Today

- Booting
- Process abstraction
- Dual mode execution

Course Objectives Review

- Classical OS
    - Emphasis on the *why*
- Agile class
- Synching with the Project

https://www.facebook.com/groups/169380229860838/

Review

- What do we call the core of the OS?

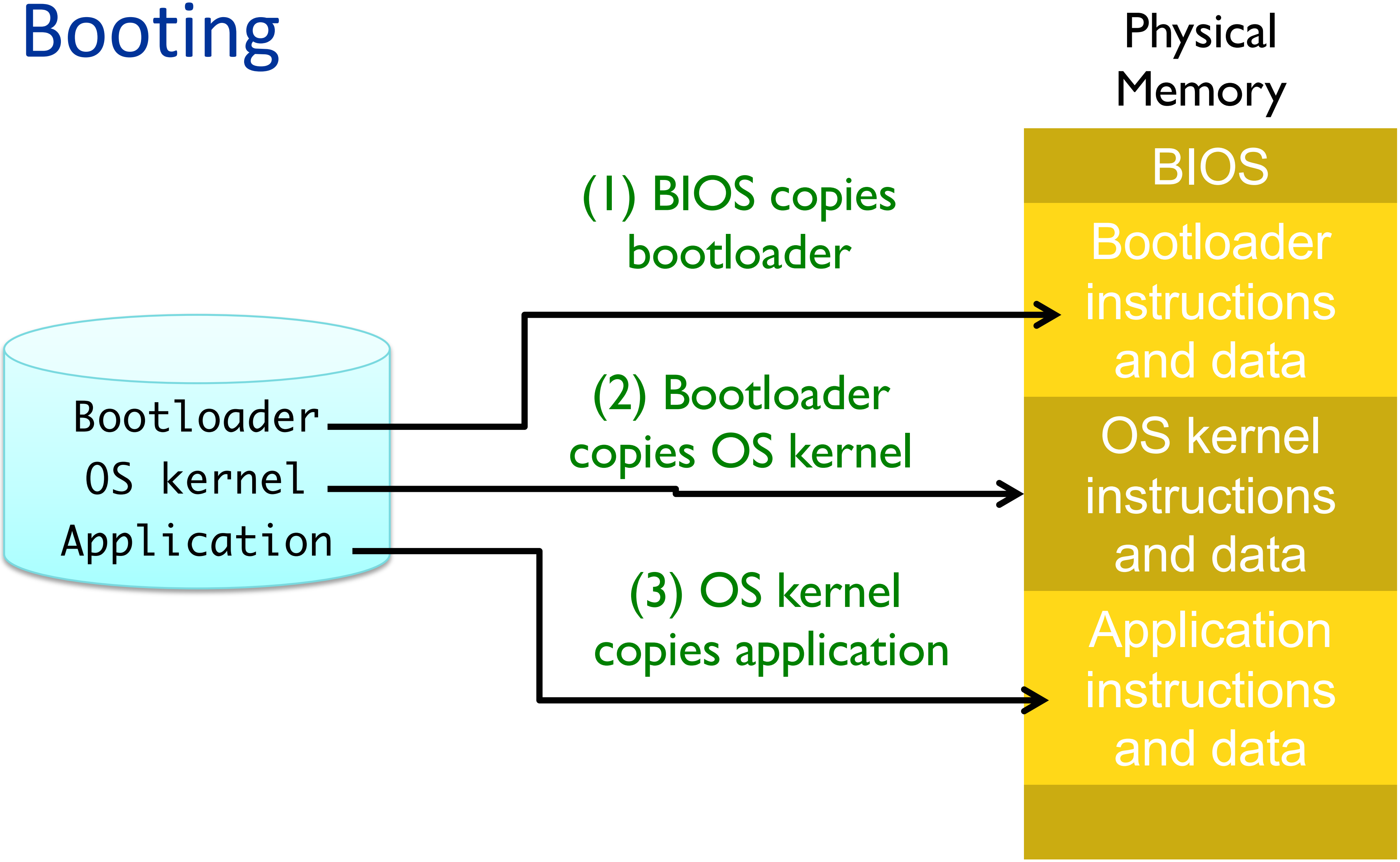
How do we get the OS party started?

BOOTING

System Boot

- ***Booting***: Loading the kernel to render a computer usable
- When power initialized on system, execution starts at a *fixed* memory location
    - Firmware ROM used to hold initial boot code
- OS must be available to hardware so hardware can start it
    - Small piece of code – ***bootstrap loader***—locates the kernel, loads it into memory, and starts it
        - stored in ROM or EEPROM
    - Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then running

Booting



Physical Memory

BIOS

Bootloader instructions and data

OS kernel instructions and data

Application instructions and data

Bootloader

OS kernel

Application

(1) BIOS copies bootloader

(2) Bootloader copies OS kernel

(3) OS kernel copies application

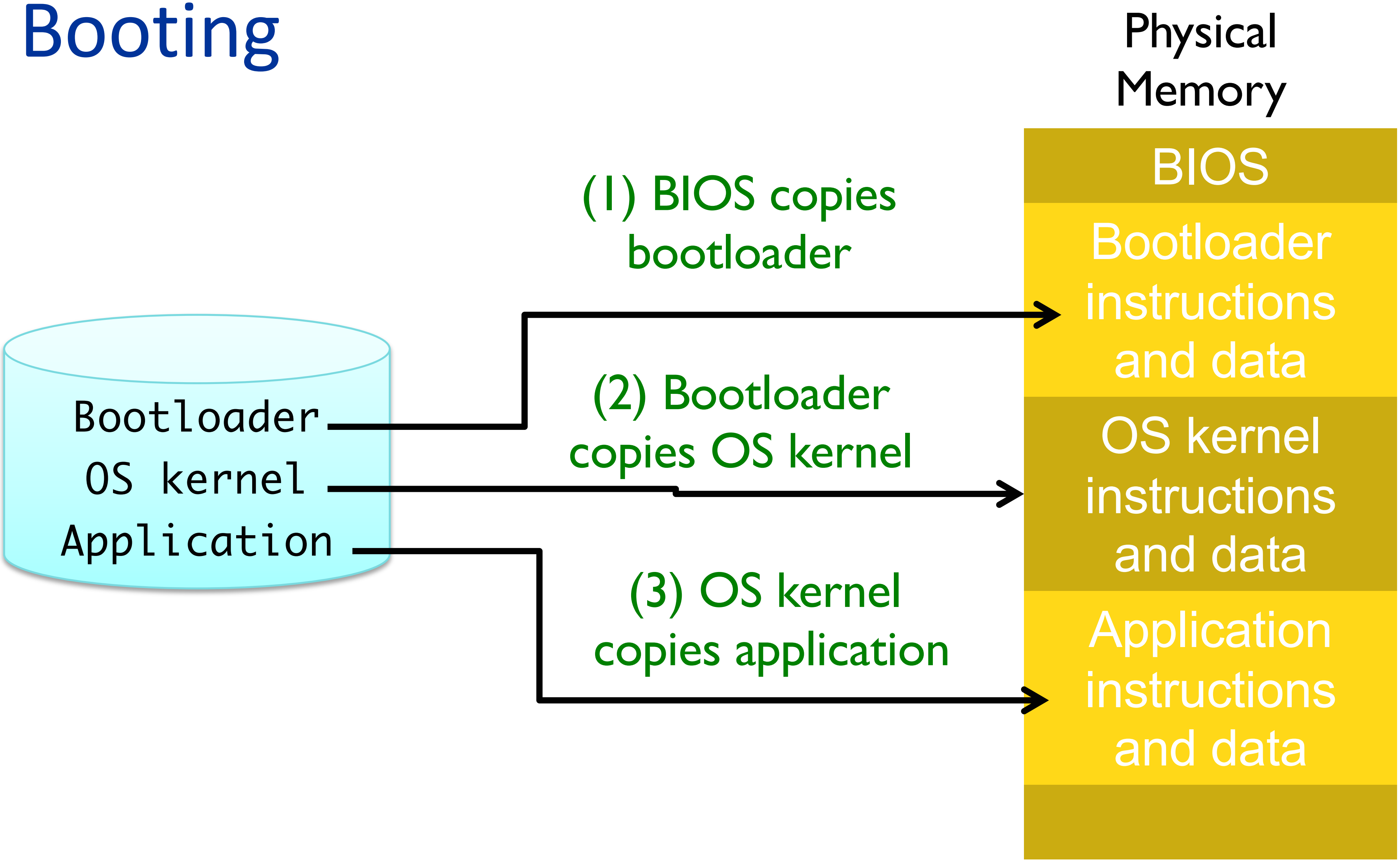
Basic Input/Output System (BIOS)

- A number of small programs and subroutines:
  - Power on self test (POST)
  - System configuration utility
    - Settings stored in small amount of battery backed CMOS memory.
  - A set of routines for performing basic operations on common input/output devices. Such as…
    - Read/write a specified C:H:S from disk
    - Read character from keyboard
    - Display character on the screen
  - OS bootstrap program
- Stored on a Flash ROM that is part of the computer's address space.

Bootstrap Process

- Program Counter (PC) is initialized to the address of the POST program contained in the BIOS
- The last instruction of the POST jumps to the address of the *bootstrap program*, also contained in the BIOS.
- The bootstrap program uses the BIOS routines to load a program contained in the *Master Boot Record* (MBR) of the boot disk into memory at a known address.
    - MBR = first sector on the disk (512 bytes).
    - Boot disk is identified by data stored in the configuration CMOS.
- The last instruction in the bootstrap program jumps to the address at which the MBR program was loaded.
- The MBR program loads the OS kernel.
    - Often indirectly by loading another program (a *secondary boot loader*) that then loads the kernel

Booting



Bootloader

OS kernel

Application

(1) BIOS copies bootloader

(2) Bootloader copies OS kernel

(3) OS kernel copies application

Physical Memory

BIOS

Bootloader instructions and data

OS kernel instructions and data

Application instructions and data

Design Questions

- Why don’t we store the whole kernel in ROM?
  - Why do we need a bootloader?

- Consider:
  - What are the characteristics of ROM?
  - What are the characteristics of the kernel?

Design Questions

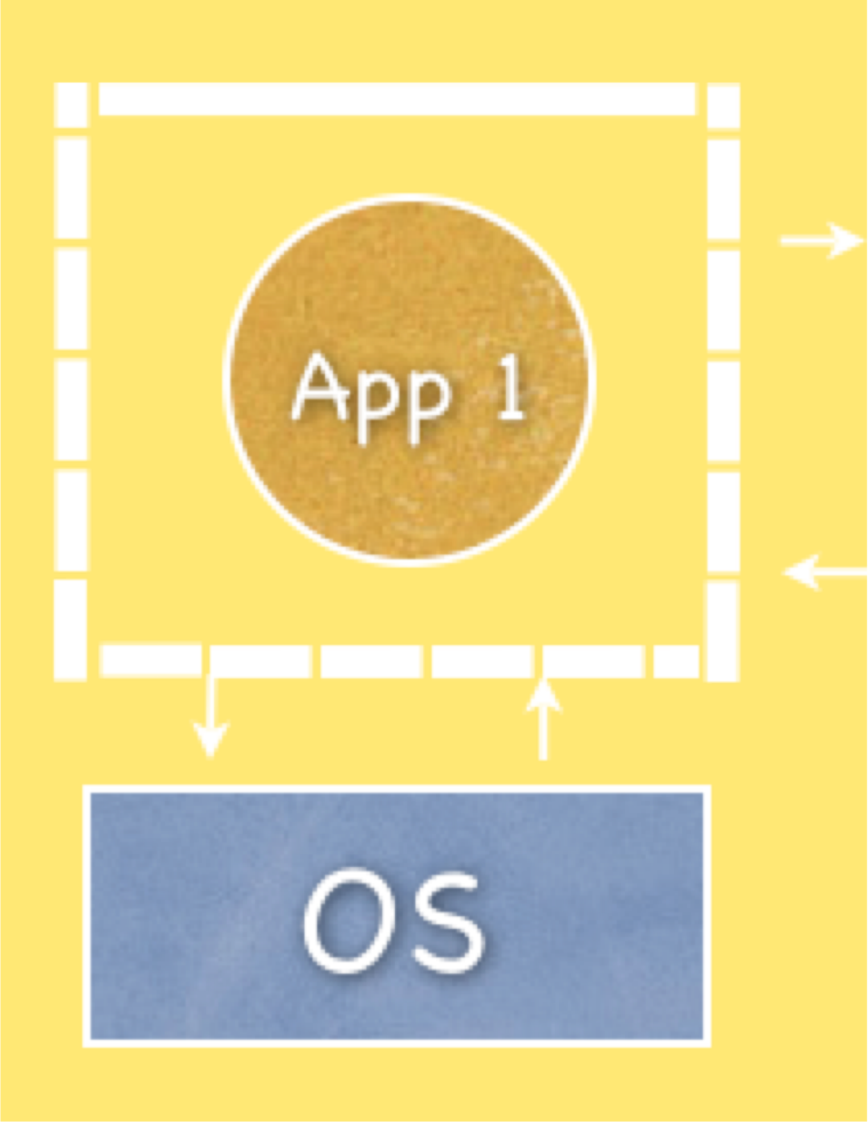
- Why don’t we store the whole kernel in ROM?
  - Why do we need a bootloader?
- Issues
  - Size of kernel
  - Updatability of kernel
    - What happens if there is an error in kernel?
  - ROM – slow, expensive, small
- Common solution: Add a level of indirection

> "All problems in computer science can be solved by another level of indirection.” – David Wheeler (except for too many levels of indirection)

Review

- What goals do the interfaces of the OS enable?
- What is the basic unit of execution in an OS?
- What resources does that unit require?

Goals for the Process: Boxes in the Application



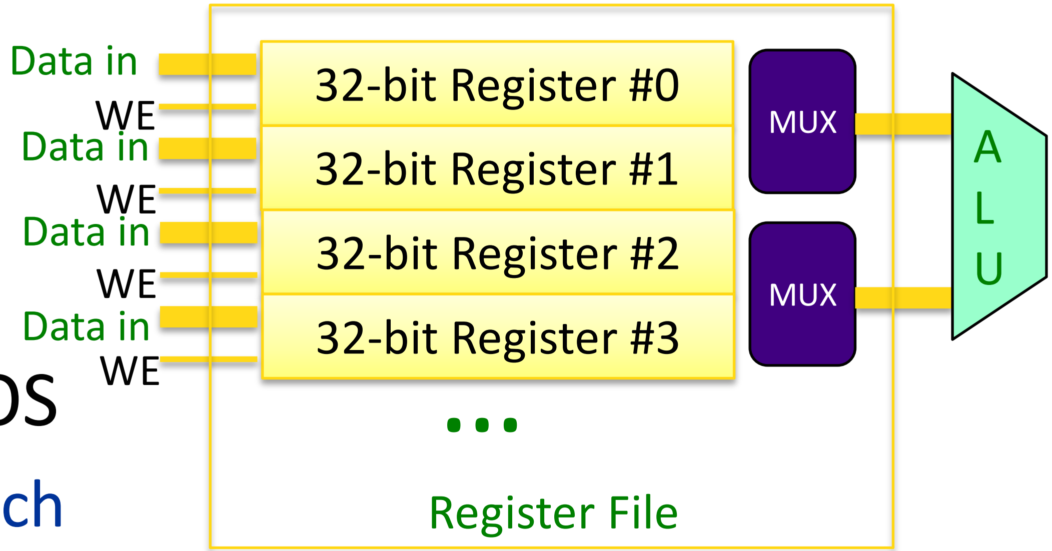
- An abstraction for protection
  - Represents an application program executing with restricted rights
- Restricting rights must not hinder functionality
  - Must still allow efficient use of hardware
  - Must still enable safe communication

Process Resource: CPU Time

- CPU: Central Processing Unit
- PC points to next instruction
- CPU loads instruction, decodes it, executes it, stores result
- Process “given” CPU by OS
    - **Mechanism**: context switch
    - **Policy**: CPU scheduling

Program Counter (PC): **Memory address of next instr**

Instruction Register (IR): **Instruction contents (bits)**

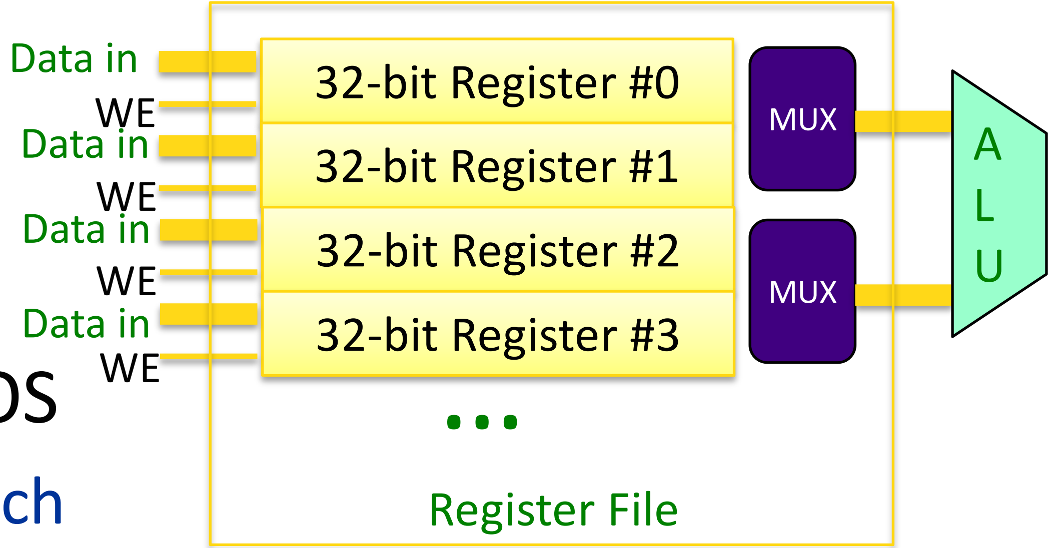


Process Resource: CPU Time

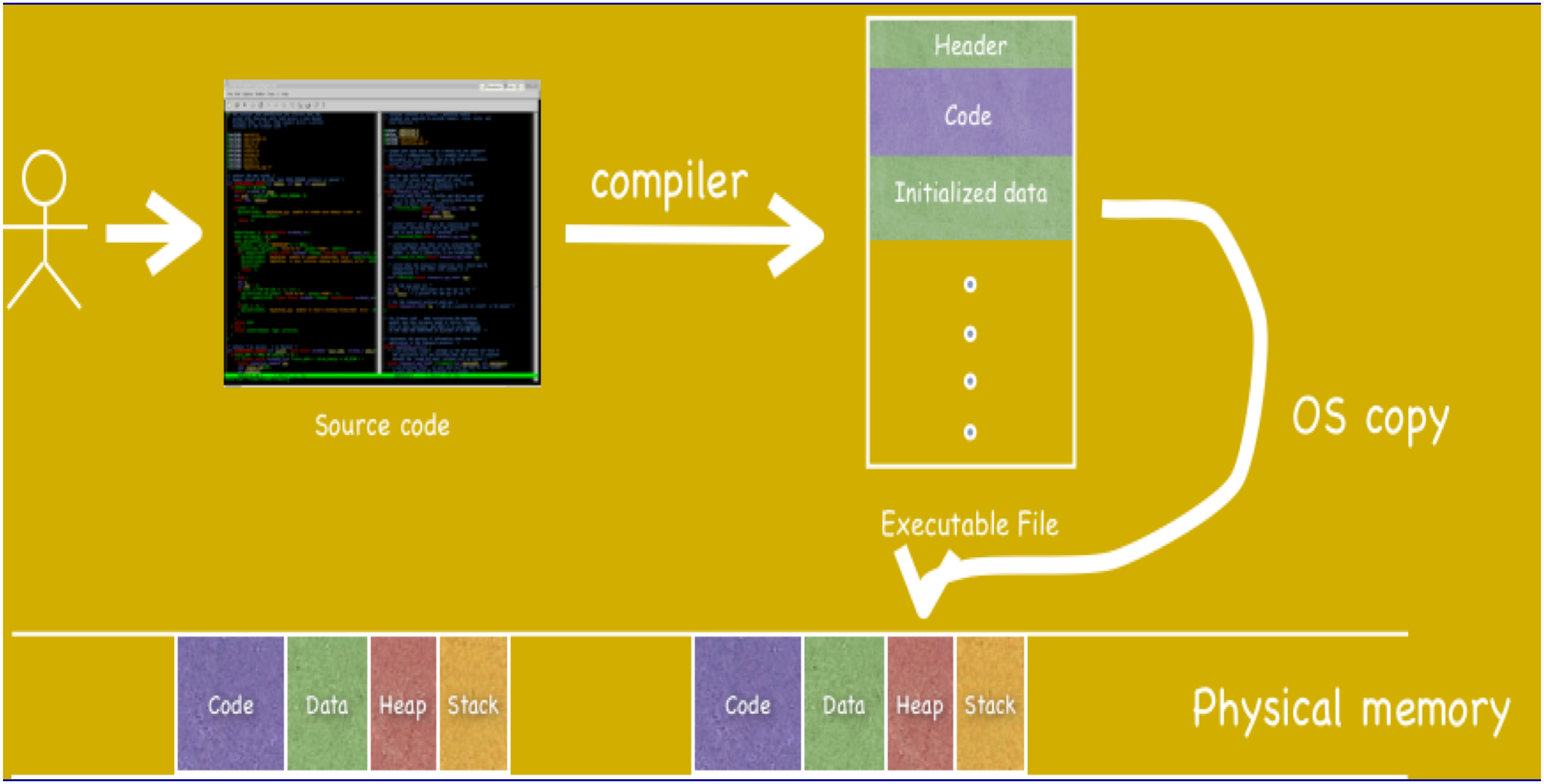
- CPU: Central Processing Unit
- PC points to next instruction
- CPU loads instruction, decodes it, executes it, stores result
- Process "given" CPU by OS
  - **Mechanism**: context switch
  - **Policy**: CPU scheduling

Program Counter (PC): **Memory address of next instr**

Instruction Register (IR): **Instruction contents (bits)**

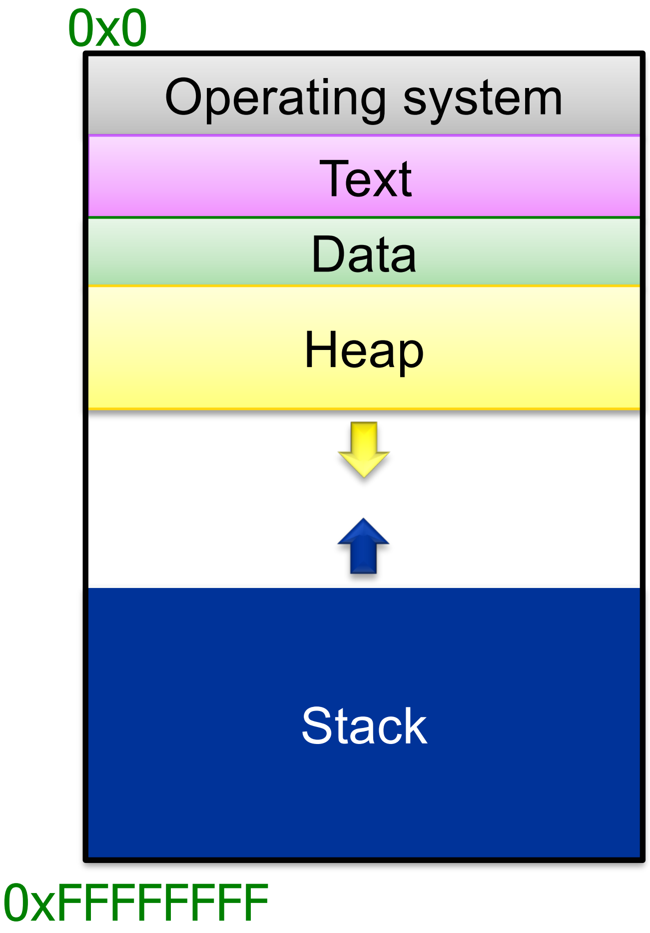


Required for process to execute and make progress!


compiler
Source code
Header
Code
Initialized data
Executable File
OS copy
Code
Data
Heap
Stack
Code
Data
Heap
Stack
Physical memory

Process Resource: Main Memory

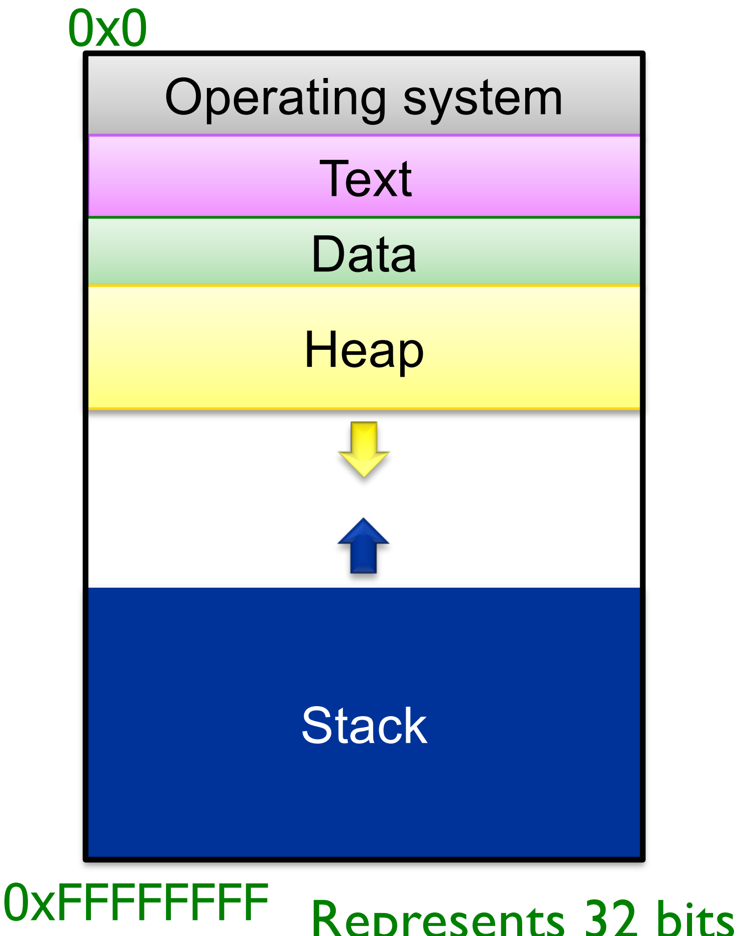
- Process must store:
  - Text: code instructions
  - Data: global and ***static*** (known at compile time) variables
  - Heap: dynamically requested memory at runtime (malloc, new, etc.)
  - Stack: store local variables and compiler-generated function call state (e.g., saved registers)



Why do the heap and stack grow towards each other?
What would an alternative organization look like?

Process Resource: Main Memory

- Process must store:
  - Text: code instructions
  - Data: global and ***static*** (known at compile time) variables
  - Heap: dynamically requested memory at runtime (malloc, new, etc.)
  - Stack: store local variables and compiler-generated function call state (e.g., saved registers)



Represents 32 bits → 2^{32} locations

Required for process to store instructions (+data)!

Process Resource: I/O

- Allows processes to interact with a variety of devices (i.e., everything that isn't a CPU or main memory).
- Enables files, communication, human interaction, etc.
- Learn about or change the state of the outside world.

Disk

Wireless Network

Keyboard / Mouse

Does a process ***require*** I/O?

HOW CAN THE OS ENFORCE RESTRICTED RIGHTS?

How can the OS enforce restricted rights?

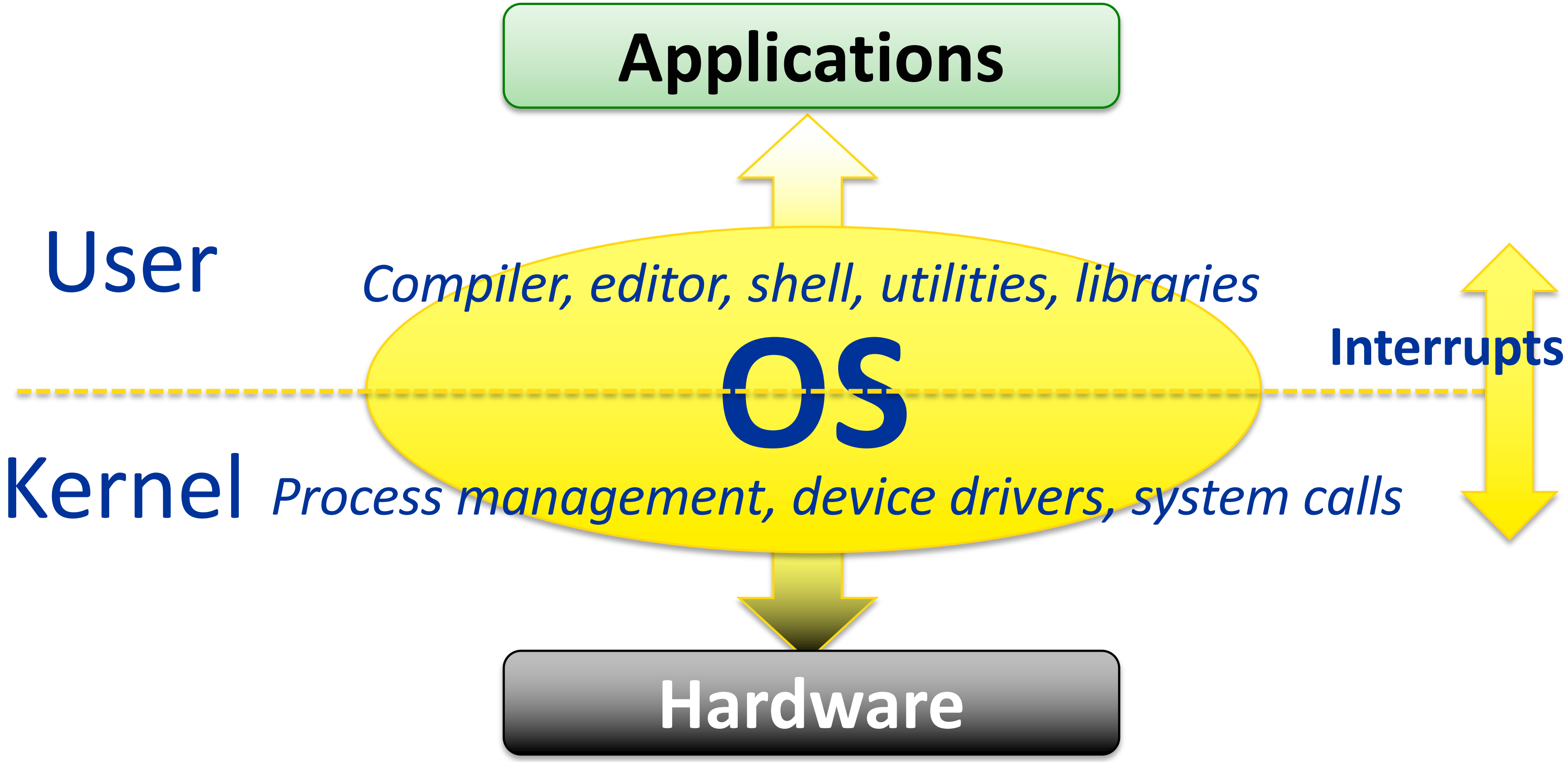
- Consider: OS interprets each instruction
    - Every instruction must be validated/executed by the [privileged] OS

- Good solution?
    - No! Slow
    - Most instructions are safe: can we just run them in hardware?

How can the OS enforce restricted rights?

- Consider: Dual Mode Execution
  - ***User mode***: access is restricted
  - ***Kernel mode***: access is unrestricted
  - Supported by the hardware
    - Mode is indicated by a bit in the process status register

Process Modes: User and Kernel



Applications

User

Compiler, editor, shell, utilities, libraries

OS

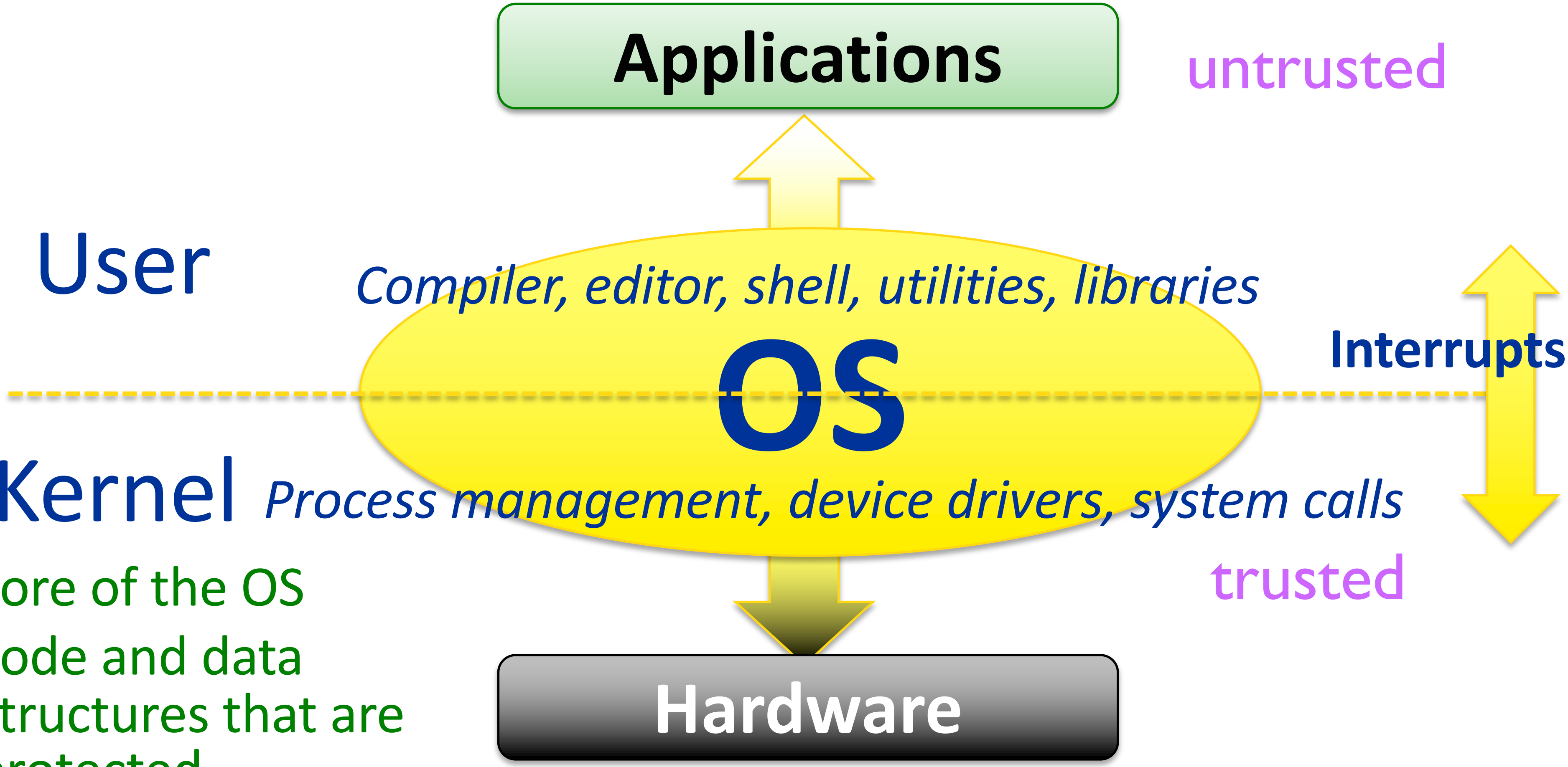
Interrupts

Kernel

Process management, device drivers, system calls

Hardware

Process Modes: User and Kernel



Applications

untrusted

User

Compiler, editor, shell, utilities, libraries

OS

Interrupts

Kernel *Process management, device drivers, system calls*

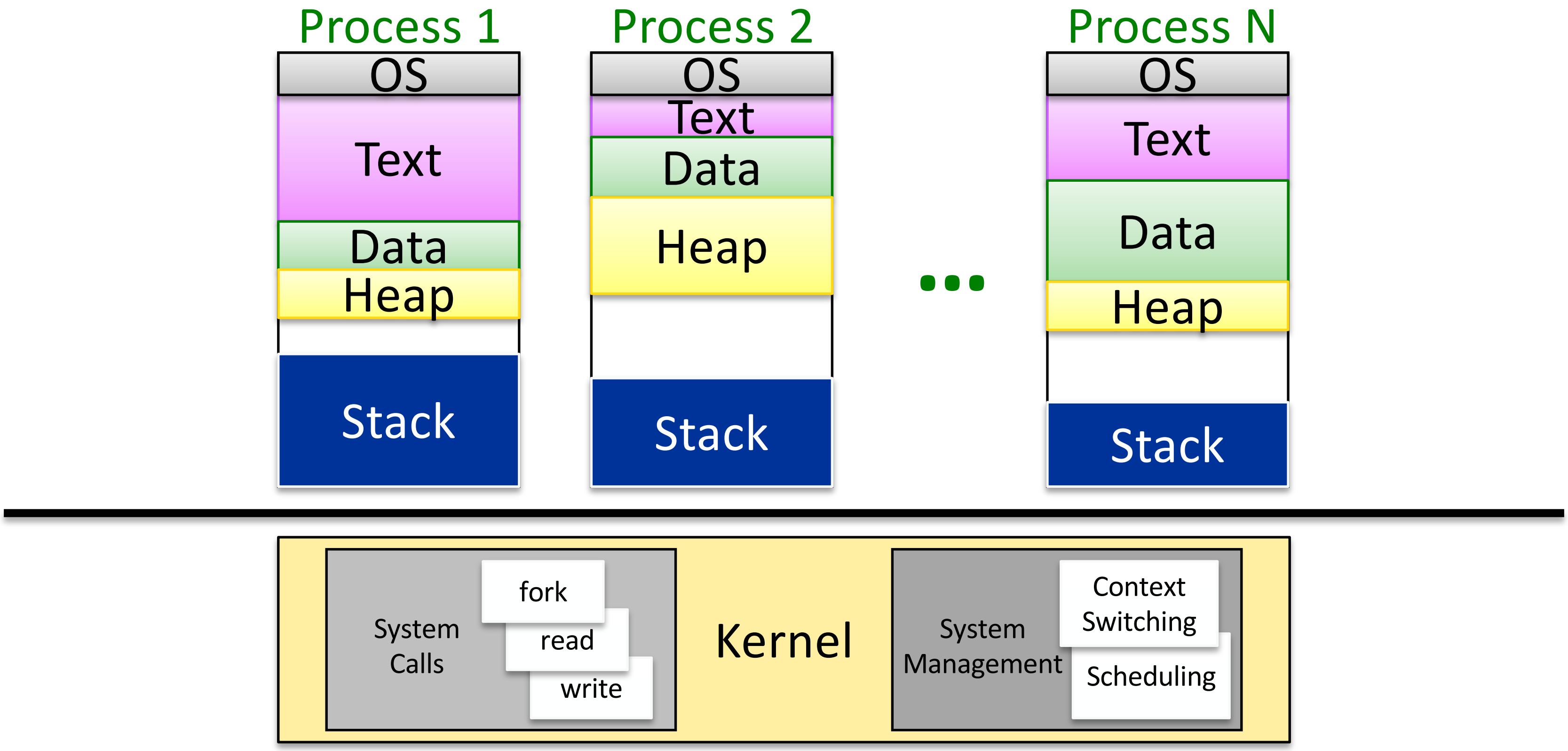
trusted

Hardware

- core of the OS
- code and data structures that are protected,
  can be accessed only in the kernel mode

Mode stored in a register

Kernel vs. Userspace: Model



Kernel vs. User Mode: Privileged Instructions

- User processes may not:
  - address I/O directly
  - use instructions that manipulate the OS's memory (e.g., page tables)
  - set the mode bits that determine user or kernel mode
  - disable and enable interrupts
  - halt the machine
- But in kernel mode, the OS does all these things.

OS: Taking Control of the CPU

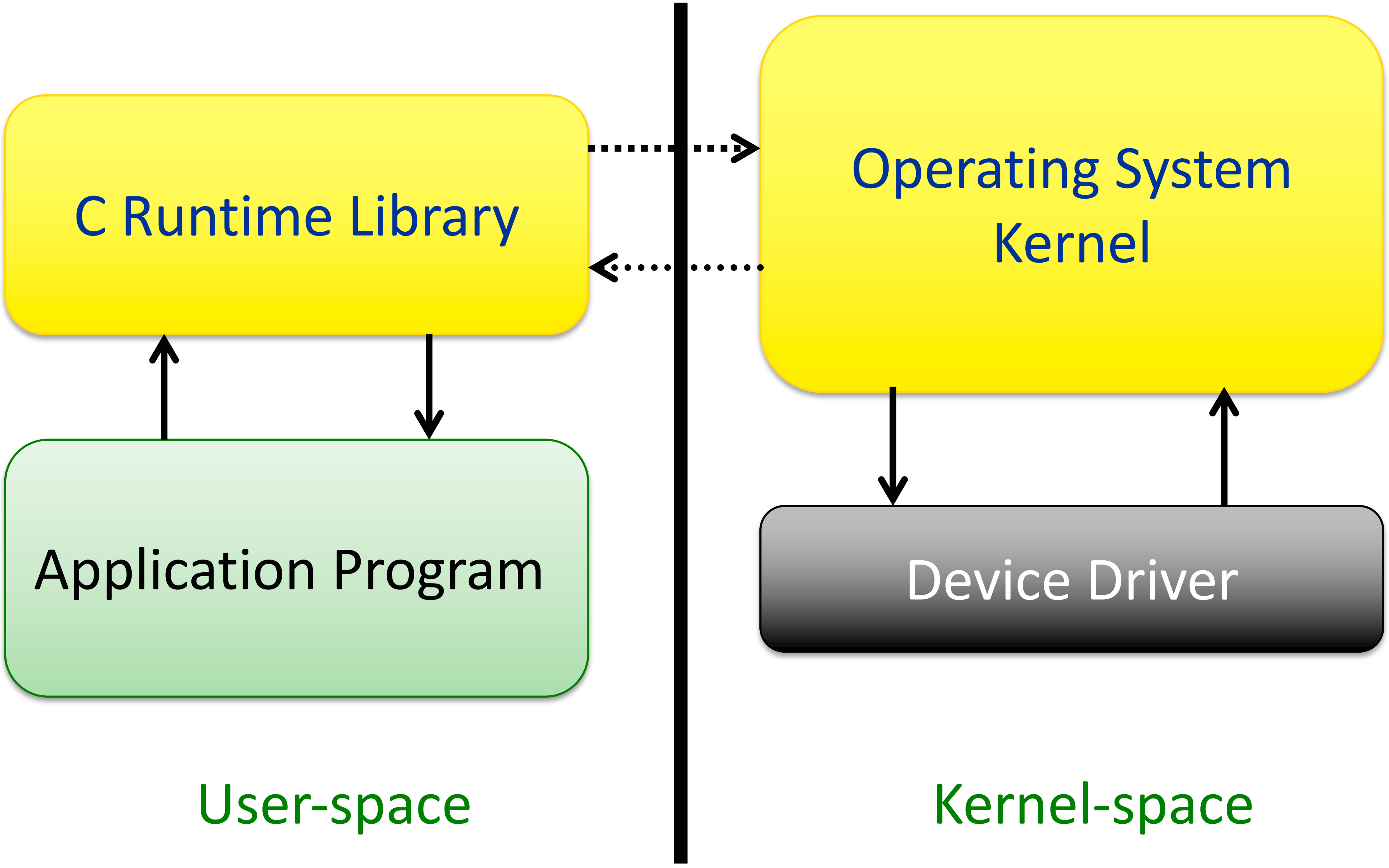
The terminology here is, unfortunately, muddy

1. System call/Trap – user requests service from the OS
2. Exception – user process has done something that requires help
3. (Hardware) interrupt – a device needs attention from the OS

System call often implemented as a special case of exception: execute intentional exception-generating instruction.

SYSTEM CALLS & LIBRARIES

How system calls work



C Runtime Library

Application Program

Operating System Kernel

Device Driver

User-space

Kernel-space

Synchronous request

Common Functionality

- Some functions useful to many programs
  - I/O device control
  - Memory allocation
- Place these functions in kernel
  - Explicitly called by programs (***system calls***)
  - Or accessed implicitly as needed (***exceptions***)
- Design questions:
  - What should these functions be?
  - How many programs should benefit?
  - Might kernel get too big?

How about a function like `printf()`?

Recall: What does `printf()` do?

A. `printf()` is a system call (why?)

B. `printf()` is not a system call (why not, what is it?)

Why make system calls?

A. Reliability: Kernel code always behaves the same.
B. Security: Programs can't use kernel code in unintended ways.
C. Usability: Kernel code is easier / adds value for programmers to use.
D. More than one of the above.
E. Some other reason(s).

Looking Ahead

- Git
- Project 1 Introduction