CS 31: Introduction to Computer Systems

20-21: Operating Systems & Processes

April 8-10, 2019

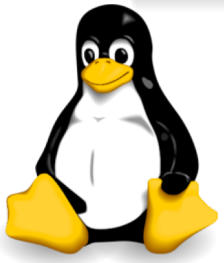
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources



Complex devices
Compute & I/O

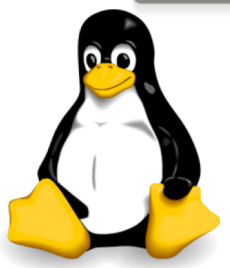
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources



Complex devices
Compute & I/O

OS Big Picture Goals

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

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

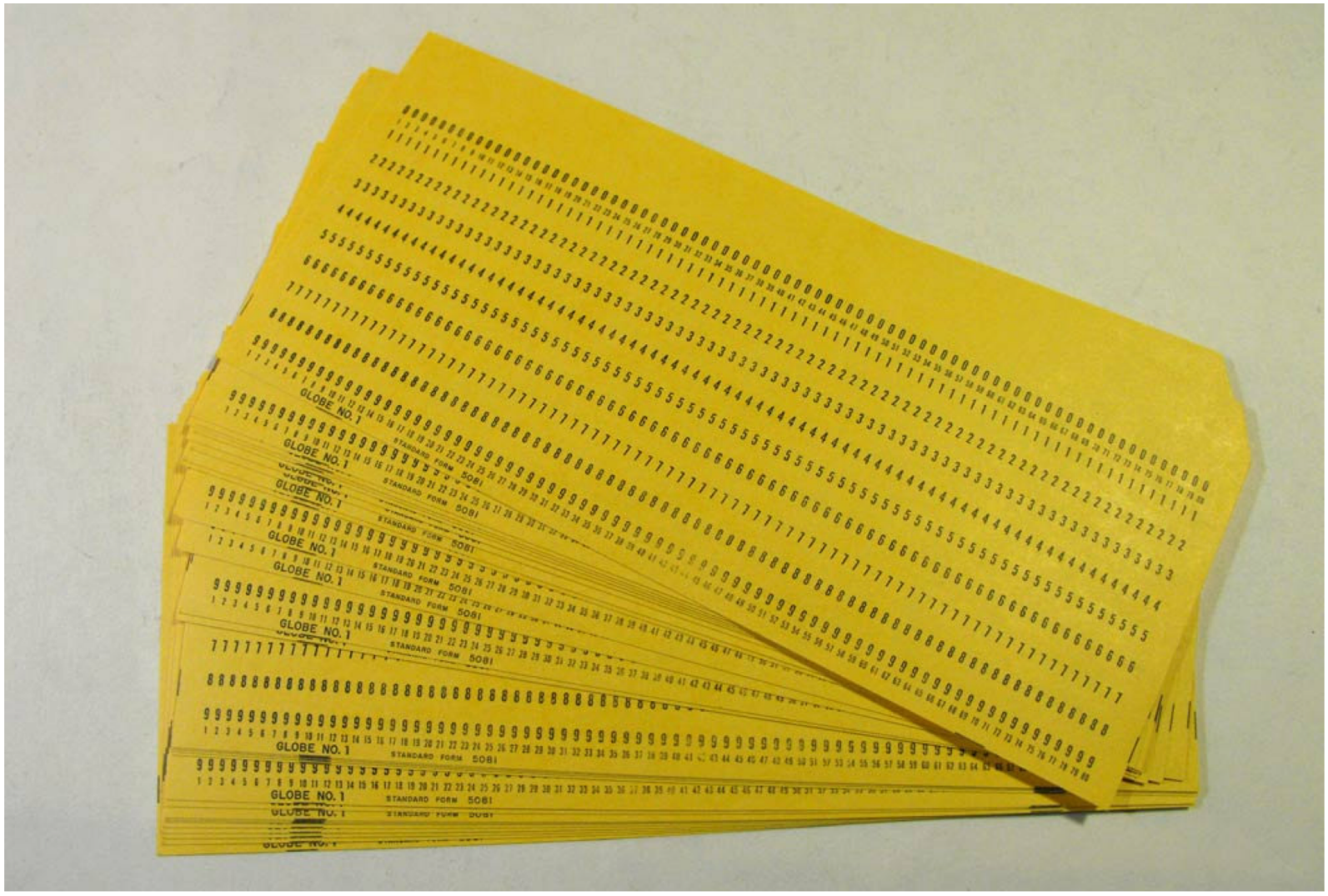
2. Resource sharing

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?)

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.

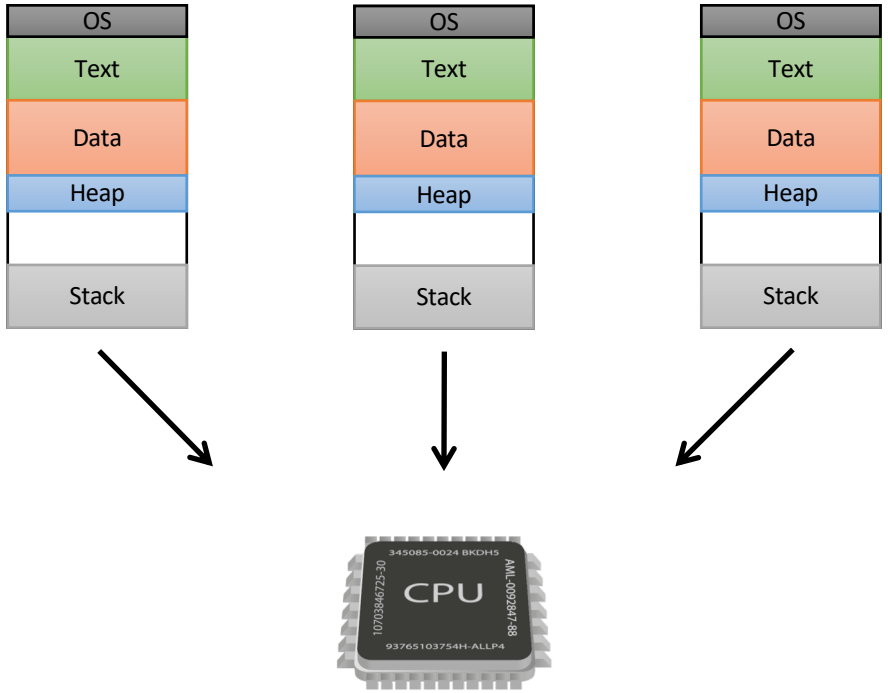
B. The hardware won't execute as quickly.

C. The hardware's resources won't be used as efficiently.

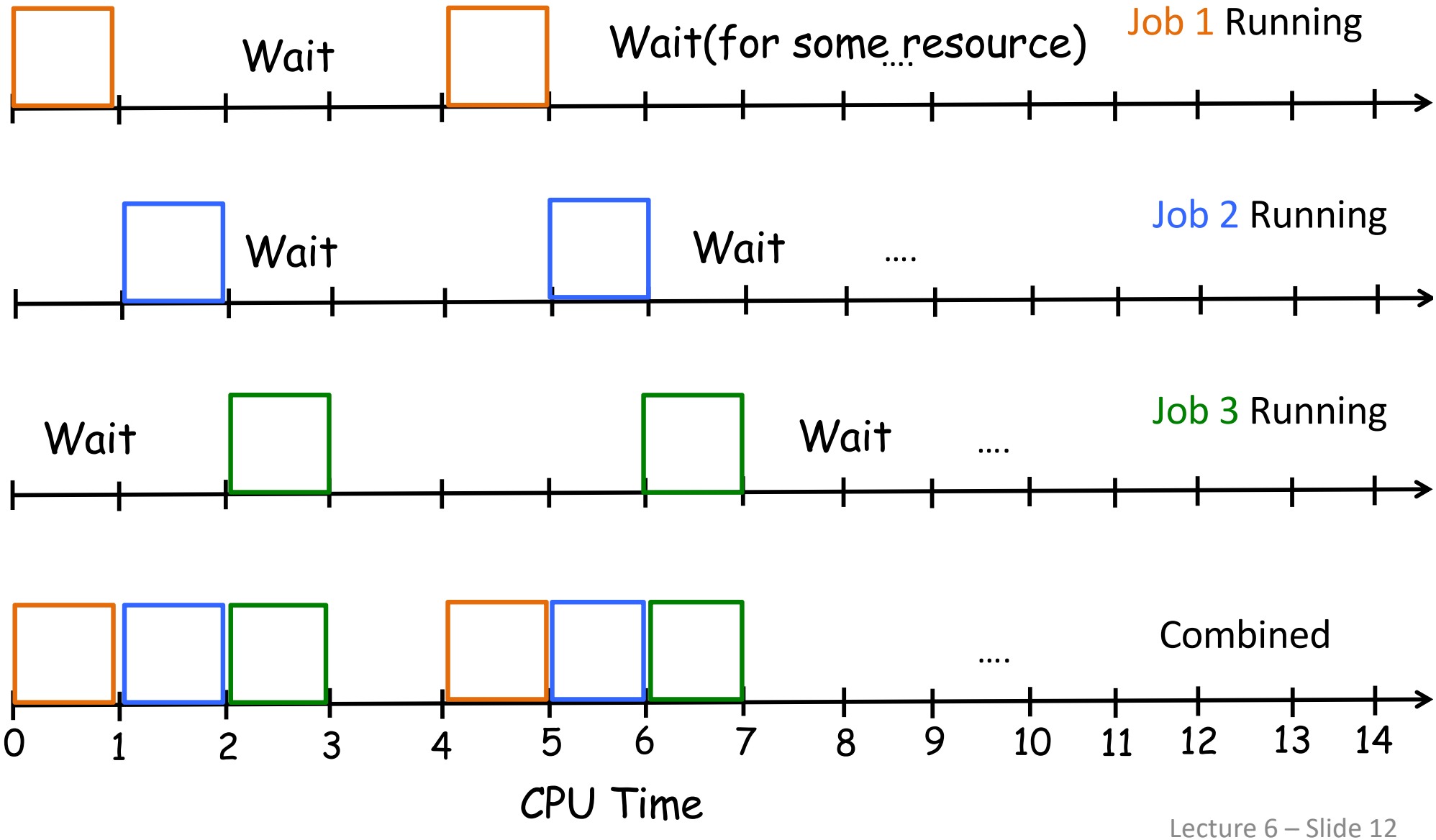
D. Some other reason(s). (What?)

Today: Multiprogramming

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



Multiprogramming



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
- Allocates basic resources
  - CPU time, memory space
  - Controls I/O devices: display, keyboard, disk, network

TRON

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?

OS Kernel

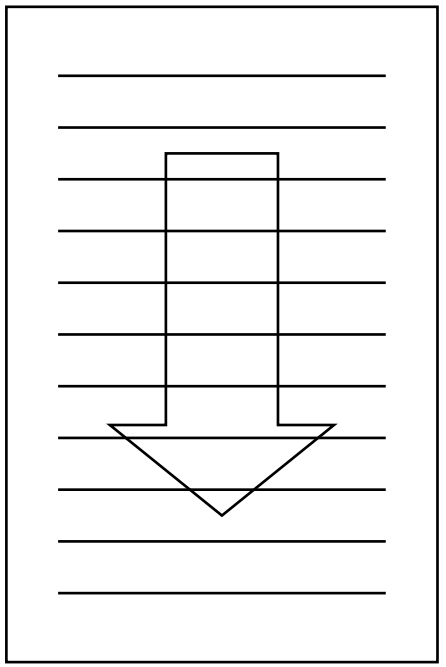
- Big Design Issue: How do we make the OS efficient, reliable, and extensible?

- 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.)

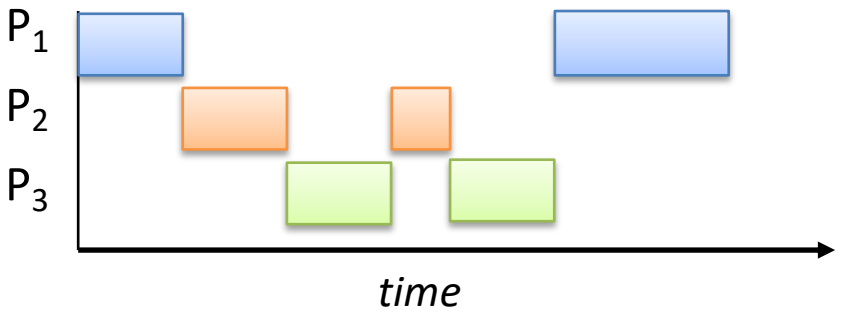
Machine State of a Process

- 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)

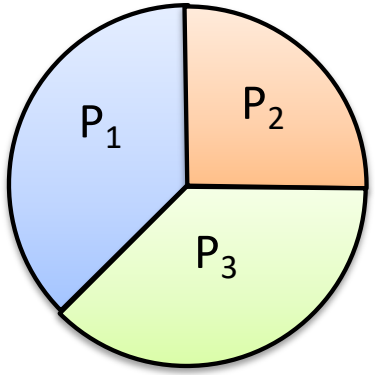
Must keep track of these for every running process !

Resource Sharing

CPU: Time



Memory: Space



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.

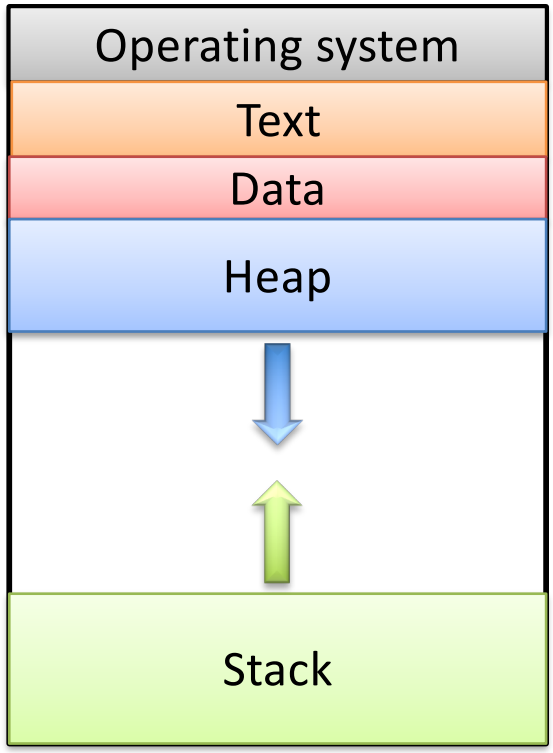
Timesharing: Sharing the CPUs

- 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, …
- Mechanism: context switching)

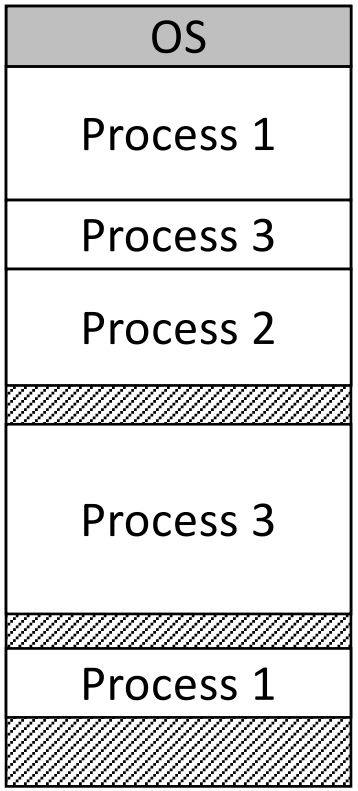
Resource: Memory

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



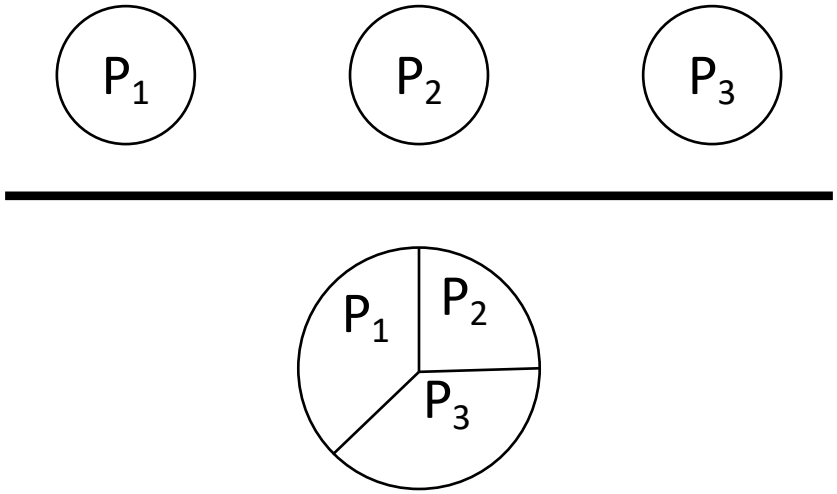
Memory

- 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, 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.

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.

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

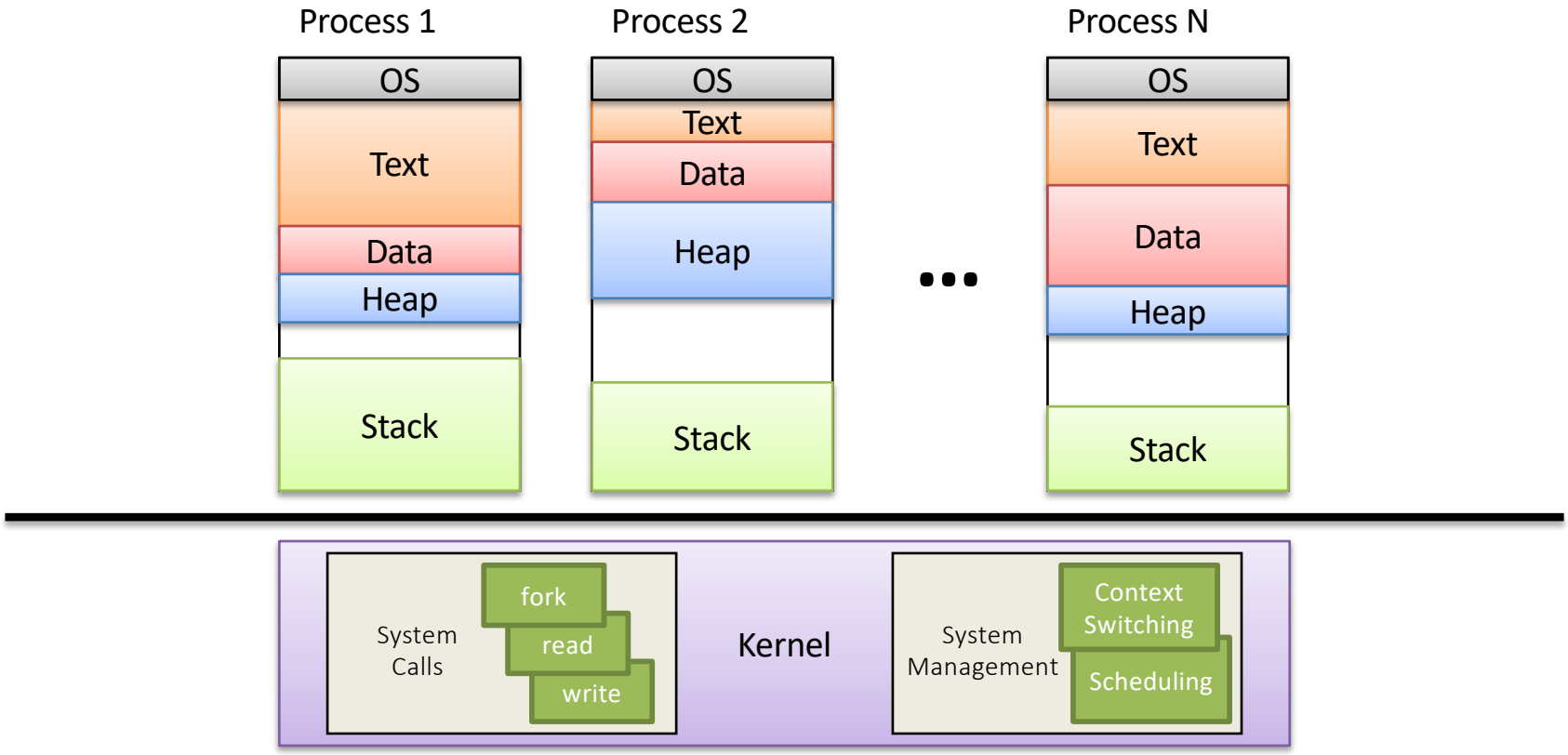
- Is the kernel itself a process?
  - No, it supports processes and devices

- OS only runs when necessary…
  - as an extension of a process making system call
  - in response to a device issuing an interrupt

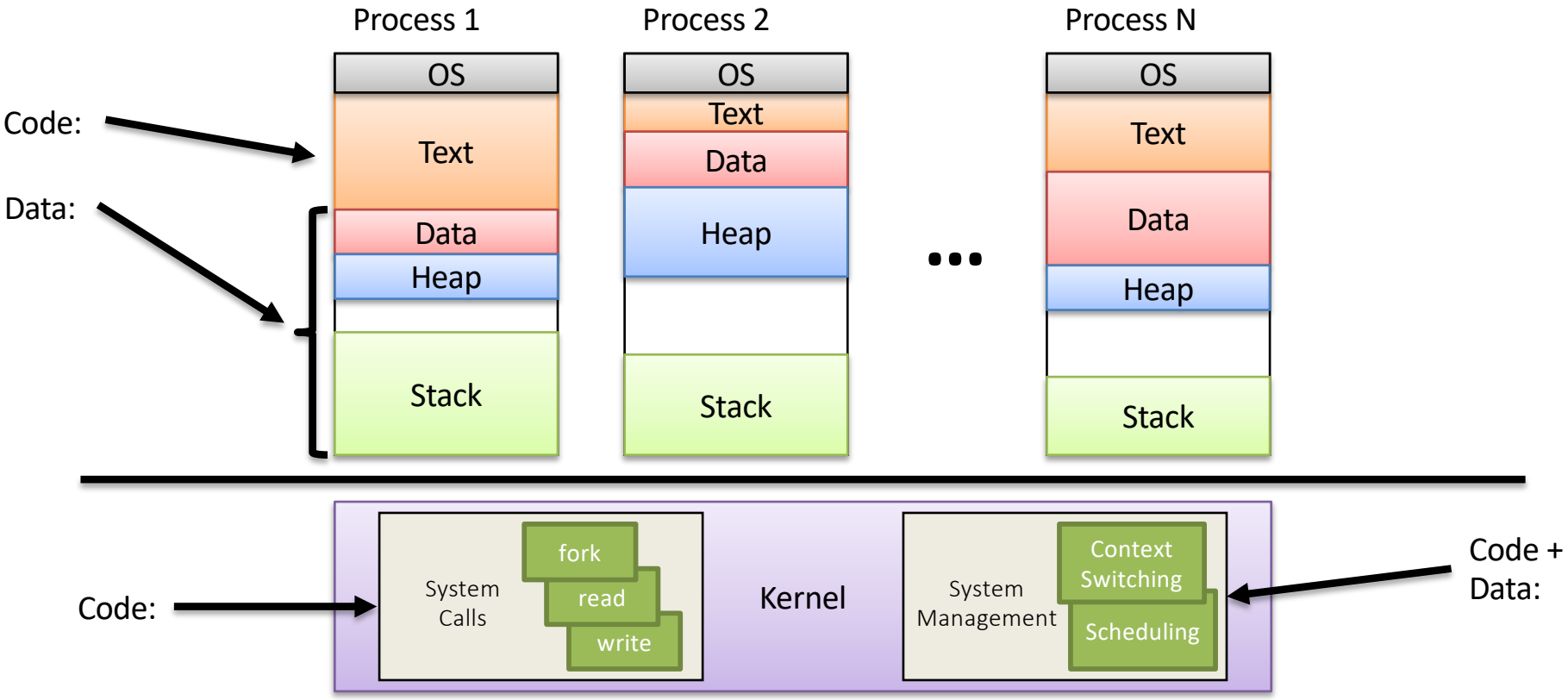
Process vs. Kernel

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

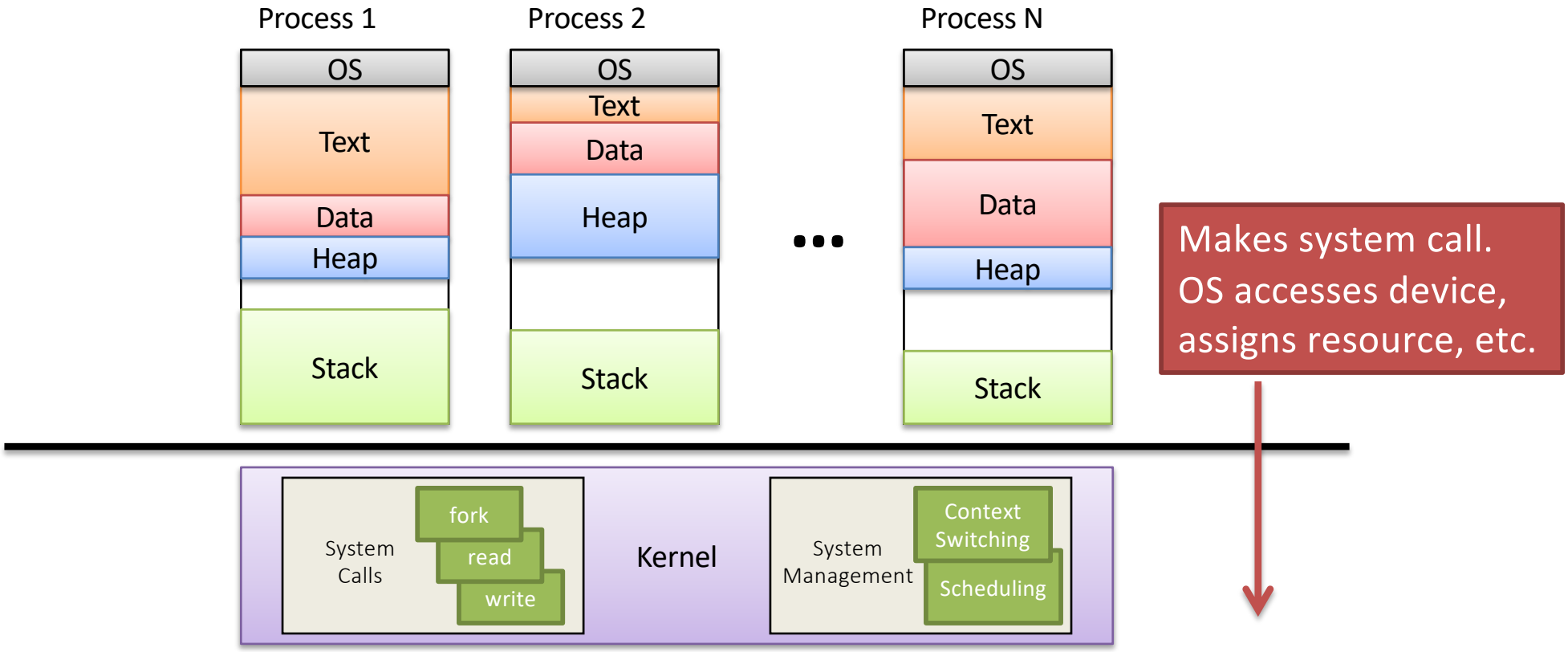
Kernel vs. Userspace: Model



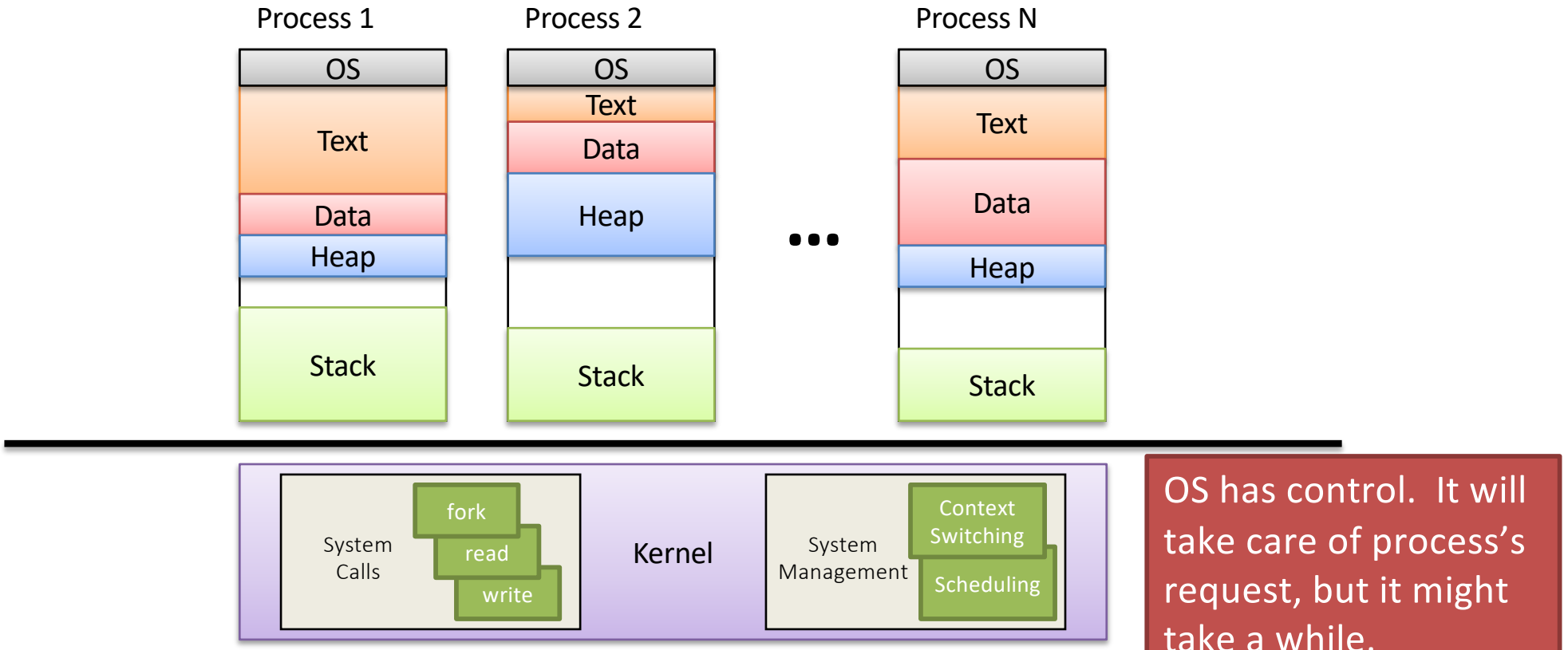
Kernel vs. Userspace: Model



Kernel vs. Userspace: Model



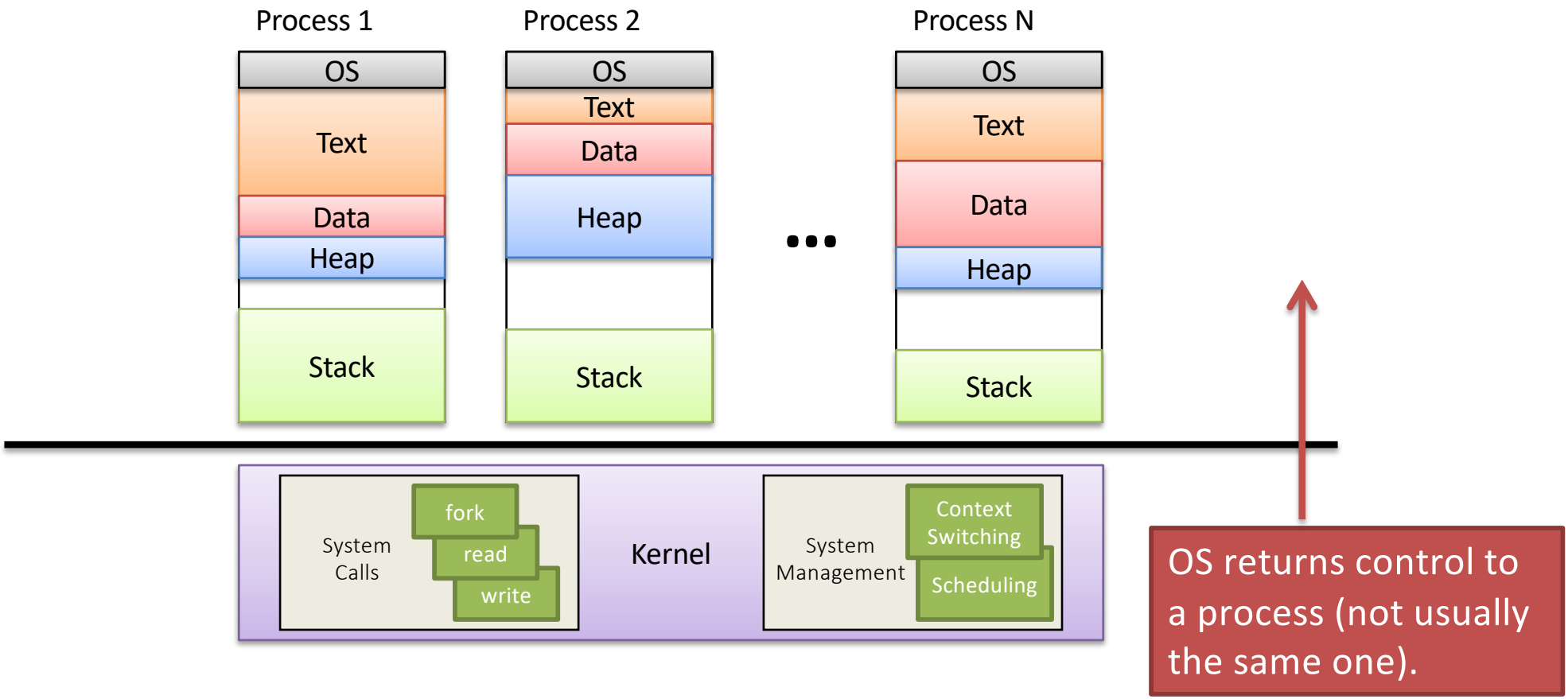
Kernel vs. Userspace: Model



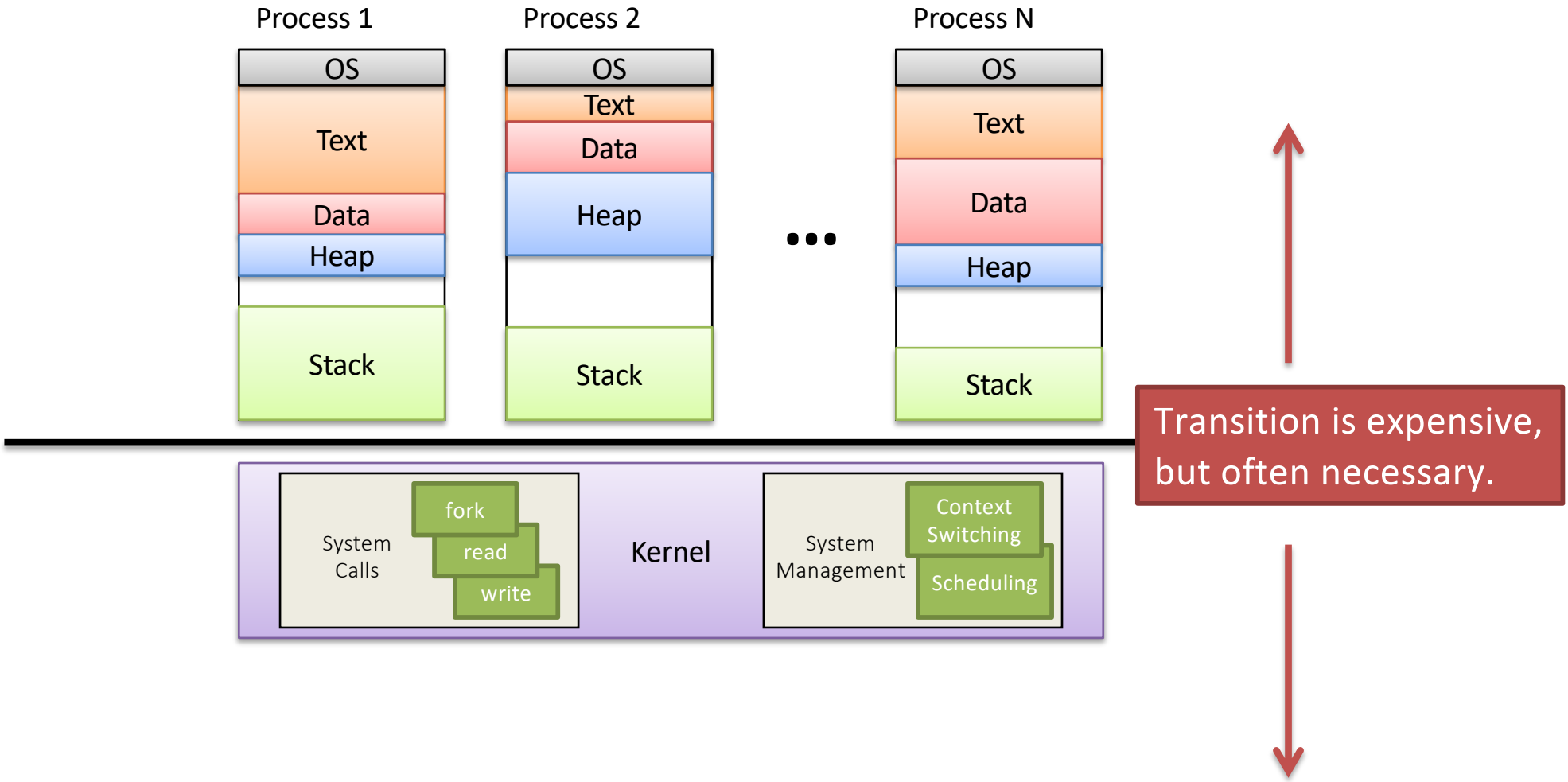
OS has control. It will take care of process's request, but it might take a while.

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

Kernel vs. Userspace: Model

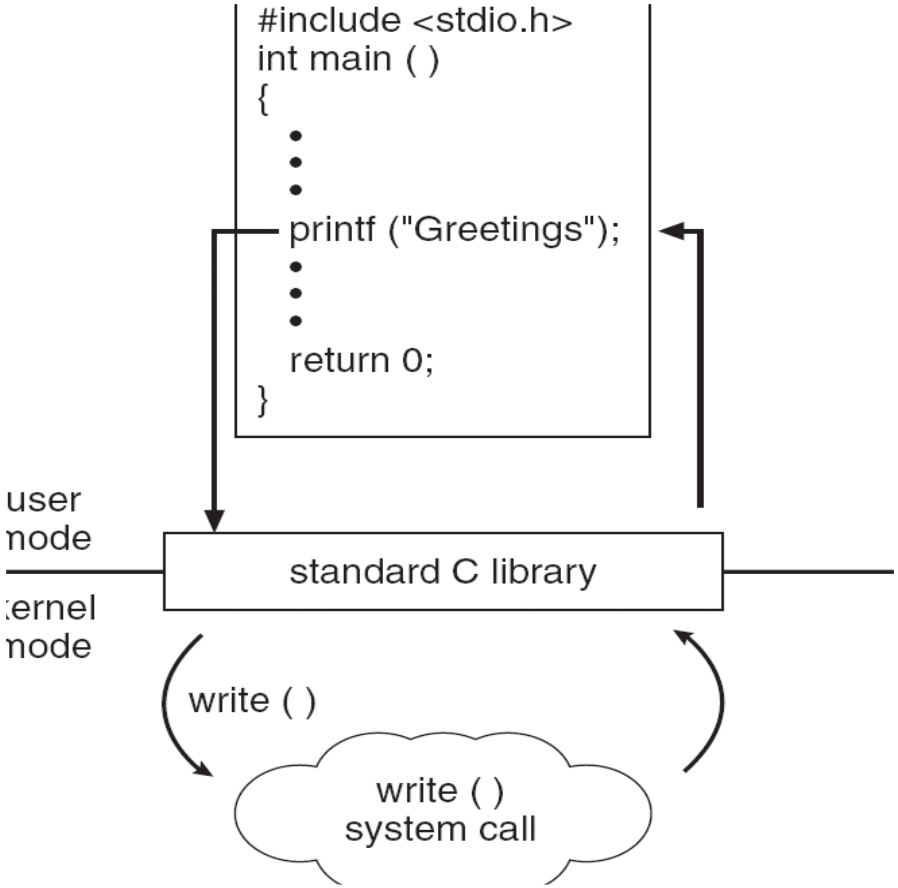


Kernel vs. Userspace: Model



System call 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?).

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.

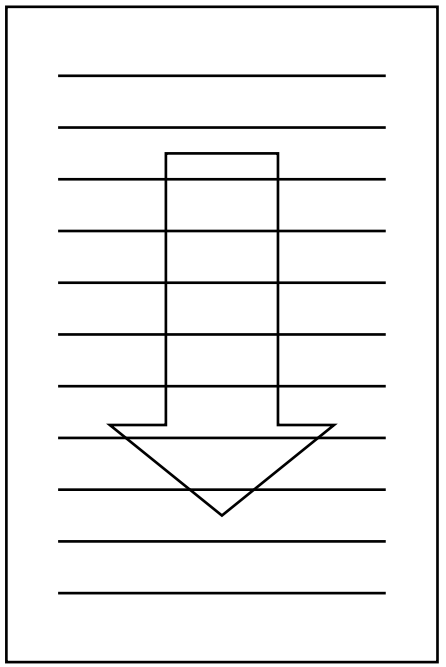
4. OS is free to perform a context switch.

Up next…

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

Anatomy of a Process

- Abstraction of a running program
  - a dynamic “program in execution”
- OS keeps track of process state
  - What each process is doing
  - Which one gets to run next
- Basic operations
  - Suspend/resume (context switch)
  - Start (spawn), terminate (kill)

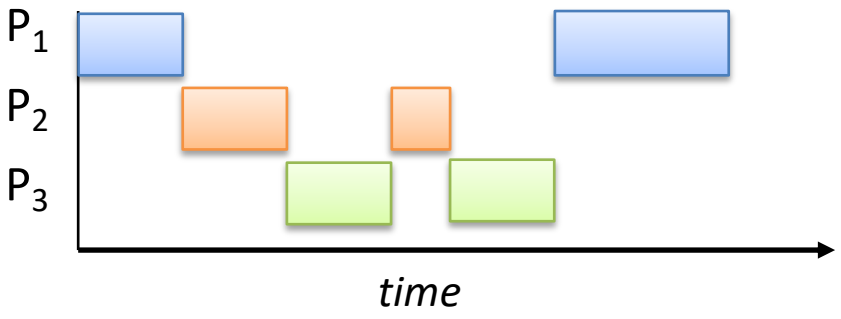


Processes

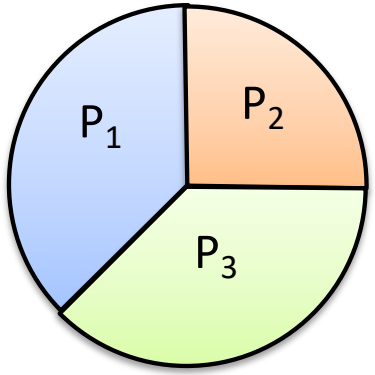
- Process: dynamic execution context of an executing program
- A process is not a program!
- A process is one instance of a program in execution. Many processes can be running the same program. Processes are independent entities.
- Several processes may run the same program, but each is a distinct process with its own state (e.g., Tabs in Chrome).
- A process executes sequentially, one instruction at a time

Resource Sharing

CPU: Time



Memory: Space



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, …
- Mechanism: context switching)

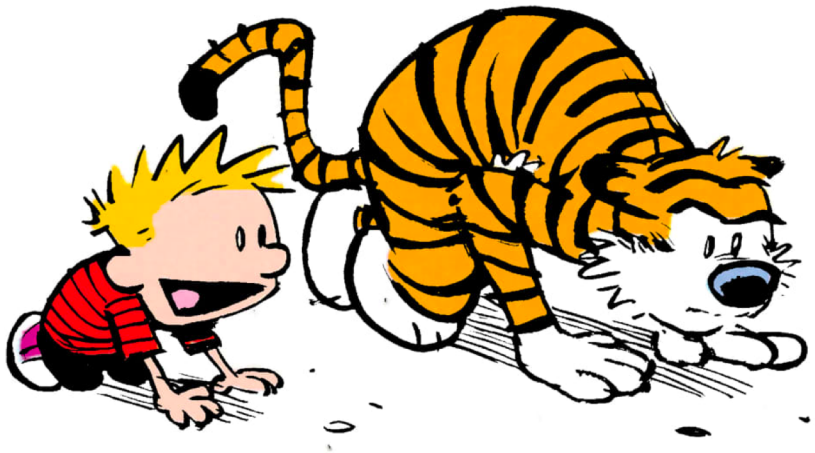
How is Timesharing Implemented?

- Kernel keeps track of progress of each process
- Characterizes state of process's progress
  - Running: actually making progress, using CPU

(C) Calvin and Hobbes

How is Timesharing Implemented?

- Kernel keeps track of progress of each process
- Characterizes state of process's progress
  - Ready: able to make progress, but not using CPU



(C) Calvin and Hobbes

How is Timesharing Implemented?

- Kernel keeps track of progress of each process
- Characterizes state of process's progress
  - Blocked: not able to make progress, can't use CPU

(C) Calvin and Hobbes

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, …

Why might a process be blocked (unable to make progress / use CPU)?

A. It's waiting for another process to do something.

B. It's waiting for memory to find and return a value.

C. It's waiting for an I/O device to do something.

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

E. Some other reason(s).

Why might a process be blocked (unable to make progress / use CPU)?

A. It's waiting for another process to do something.

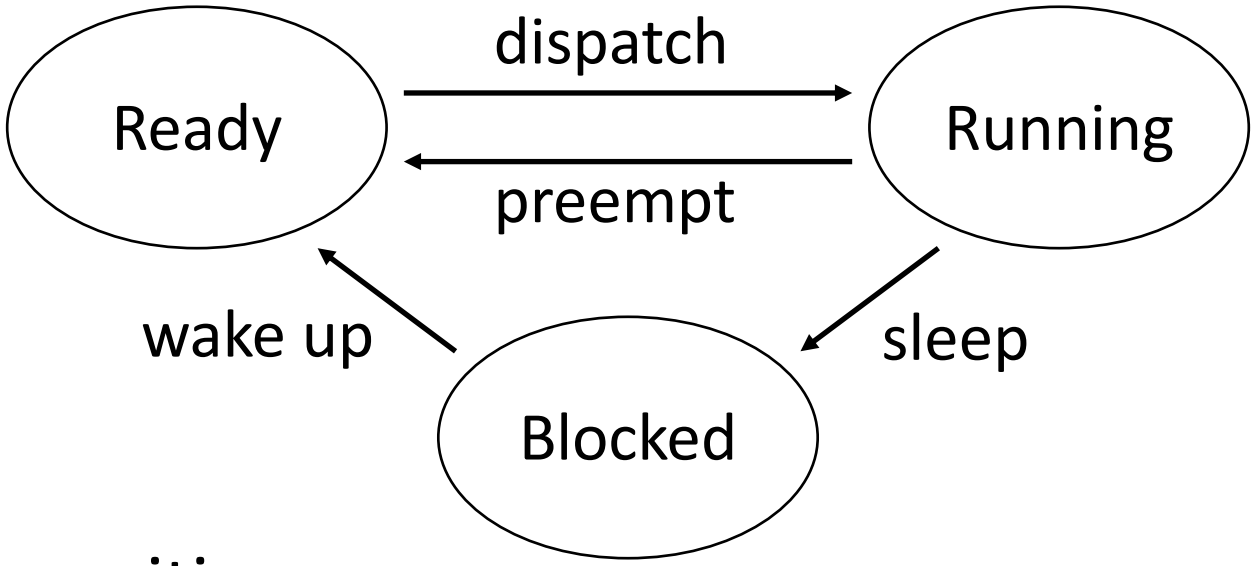
B. It's waiting for memory to find and return a value.

C. It's waiting for an I/O device to do something.

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

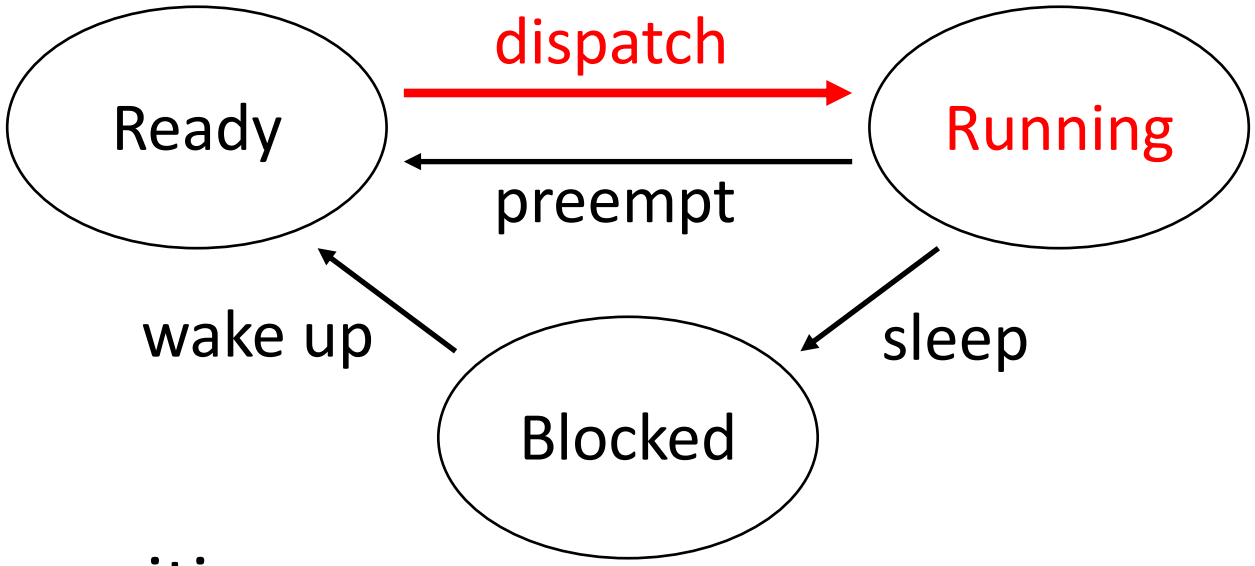
E. Some other reason(s).

Process State Diagram



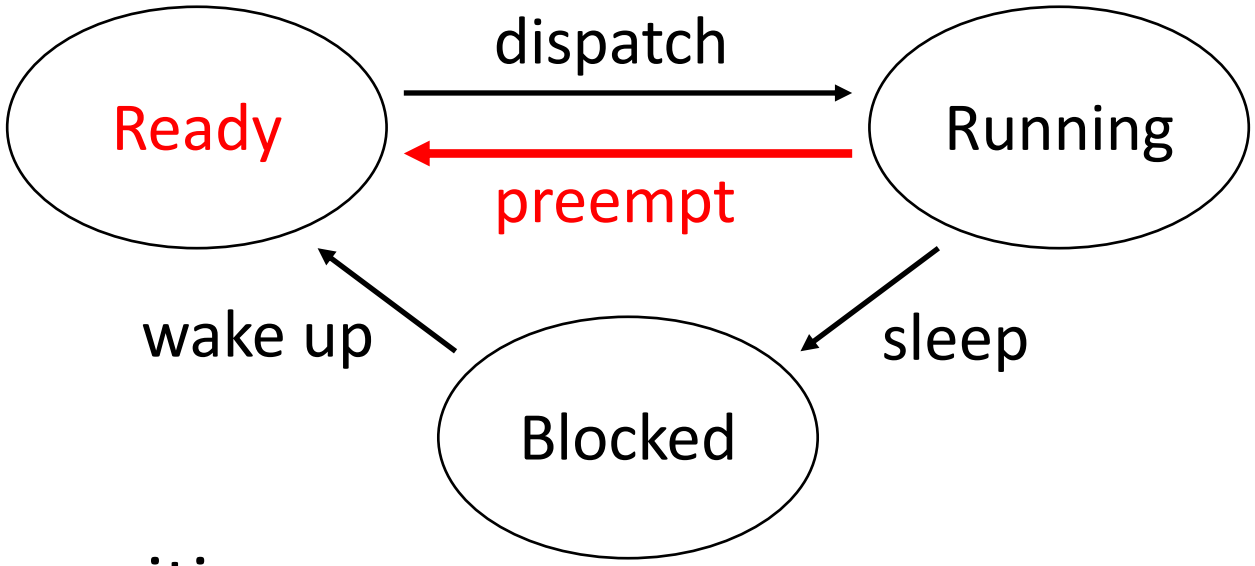
- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready

Process State Diagram



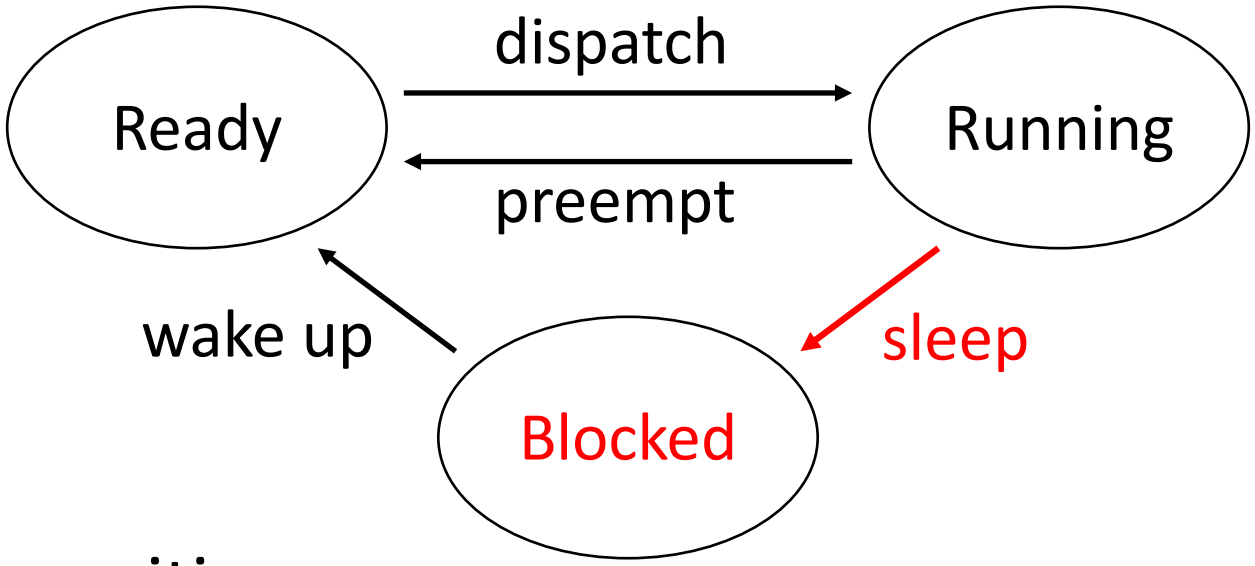
- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready

Process State Diagram



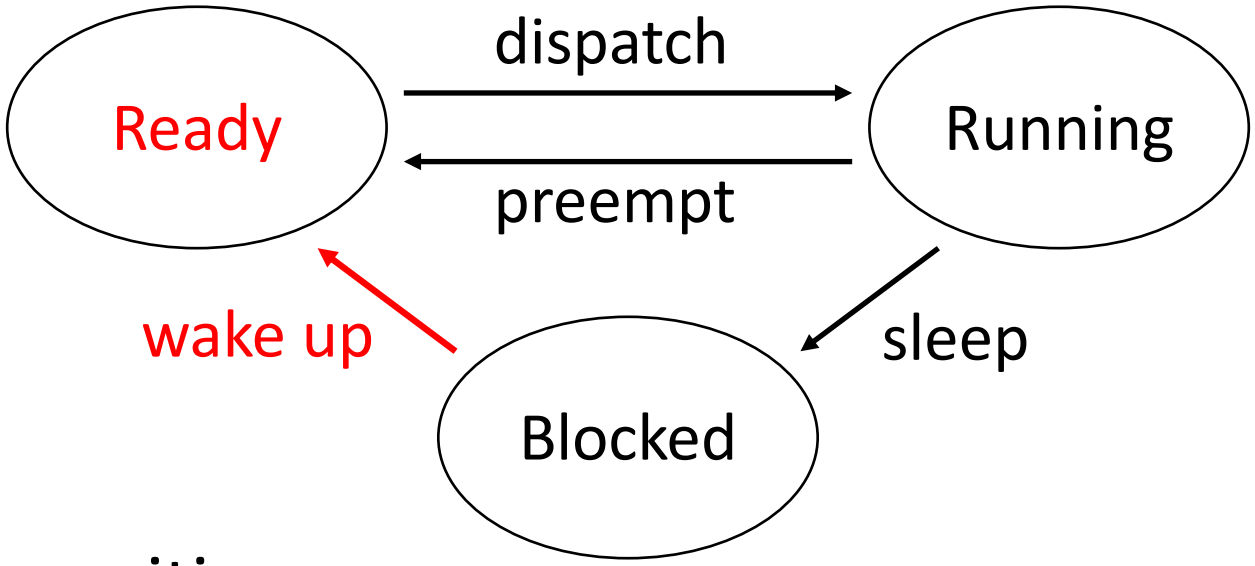
- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready

Process State Diagram



- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready

Process State Diagram



- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready

Kernel Maintains Process Table

Process ID (PID)	State	Other info
1534	Ready	Saved context, …
34	Running	Memory areas used, …
487	Ready	Saved context, …
9	Blocked	Condition to unblock, …

- Table: List of processes and their states
  - Also sometimes called "process control block (PCB)"
- Other state info includes
  - contents of CPU context
  - areas of memory being used
  - other information

Values of registers in use by process

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

Context Switching

- Allocating CPU from one process to another
  - First, save context of currently running process
  - Next, load context of next process to run

Context Switching

- Allocating CPU from one process to another
  - First, save context of currently running process
  - Next, load context of next process to run
- Loading the context
  - Load general registers, stack pointer, etc.
  - Load program counter (must be last instruction!)

How a Context Switch Occurs

- Process makes system call or is interrupted
  - These are the only ways of entering the kernel

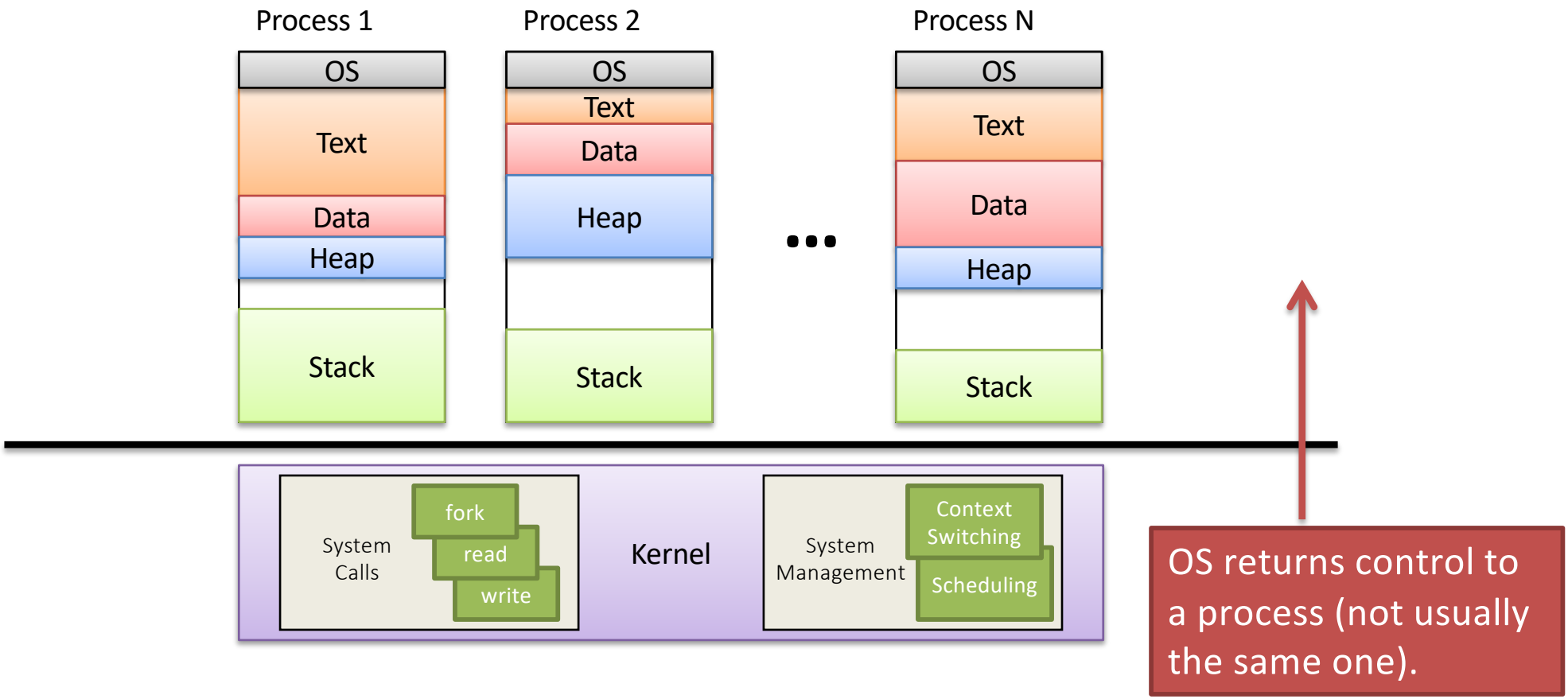
How a Context Switch Occurs

- In hardware
  - Switch from user to kernel mode: amplifies power
  - Go to fixed kernel location: interrupt/syscall handler

How a Context Switch Occurs

- In software (in the kernel code)
  - Save context of last-running process
  - Conditionally
    - Select new process from those that are ready
    - Restore context of selected process
  - OS returns control to a process from interrupt/syscall

Context Switch: Loading the PC last



process wakes up by whatever instruction is pointed to by the program counter

Why shouldn't processes control context switching?

A. It would cause too much overhead (costs too much resources: time/mem.).

B. They could refuse to give up the CPU.

C. They don't have enough information about other processes.

D. Some other reason(s).

Why shouldn't processes control context switching?

A. It would cause too much overhead (costs too much resources: time/mem. – have to implement their own context switching) - true

B. They could refuse to give up the CPU (also true)

C. They don't have enough information about other processes (more fundamental problem)

D. Some other reason(s).

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

The CPU Scheduling Problem

- Given multiple processes, but only one CPU
- How much CPU time does each process get?
- Which process do we run next?

- Possibilities
  - Keep CPU till done
  - Each process uses CPU a bit and passes it on
  - Each process gets proportional to what they pay

Which CPU scheduling policy is the best?

A. Processes keep CPU until done (maximize throughput)

B. Processes use a fraction of CPU and pass it on (ensure fairness)

C. Processes receive CPU in proportion to their priority or what they pay (prioritize importance)

D. Other (explain)

There is No Single Best Policy

- Depends on the goals of the system
- Different for…
  - Your personal computer
  - Large time-shared (super) computer
  - Computer controlling a nuclear power plant
- Often have multiple (conflicting) goals

Common Policies

- Details beyond scope of this course (Take OS)

- Different classes of processes
  - Those blessed by administrator (high/low priority)
  - Everything else

Common Policies

- Special class gets special treatment (varies)
- Everything else: *roughly* equal time quantum
  - "Round robin"
  - Give priority boost to processes that frequently perform I/O
  - Why?
- "I/O bound" processes frequently block.
  - If we want them to get equal CPU time, we need to give them the CPU more often.

Linux's Policy

(You're not responsible for this.)

- Special "real time" process classes (high prio)
- Other processes:
  - Keep red-black BST of process, organized by how much CPU time they've received.
  - Pick the ready with process that has run for the shortest time thus far.
  - Run it, update it's CPU usage time, add to tree.
- Interactive processes: Usually blocked, low total run time, high priority.

Managing Processes

- Processes created by calling `fork()`
  - "Spawning" a new process
- "Parent" process spawns "Child" process
  - Brutal relationship involving "zombies", "killing" and "reaping". (I'm not making this up!)
- Processes interact with one another by sending signals.

Managing Processes

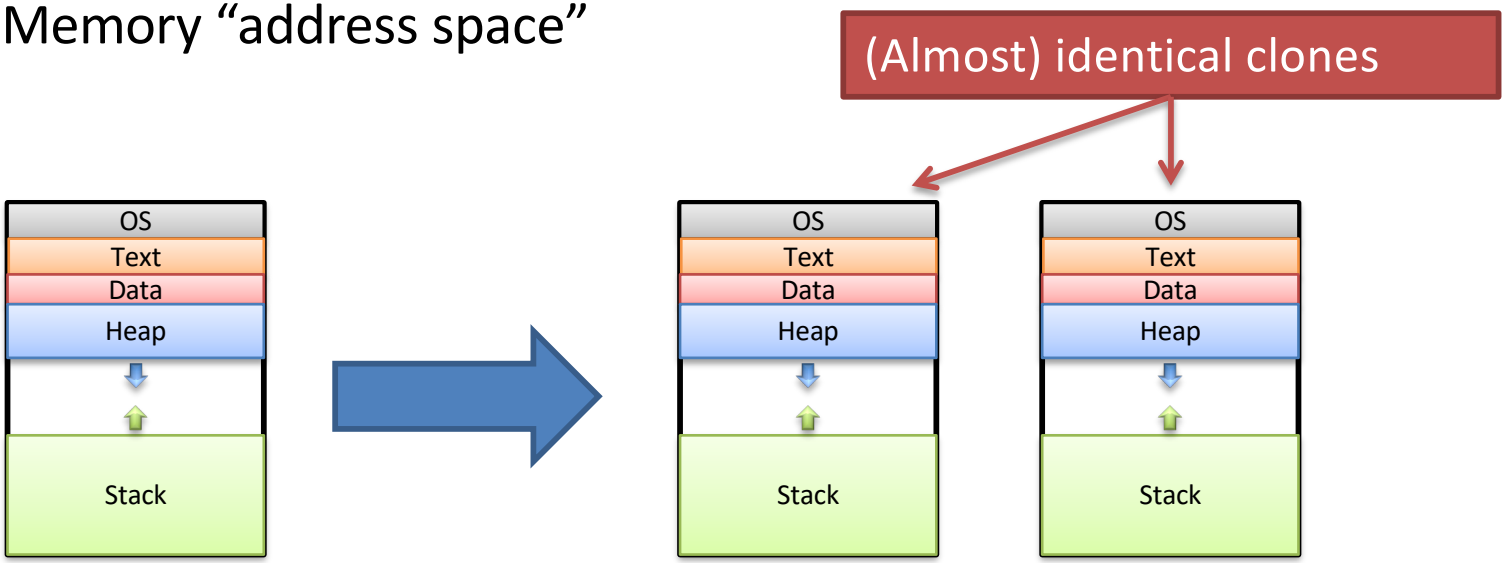
- Given a process, how do we make it execute the program we want?

- Model: fork() a new process, execute program

`fork()`

- System call (function provided by OS kernel)

- Creates a duplicate of the requesting process
  - Process is cloning itself:
    - CPU context
    - Memory “address space”



`fork()` return value

- The two processes are identical in every way, except for the return value of `fork()`.
  - The child gets a return value of 0.
  - The parent gets a return value of child's PID.

```
pid_t pid = fork(); // both continue after call
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Which process executes next? Child? Parent? Some other process?

Up to OS to decide. No guarantees. Don't rely on particular behavior!

How many hello's will be printed?

```
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
```

A. 6

B. 8

C. 12

D. 16

E. 18

How many hello's will be printed?

```
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
```

A. 6

B. 8

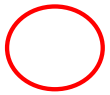
C. 12

D. 16

E. 18

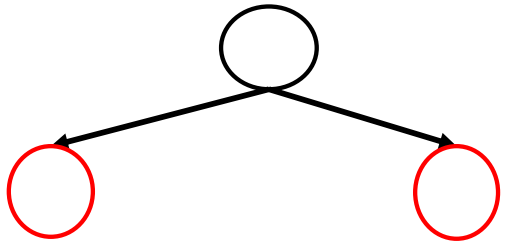
How many hello's will be printed?

```
int main(){
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
}
```



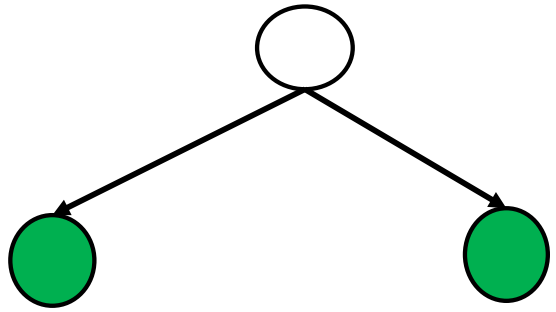
How many hello's will be printed?

```
int main(){
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
}
```



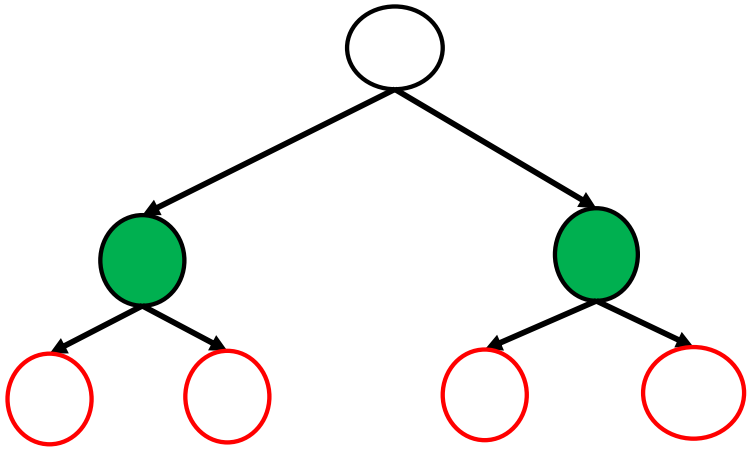
How many hello's will be printed?

```
int main(){
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
}
```



How many hello's will be printed?

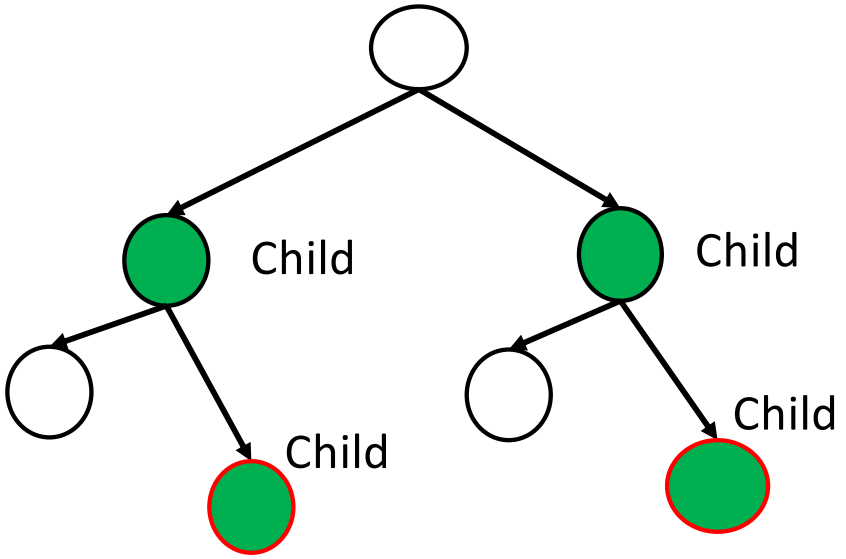
```
int main(){
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
}
```



Processes in green execute the print statement

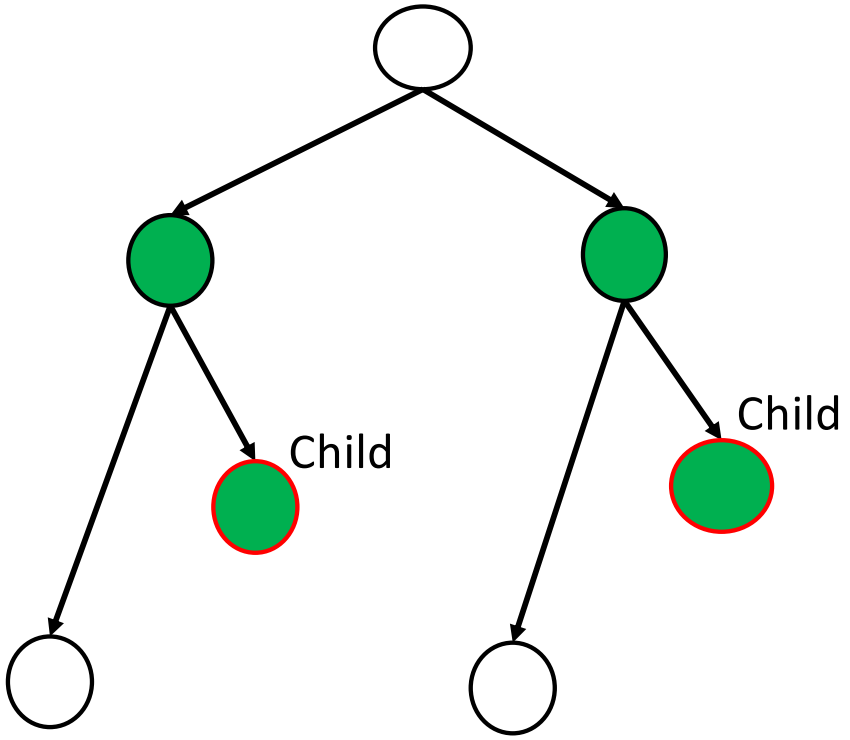
How many hello's will be printed?

```
int main(){
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
}
```



How many hello's will be printed?

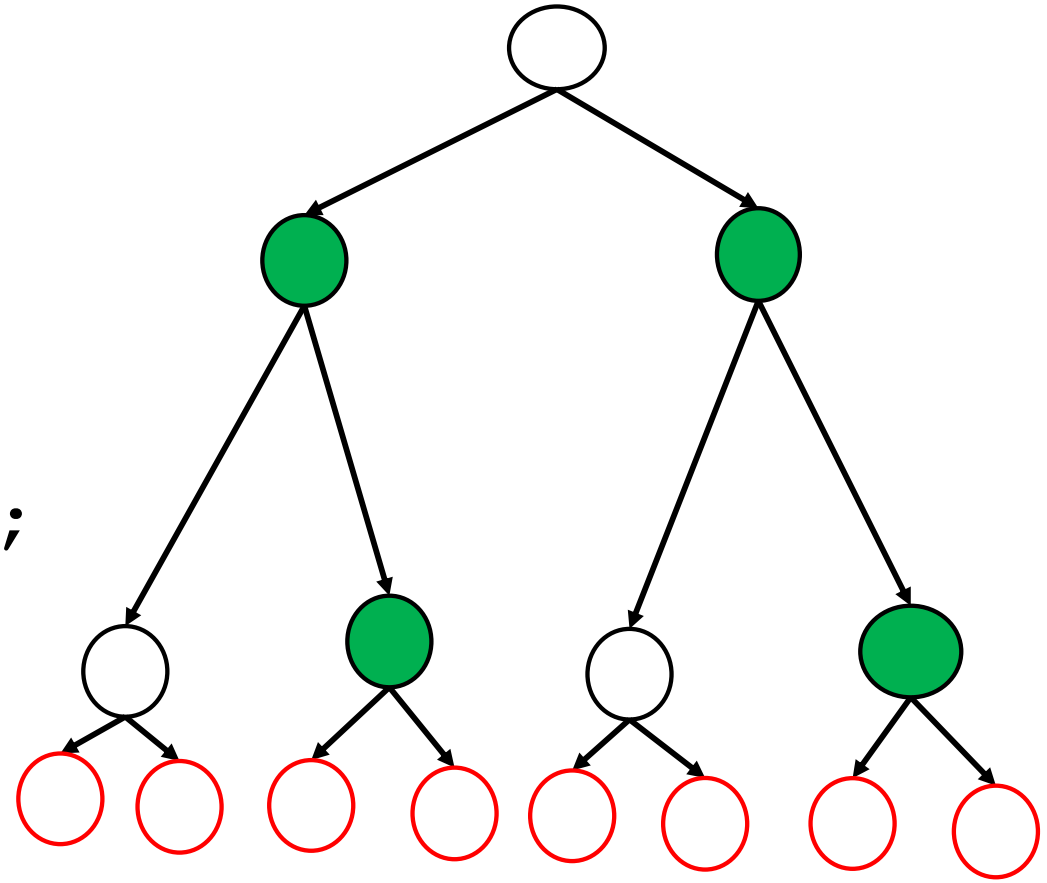
```
int main(){
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
}
```



Parent does not enter the if-condition block

How many hello's will be printed?

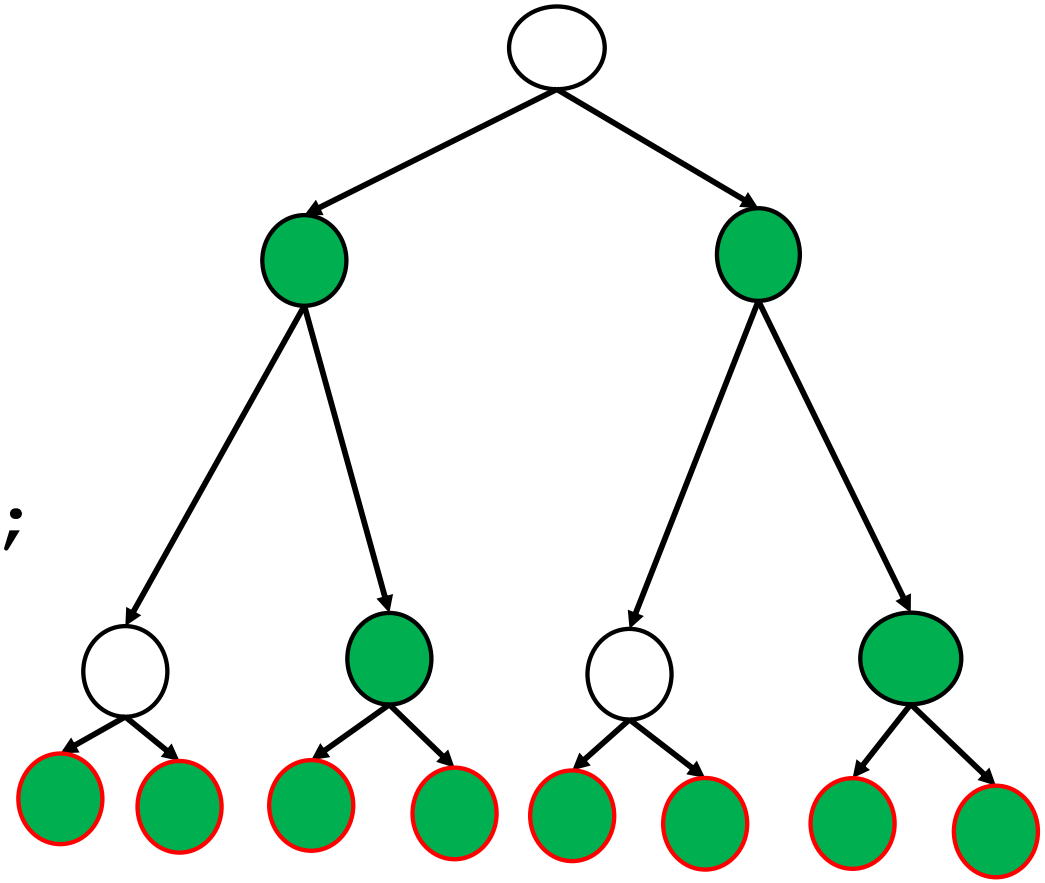
```
int main(){
fork();
printf("hello");
if (fork()) {
   printf("hello");
}
fork();
printf("hello");
}
```



Outside the if-condition, all the processes execute the fork call.

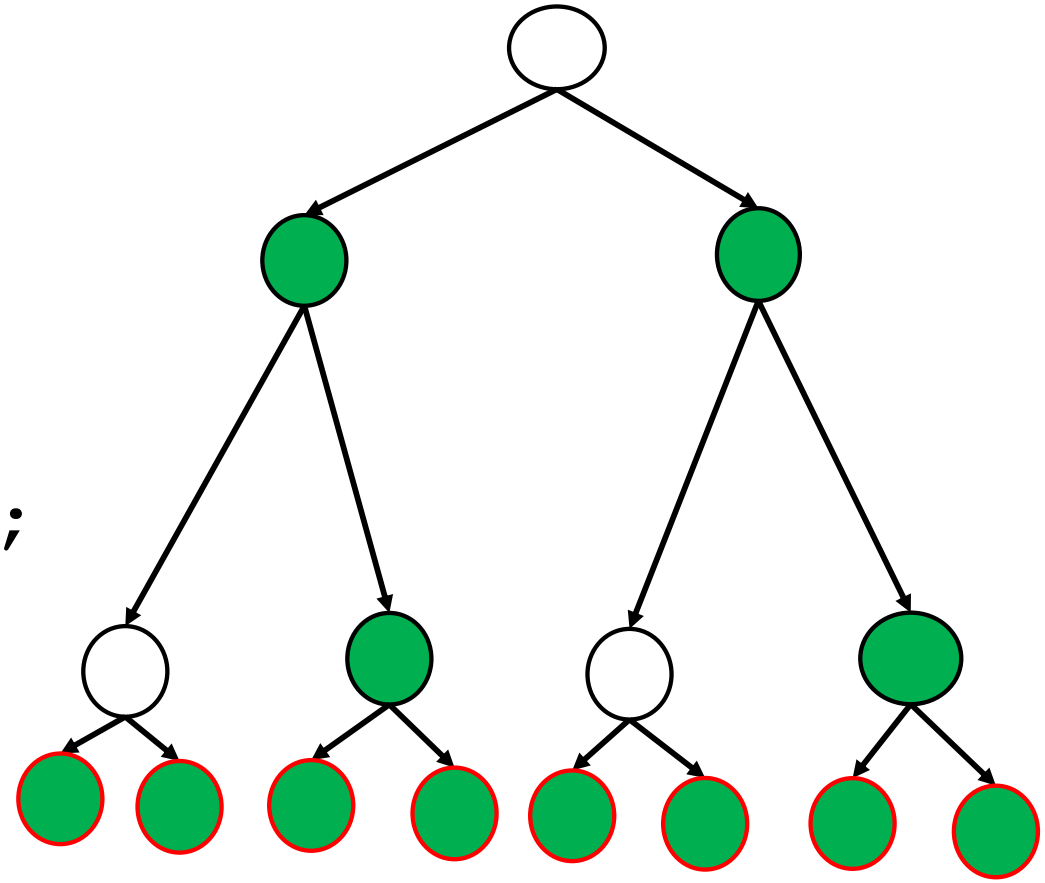
How many hello's will be printed?

```
int main(){
fork();
printf("hello");
if (fork()) {
  printf("hello");
}
fork();
printf("hello");
}
```



How many hello's will be printed?

```
int main(){
fork();
printf("hello");
if (fork()) {
   printf("hello");
}
fork();
printf("hello");
}
```



Total of 12 print statements

Common `fork()` usage: Shell

- A “shell” is the program controlling your terminal (e.g., bash).
- It `fork()`’s to create new processes, but we don’t want a clone (another shell).
- We want the child to execute some other program: `exec()` family of functions.

exec()

- Family of functions (execl, execlp, execv, …).
- Replace the current process with a new one.
- Loads program from disk:
  - Old process is overwritten in memory.
  - Does not return unless error.

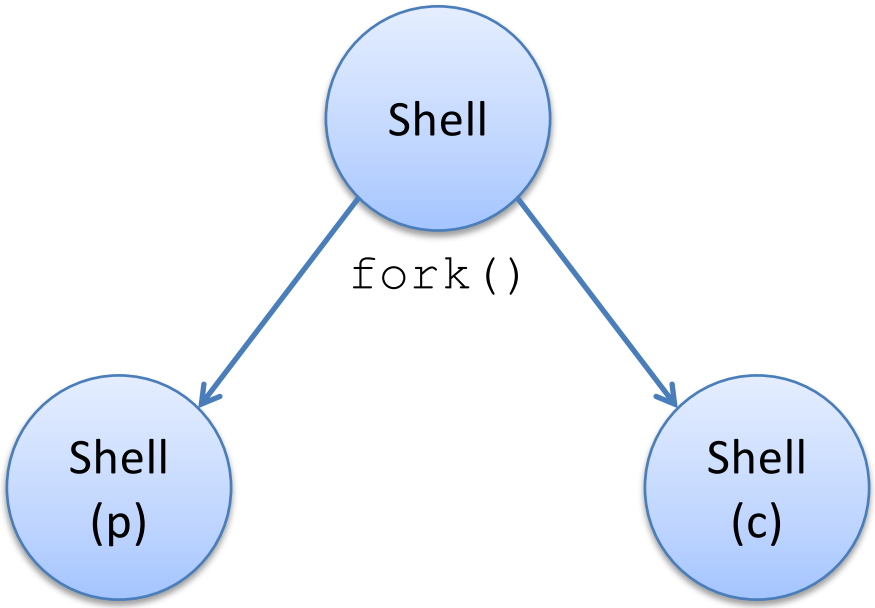
Common `fork()` usage: Shell

1. `fork()` child process.
2. `exec()` desired program to replace child's address space.
2. `wait()` for child process to terminate.
3. repeat…

The parent and child each do something different next.

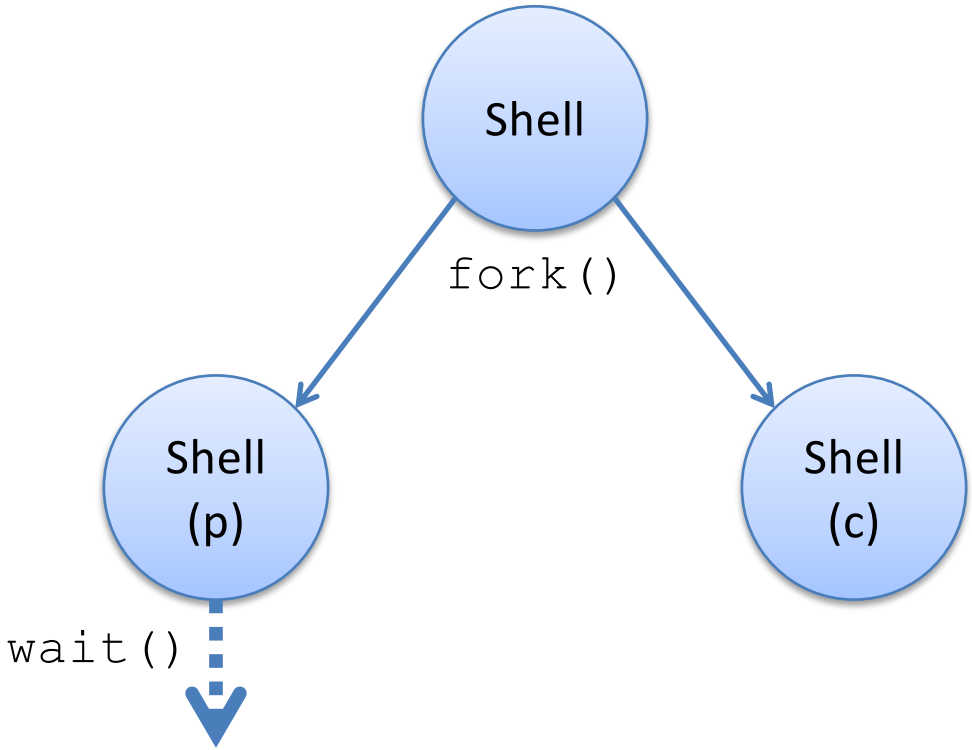
Common `fork()` usage: Shell

1. `fork()` child process.



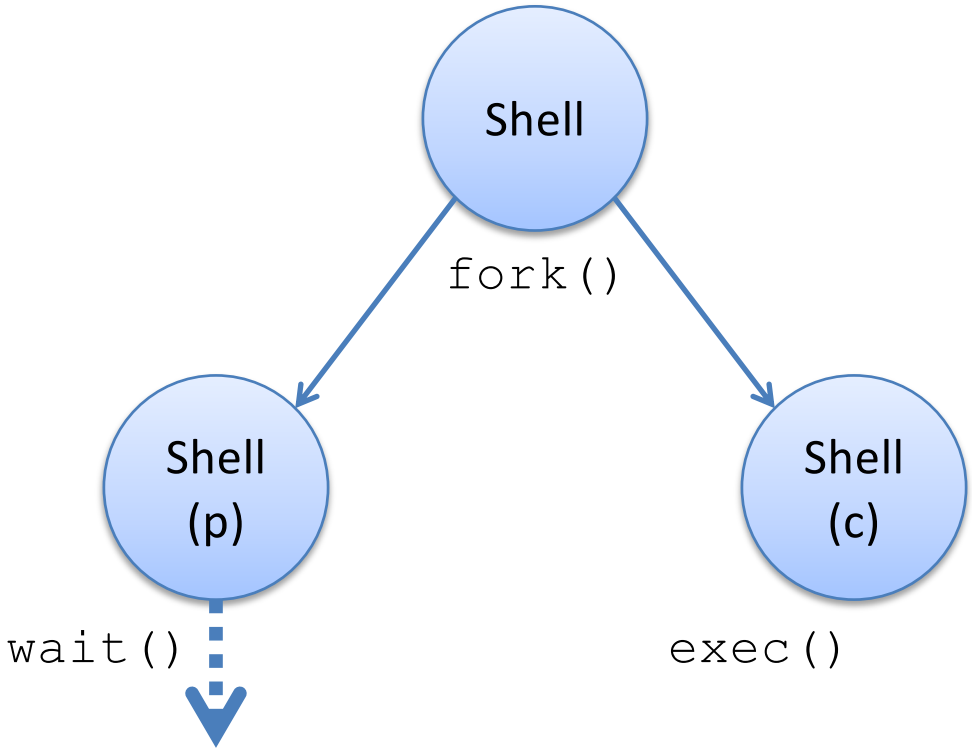
Common `fork()` usage: Shell

2. parent: `wait()` for child to finish



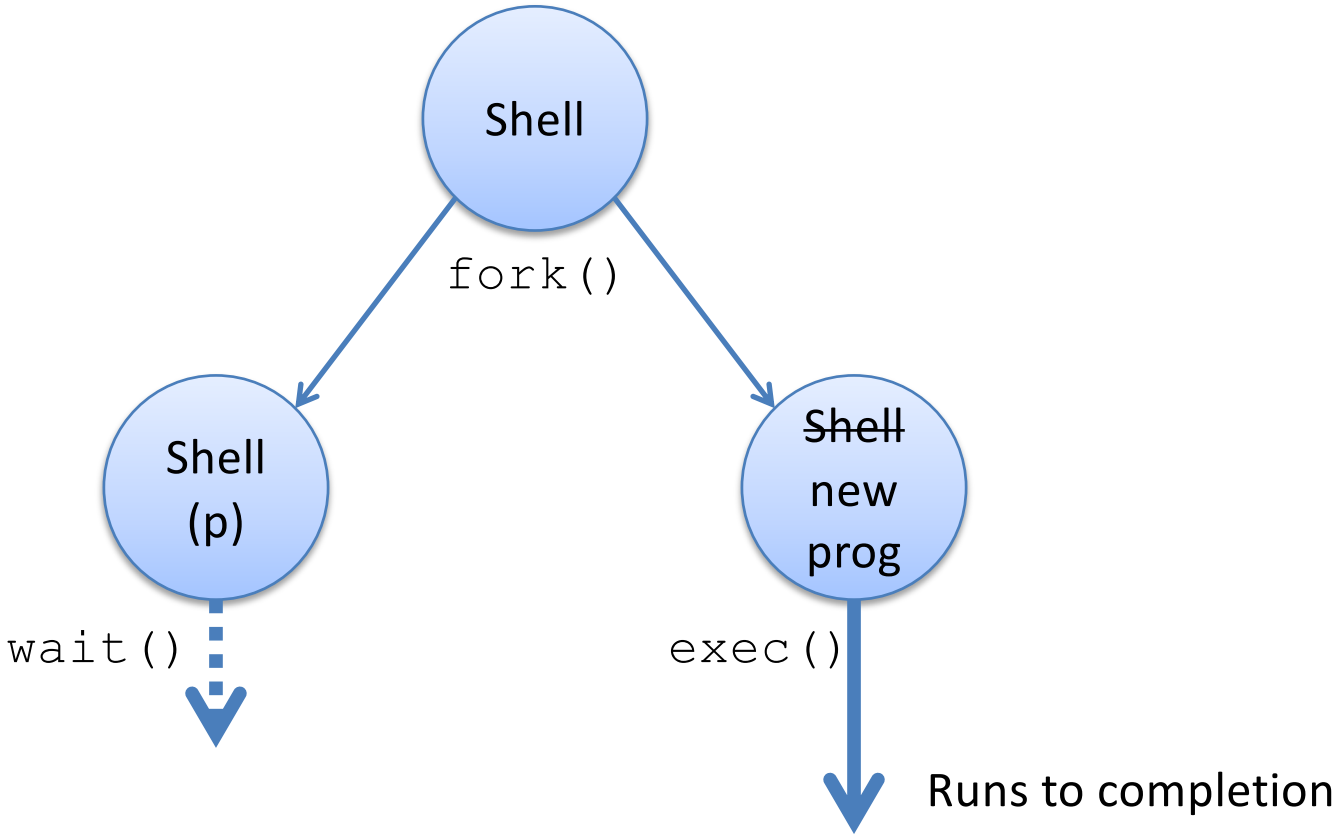
Common `fork()` usage: Shell

2. child: `exec()` user-requested program



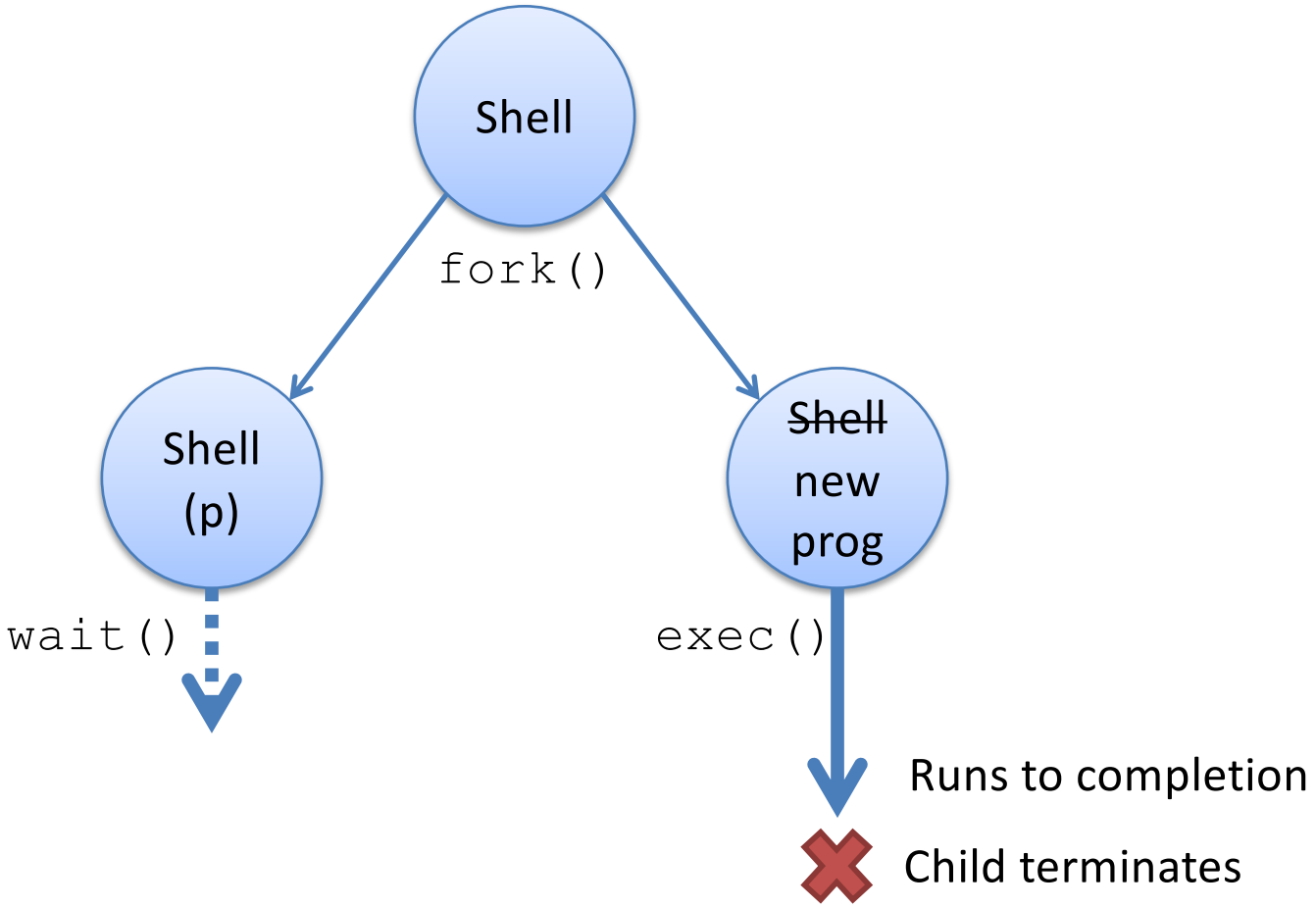
Common `fork()` usage: Shell

2. child: `exec()` user-requested program



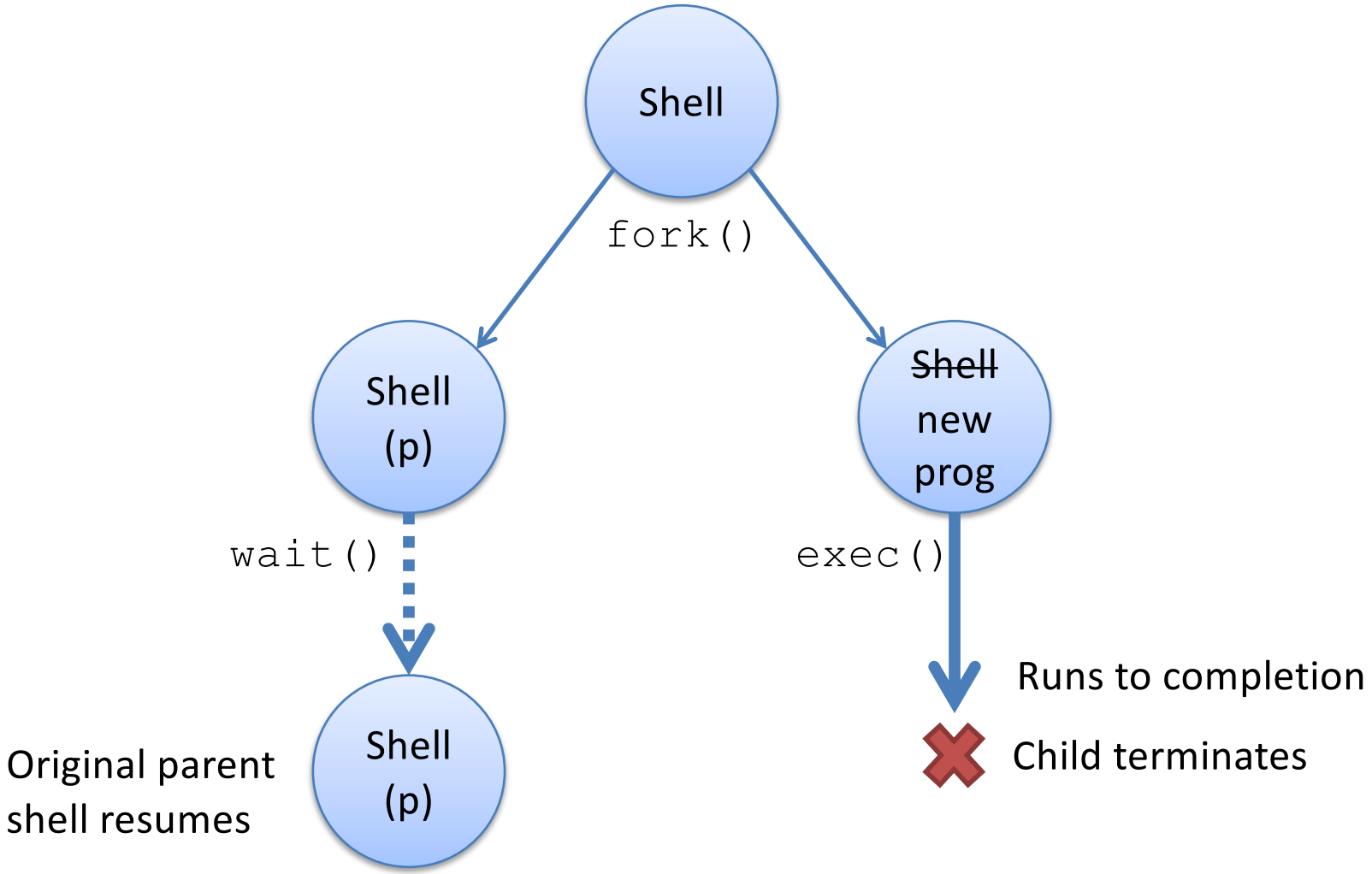
Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Process Termination

- When does a process die?
  - It calls `exit(int status);`
  - It `returns` (an int) from main
  - It receives a termination signal (from the OS or another process)

- Key observation: the dying process *produces status information*.

- Who looks at this?
- The parent process!

Reaping Children

(Bet you didn't expect to see THAT title on a slide when you signed up for CS 31?)

- `wait()`: parents reap their dead children
  - Given info about why child died, exit status, etc.

- Two variants:
  - wait(): wait for and reap next child to exit
  - waitpid(): wait for and reap specific child

- This is how the shell determines whether or not the program you executed succeeded.

Common `fork()` usage: Shell

1. `fork()` child process.

2. `exec()` desired program to replace child's address space.

3. `wait()` for child process to terminate.
   - Check child's result, notify user of errors.

4. repeat…

"Zombie" Processes

- Zombie: A process that has terminated but not been reaped by parent. (AKA defunct process)
- Does not respond to signals (can't be killed)
- OS keeps their entry in process table:
  - Parent may still reap them, want to know status
  - Don't want to re-use the process ID yet

Basically, they're kept around for bookkeeping purposes, but that's much less exciting...

Signals

- How does a parent process know that a child has exited (and that it needs to call wait)?

- Signals: inter-process notification mechanism
  - Info that a process (or OS) can send to a process.
    - Please terminate yourself (SIGTERM)
    - Stop NOW (SIGKILL)
    - Your child has exited (SIGCHLD)
    - You've accessed an invalid memory address (SIGSEGV)
    - Many more (SIGWINCH, SIGUSR1, SIGPIPE, …)

Signal Handlers

- By default, processes react to signals according to the signal type:
  - SIGKILL, SIGSEGV, (others): process terminates
  - SIGCHLD, SIGUSR1: process ignores signal
- You can define "signal handler" functions that execute upon receiving a signal.
  - Drop what program was doing, execute handler, go back to what it was doing.
  - Example: got a SIGCHLD? Enter handler, call `wait()`
  - Example: got a SIGUSR1? Reopen log files.
- Some signals (e.g., SIGKILL) cannot be handled.

Summary

- Processes cycled off and on CPU rapidly
  - Mechanism: context switch
  - Policy: CPU scheduling
- Processes created by `fork()`ing
- Other functions to manage processes:
  - exec(): replace address space with new program
  - exit(): terminate process
  - wait(): reap child process, get status info
- Signals one mechanism to notify a process of something