CS 241: Systems Programming
Lecture 16. Processes

Spring 2024

Prof. Stephen Checkoway

Processes are instances of running programs

Every time we open a program or run one on the command line, the kernel creates a new **process**

Each process has

- memory allocated to it by the kernel for code and data (including the stack and the heap)
- a process state (next slide)
- a process id
- a table of open files (including stdin/stdout/stderr)
- other data including a user id, group id, and various permissions

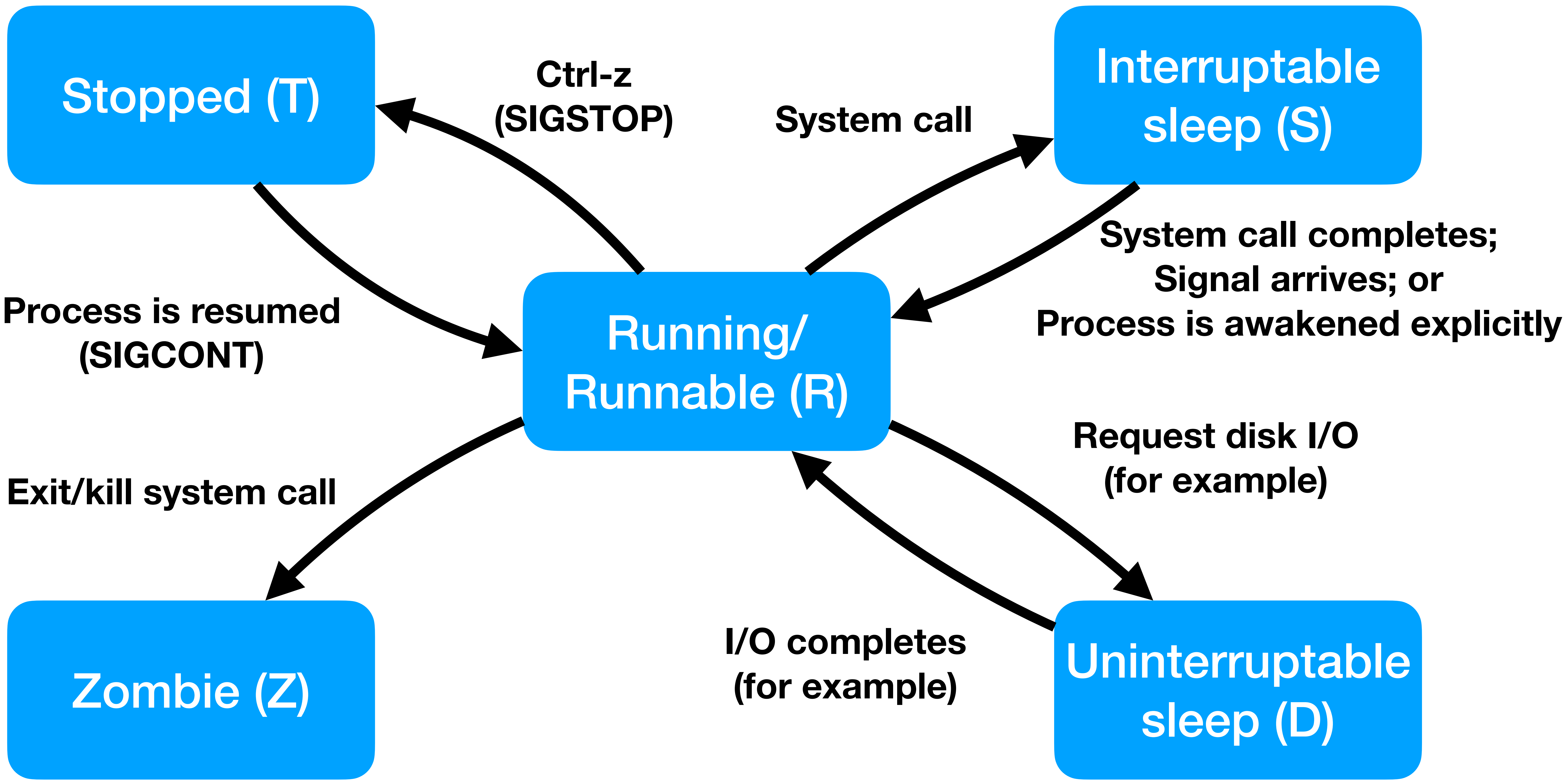
The kernel is responsible for running processes

Process states

Every process in the system is in one of several states

- Running/Runnable — Process is running on a CPU or able to run
- Interruptable sleep — Process is asleep but can be awakened via a signal
- Uninterruptable sleep — Process is asleep but will not wake for a signal
- Stopped — Process has been suspended (e.g., ctrl-Z)
- Zombie — Process has exited but is still in the process table until its parent uses the wait system call to “reap” it

Process state transitions



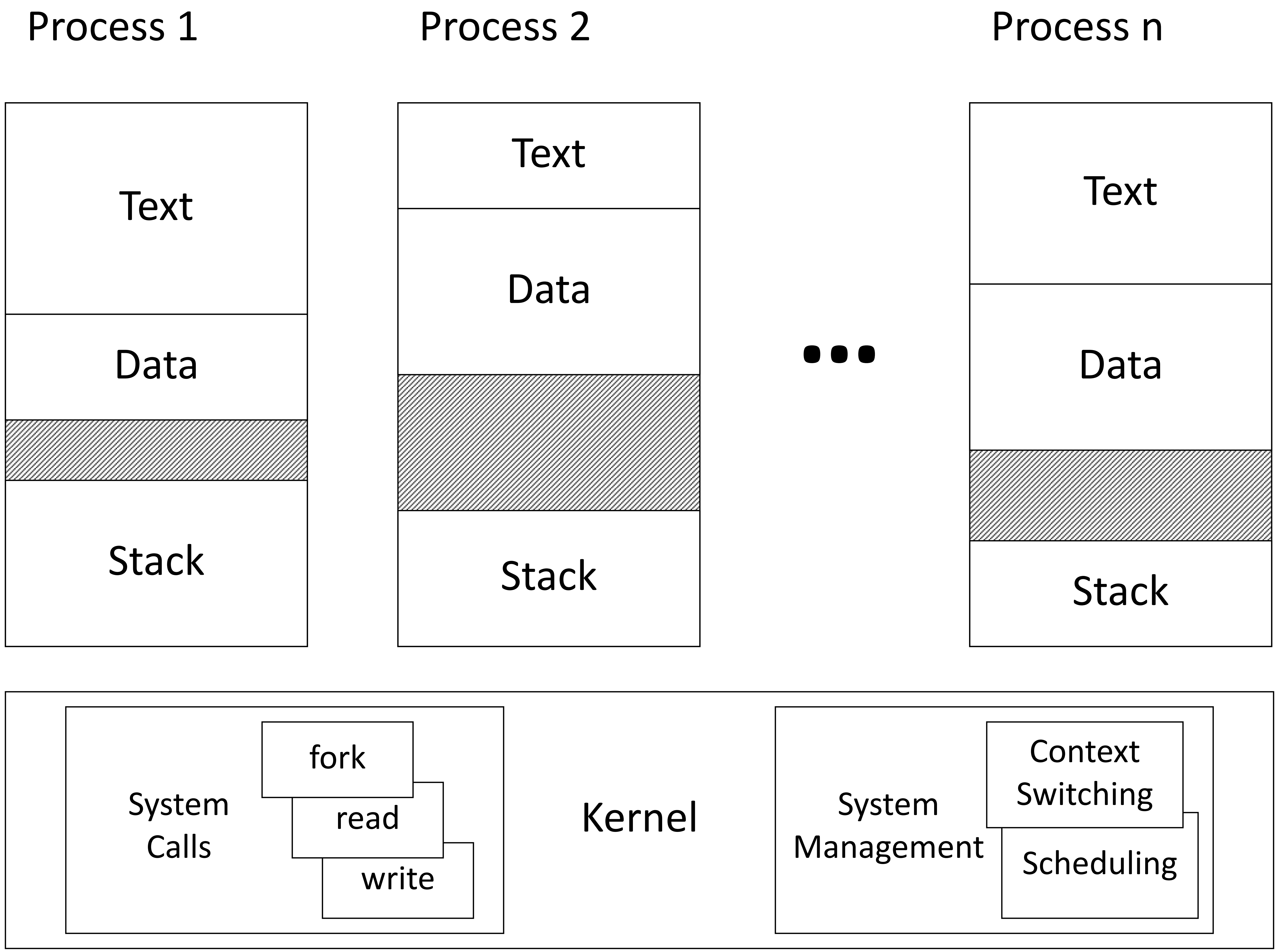
Printing the process state

```
$ ps -e -o pid,state,command
```

This will print the process ID, process state, and command name of every process on the system

```
    PID S COMMAND
      1 S /sbin/init splash
      …
1156303 R sshd: steve@pts/0
1156310 S -bash
1156474 S /usr/libexec/tracker-store
1156493 R ps -e -o pid,state,command
```

Process and Kernel Model



System calls

A process makes a **system call** to request the kernel take an action on the process's behalf

Examples include

- Opening/creating/reading/writing/closing files and directories
- Sending network packets
- Running new processes
- Exiting (a program stops running by making a request to the kernel)
- Suspending/resuming other processes
- Sending information between processes

The kernel is responsible for

Handling system calls (either performing the requested action or denying the request)

Managing hardware resources including

- CPUs
- GPUs
- Timers
- Networking hardware

Process control

- Starting/stopping processes
- Switching which process is running on a CPU

How does the kernel get control

To perform its tasks, the kernel must get control of the CPU

Running process can give up control voluntarily

- Process makes a blocking system call, e.g., reading a file
- Control goes to kernel, which dispatches a process

Or, CPU is forcibly taken away: preemption

- While kernel is running, it configures a hardware timer
- When timer expires, the hardware interrupts the CPU
- The interrupt forces control to go to kernel

User and kernel mode

Modern OSes leverage hardware support to have distinct operating modes: user mode and kernel mode

Modern hardware has privileged instructions for managing processes and hardware that can only run in kernel mode

- If user programs attempt to run them, the hardware **traps** into the kernel

Why do you think modern processors support user vs. kernel mode?

A. It’s faster to manage hardware in kernel mode than in user mode

B. It’s safer to prevent applications from interacting with hardware like hard drives/solid state drives directly

C. Government purchasing regulations discourage selling computers without it

D. It prevents buggy or malicious applications from interfering with the kernel’s operation

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

Switching between user and kernel mode

Process makes system call or is interrupted (by hardware or software)

- These are the only ways of entering the kernel

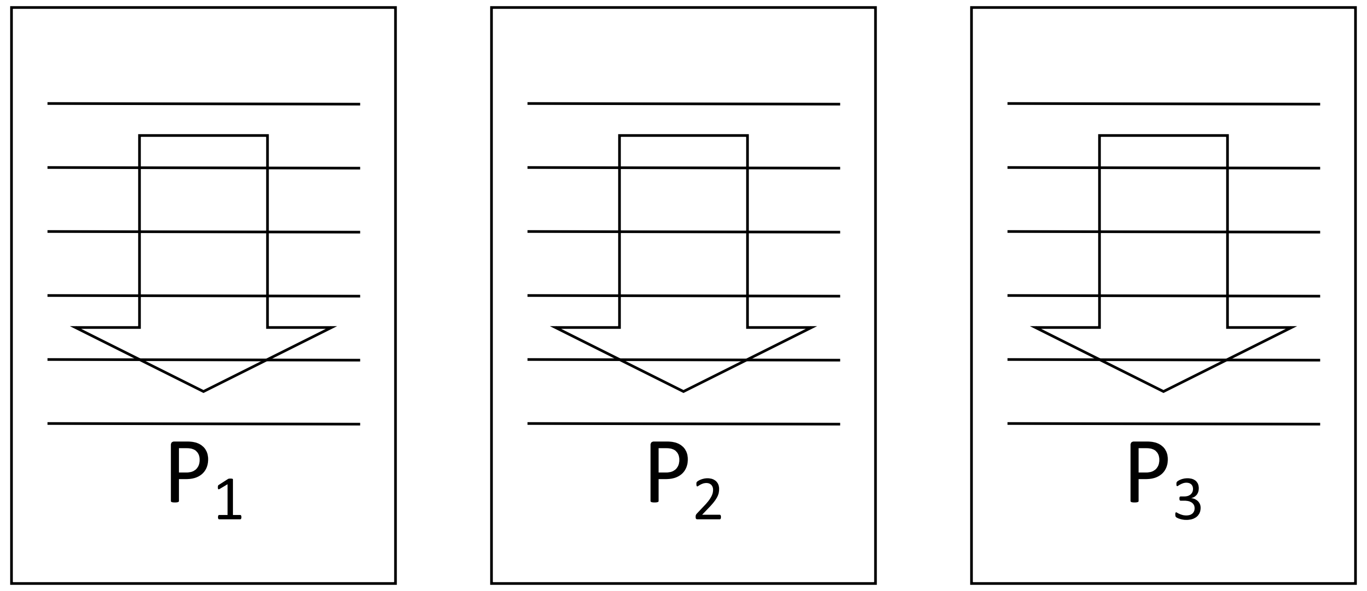
First: In hardware

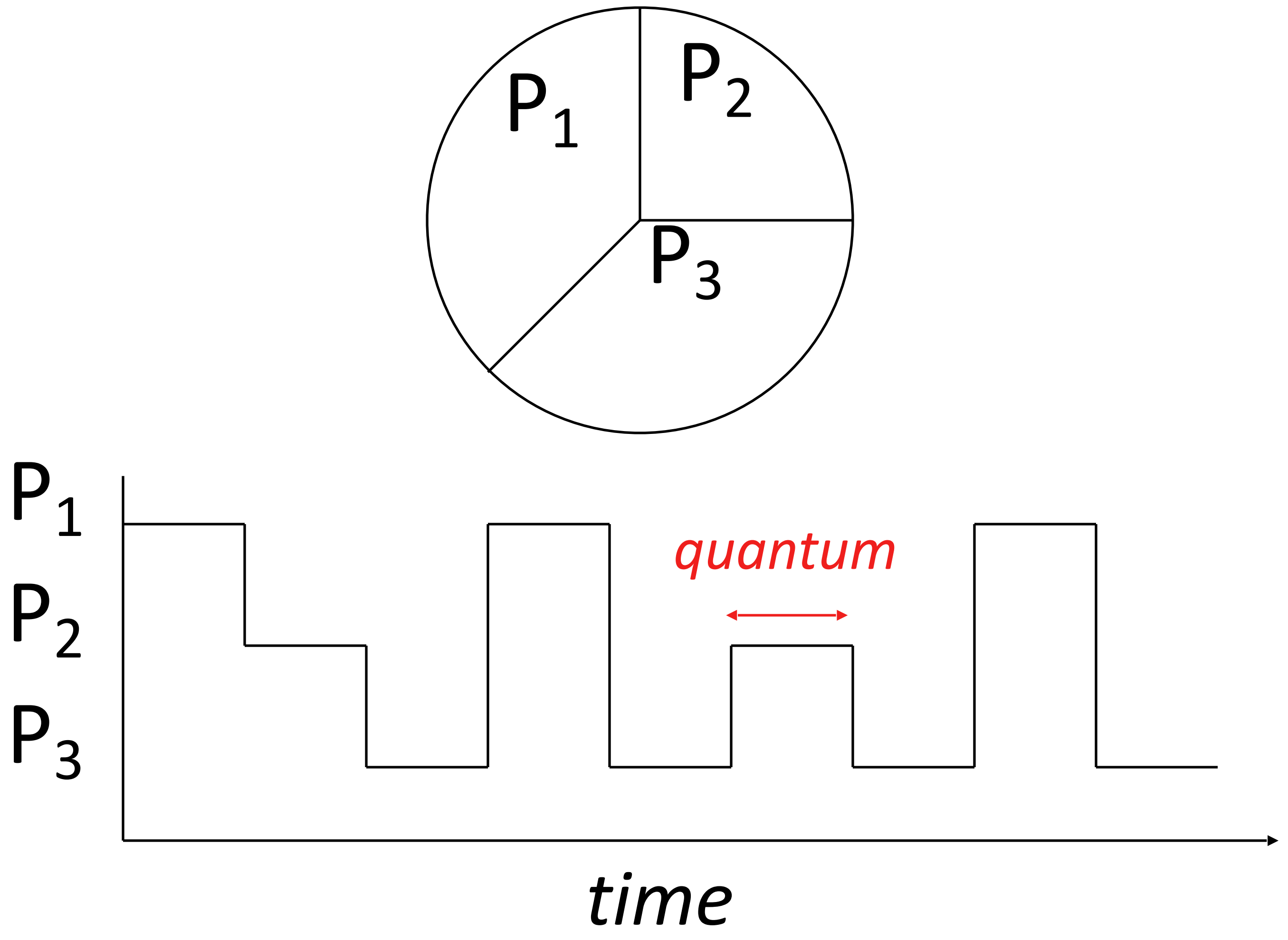
- Switch from user to kernel mode
- Go to fixed kernel location: interrupt/trap handler

Next: In software (in the kernel)

- Run handler code
- Return from interrupt/trap
    - Return to user mode

Timesharing





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

Switching between different processes for timing sharing is not free. It takes time and memory and energy (think about your phone’s battery) away from doing useful work (running the processes). A different approach would be to run each process in order until it has completed. This is called batch processing.

What are the benefits and drawbacks of timesharing? (Think of some of each.)

A. Select A when you have an answer

How is timesharing implemented?

Kernel keeps track of the state of each process

Characterizes state of process’s progress

- Running: actually making progress, using CPU
- Runnable: able to make progress, but not using CPU
- Sleeping: not able to make progress, can’t use CPU
- Etc.

Kernel selects a process in the Runnable, lets it run

- Eventually, the kernel gets back control
- Selects another ready process to run and **performs a context switch**
- And repeat

Context

Every process has an execution context consisting of

- General purpose register values
- Floating point register values
- Stack pointer (used for managing stack frames)
- Instruction pointer (the address of the next instruction in memory to execute)
- Other hardware-specific state

How a context switch occurs

Switch to kernel

In software (in the kernel)

- Save context of last-running process
- Conditionally
    - Select new process from those that are ready
    - Restore context of selected process
- Return to user mode

Multi-processor system

Same time-sharing behavior occurs but now the kernel has multiple CPUs it can schedule processes on

More complicated because now multiple processes really are running at once!

Not all CPUs in the system are the same, particularly in devices like phones

- Fast, power-hungry CPUs
- Slow, low-power CPUs
- Kernel has to decide which processes to allocate to which CPUs

Manipulating processes from the command line

Listing running processes: ps (you're going to implement this!), pgrep

Stopping (suspending) a **foreground** process: ctrl-z

- Foreground processes are those that write to/read from the terminal

Running a stopped process:

- fg — runs a stopped process and makes it a foreground process
- bg — runs a stopped process and makes it a background process

Causing a process to exit

- kill — sends one of several **signals** to processes, SIGTERM (the default) terminates the process