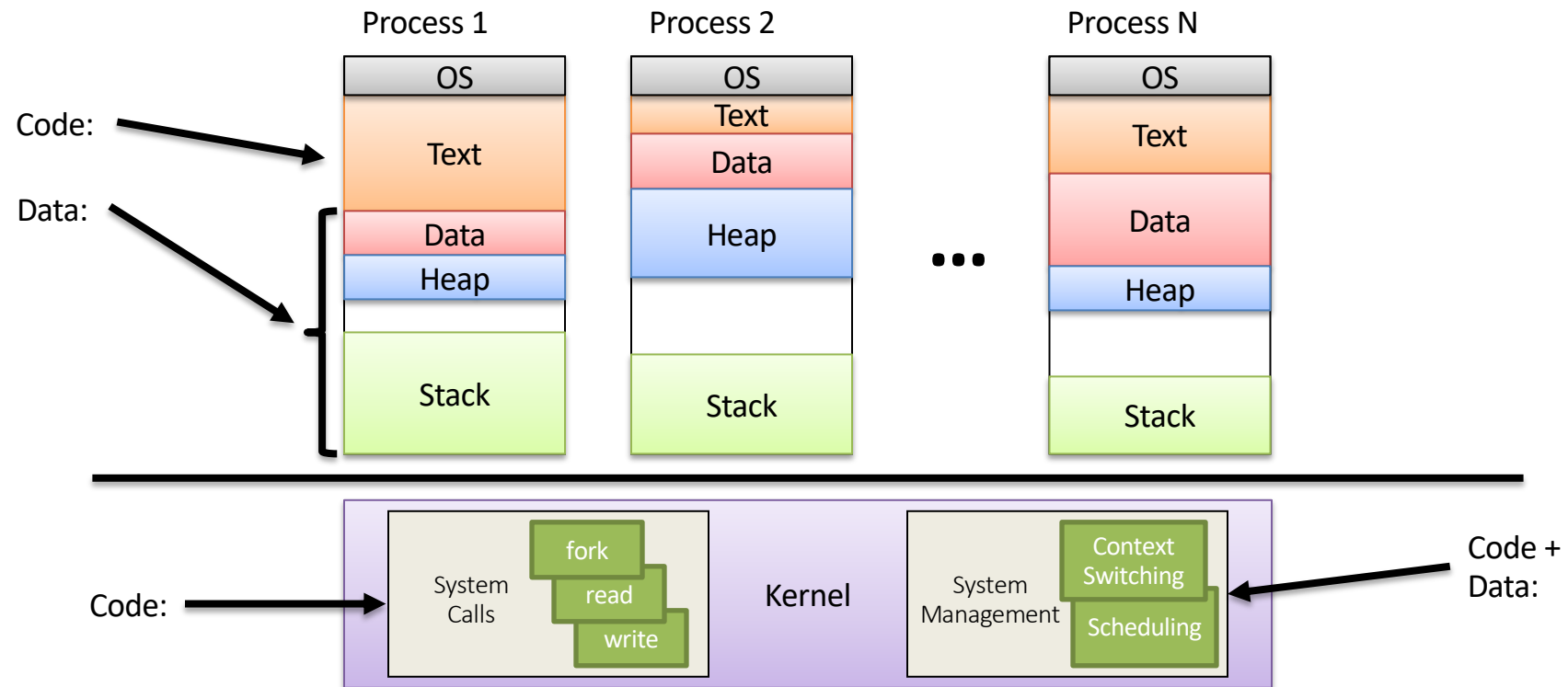
Process vs. Kernel

- The kernel is the code that supports processes
 - System calls: `fork ()`, `exit ()`, `read ()`, `write ()`, ...
 - System management: context switching, scheduling, memory management

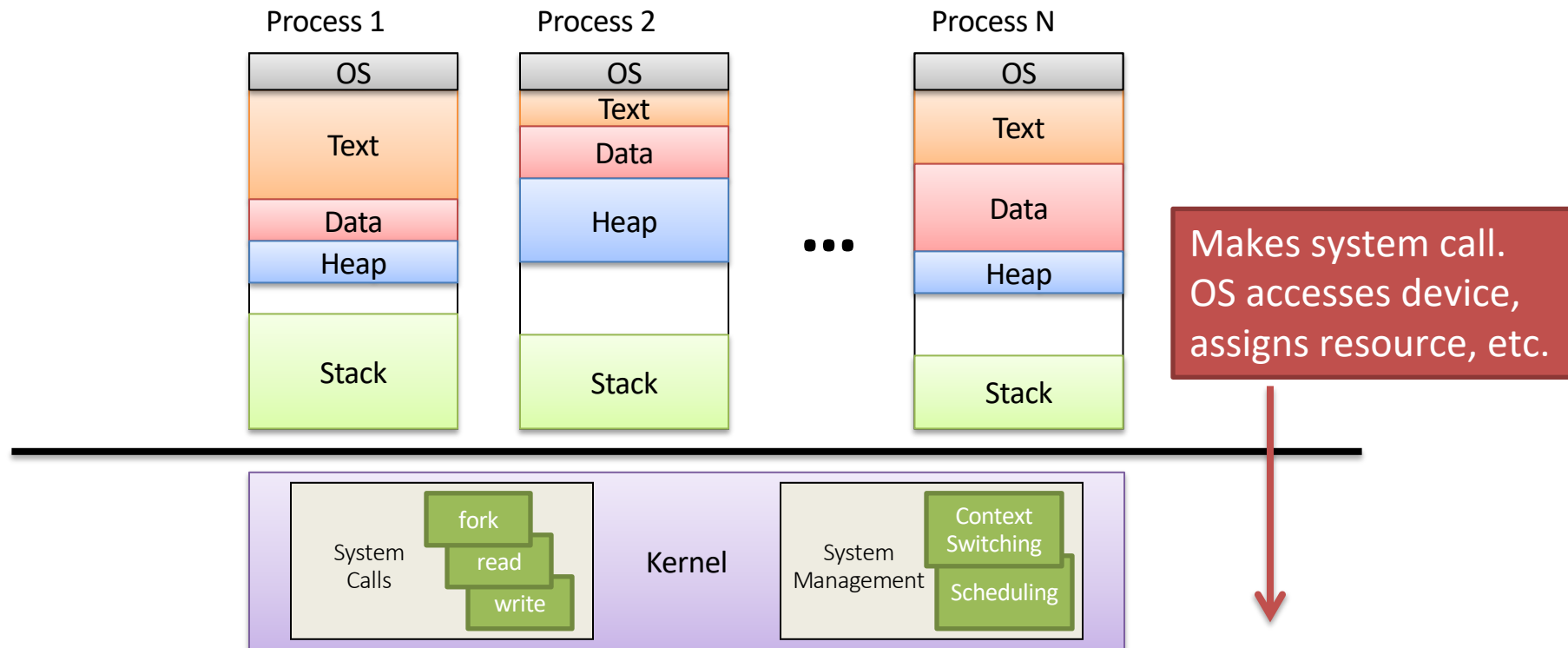
Kernel vs. Userspace: Model



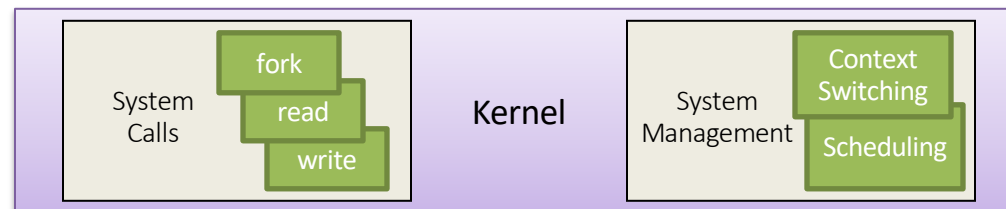
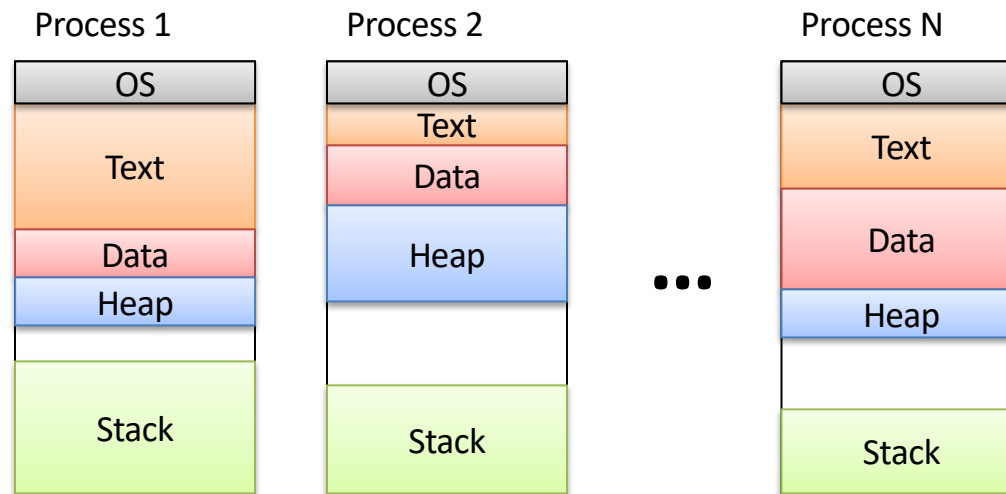
Kernel vs. Userspace: Model



Kernel vs. Userspace: Context Switch



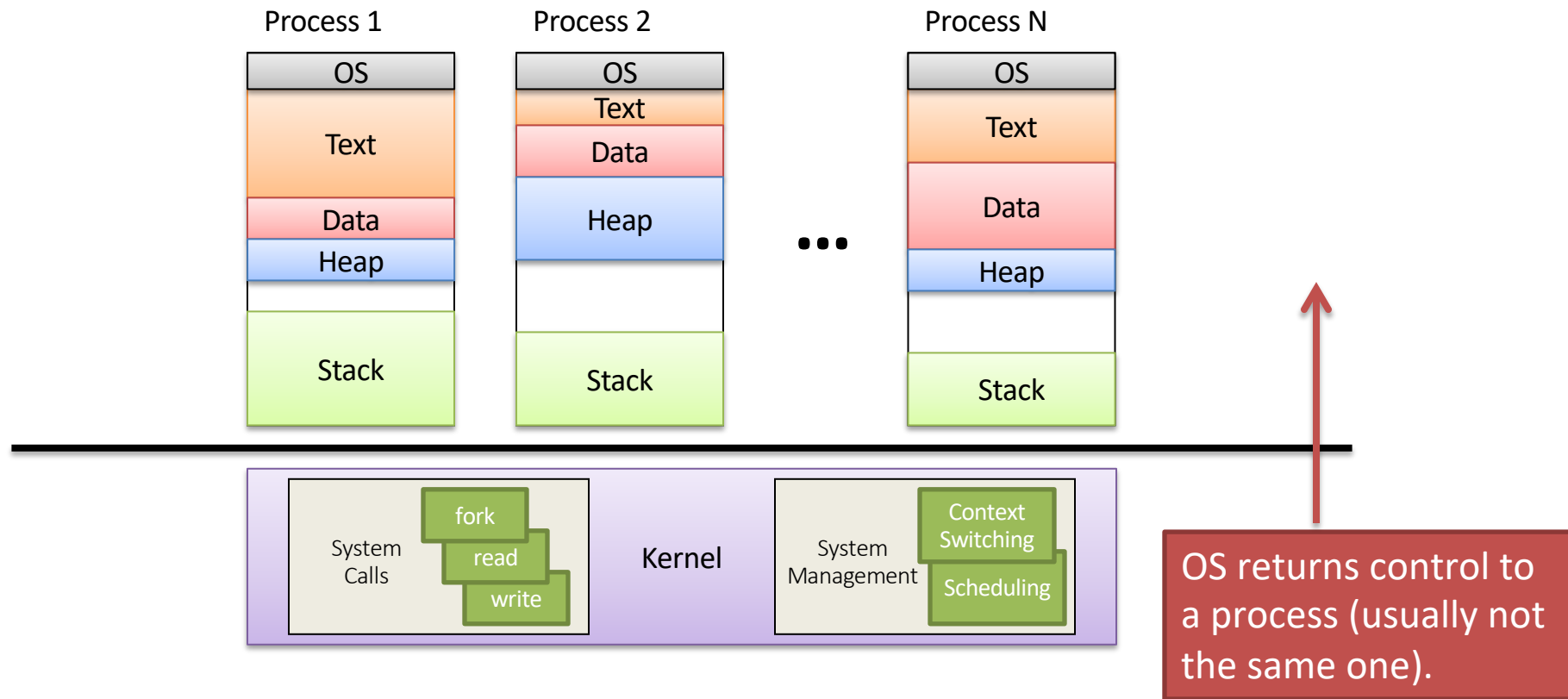
Kernel vs. Userspace: Context Switch



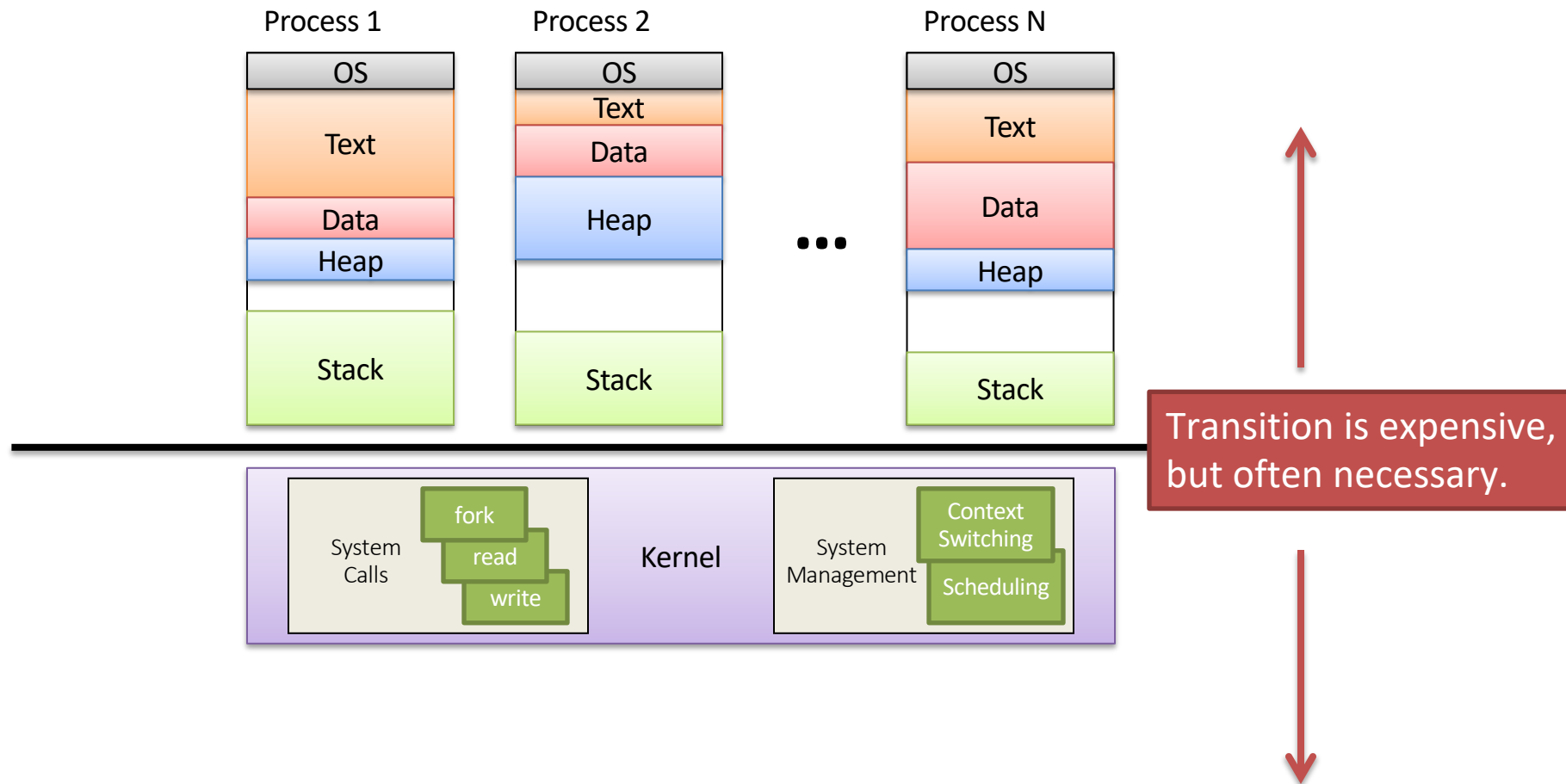
OS has control. It will take care of process's request, but it might take a while.

It can context switch (and usually does at this point).

Kernel vs. Userspace: Context Switch

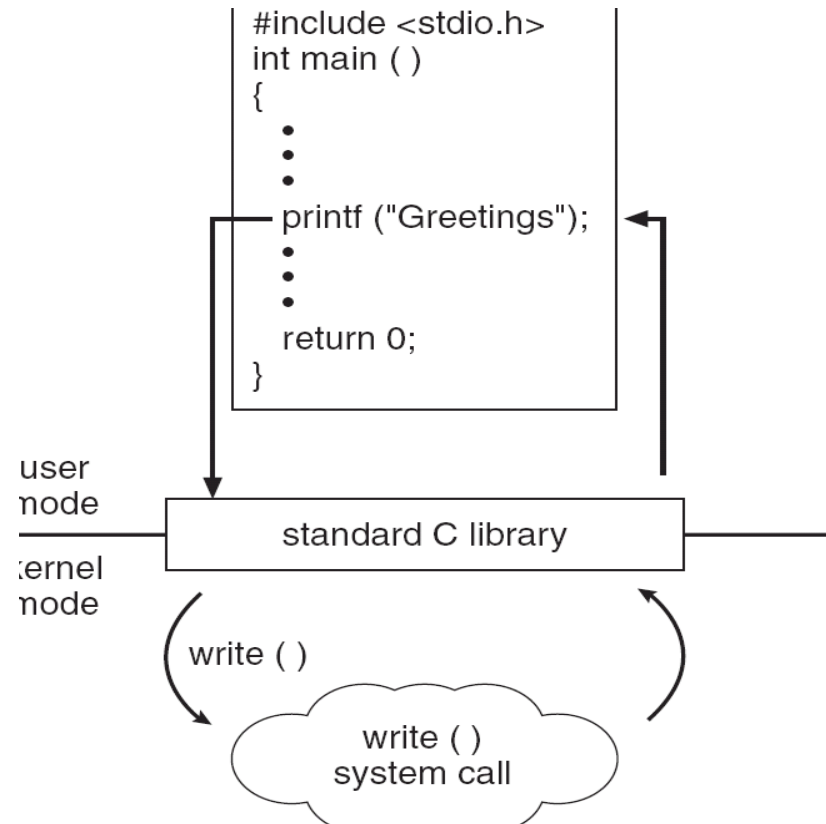


Kernel Kernel vs. Userspace: Context Switch



Standard C Library Example

C program invoking printf() library call, which calls write() system call



Control over the CPU

- To context switch processes, kernel must get control:
 1. **Running process can give up control voluntarily**
 - To block, call `yield ()` to give up CPU
 - Process makes a blocking system call, e.g., `read ()`
 - Control goes to kernel, which dispatches new process
 2. **CPU is forcibly taken away: preemption**

How might the OS forcibly take control of a CPU?

- A. Ask the user to give it the CPU.
- B. Require a program to make a system call.
- C. Enlist the help of a hardware device.
- D. Some other means of seizing control (how?).

CPU Preemption

1. While kernel is running, set a hardware timer.
2. When timer expires, a hardware interrupt is generated. (device asking for attention)
3. Interrupt pauses process on CPU, forces control to go to OS kernel.
4. OS is free to perform a context switch.

Up next...

- How we create/manage processes.
- How we provide the illusion of the same enormous memory space for all processes.