Today

- System debugging
- Dual Mode
- Processes
    - Handling interrupts

Review

- What are the benefits of version control?
- How is the bcc-compatible version of C different from the gcc-compatible version of C?

What is your debugging process?

DEBUGGING

Debugging as Engineering

- Much of your time in this course will be spent debugging
  - In industry, **50%** of software dev is debugging
  - Even more for kernel development
- How do you reduce time spent debugging?
  - Produce working code with smallest effort
- Optimize a process involving you, code, computer

Kernighan's Law: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

Debugging as Science

- **Understanding** → design → code
  - not the opposite
- Form a hypothesis that explains the bug
  - Which tests work, which don't? Why?
  - Add tests to narrow possible outcomes – what's the minimal input required to fail the test & reproduce the bug?
- Use best practices
  - Always walk through your code line by line
  - Unit tests – narrow scope of where problem is
  - Develop code in stages, with dummy stubs for later functionality

Project 1

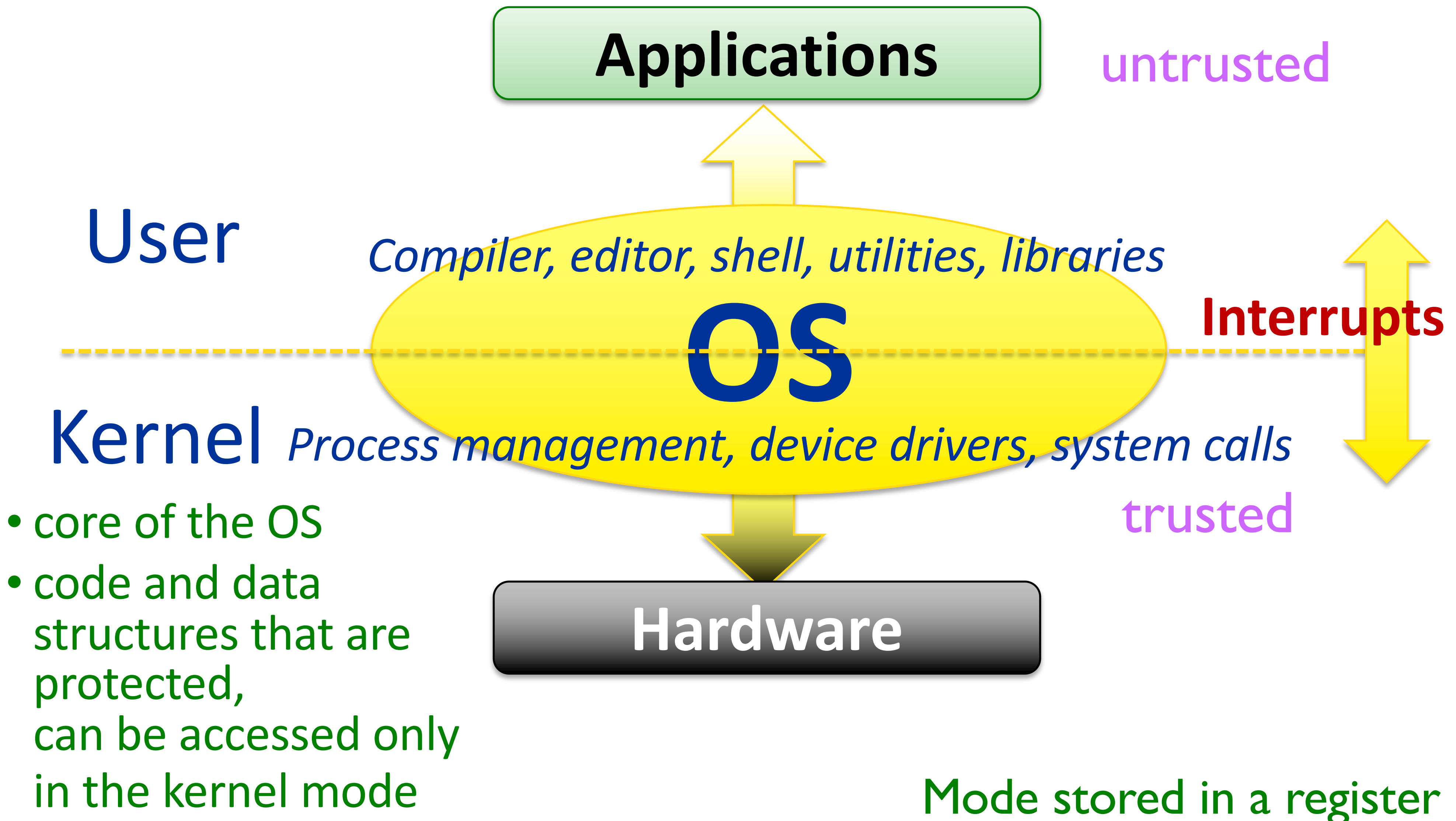
- Reload the Project 1 page whenever you return
- ~100 [final] lines of code
  - More lines written along the way for testing various pieces
- What should be in/updated in GitHub
  - kernel.c
  - Bash scripts and, optionally, Makefile
- What should **not** be in GitHub
  - Executables, the floppy image, log files from bochs, anything generated
    - Your scripts should generate these, so they do not need to be in GitHub

DUAL MODE

Review

- What is a process?
  - What resources does it require?
- Why do operating systems have dual modes?
  - What are those two modes?
- What causes a switch between the two modes?

Process Modes



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

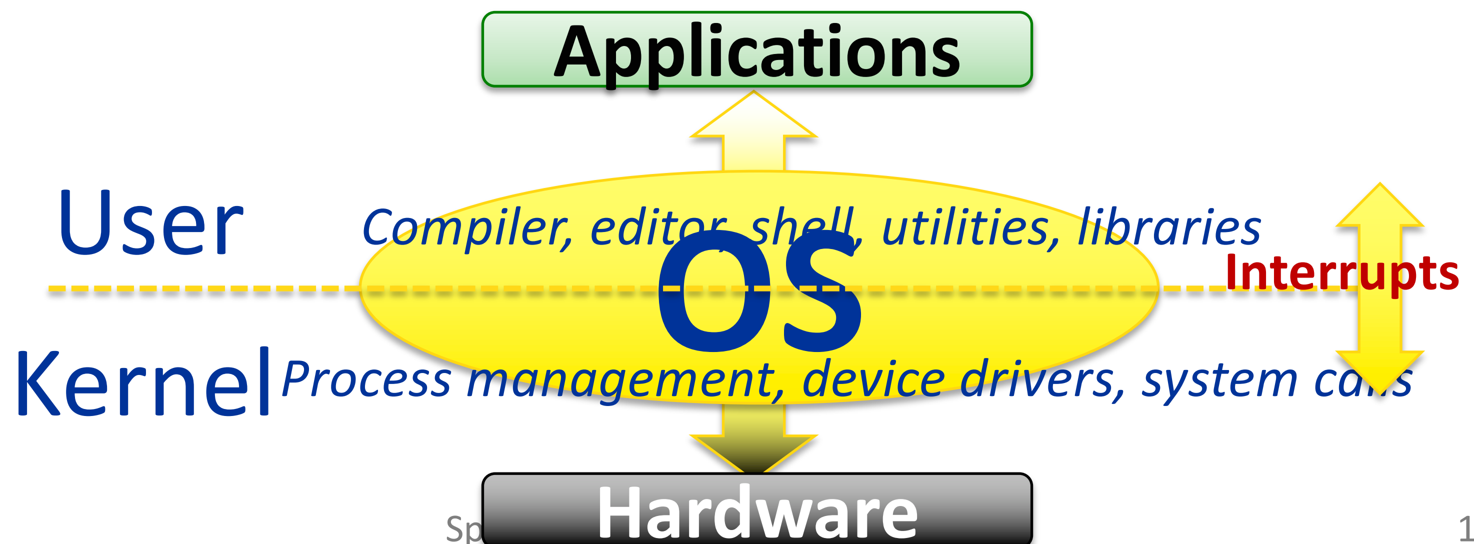
Mode stored in a register

OS is Interrupt-Driven

- Sits, waiting for something to happen
  - Alternative model: polling
- A ***trap*** or ***exception*** is a software-generated interrupt caused by a user request or an error

"Hey! Look at me!
I'm ready to do something!"

"Oopsies! I divided by 0!"



Exceptions: trap, fault, interrupt

	intentional happens every time	**unintentional** contributing factors
synchronous caused by an instruction	**trap: system call** user program requests. Examples: open, close, read, write, fork, exec, exit, wait, kill	**fault/exception** invalid or protected address or opcode, page fault, overflow, etc.
asynchronous caused by some other event	"software interrupt" software requests an interrupt to be delivered at a later time	**interrupt** caused by an external event (not related to instruction that just executed): I/O op completed, clock tick, power fail, etc.

Discussion: Interrupts vs Polling

- Why should OS's be interrupt-driven instead of polling?

How/When should the OS Kernel's code execute?

A. The kernel code is always executing.
B. The kernel code executes when a process asks it to.
C. The kernel code executes when the hardware needs it to.
D. The kernel code should execute as little as possible.
E. The kernel code executes at some other time(s).

How/When should the OS Kernel's code execute?

A. The kernel code is always executing
   - We don't want the kernel executing → it is taking valuable resources away from applications

B. The kernel code executes when a process asks it to.
   - Yes, through system calls

C. The kernel code executes when the hardware needs it to.
   - Yes, through interrupts

D. The kernel code should execute as little as possible.
   - Yes (see A)

E. The kernel code executes at some other time(s).

Same Question, Different Resource

- “How much of the system’s memory should the OS use?”
- Hopefully not much… just enough to get its work done.
- Leave the rest for the user!

Review: System Calls

- User programs are not allowed to access system resources directly
  - must ask OS to do that on their behalf
- System calls: set of functions for user programs to request for OS services
  - Run in *kernel* mode
  - Invoked by special instruction (trap/interrupt) causing the kernel to switch form user mode
  - When the system call finishes, processor returns to the user program and runs in user mode.

Why should processes make system calls?

A. Reliability: Kernel code always behaves the same.

B. Security: Programs can’t use kernel code or devices 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).

Why should processes make system calls?

A. Reliability: Kernel code always behaves the same.
   - We kind of assume this property of the kernel

B. Security: Programs can't use kernel code or devices in unintended ways.

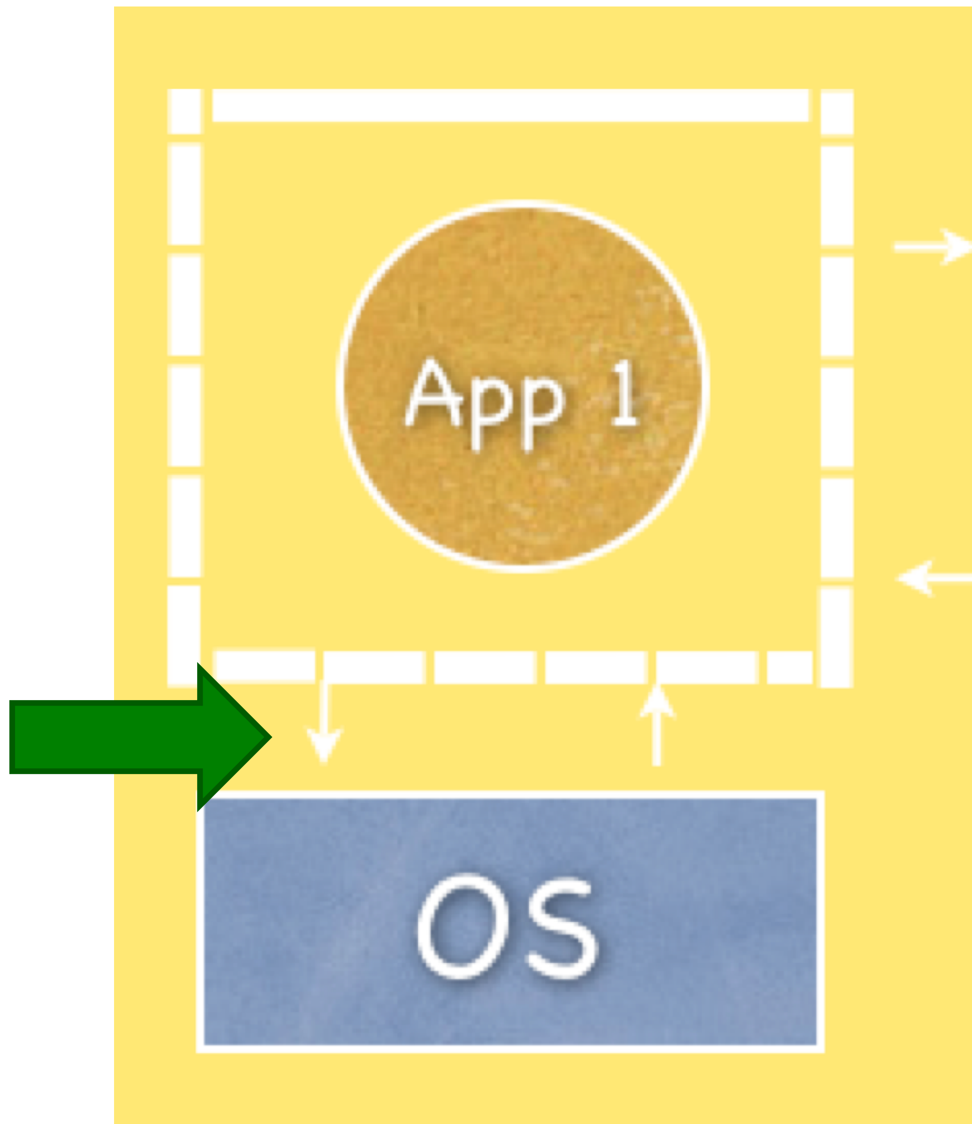
C. Usability: Kernel code is easier / adds value for programmers to use.
   - Adds common functionality that all apps benefits from

D. More than one of the above.

E. Some other reason(s).

A & B & C
(so D!)

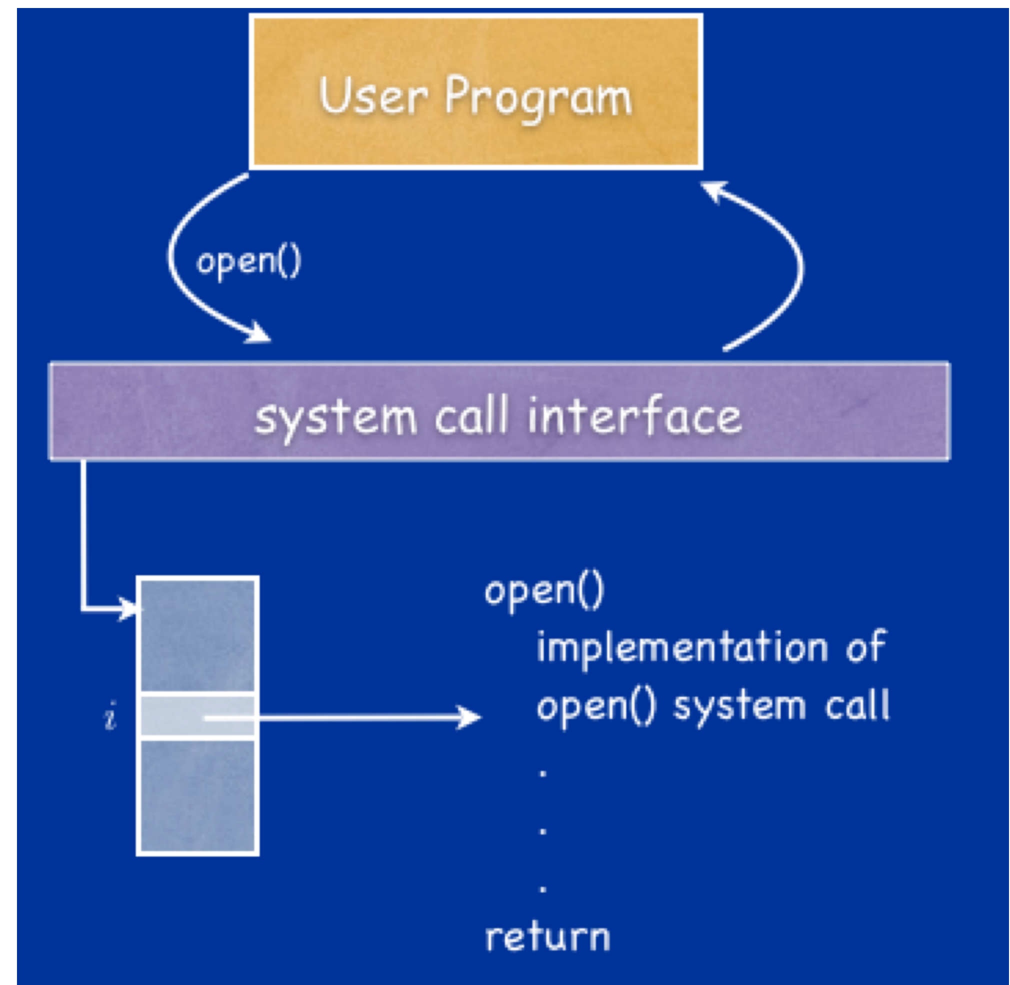
Review: Processes and Protection



- 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

System Calls

- A request by a user-level process to call a function in the kernel is a *system call*
  - Examples: read(), write(), exit()
- The interface between the application and the operating system (API)
  - Mostly accessed through *system-level libraries*
- Parameters passed according to calling convention
  - registers, stack, etc

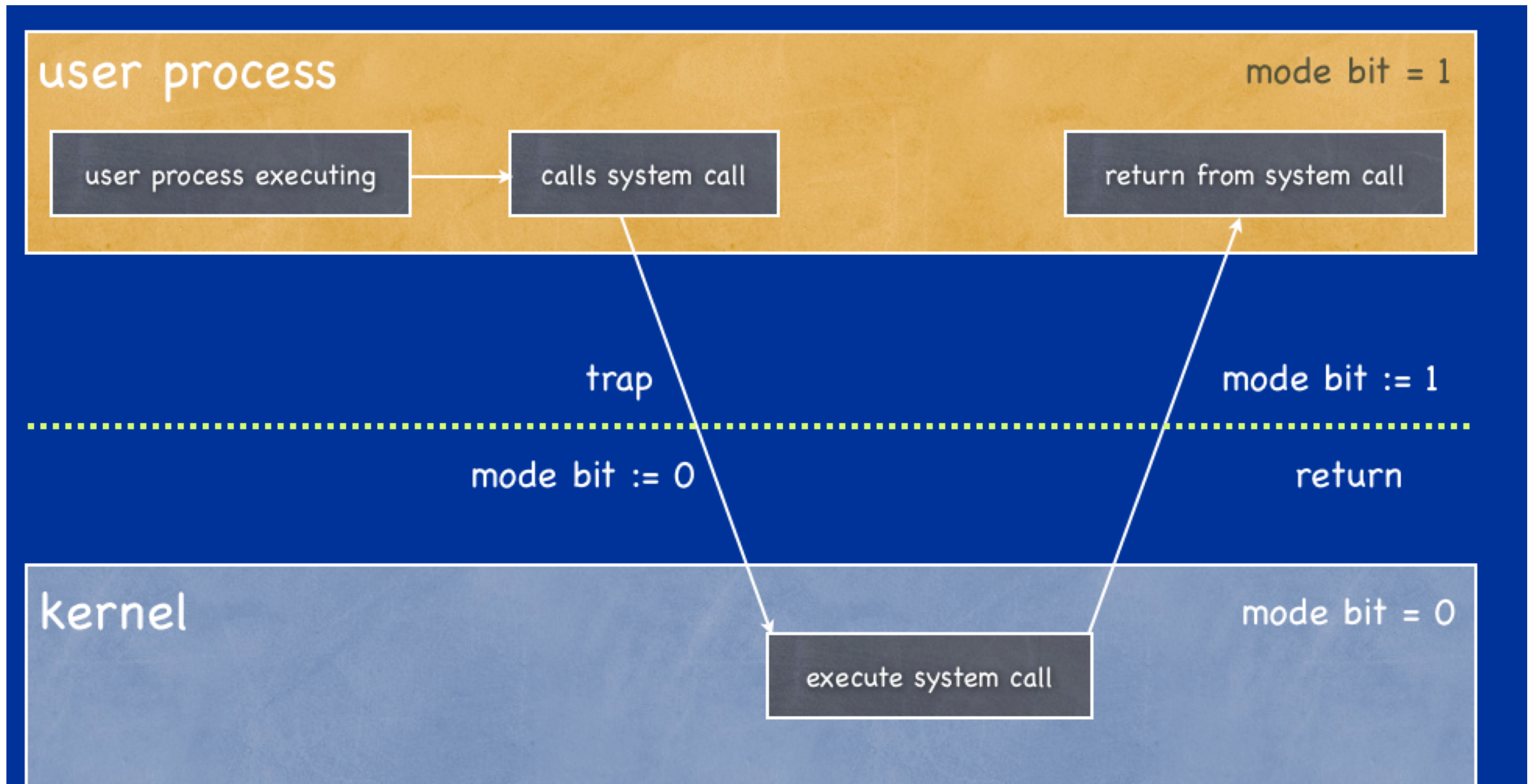


Project 2

System Calls: A Closer Look

- User process executes a trap instruction
- Hardware calls the OS at the system-call handler, a pre-specified location
- OS then:
  - identifies the required service and parameters
    - e.g., open(filename, O_RDONLY)
  - executes the required service
  - sets a register to contain the result of call
  - Executes a Return from Interrupt (RTI) instruction to return to the user program
- User program receives the result and continues

System Calls: A Closer Look

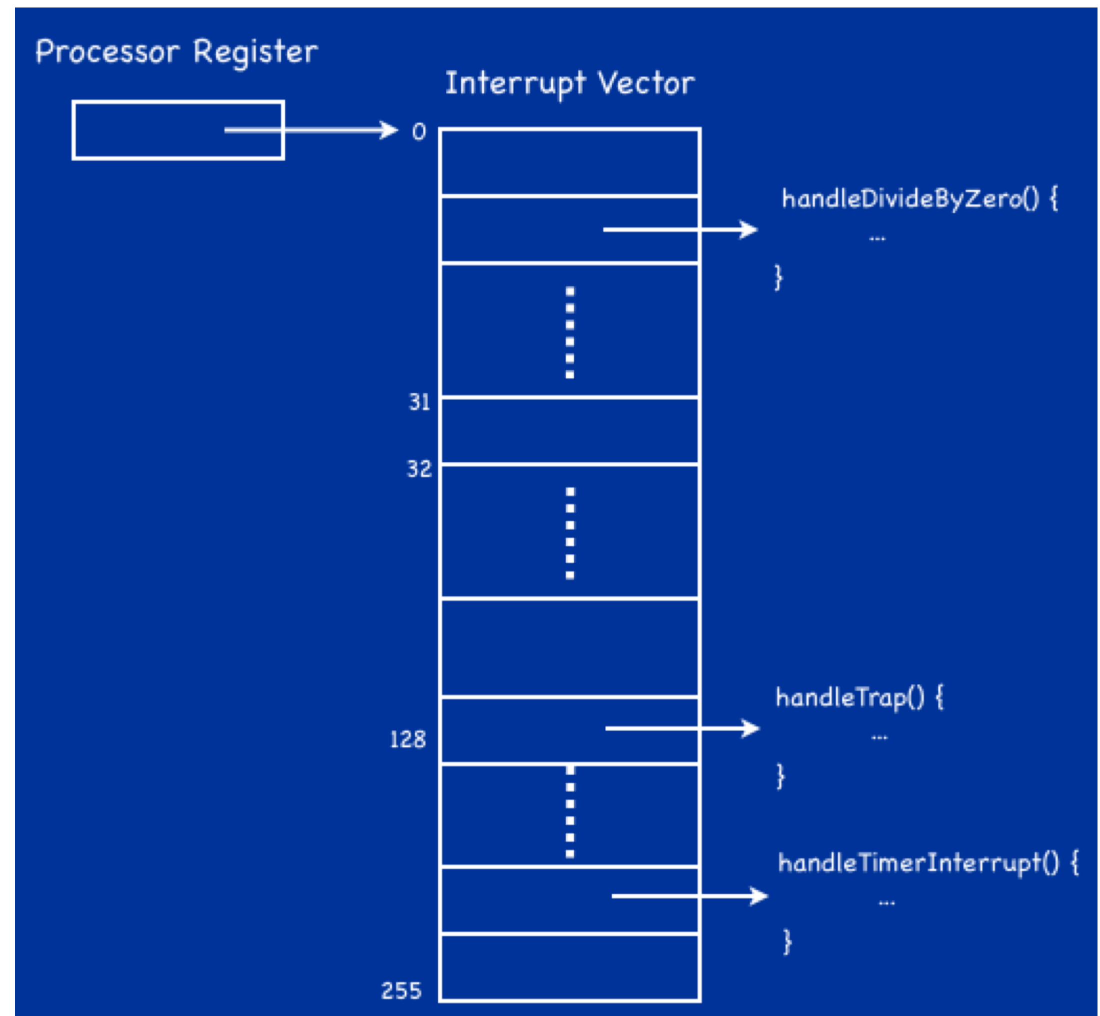


How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- ***Atomic*** transfer of control
  - A single instruction changes:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

User Mode to Kernel Mode: Details

- OS saves state of user program
- Hardware identifies why boundary is crossed
  - system call?
  - interrupt? then which hardware device?
  - which exception?
- Hardware selects entry from interrupt vector
- Appropriate handler is invoked



Project 2

How Process Works

1. Interrupt transfers control to the ***interrupt service routine (ISR)***
   - ISR is part of BIOS or OS
   - Generally, transferred through the **interrupt vector**, which contains the addresses of all the service routines
2. Interrupt architecture must save the address of the interrupted instruction
3. Figure out which system call made
4. Verify parameters
5. Execute request
6. Back to the calling instruction.

Vectored Interrupts

- Each device is assigned an ***interrupt request number*** (IRQ).
- The device's IRQ is used as an index into the ***interrupt vector***
  - The value at each index is the address of the ISR associated with the interrupt.
- The value from the interrupt vector is loaded into the PC

Saving the State of the Interrupted Process

- Privileged hw register points to ***Exception*** or ***Interrupt Stack***
  - on switch, hw pushes some of interrupted process registers (SP, PC, etc) on exception stack before handler runs. Why?
  - then handler pushes the rest
  - On return, do the reverse
- Why not use user-level stack?
  - reliability: even if user's stack points to invalid address, handlers continue to work
  - security: kernel state should not be stored in user space
    - could be read/written by user programs
- One interrupt stack per processor/process/thread

(stopped here)

Question

The interrupt vector is used to determine the action taken by the OS when:

A. An exception occurs
B. An interrupt occurs
C. A system call is executed
D. All of the above
E. None of the above

Switching Back!

- From an interrupt, just reverse all steps!
  - asynchronous, so not related to executing instruction
- From exception and system call, increment PC on return
  - synchronous, so you want to execute the *next* instruction, not the same one again!
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by the hardware

Dual Mode Execution: One Piece of the Protection Pie

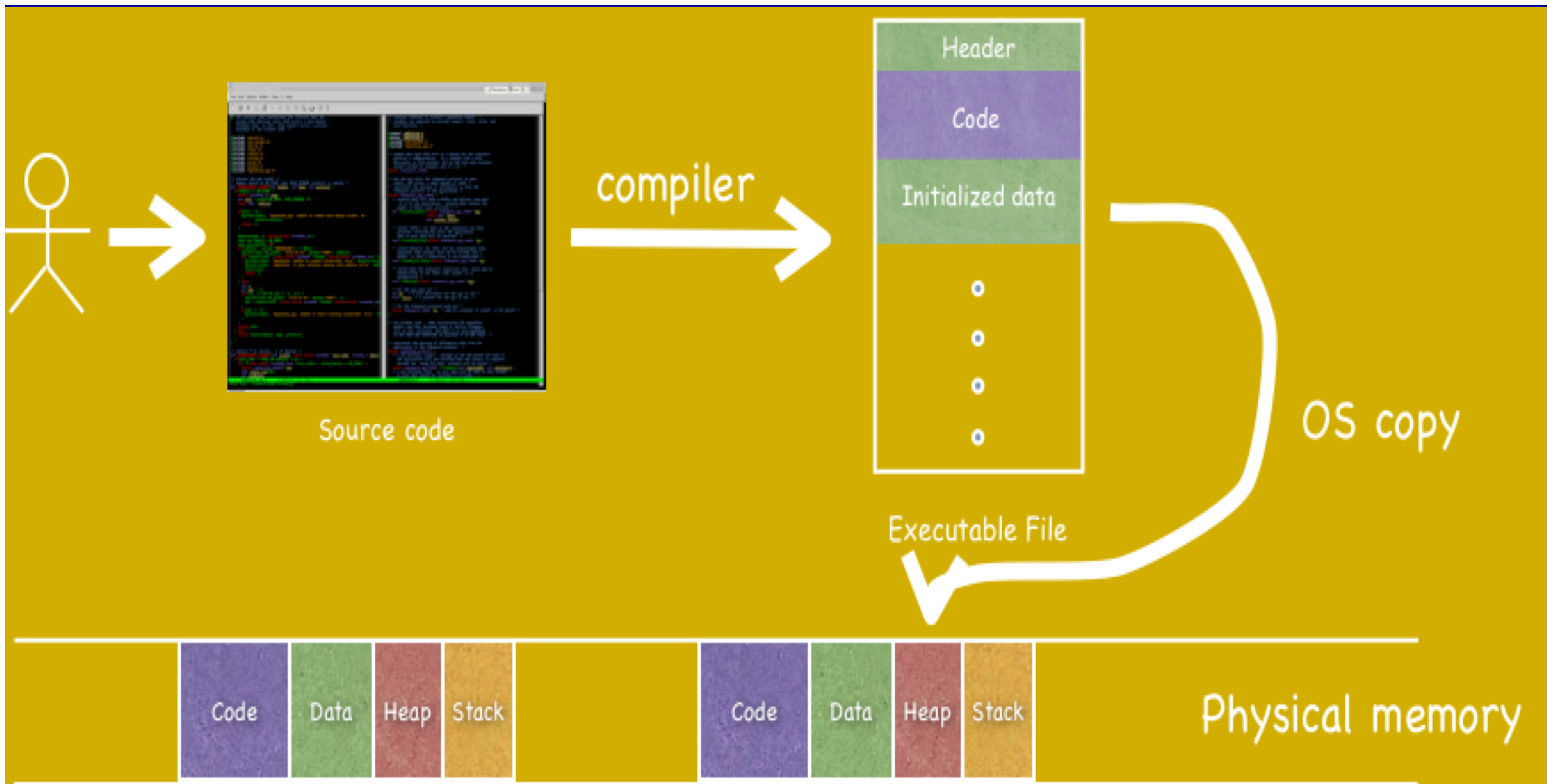
- For efficient protection, the hardware must support at least 3 features:
  - Privileged instructions
  - Timer interrupts
  - Memory protection

Dual Mode Execution: One Piece of the Protection Pie

- Privileged instructions
  - Instructions only available in kernel mode
  - In user mode, no way to execute potentially unsafe instructions
  - Prevents user processes from, for instance, halting the machine
  - Implementation: mode status bit in the process status register

Dual Mode Execution: One Piece of the Protection Pie

- Timer interrupts
  - Kernel must be able to periodically regain control from running process
  - Prevents process from gaining control of the CPU and never releasing it
  - Implementation: hardware timer can be set to expire after a delay and pass control back to the kernel


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

Dual Mode Execution: One Piece of the Protection Pie

- Memory protection
  - In user mode, memory accesses outside a process' memory region are prohibited
  - Prevents unauthorized access of data
  - Implementation: We'll return to this later in the course

Summary

- Operating System provides protection through dual-mode execution
  - Mode changes through interrupts (e.g., time slice), exceptions, or system calls.
  - A status bit in a protected processor register indicates the mode
  - Privileged instructions can only be executed in kernel mode