CS 31: Intro to Systems Operating Systems Overview

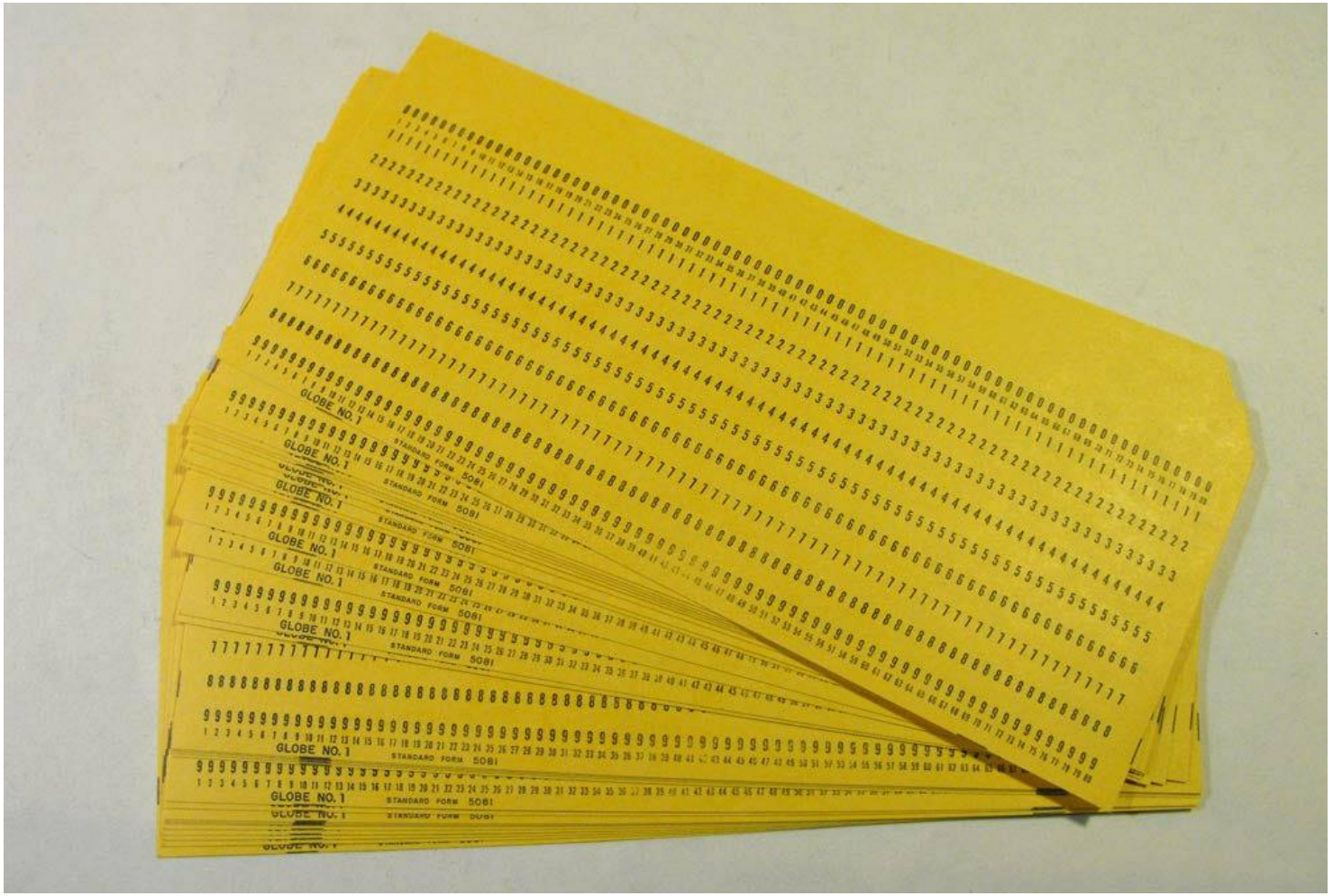
Kevin Webb
Swarthmore College
March 29, 2016

Reading Quiz

OS: Turn undesirable into desirable

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

Before Operating Systems



Running multiple programs

- When I/O issued, CPU not needed
  - Allow another program to run
  - Requires yielding and sharing memory
- What if one running program
  - Monopolizes CPU, memory?
  - Reads/writes another's memory?
  - Uses I/O device being used by another?

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

A. 1-10

B. 20-40

C. 40-80

D. 80-160

E. 160+

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
  - Originally called the nucleus in the 60's

The Kernel

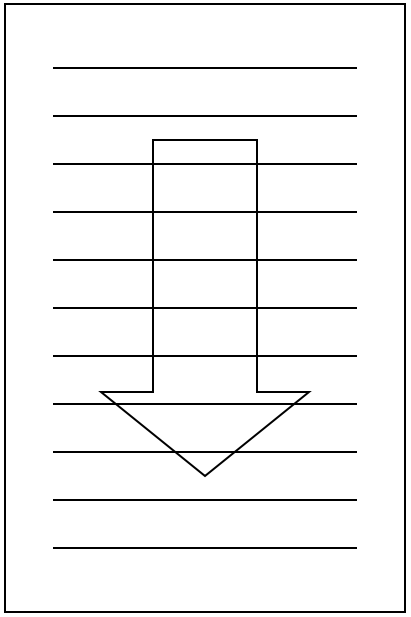
- All programs depend on it
  - Loads and runs them
  - Exports system calls to programs
- Works closely with hardware
  - Accesses devices
  - Responds to interrupts
- Allocates basic resources
  - CPU time, memory space
  - Controls I/O devices: display, keyboard, disk, network

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 implicitly
- What should functions be?
  - How many programs should benefit?
  - Might kernel get too big?

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.

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)

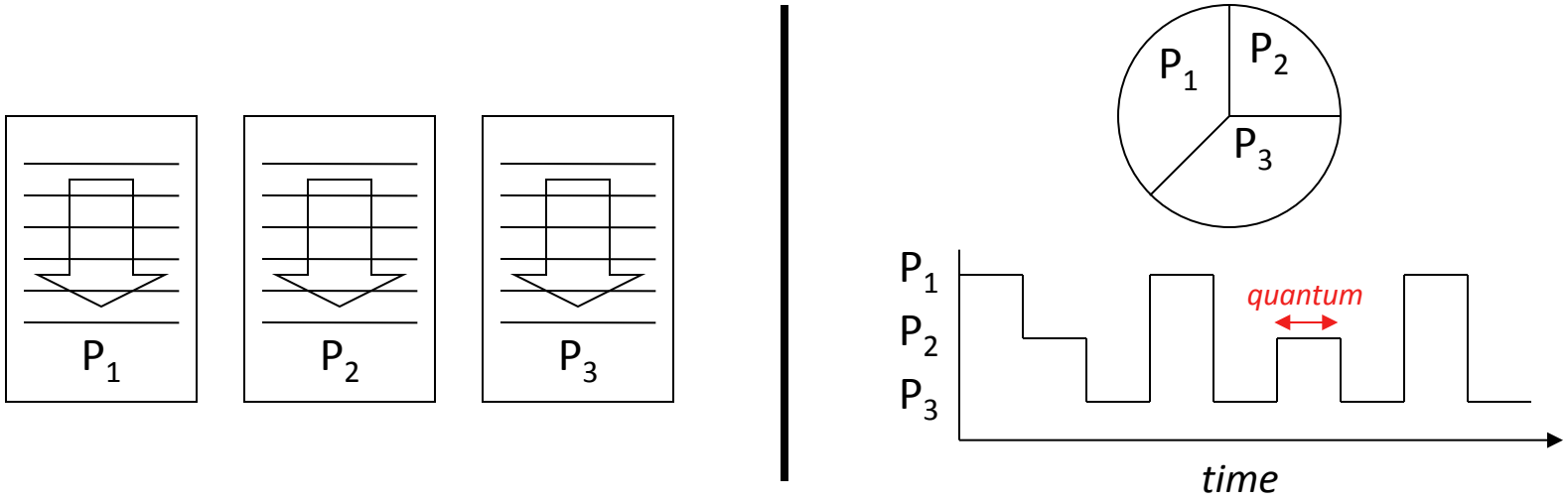
Must keep track of these for every running process !

CPU

- 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)

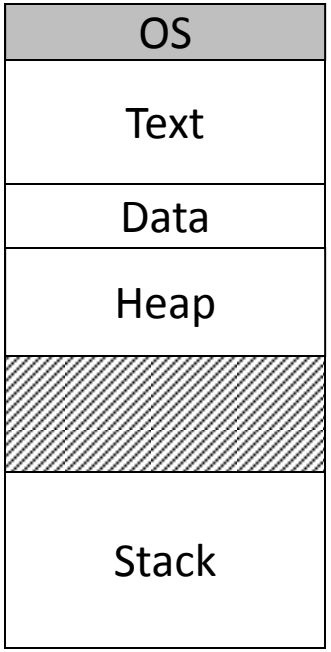
Timesharing: Sharing the CPU



- Multiple processes, single CPU (uniprocessor)
- Conceptually, each process makes progress over time
- In reality, each periodically gets quantum of CPU time
- Illusion of parallel progress by rapidly switching CPU

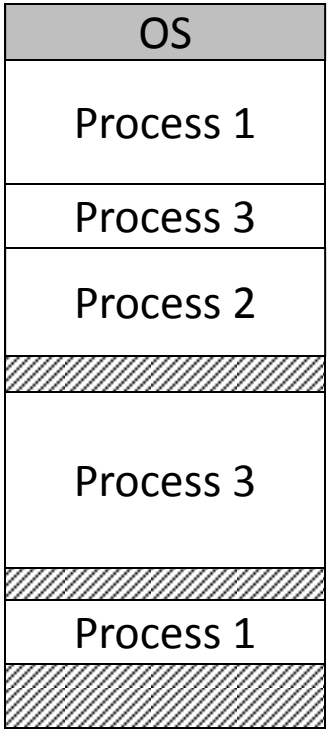
Memory

- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.



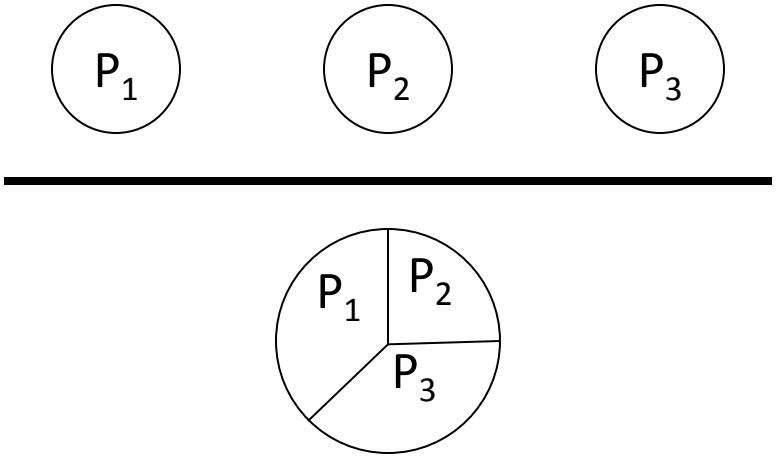
Memory

- 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, and 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.

The Kernel

- So…how / when should the kernel execute?

- What would you do if you were to build such a thing?

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.

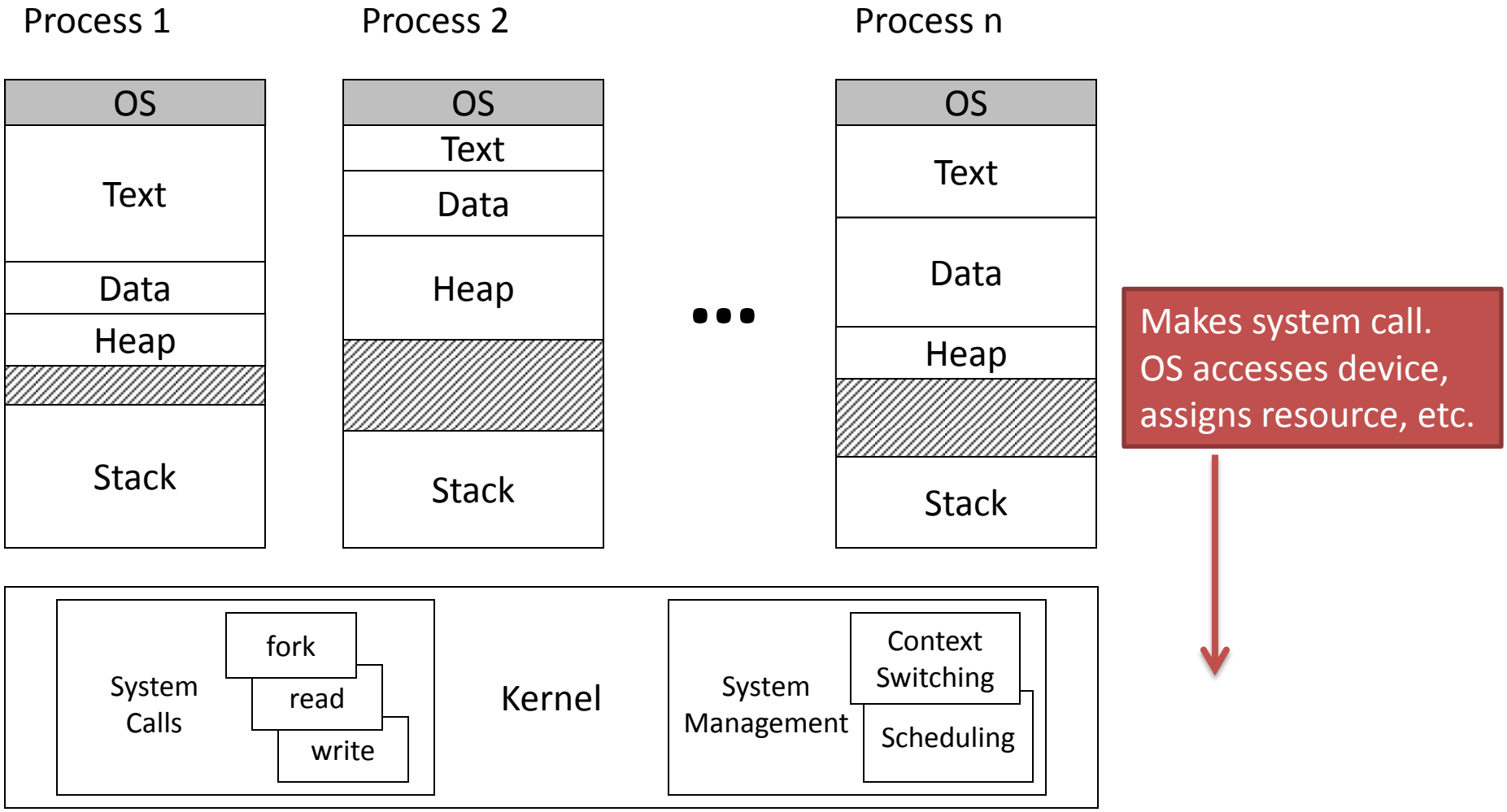
D. More than one of the above. (Which ones?)

E. None of the above.

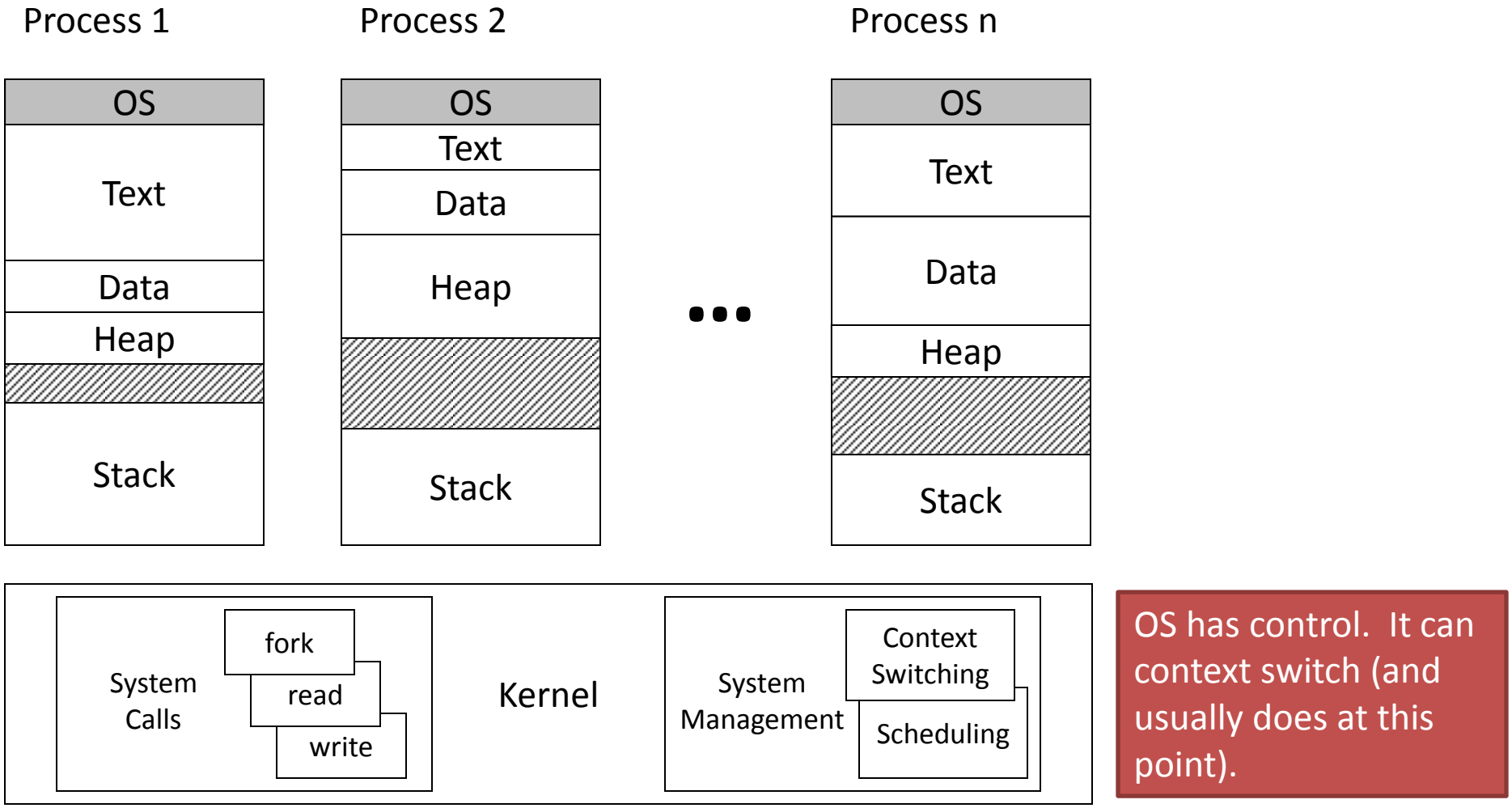
Process vs. Kernel

- The kernel is the code that supports processes
  - System calls: fork ( ), exit ( ), read ( ), write ( ), …
  - System management: context switching, scheduling
- When does the kernel run?
  - When system call or hardware interrupt occurs
- Is the kernel a process?
  - No, it supports processes and devices
  - Runs as an extension of process making system call
  - Runs in response to device issuing interrupt

Process and Kernel Model

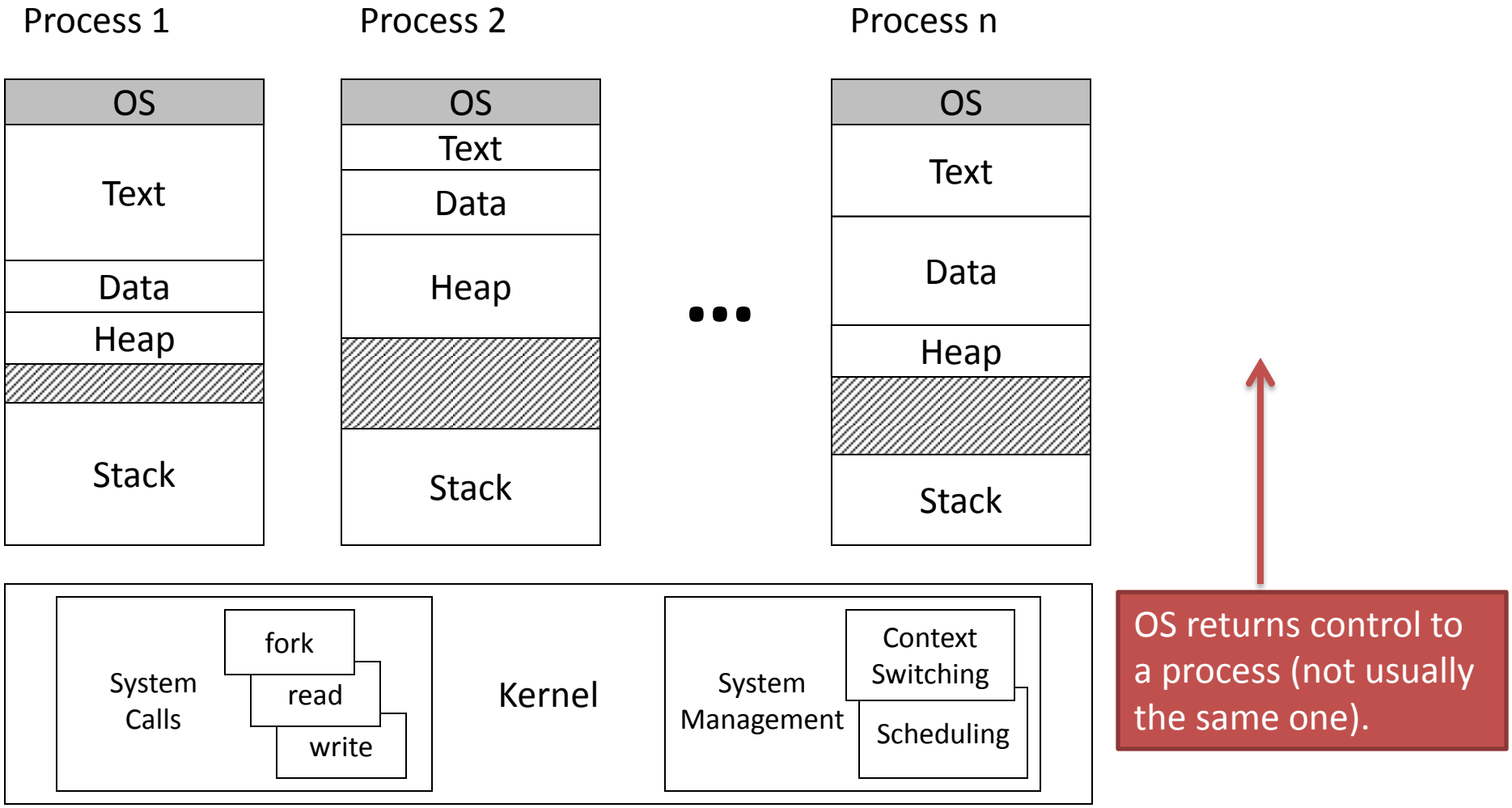


Process and Kernel Model

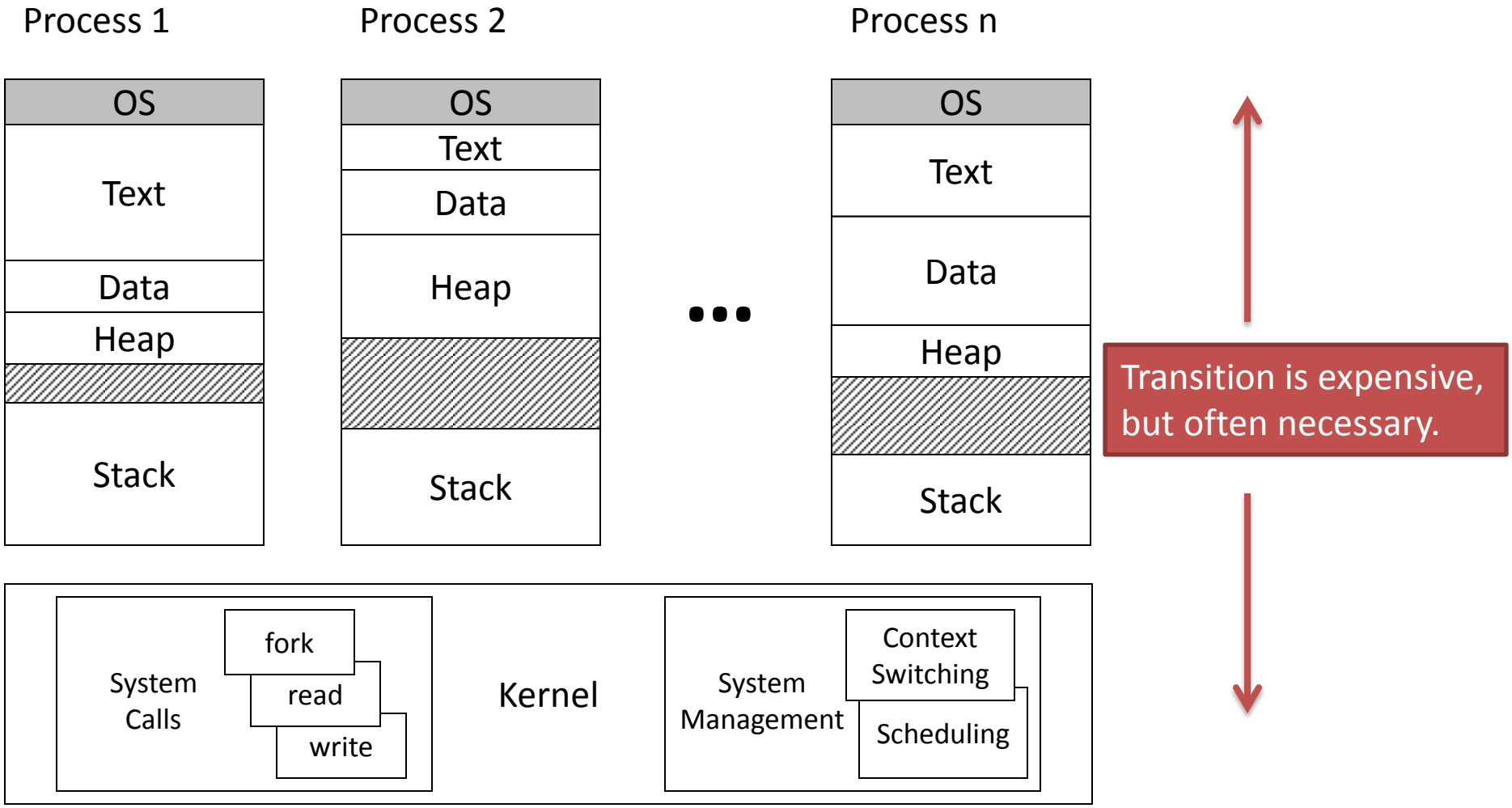


OS has control. It can context switch (and usually does at this point).

Process and Kernel Model



Process and Kernel Model



How Does Kernel Get Control

- To allow process to run, kernel must get control
- Running process can give up control voluntarily
  - Process makes a blocking system call, e.g., read ()
  - To block, call yield () to give up CPU
  - Control goes to kernel, which dispatches a process
- Or, CPU is forcibly taken away: preemption
  - While kernel is running, it sets a timer
  - When timer expires, interrupt is generated
  - Hardware forces control to go to kernel

Up next…

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