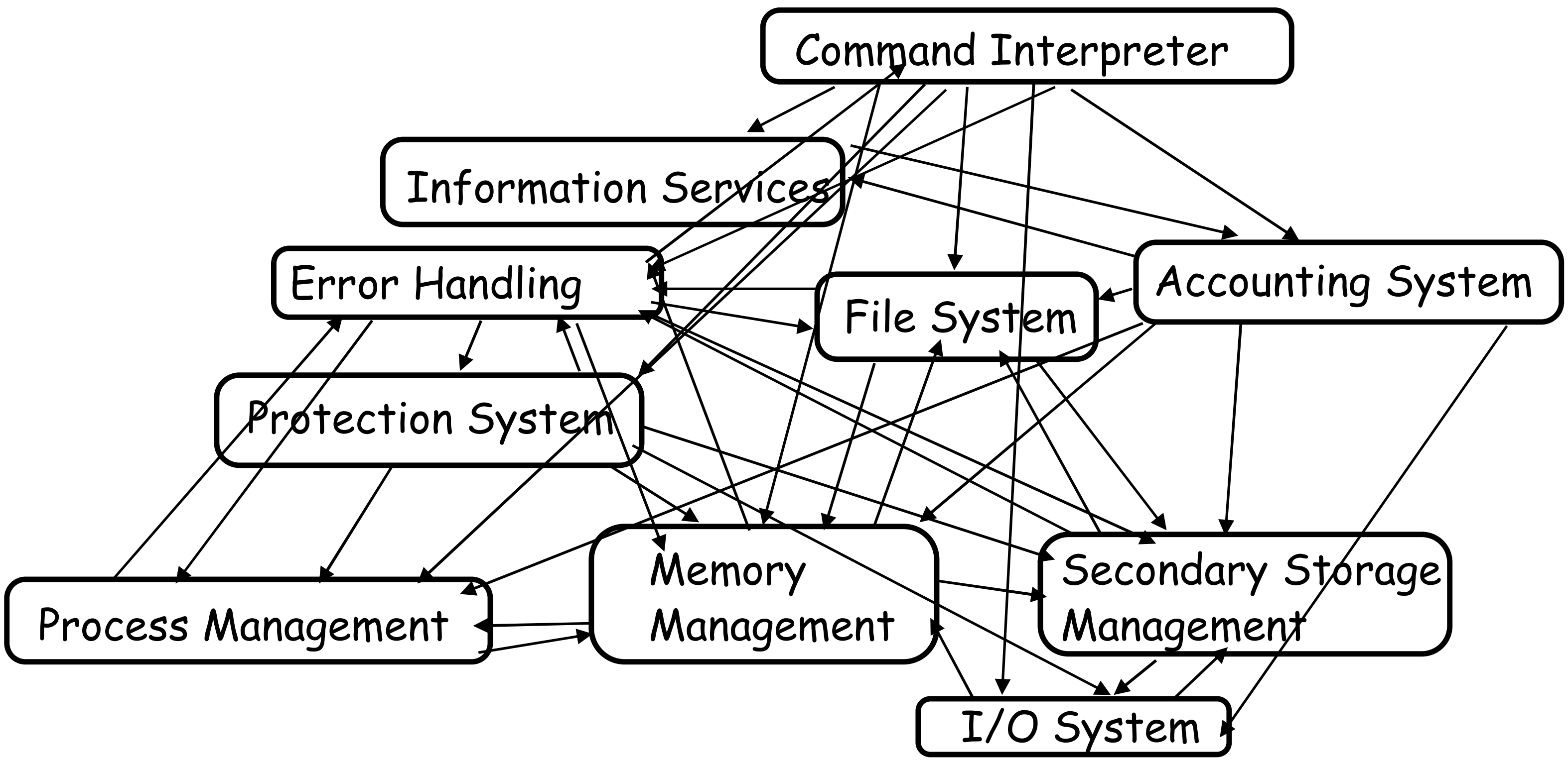
# CSE 451: Operating Systems
# Winter 2024

## Module 2
## Kernel Abstraction

Gary Kimura

# DEVELOPING AND DEBUGGING LARGE SYSTEMS



# Some Engineering Advice

- 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
  - When at all possible, code and test changes incrementally

# The science of debugging

- 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
  - Use best practices
    - Always walk through your code line by line
    - Module tests – narrow scope of where problem is
    - Develop code in stages, with dummy replacements for later functionality


# HARDWARE MODES

# Hardware Modes

- Who actually gets to control the hardware?
- The Application?
  - Advantages
  - Disadvantages (aka, what can possibly go wrong?)
- The Operating System?
  - Acting on behalf of the application
  - Advantages?
  - Disadvantages?

# Challenge: Protection using Restrictions

- How do we execute code with restricted privileges?
  - Either because the code is buggy or if it might be malicious
- Some examples:
  - A script running in a web browser
  - A program you just downloaded off the Internet
  - A program you just wrote that you haven't tested yet
  - Or the program that gets stuck in an infinite loop

# Hardware Support:  Dual-Mode Operation

- Kernel mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode
  - Limited privileges (How is this done?)
  - Only those granted by the operating system  kernel
- On the x86, mode stored in EFLAGS register
- On the MIPS, mode in the status register

# Hardware Support: Dual-Mode Restrictions

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel
  - To prevent user from reading data it shouldn't
- Timer
  - To regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

# Privileged instructions

- Examples
  - Halt Processor — Privileged
  - Disable interrupts — Privileged
  - Change mode — Privileged
  - Load and store — No, but there is a but…
- What should happen if a user program attempts to execute a privileged instruction?
  - An Exception is raised, and the OS takes control

# How to use the two modes

- It is a little naïve but okay to say that the OS only runs in kernel mode and user apps run in user mode.
  - Is that why they're called kernel mode and user mode?
- Important to understand when and how the system switches between the modes.
  - From Kernel Mode to User Mode
  - From User Mode to Kernel Mode

# Mode Switch (Kernel to User)

- Without getting into what is running here is generally how one goes from kernel mode to user mode
  1. New process/new thread start
     - Jump to first instruction in program/thread
  2. Return from interrupt, exception, system call
     - Resume suspended execution
  3. Process/thread context switch
     - Resume some other process
  4. User-level upcall (UNIX signal)
     - Asynchronous notification to user program

# Mode Switch (User to Kernel)

- From user mode to kernel mode
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior or malicious behavior!
  - System calls (aka protected procedure call, or a trap)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

# Device Interrupts: Example

- Here is the situation: The OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
  - One solution is polling: Kernel waits until I/O is done
  - Another solution are Interrupts: Kernel can do other work in the meantime
- Example: Device access to memory
  1. Programmed I/O: CPU reads and writes to device
  2. Device has Direct memory access (DMA)
  3. When I/O completes the Device interrupts the CPU

# How do Device Interrupts work?

- Where does the CPU run after an interrupt? Kernel
- What stack does it use? Kernel Stack
- Is the work the CPU had been doing before the interrupt lost forever? No
- If not, how does the CPU know how to resume that work? We'll see

# Example of an Interrupt: Hardware Timer

Hardware device that periodically interrupts the processor

- Returns control to the kernel handler
- Interrupt frequency set by the kernel and not by user code

Side note: Interrupts can be temporarily deferred by the kernel

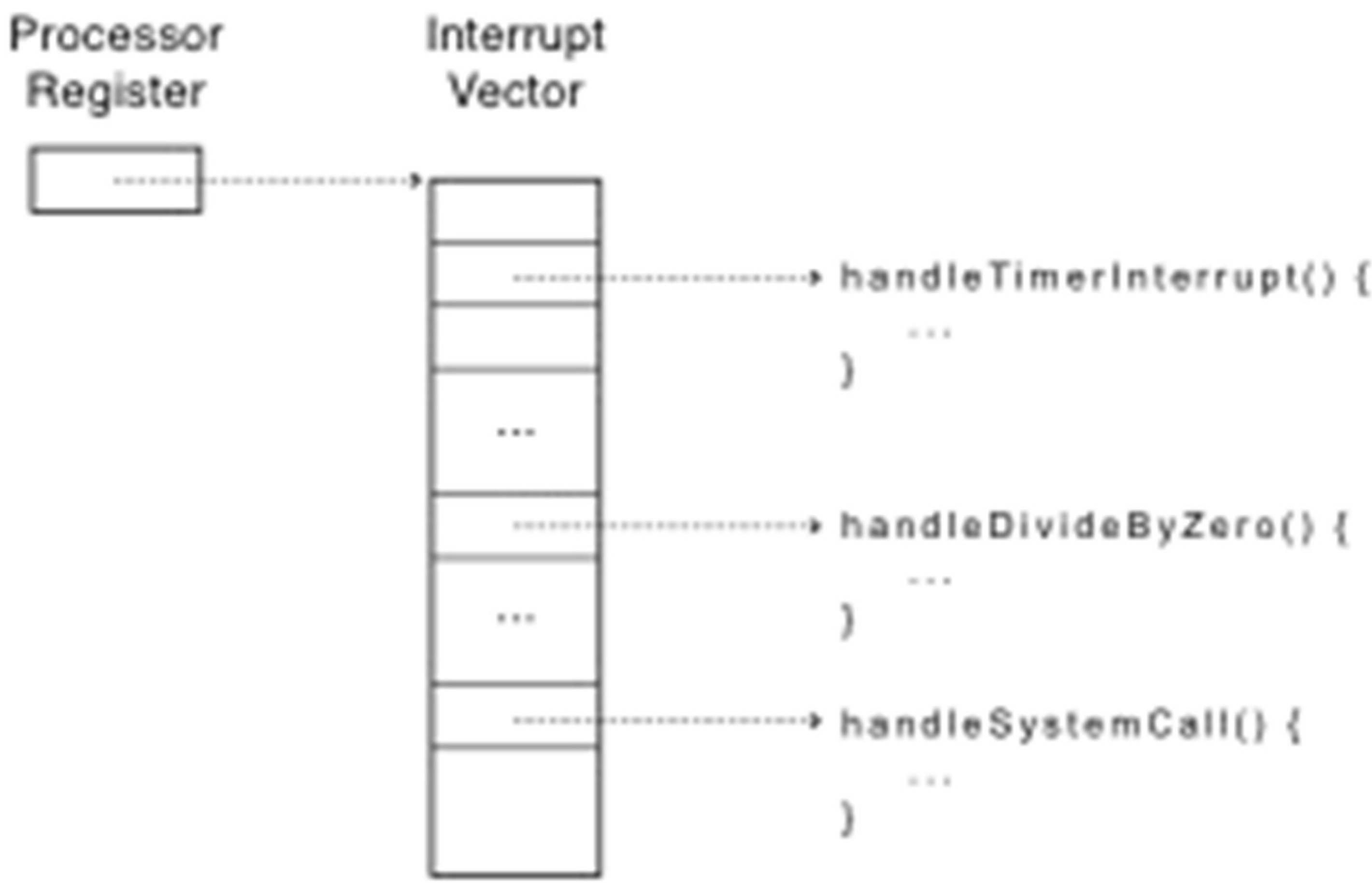
- But not by user code!
- Interrupt deferral crucial for implementing mutual exclusion

# How do we take interrupts safely?

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

# Interrupt Vector

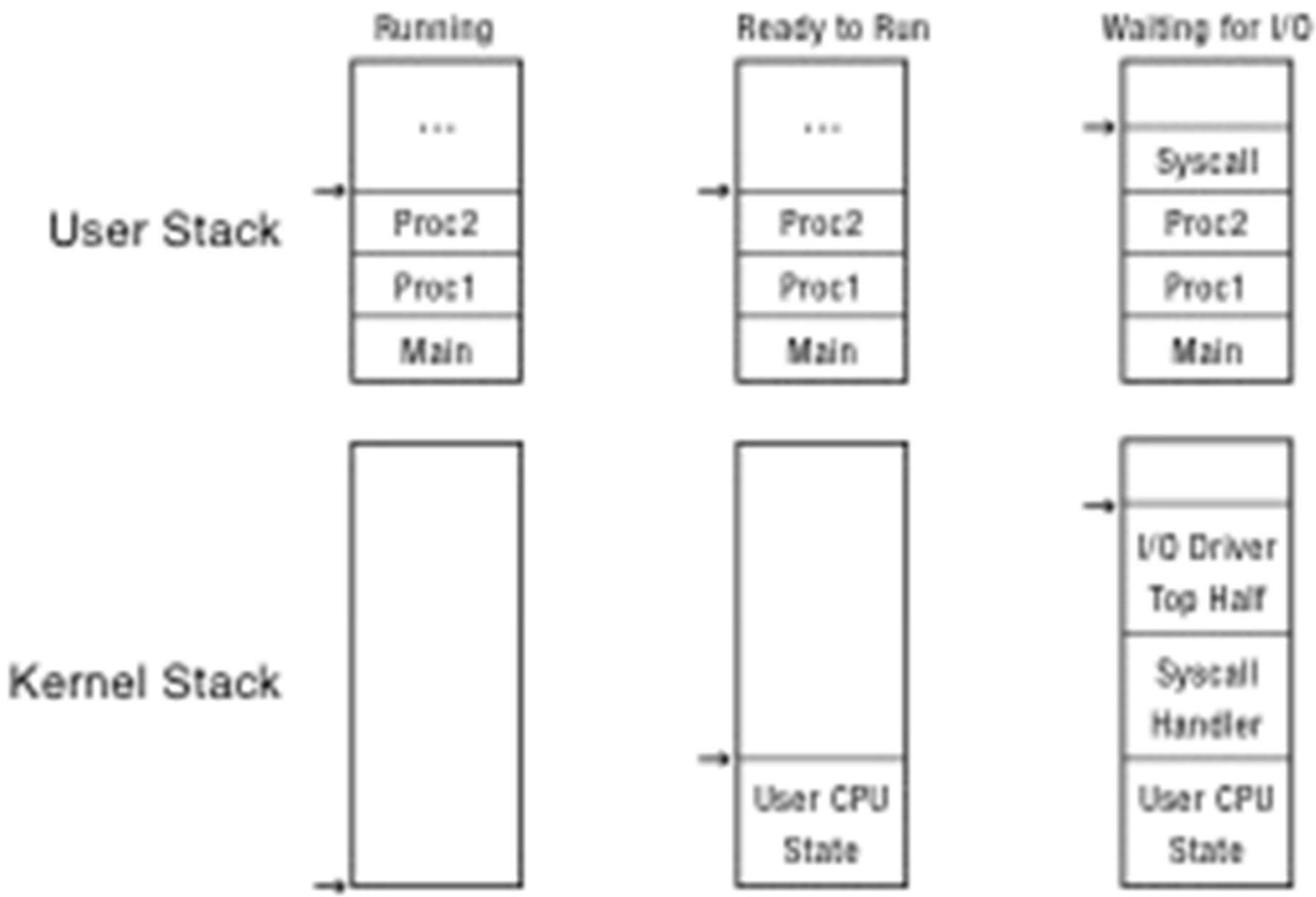
- Table set up by OS kernel; pointers to code to run on different events



# Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a process/thread has both: kernel and user stack
- Why can't the interrupt handler run on the stack of the interrupted user process?

# Interrupt Stack



# Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86
    - CLI: disable interrrupts
    - STI: enable interrupts
    - Only applies to the current CPU (on a multicore)
- We'll need this to implement synchronization in  chapter 5

# Interrupt Handlers

- Often part of a device driver
- Non-blocking, run to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
    - Linux: semaphore
- Rest of device driver runs as a kernel thread

# Case Study: MIPS Interrupt/Trap

- Two entry points: TLB miss handler, everything else
- Save type: syscall, exception, interrupt
  - And which type of interrupt/exception
- Save program counter: where to resume
- Save old mode, interruptable bits to status register
- Set mode bit to kernel
- Set interrupts disabled
- For memory faults
    - Save virtual address and virtual page
      - Jump to general exception handler

# Case Study: x86 Interrupt

- Save current stack pointer
- Save current program counter
- Save current processor status word (condition  codes)
- Switch to kernel stack; put SP, PC, PSW on stack
- Switch to kernel mode
- Vector through interrupt table
- Interrupt handler saves registers it might clobber

# At end of Interrupt Handler

- Handler restores saved registers
- Atomically return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

# Summary: Entering the Kernel

As a rule of thumb the kernel gets executed (entered) through interrupts, exceptions, and system calls.

- Interrupts – a device needs servicing; the OS will continue the interrupted process when able
- Exceptions – a process did something that the OS needs to fix
- System calls – a process is asking the OS to perform a privileged operation

Exceptions and System calls serve a different scenario than Interrupts, but share much of the same mechanism

# Exceptions and System Calls

Examples of exceptions

- divide by zero
  - ✓ overflow or underflow
- illegal Instruction
- load/store from a protected location

Examples of system calls

- open/create a file
- read/write from a file
- allocate memory (e.g., malloc)
  - ✓ Sometimes these are handled in user mode libraries

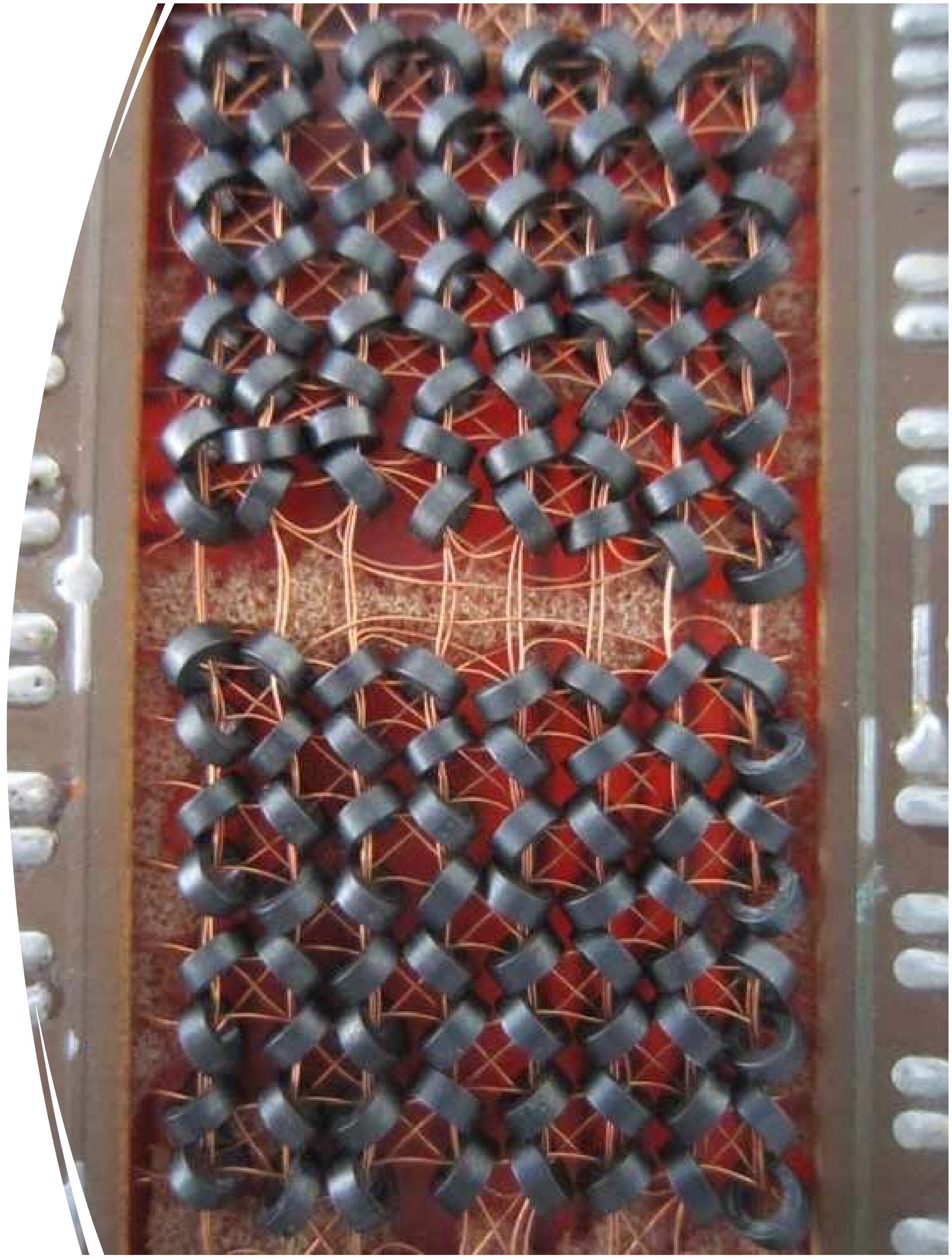
# Dealing with Exceptions

- OS can choose to fix the program's exception
  - For example, make an illegal memory address legal
- OS can choose to alert the program of the exception
  - For example, divide by zero
- OS can choose to terminate the program
- Are there other choices?

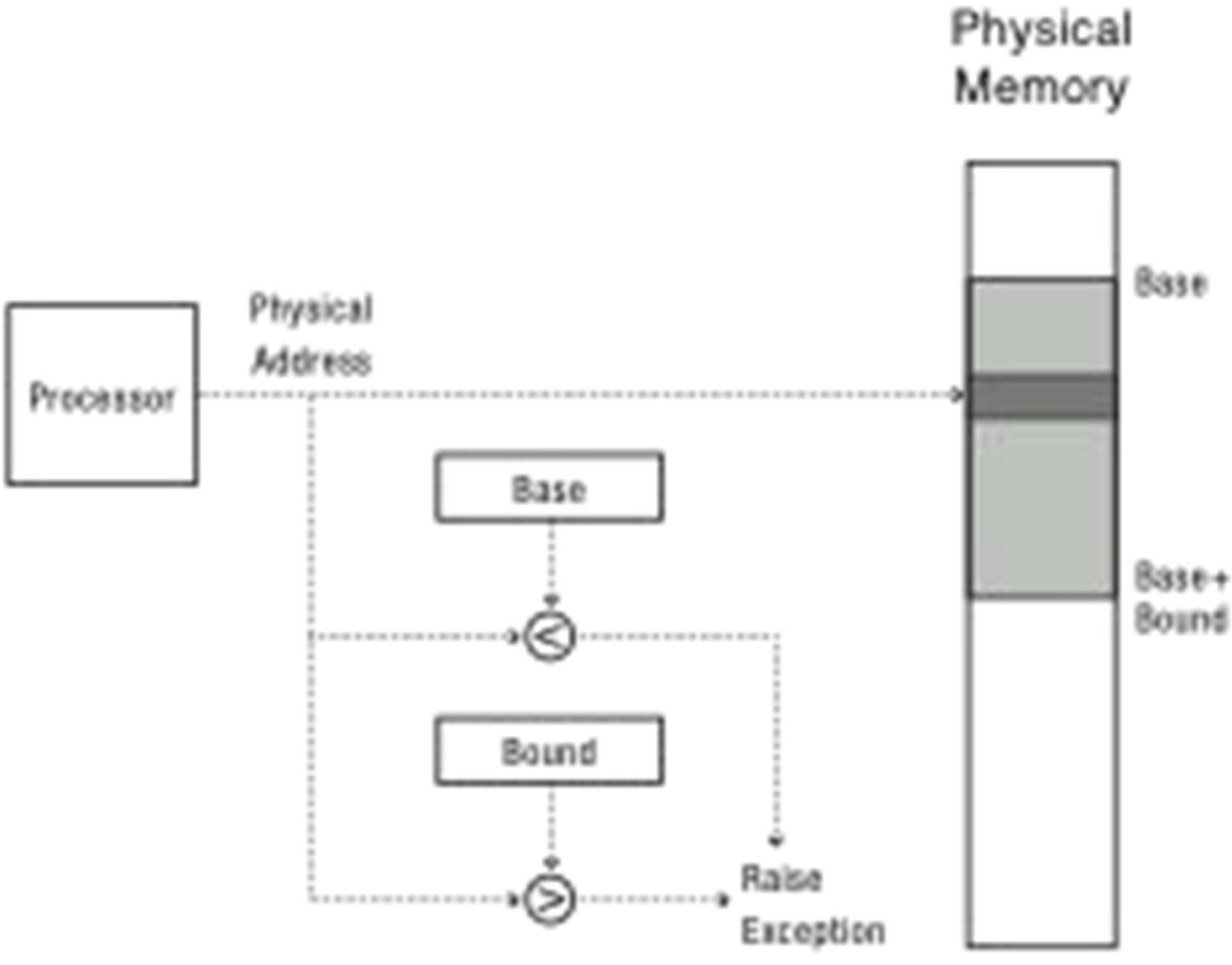
# Dealing with System Calls

- Locate arguments
  - In registers or on user stack
  - *Translate* user addresses into kernel addresses
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back into user memory
  - *Translate* kernel addresses into user addresses

# MEMORY LAYOUT



# Simple Memory Protection

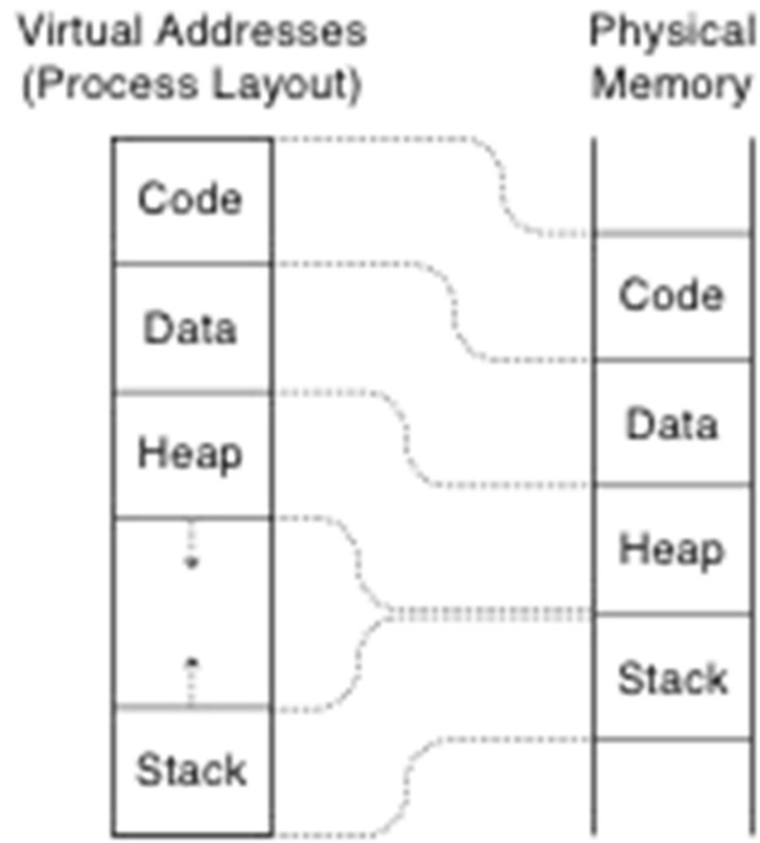


# Towards Virtual Addresses

- Problems with base and bounds?

# Virtual Addresses

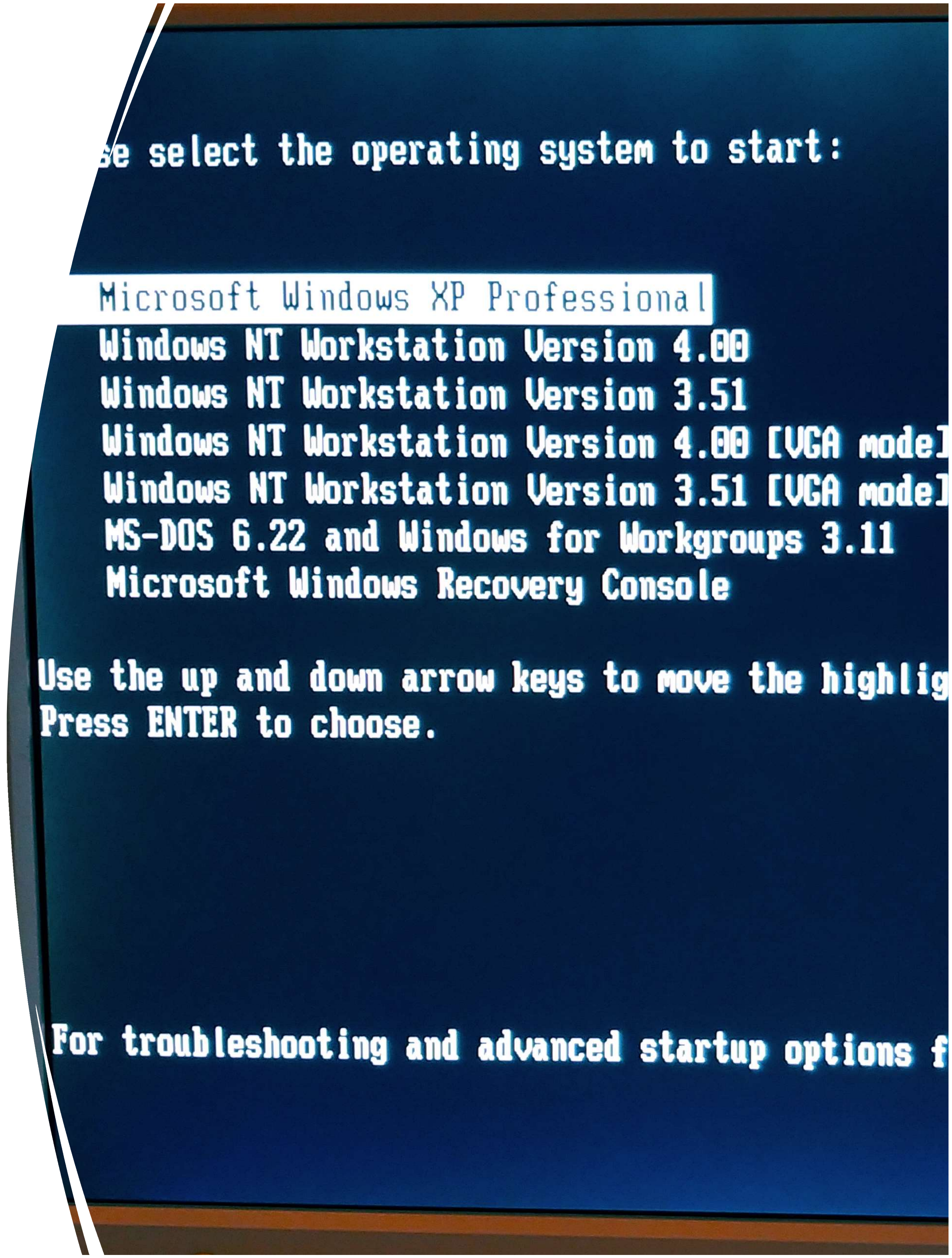
- Translation done in hardware, using a table
- Table set up by operating system kernel



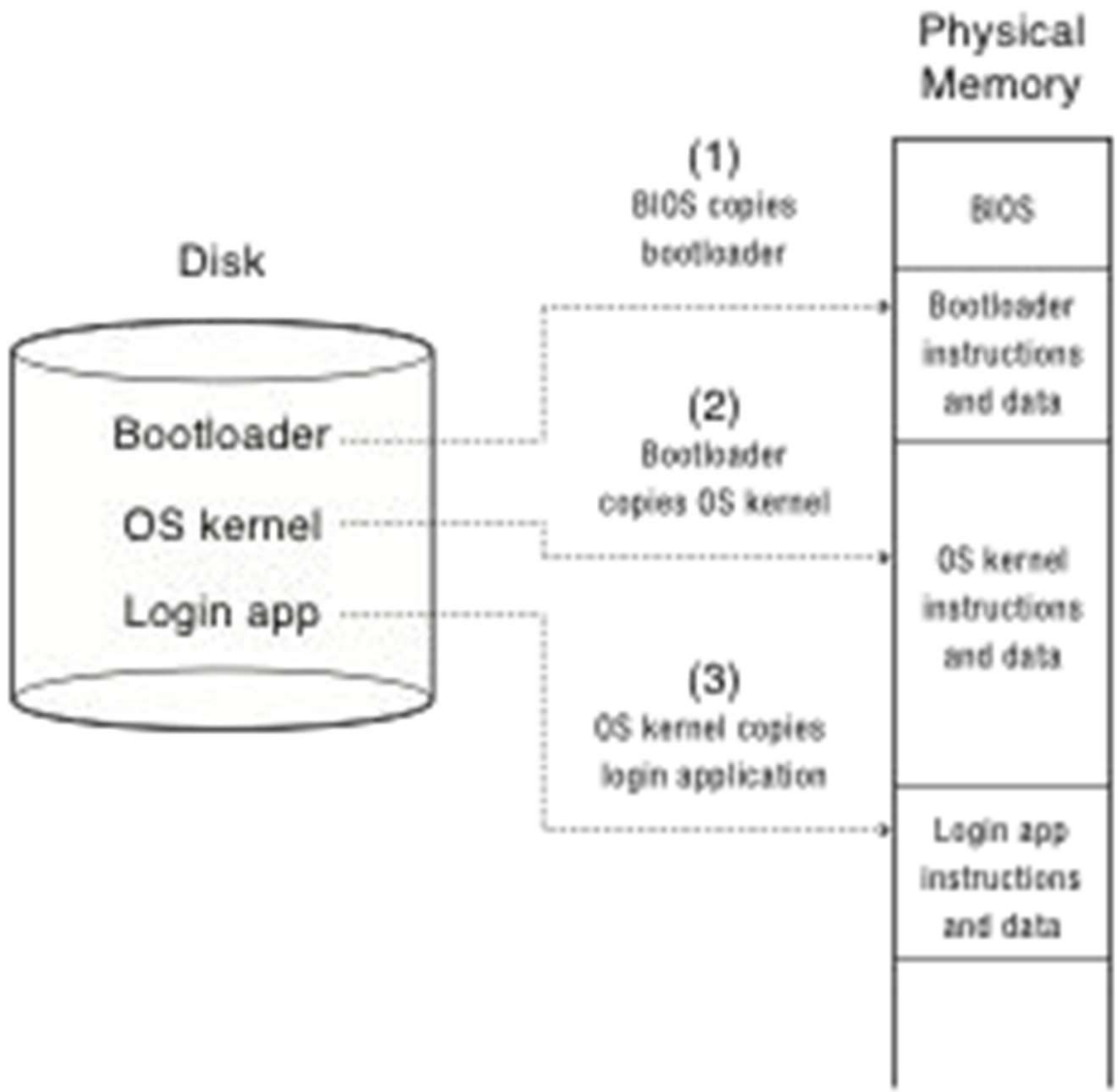
# Division between User and Kernel memory

| | |
|---|---|
| User virtual address space: | 0x00000000 and 0x7FFFFFFF |
| Kernel virtual address space: | 0x80000000 and 0xFFFFFFFF |

# HOW DO WE BOOT THIS THING?



# Booting


Physical
Memory
Disk
Bootloader
OS kernel
Login app
(1)
BIOS copies
bootloader
(2)
Bootloader
copies OS kernel
(3)
OS kernel copies
login application
BIOS
Bootloader
instructions
and data
OS kernel
instructions
and data
Login app
instructions
and data

# Next up

Processes: Chapter 3 (first part) and Chapter 4