Project 2, Interrupts, and Scheduling

CS 4411

Spring 2020

Announcements

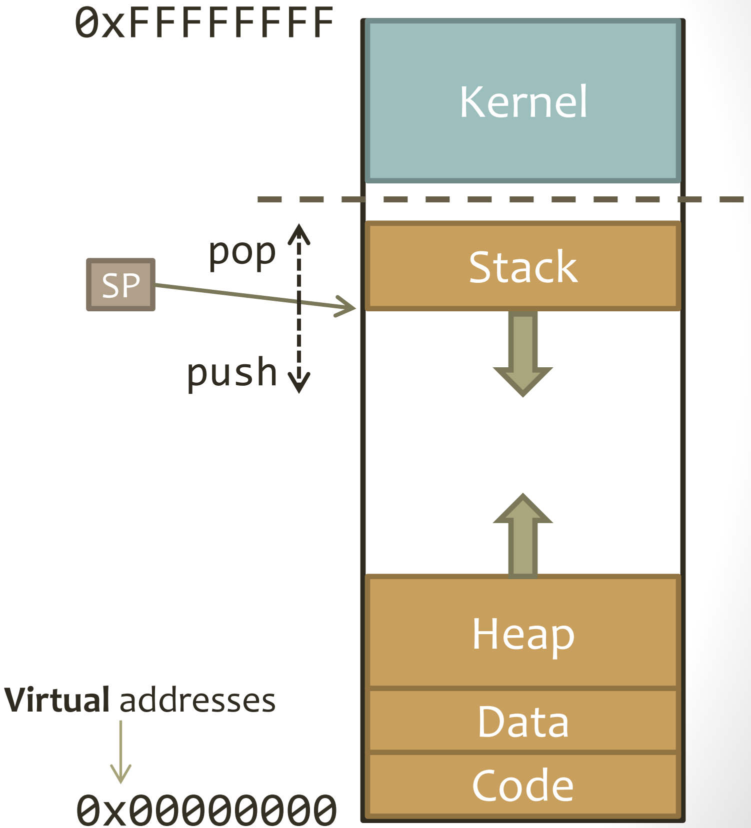
- Office hours
- Regrades
- Piazza

Outline for Today

- Arrays and Stacks
- Project 2 Overview
- Interrupt Handling
  - Privilege Modes
  - Timer Interrupts
- Scheduling with Quanta

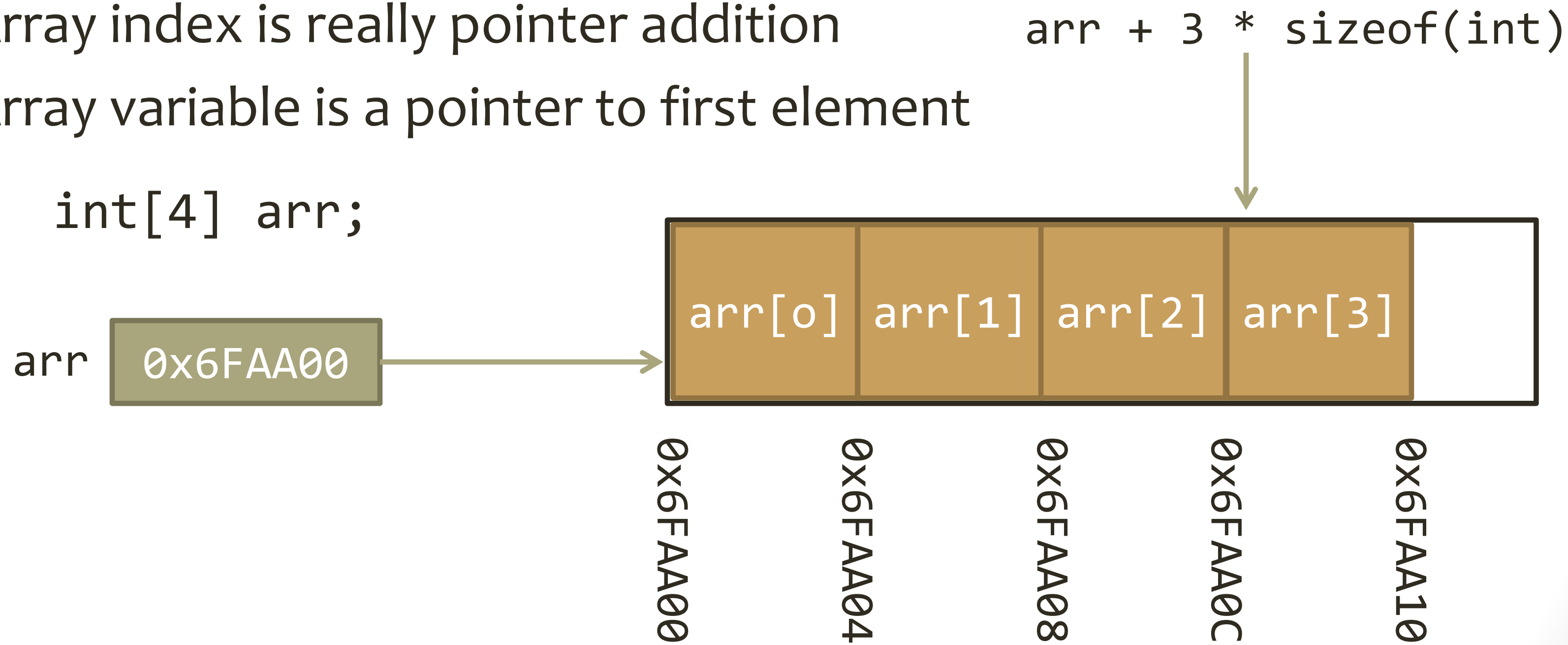
On P1: A Note About Stacks

- Standard process layout: Stack "grows downward"
- What does this mean?
- push instruction:
  - **Decrements** SP
  - Stores register to memory at SP
- pop instruction:
  - Reads memory at SP into register
  - **Increments** SP

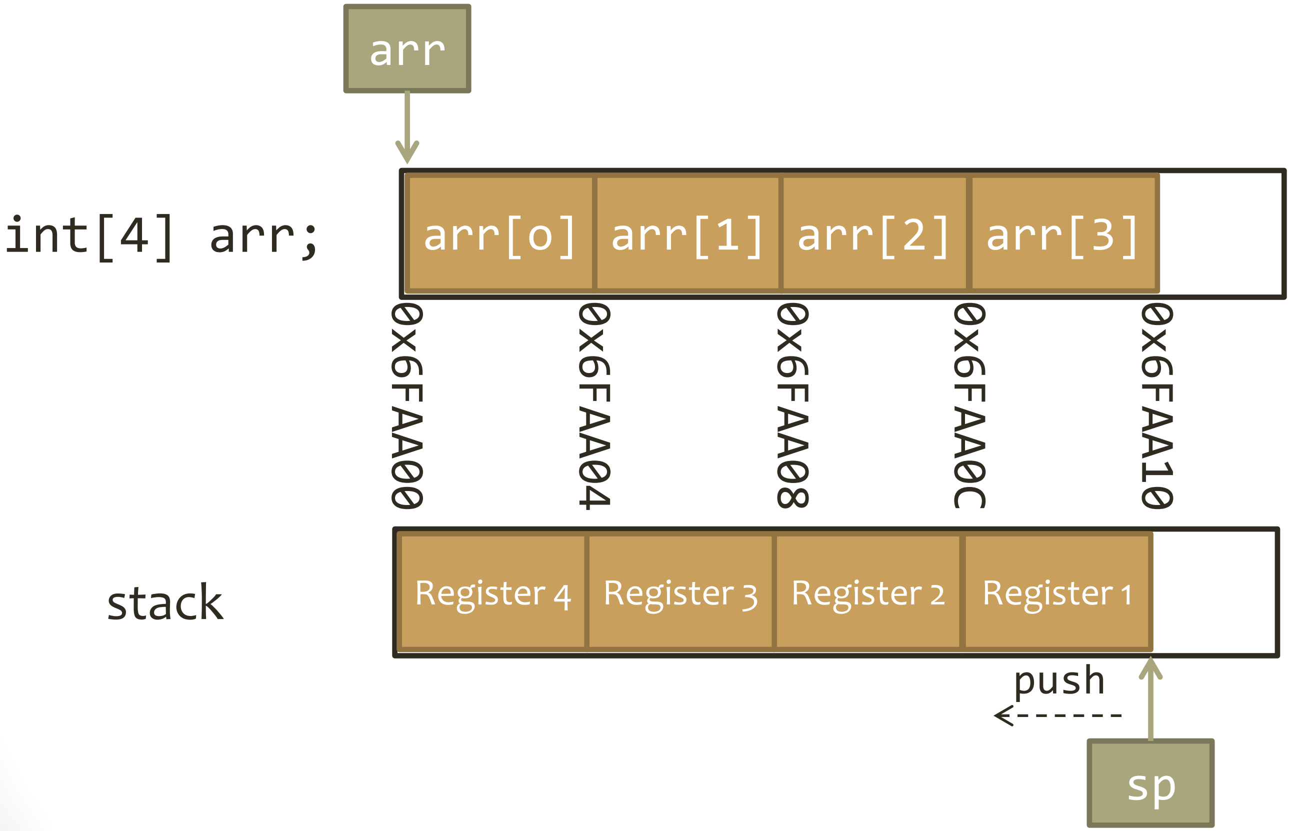


Compare to Arrays

- Arrays in C are contiguous memory
- Array index is really pointer addition
- Array variable is a pointer to first element



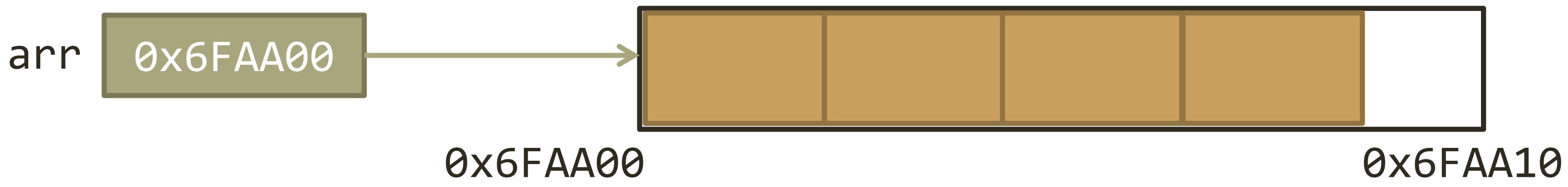
Arrays vs. Stacks



malloc() Behavior

- `malloc()` is a natural fit for arrays: it returns a pointer to the **lowest** memory address in the allocated region

```
int* arr = malloc(4 * sizeof(int));
```



- Is this what you want for a thread/process's stack? (Can you use `arr` as a stack pointer?)

Outline

- Arrays and Stacks
- **Project 2 Overview**
- Interrupt Handling
    - Privilege Modes
    - Timer Interrupts
- Scheduling with Quanta

Project 2 Basics

- EGOS has a scheduler, but it's not very good
  - Round-robin algorithm
  - FIFO run queue, timer interrupts force yield
- Replace scheduling logic with Multi-Level Feedback Queue
- Measure quality of new scheduler
  - Each process's completion time and number of yields
  - Overall average CPU load

Project 2 Logistics

- One file to edit: src/grass/process.c
- When you make changes, keep the original code, and use a macro to select whether new or old code is compiled:

```
#ifdef HW_MLFQ
    proc_next = mlfq_get_next(&run_queue, level);
#else
    proc_next = queue_get(&proc_runnable);
#endif
```

Your new code ← (first branch)

Original (round-robin) code ← (second branch)

- If COMMONFLAGS in Makefile.common includes `-DHW_MLFQ` your code will be used, otherwise original code will be used

Concepts in Project 2

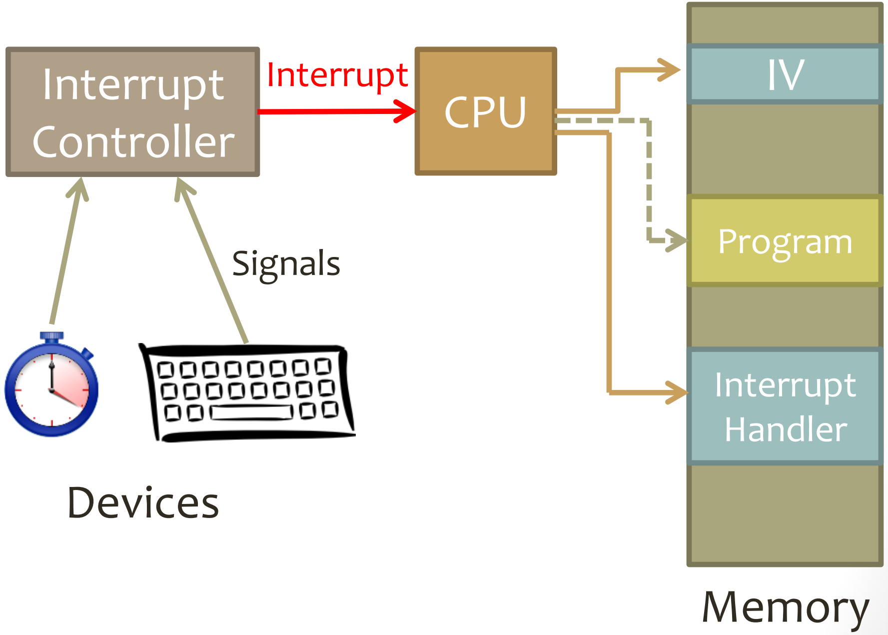
- Interrupt handling
- Context switches (again)
- Process blocking and I/O
- Scheduling decisions and bookkeeping

Outline

- Arrays and Stacks
- Project 2 Overview
- **Interrupt Handling**
  - Privilege Modes
  - Timer Interrupts
- Scheduling with Quanta

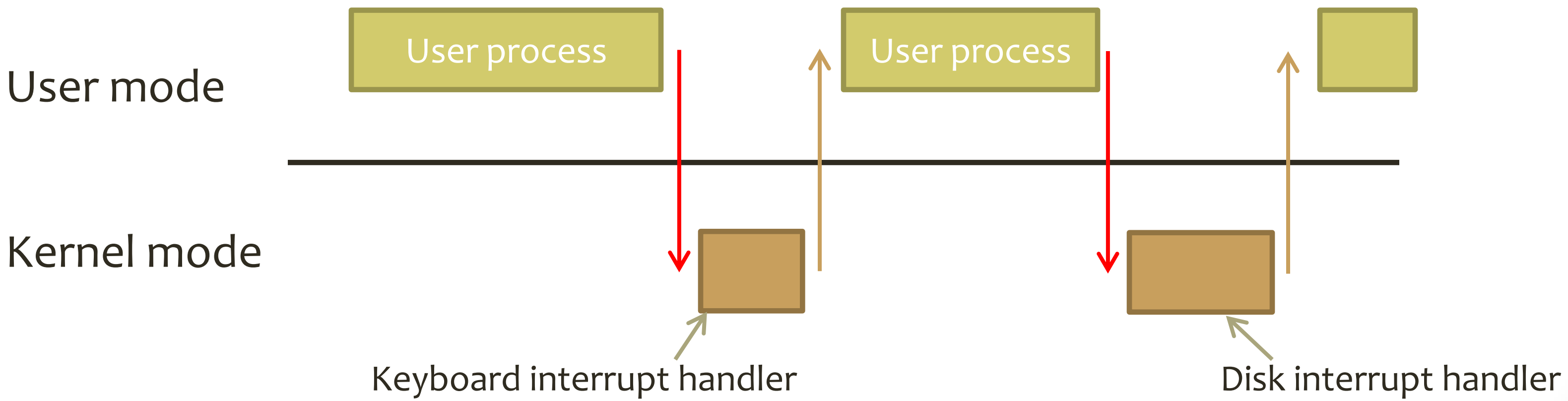
Essentials of Interrupt Handling

- Hardware-assisted
- Interrupt Vector selects where CPU jumps
  - In a fixed, known location, has an entry for each type of interrupt
- Forced context switch



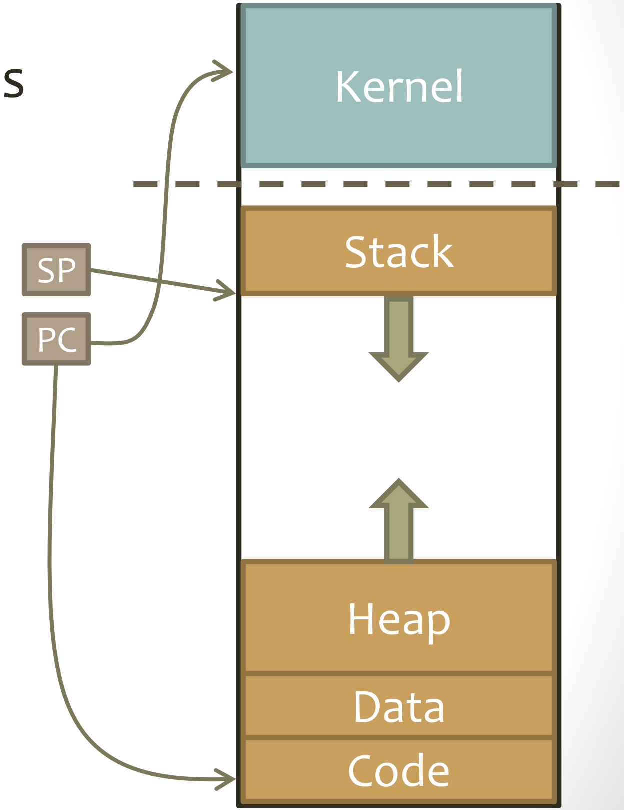
Privileges

- Interrupt handling is a privileged operation
  - HW sets kernel-mode bit
- Interrupt handlers are part of kernel
- After interrupt handler runs, return control to user process



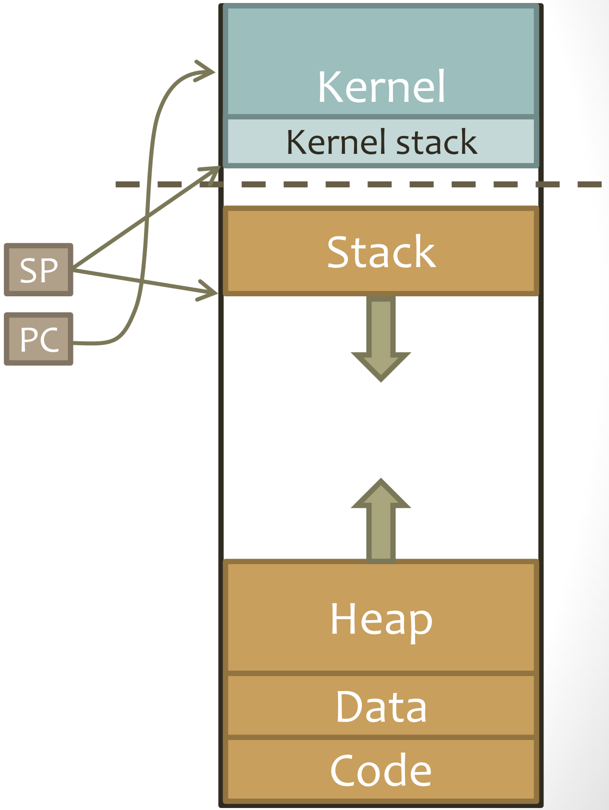
Memory Layout

- When interrupt happens, some other process is running
- To switch to interrupt handler, kernel memory must also be mapped in process's address space
  - Otherwise, how would you get to interrupt handler's code?



Memory Layout

- Interrupt handler is a program, needs a stack
- Where should its stack be?
    - Kernel, in privileged mode, has access to process's entire memory space
- Each process has a **kernel stack**
    - SP moved here every time kernel takes control
    - E.g. when interrupt handler is running

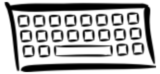


Interrupt Handling in EGOS

- Interrupts generated by "intr" module in Earth (src/earth/intr.c)
  - Simulates interrupt controller
- Kernel registers an interrupt handler that calls `proc_got_interrupt()` in process.c for all interrupts
- Interrupts **disabled (masked)** by default in kernel mode
- Interrupts only enabled:
  - When executing user-mode process
  - When waiting for I/O (even in kernel mode)
- Masked interrupts **will fire** once interrupts re-enabled

A Special Kind of Interrupt

Other Types of Interrupts

- I/O Interrupts 
  - Device has some input for you!
- Page Fault Interrupts 
  - Process needs memory!
- System Calls
  - Process wants you to do something!

Timer Interrupts 

- Ding! Time has elapsed!
- No pending task to do
- What's the point?
- Periodically returns control to the kernel, even for long-running processes
- Kernel can switch to a different process – pre-emption

Outline

- Arrays and Stacks
- Project 2 Overview
- Interrupt Handling
  - Privilege Modes
  - Timer Interrupts
- **Scheduling with Quanta**

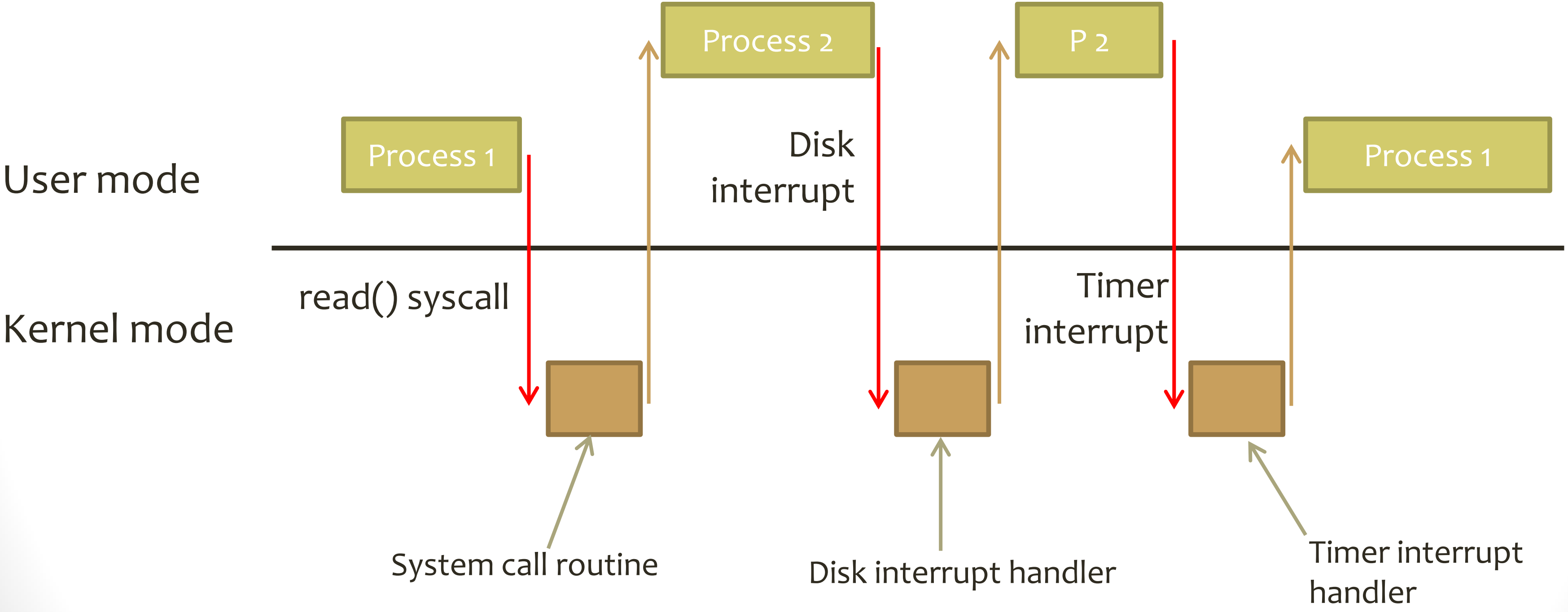
Reasons for Scheduling

- Why might control return to the kernel?

- A timer interrupt occurred
- Another kind of interrupt occurred (I/O, system call, etc)
- Process is blocked waiting for an event
- Process has terminated

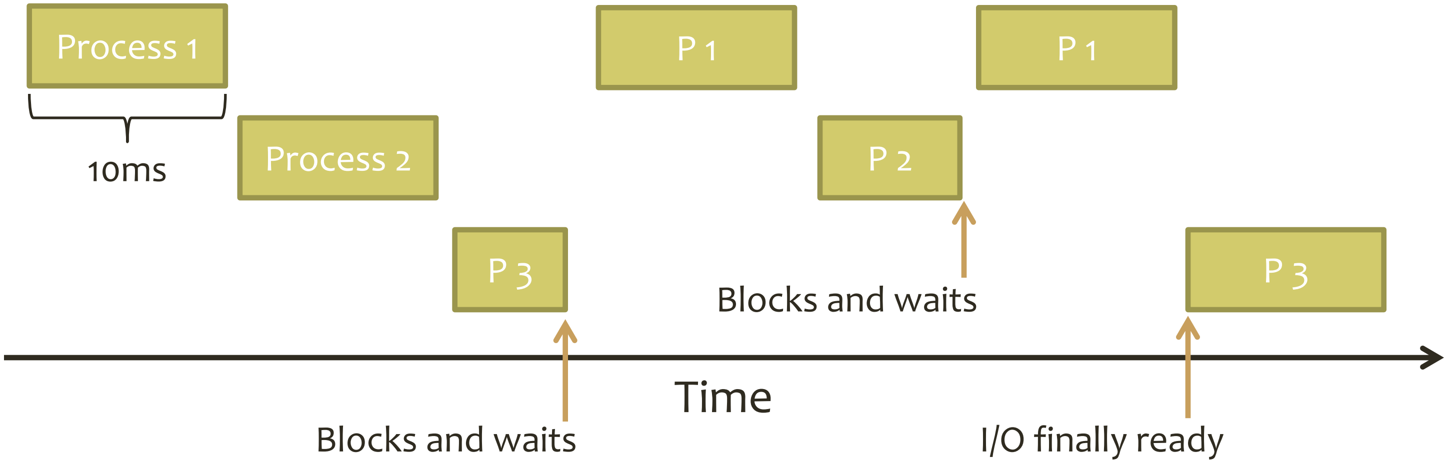
- Which of these requires the kernel to schedule a new process?

A Day in the Life



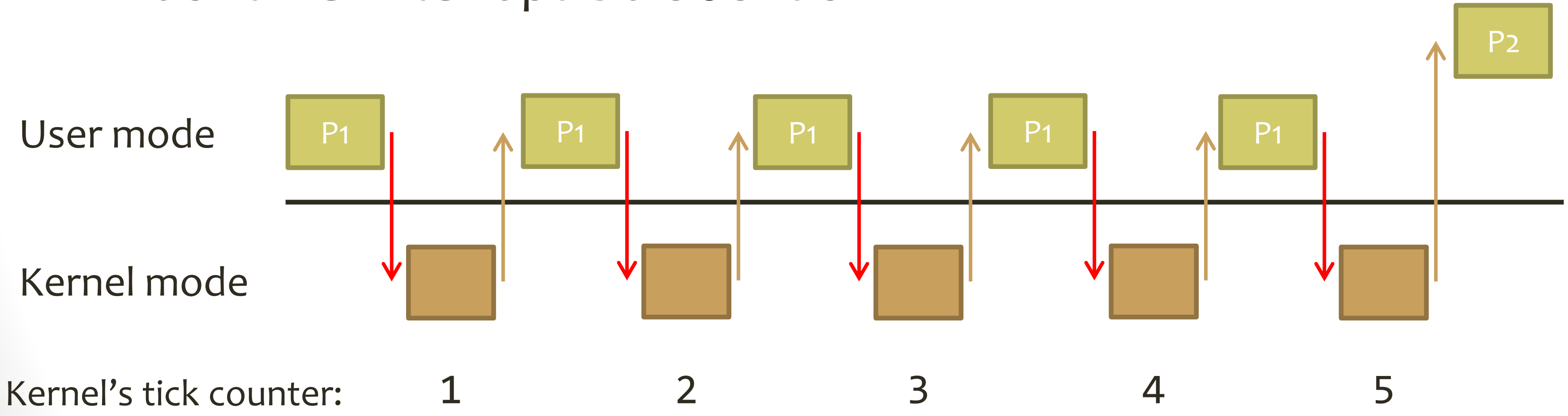
Quanta and Scheduling

- Quantum = arbitrary unit (of time)
- In a scheduler, quantum = **maximum** time a process can execute
- Round Robin with 10ms quantum:



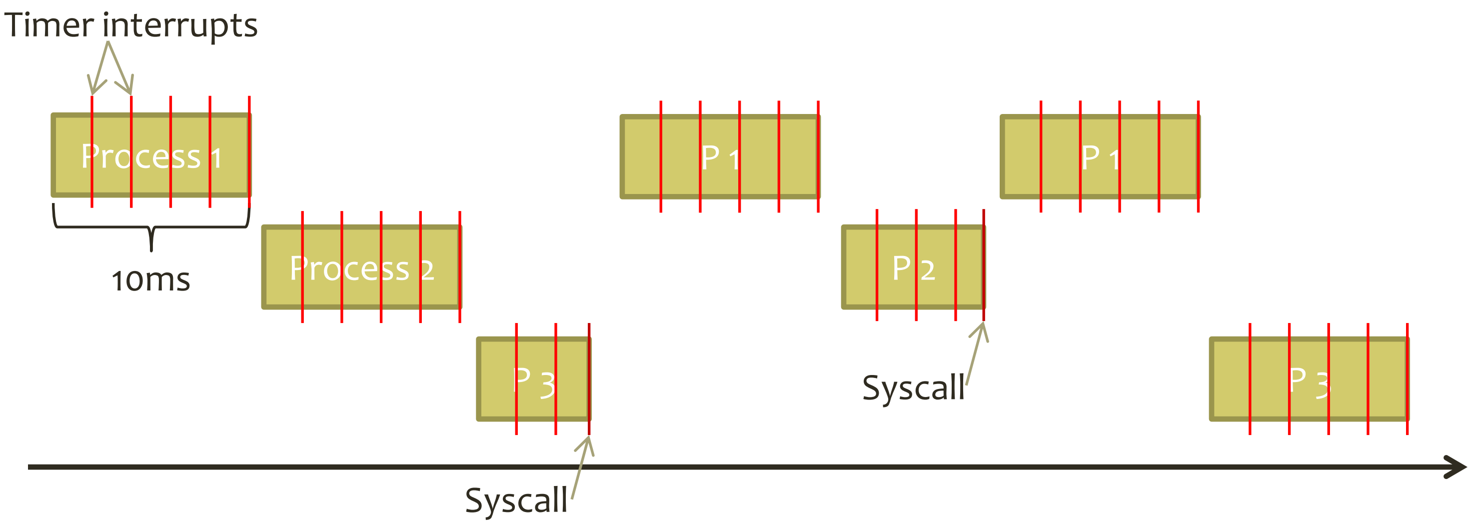
Quanta and Clock Interrupts

- Timer (clock) interrupts are how the OS measures time
- Scheduler's quantum is a multiple of **clock ticks**
- Each timer interrupt is a clock tick



Round Robin's Details

- Round Robin with 10ms quantum
- Timer interrupt (clock tick) every 2ms



On a Timer Interrupt

- Increment clock tick
- Determine if quantum is over
  - If not, interrupted process should resume running
- Make scheduling decision
- In Multi-Level Feedback Queue, what happens when a process reaches the end of a quantum without blocking?