CS 31: Introduction to Computer Systems

19-20: Operating Systems & Processes

April 14-16, 2020

SWARTHMORE COLLEGE

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?

Key OS Responsibilities

1. Hardware gatekeeping and protection

2. Simplifying abstractions for programs (e.g., files)

3. Resource sharing (memory, CPU)

The Kernel

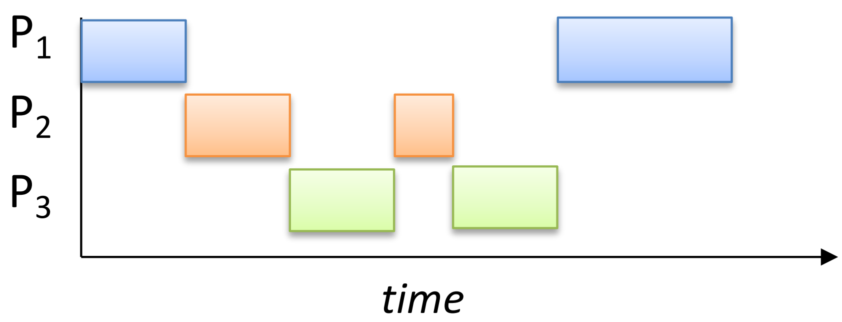
- All programs depend on it
  - Loads and runs them
  - Exports system calls to programs
- Works closely with hardware
  - Accesses devices
  - Responds to interrupts
- Allocates basic resources
  - CPU time, memory space
  - Controls I/O devices: display, keyboard, disk, network

OS: Turn undesirable into desirable

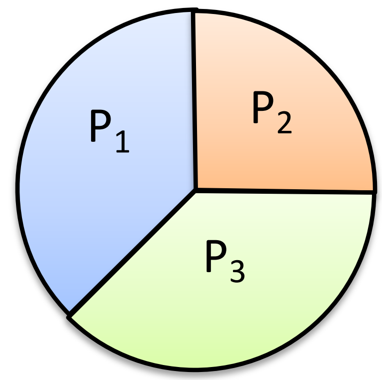
- Turn undesirable inconveniences: reality
  - Complexity of hardware
  - Single processor
  - Limited memory
- Into desirable conveniences: illusions
  - Simple, easy-to-use resources
  - Multiple/unlimited number of processors
  - Large/unlimited amount of memory

Resource Sharing

CPU: Time



Memory: Space



Reality

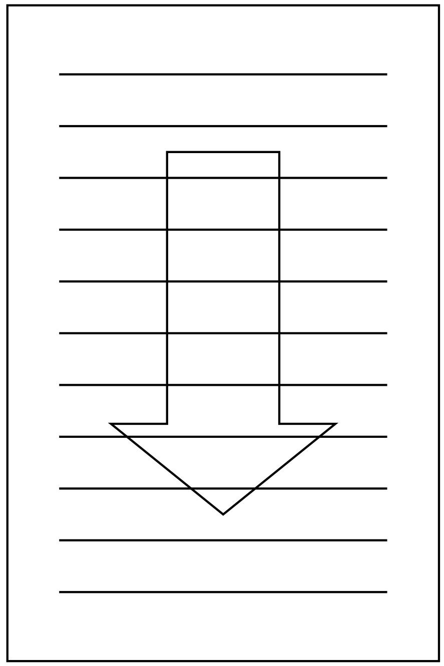
- Multiple processes
- Small number of CPUs
- Finite memory

Abstraction

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

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



Managing Processes

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

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

In creating a process, the fork() function…

A. is called <u>once</u> and returns <u>once</u>.

B. is called <u>twice</u> and returns <u>once</u>.

C. is called <u>once</u> and returns <u>twice</u>.

D. is called <u>twice</u> and returns <u>twice</u>.

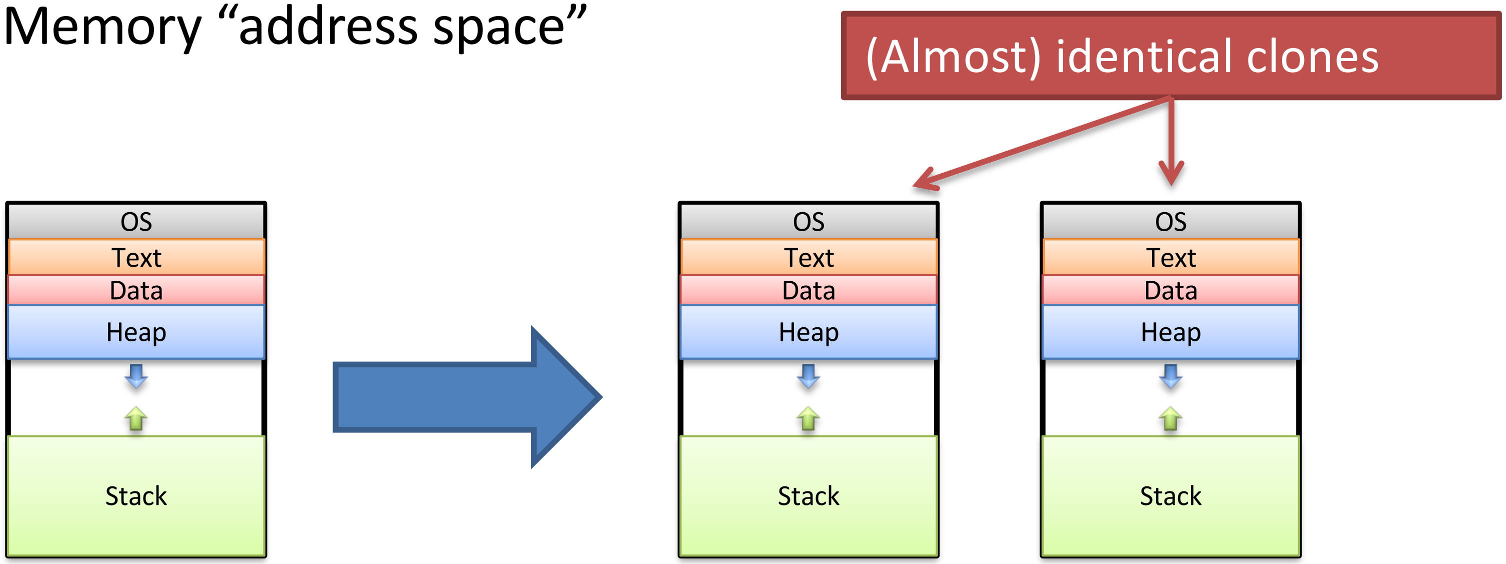
Creating a Process

- One process can create other processes to do work.
  - The creator is called the parent and the new process is the child
  - The parent defines (or donates) resources and privileges to its children
  - A parent can either wait for the child to complete, or continue in parallel

`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.

```
1.pid_t pid = fork(); // both continue after call
2. printf("A") //P&C
if (pid == 0) { //P &C
   printf("hello from child\n"); → Child
} else {  → parent
   pid_t pid_2 = fork();
   printf("hello from parent\n");
```

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

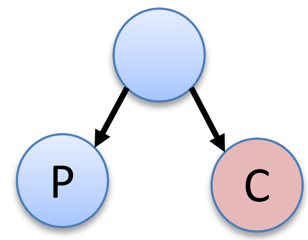
Up to OS to decide. No guarantees. Don't rely on particular behavior!

How many hello's will be printed?

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

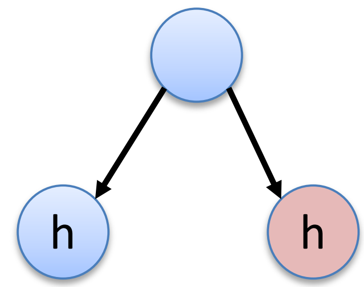
A. 6
B. 8
C. 12
D. 16
E. 18

How many hello's will be printed?




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

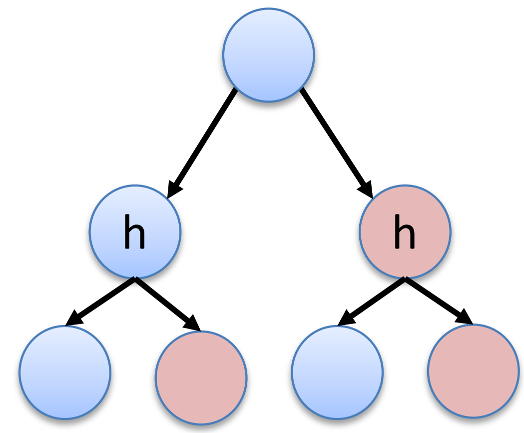
How many hello's will be printed?




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

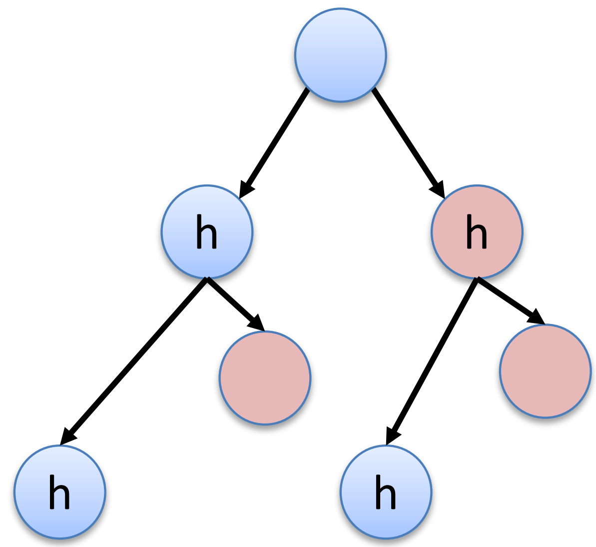
How many hello's will be printed?

```
fork();
printf("hello");
if (fork()) {//child=0
  printf("hello");
}
fork();
printf("hello");
```



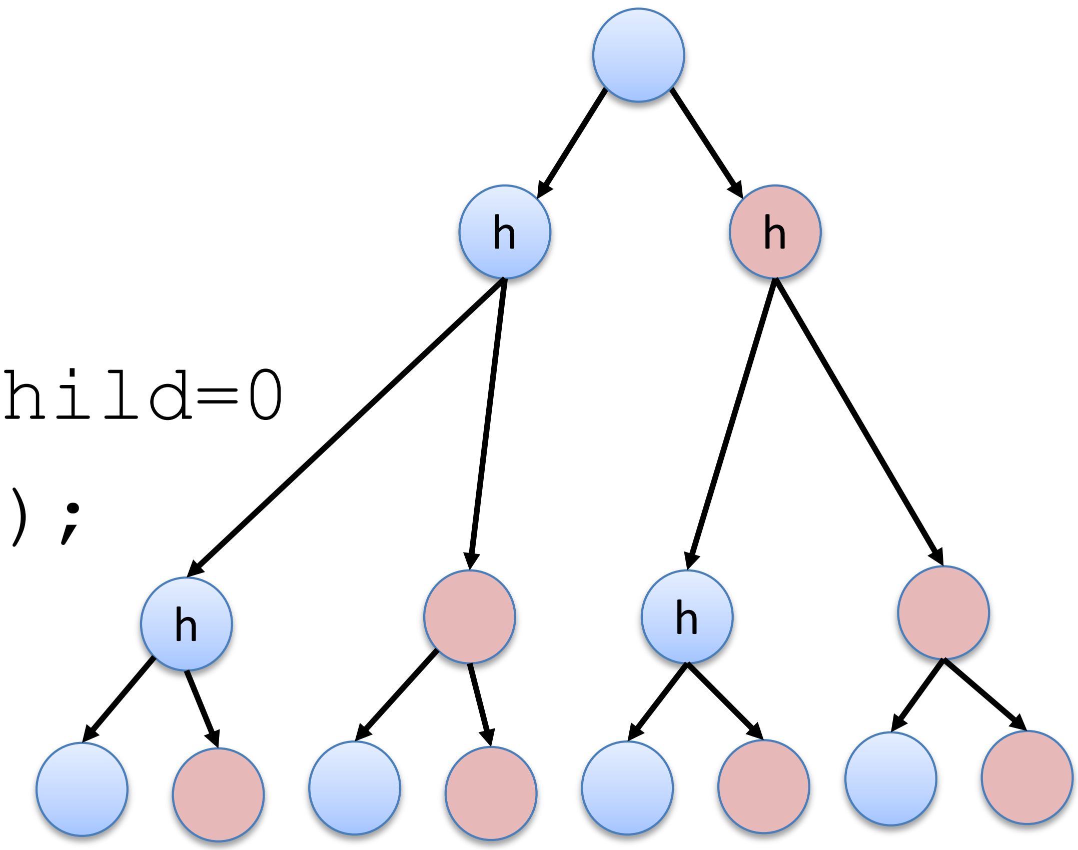
How many hello's will be printed?

```
fork();
printf("hello");
if (fork()) {//child=0
  printf("hello");
}
fork();
printf("hello");
```



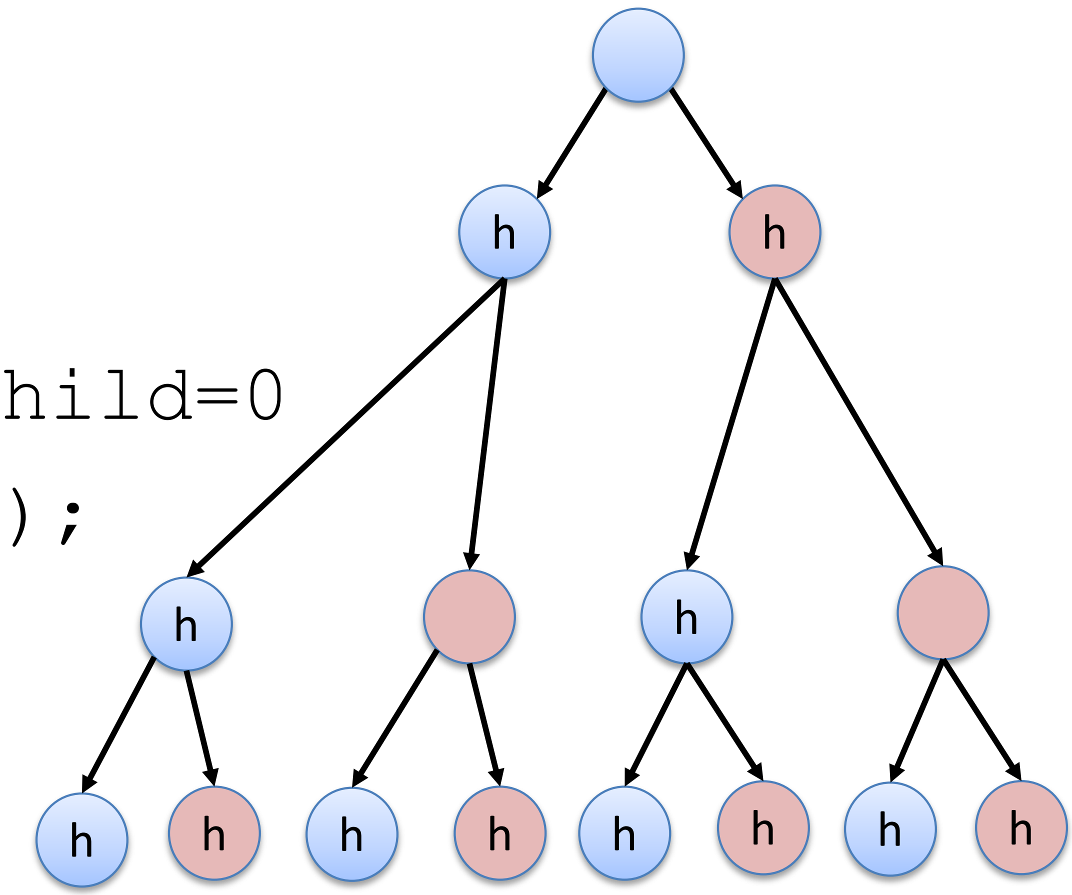
How many hello's will be printed?

```
fork();
printf("hello");
if (fork()) {//child=0
  printf("hello");
}
fork();
printf("hello");
```



How many hello's will be printed?

```
fork();
printf("hello");
if (fork()) {//child=0
  printf("hello");
}
fork();
printf("hello");
```



Print statements = 12

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.

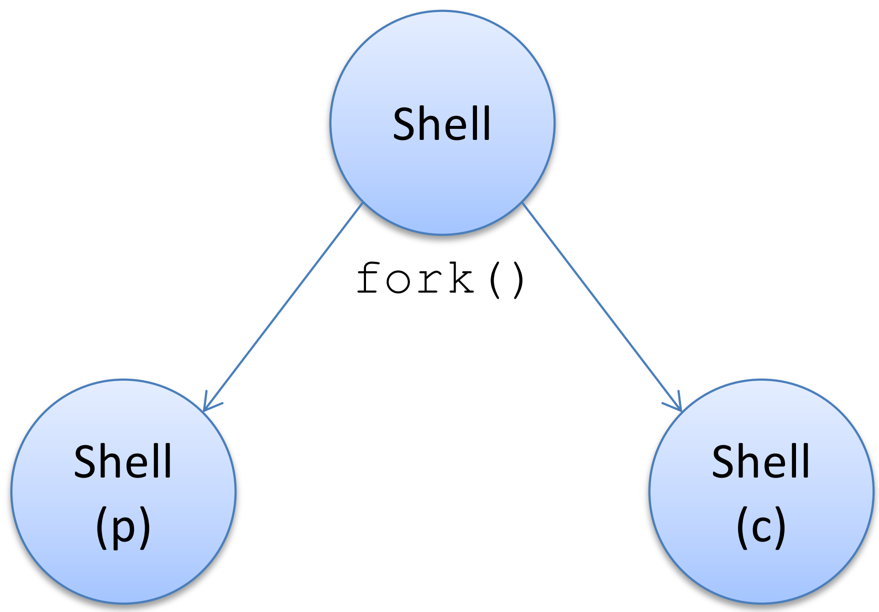
The parent and child each do something different next.

2. `wait()` for child process to terminate.

3. repeat...

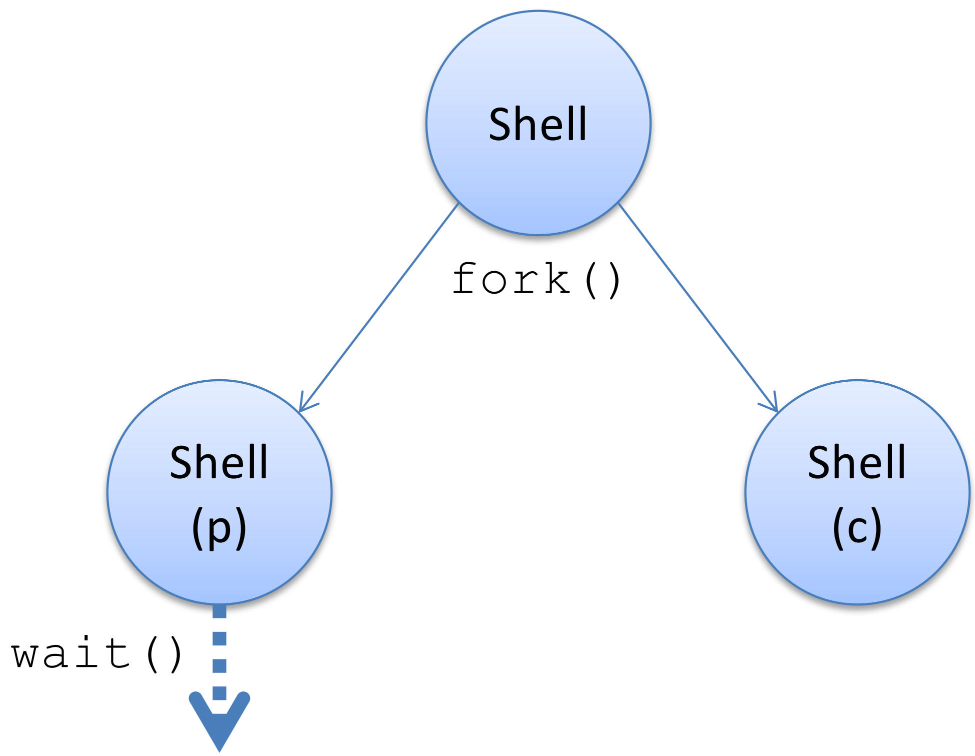
Common `fork()` usage: Shell

1. `fork()` child process.



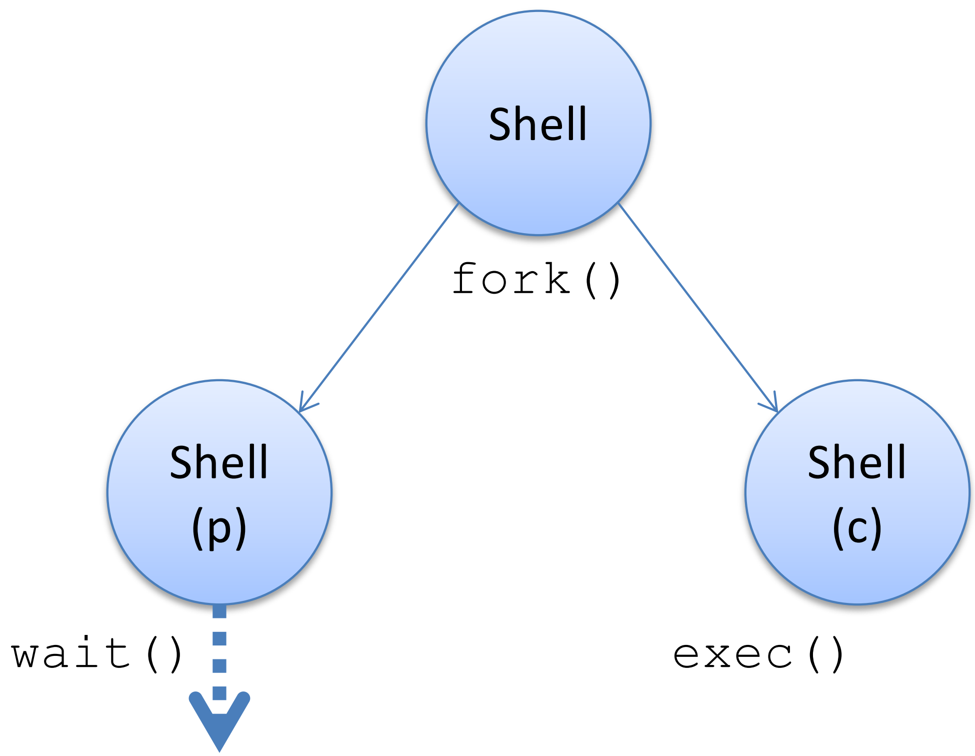
Common `fork()` usage: Shell

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



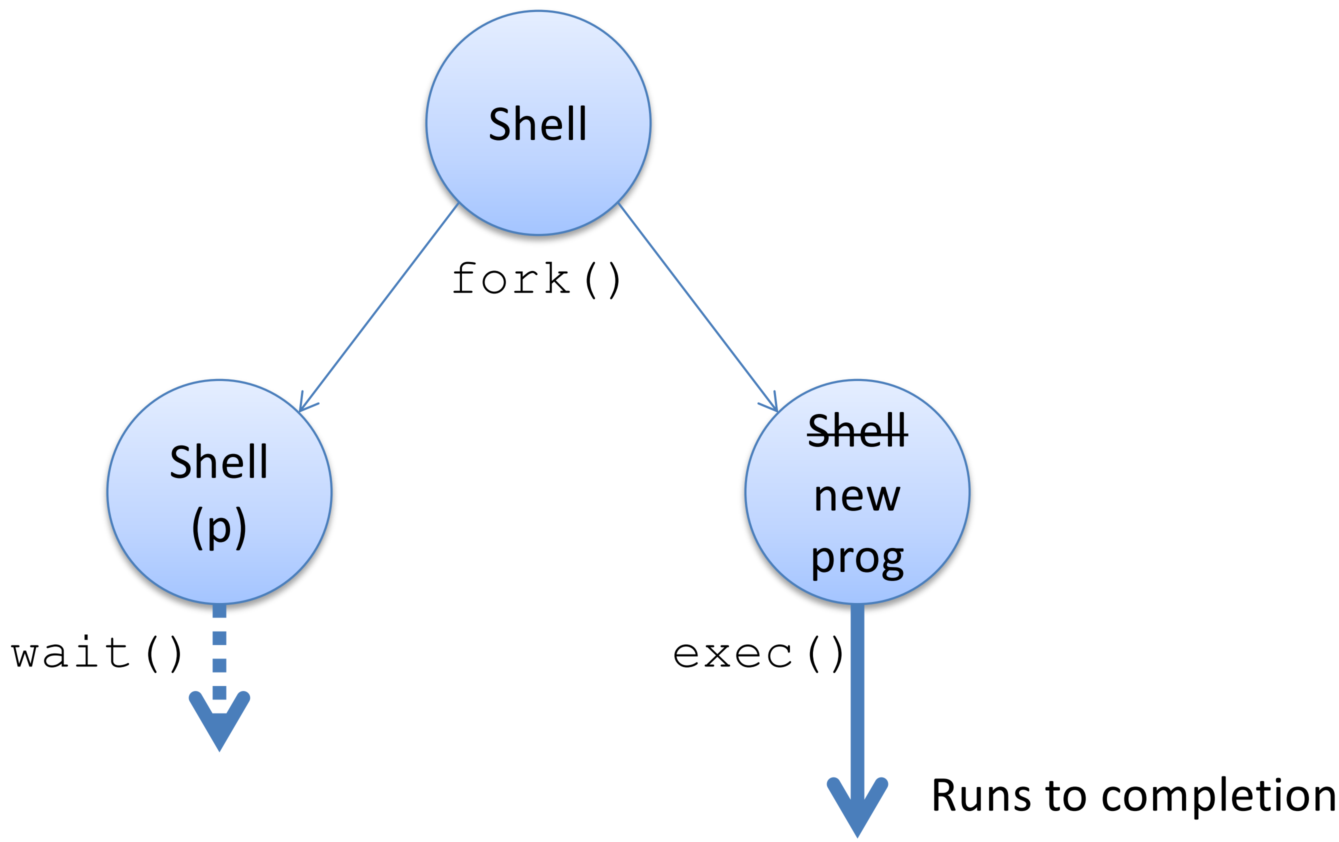
Common `fork()` usage: Shell

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



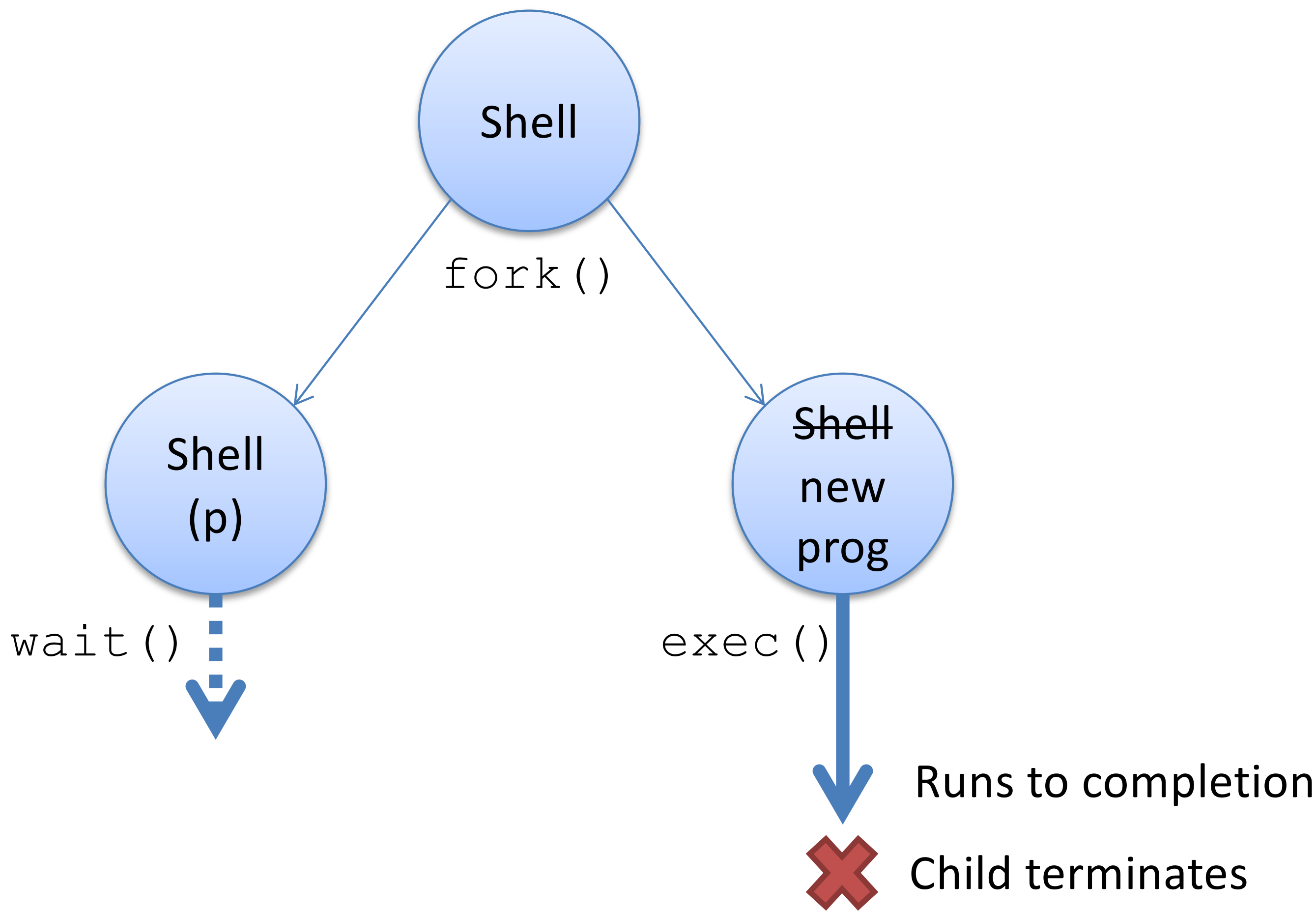
Common `fork()` usage: Shell

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



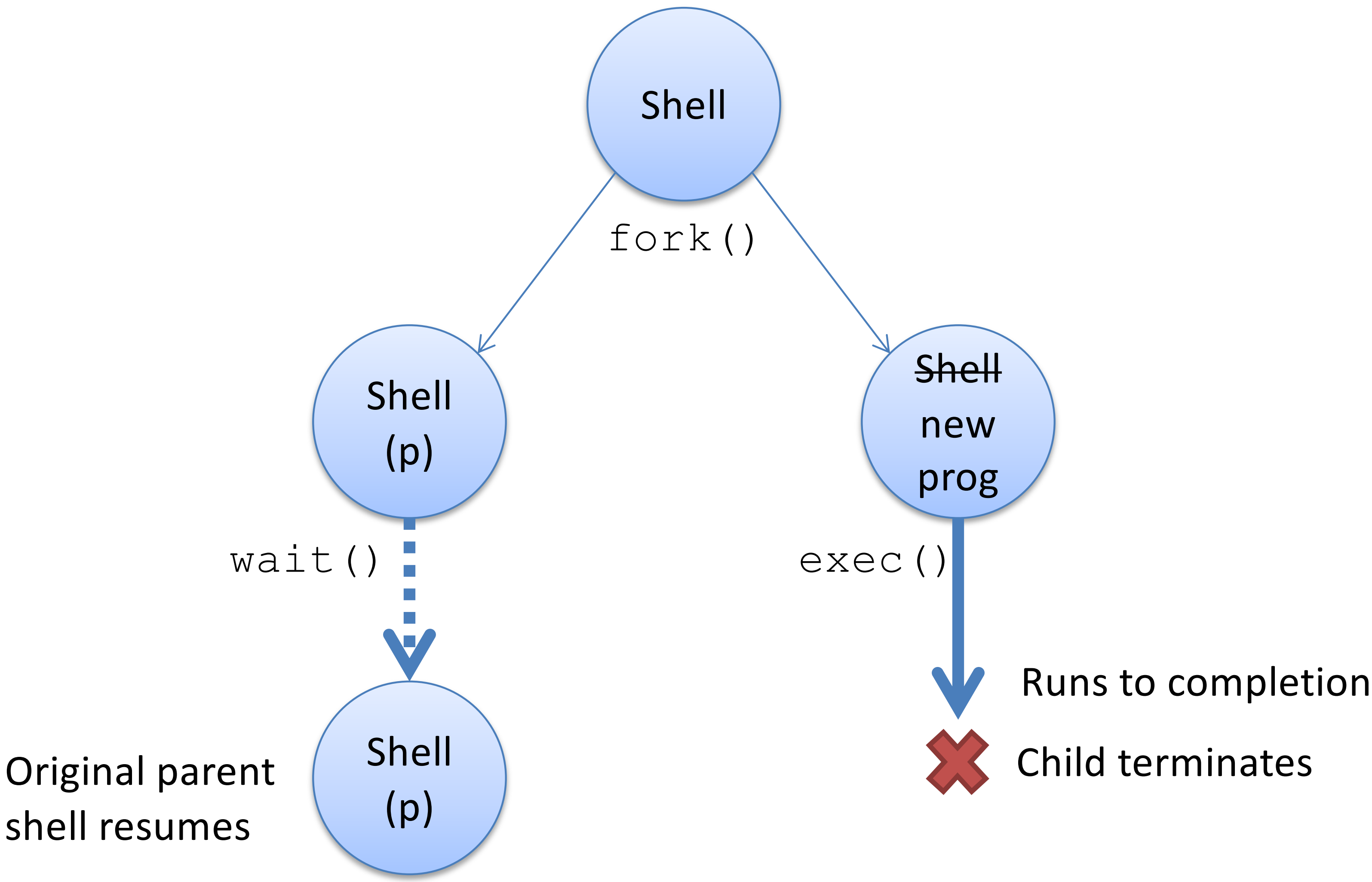
Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Process Termination

- On process termination, the OS reclaims all resources assigned to the process.

- In Unix
  - a process can terminate itself using the exit system call.
  - a process can terminate a child using the kill system call.

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…

Recall: 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, …

- 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

What should happen if dead child processes are never reaped? (That is, the parent has not `wait`()ed on them?)

A. The OS should remove them from the process table (process control block / PCB).

B. The OS should leave them in the process table (process control block / PCB).

C. The neglected processes seek revenge as undead in the afterlife.

"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...

Process Management: Summary

- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS
  - PCBs contain process state, scheduling and memory management information, etc
- A process is either New, Ready, Waiting, Running, or Terminated.
- On a uniprocessor, there is at most one running process at a time.
- The program currently executing on the CPU is changed by performing a context switch
- Processes communicate either with message passing or shared memory

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.

Key OS Responsibilities

1. Simplifying abstractions for programs

2. Resource sharing

3. Hardware gatekeeping and protection

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.)

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

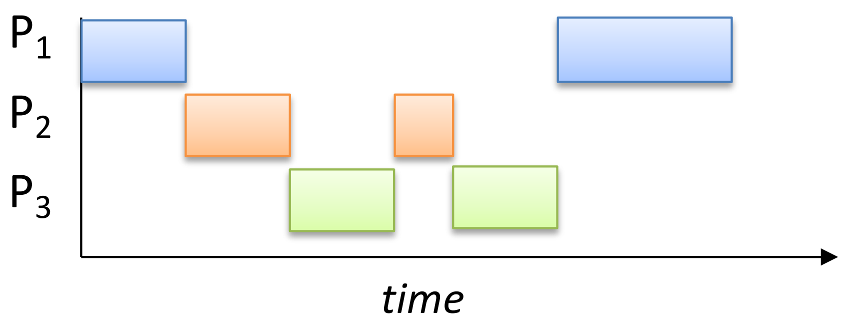
- Multiple virtual processors
  - By rapidly switching CPU use
- Multiple virtual memories
  - By memory partitioning and re-addressing
- Virtualized devices
  - By simplifying interfaces, and using other resources to enhance function

Kernel provides common functions

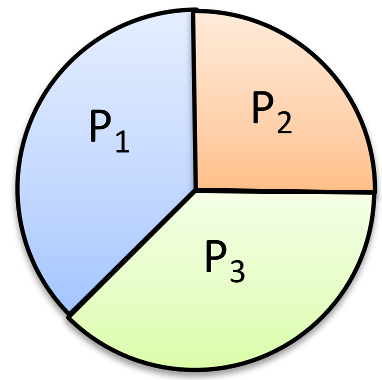
- Some functions useful to many programs
  - I/O device control
  - Memory allocation
- Place these functions in central place (kernel)
  - Called by programs (system calls)
  - Or accessed implicitly
- What should functions be?
  - How many programs should benefit?
  - Might kernel get too big?

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.

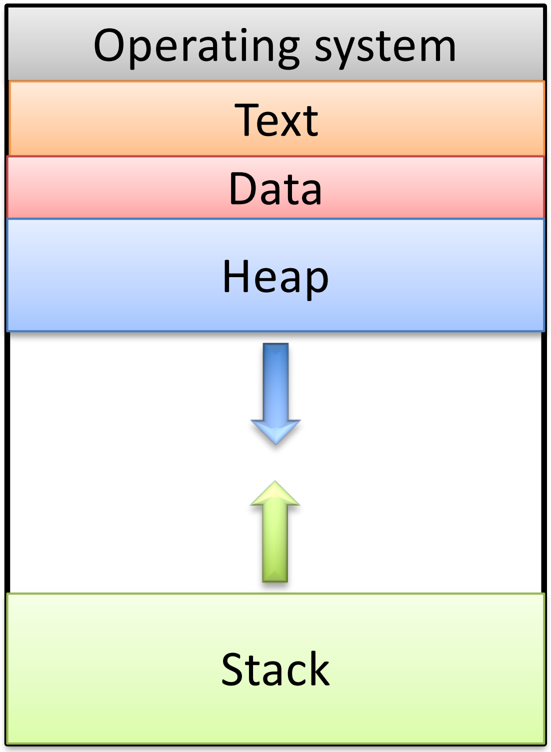
Timesharing: Sharing the CPUs

- 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, … (context switching)

Resource: Memory

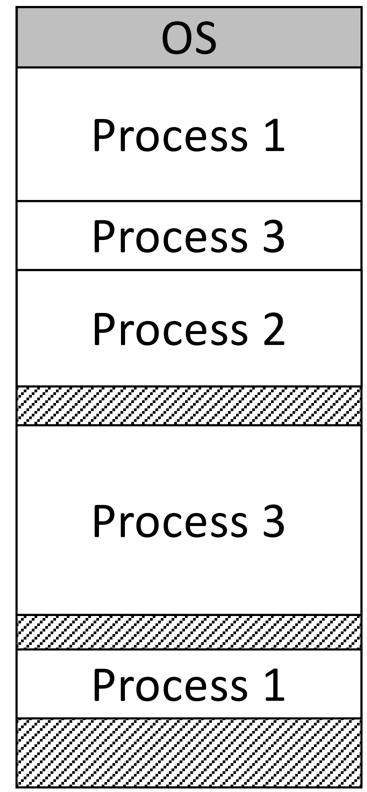
- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.



Memory

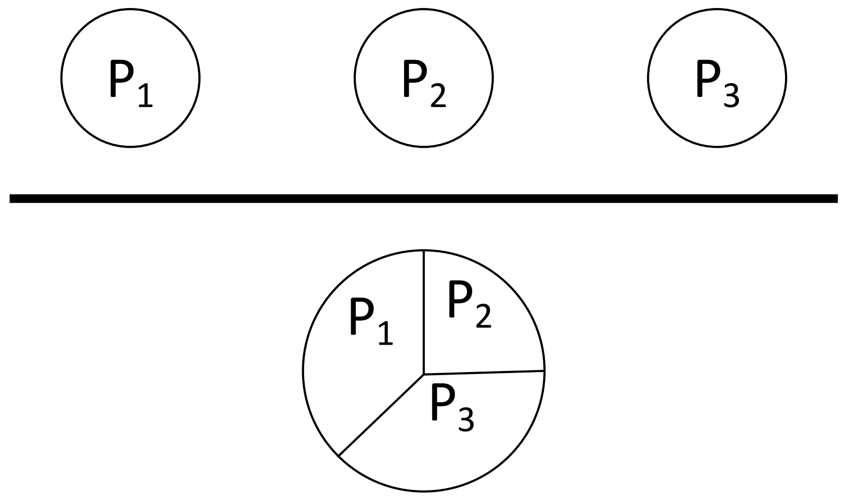
- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.

- 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.

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 Execution

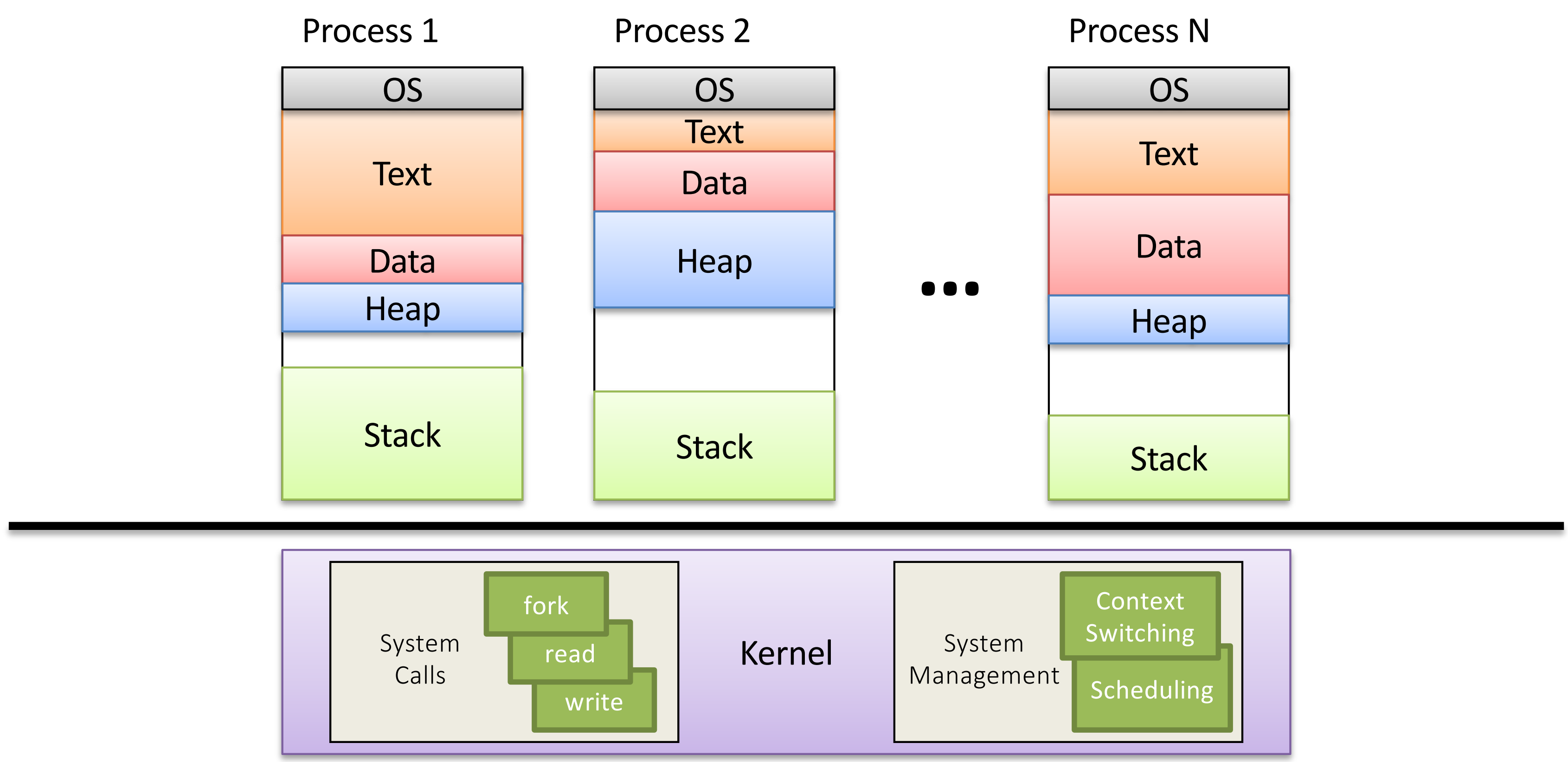
- Great, the OS is going to somehow give us these nice abstractions.

- So…how / when should the kernel execute to make all this stuff happen?

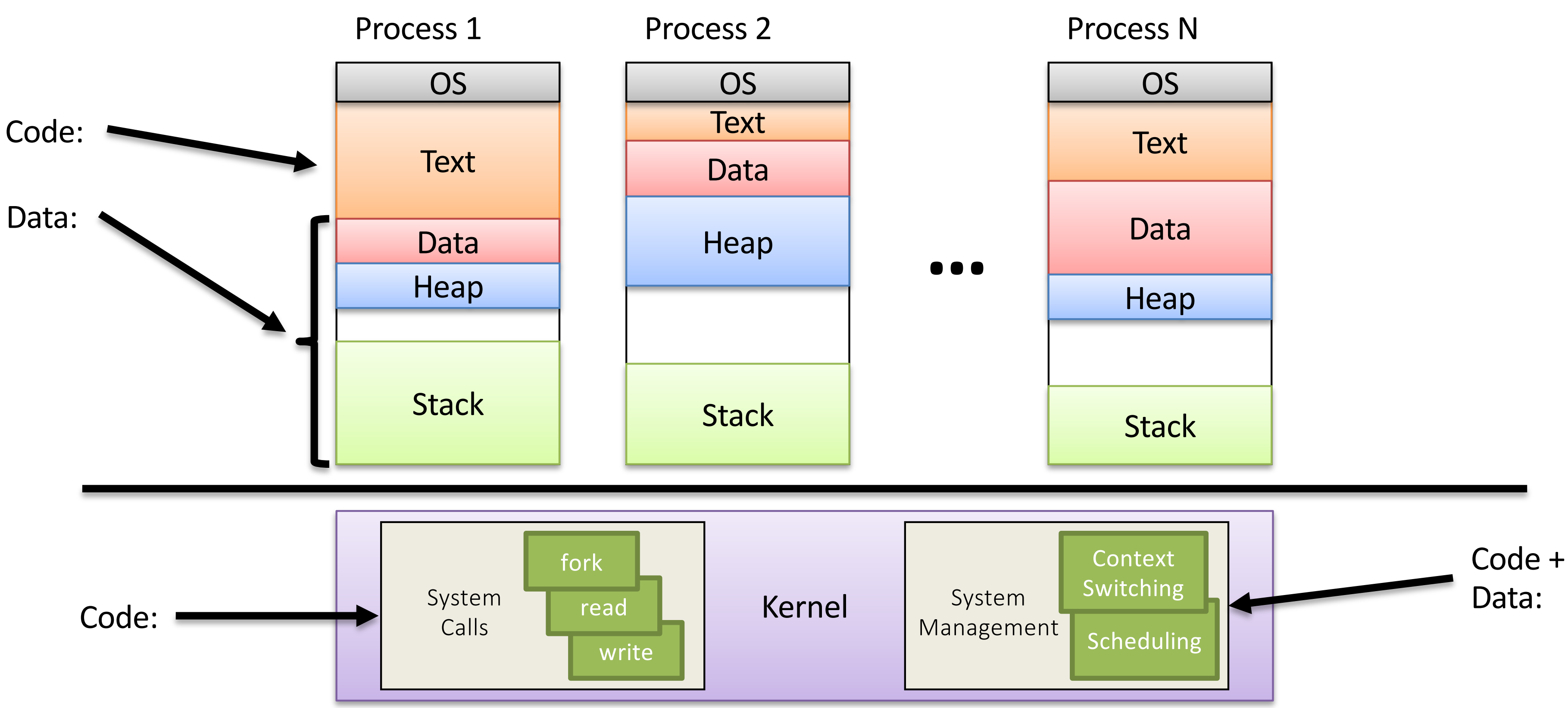
Process vs. Kernel

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

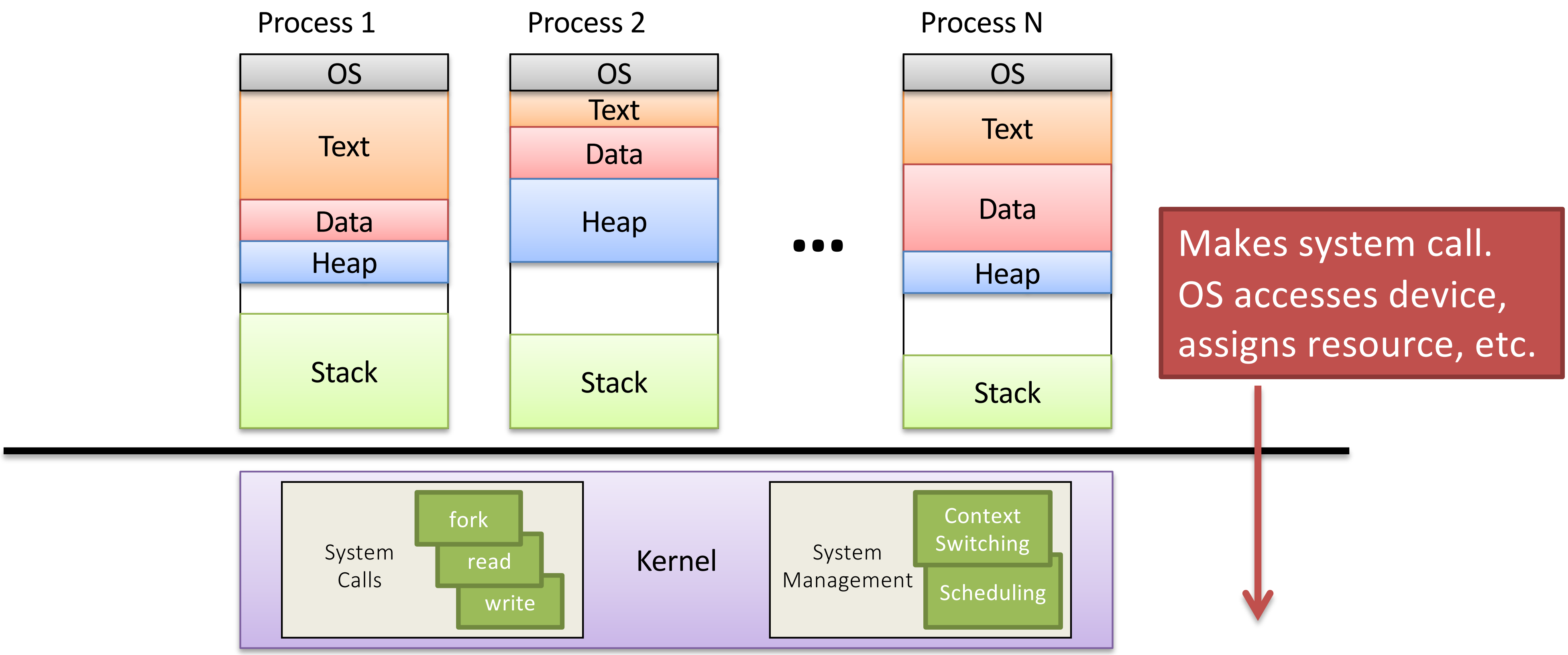
Kernel vs. Userspace: Model



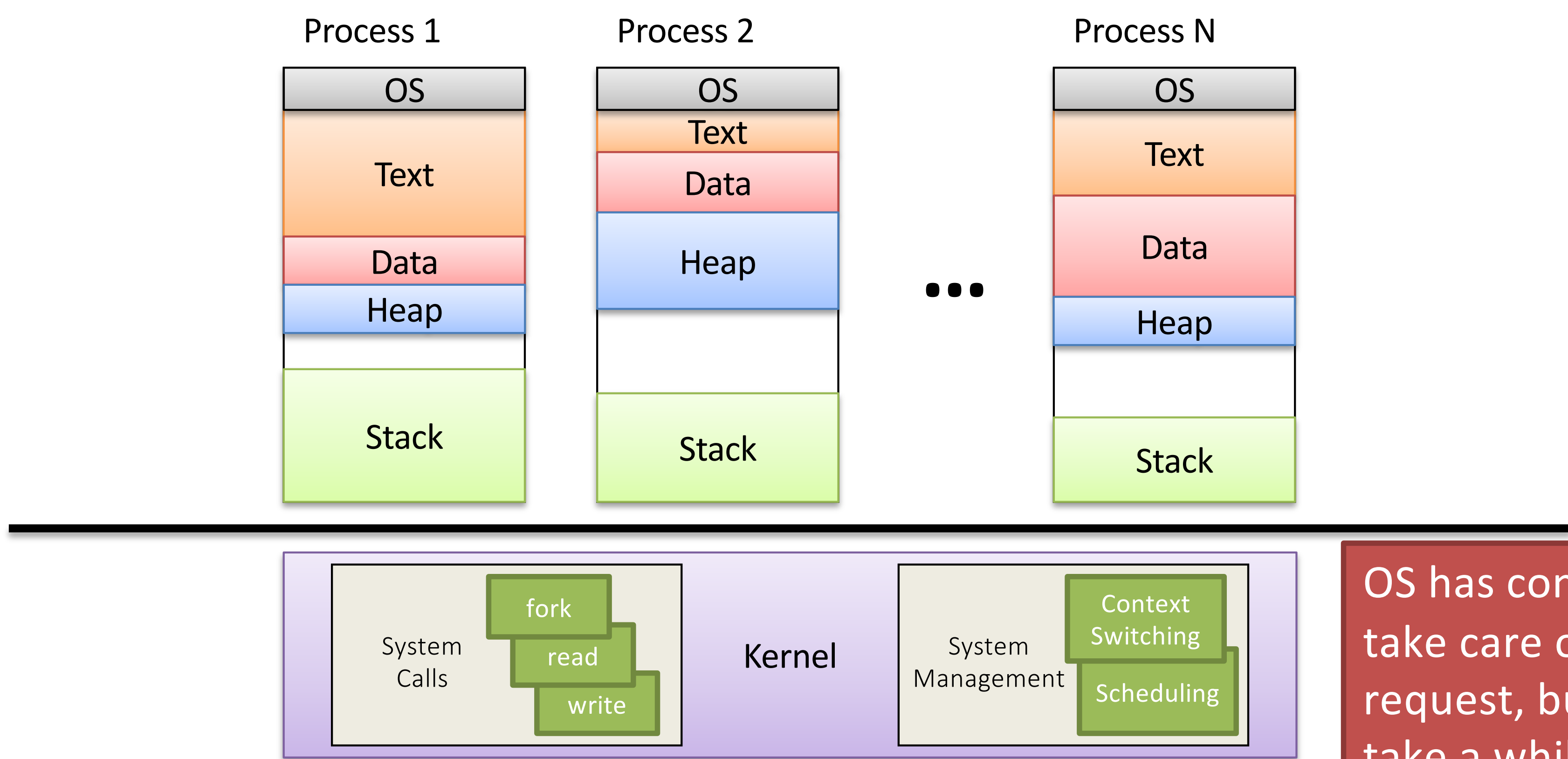
Kernel vs. Userspace: Model



Kernel vs. Userspace: Model



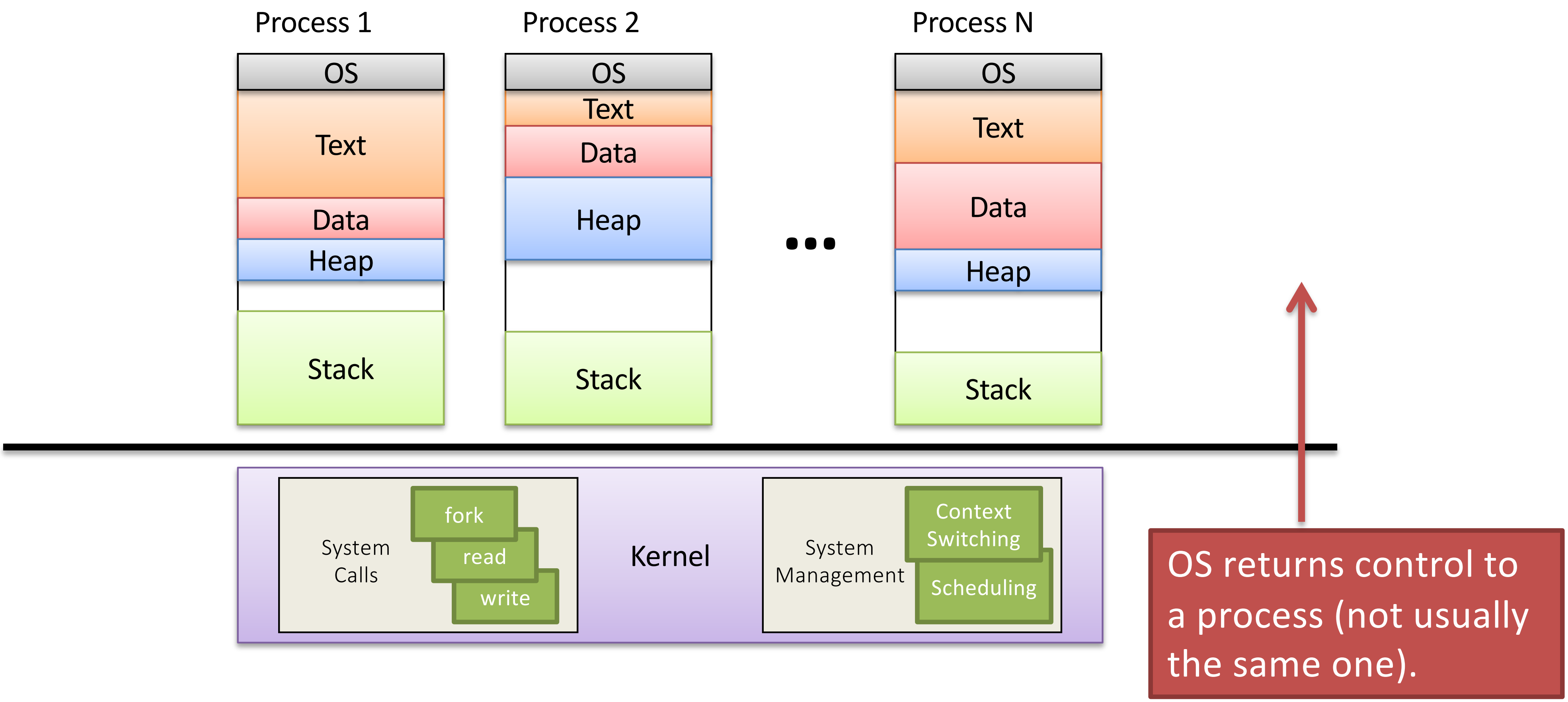
Kernel vs. Userspace: Model



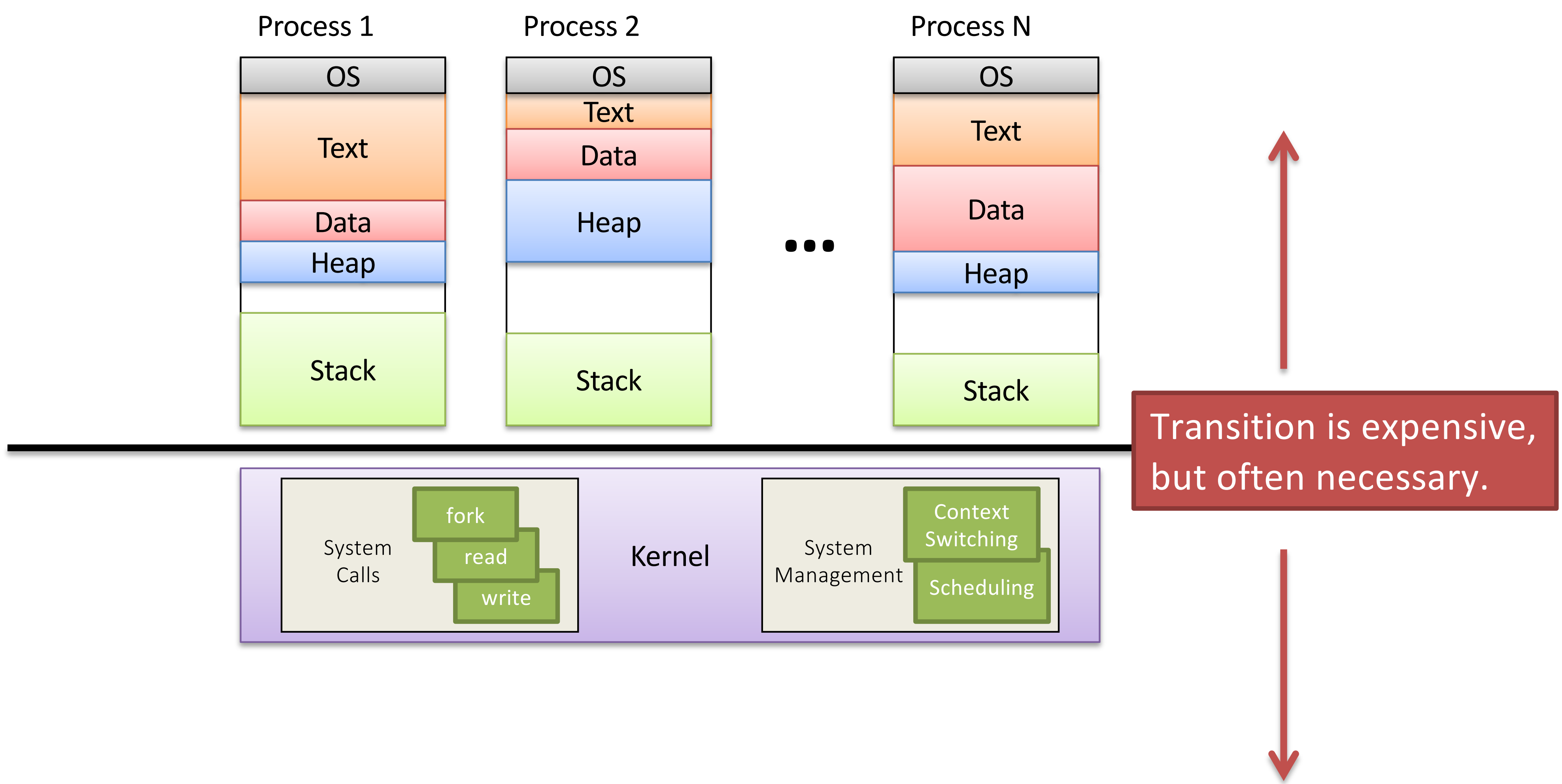
OS has control. It will take care of process's request, but it might take a while.

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

Kernel vs. Userspace: Model



Kernel vs. Userspace: Model



System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)

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

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.

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