

# Hardware Issues for Operating Systems

CS 111

Operating Systems

Peter Reiher

# Outline

- Hardware and the operating system
- Processor issues
- Buses and devices
  - Disk drives
- We'll talk about memory later

# Hardware and the Operating System

- One of the major roles of the operating system is to hide details of the hardware
  - Messy and difficult details
  - Specifics of particular pieces of hardware
  - Details that prevent safe operation of the computer
- OS abstractions are built on the hardware, at the bottom
  - Everything ultimately relies on hardware
- A major element of OS design concerns HW

# OS Abstractions and the Hardware

- Many important OS abstractions aren't supported directly by the hardware
- Virtual machines
  - There's one real machine
- Virtual memory
  - There's one set of physical memory
  - And it often isn't as big as even one process thinks it is
- Typical file abstractions
- Many others
- The OS works hard to make up the differences

# Hiding Grubby Details

- Maybe I don't have floating point hardware
- Maybe I have a RAID instead of a single hard disk
- I might have two printers with different capabilities
- I might periodically switch between using Ethernet or 802.11 for my network
- My users don't want to know any of this
- And couldn't handle it if they did

# Safety Issues

- If the machine is doing multiprocessing, failures in one process shouldn't hurt another
- If process A divides by zero, that's not process B's problem
- If process C and process D both ask to get data off the disk, they should only see their own data
- Only the OS knows enough and is trusted enough to handle safety issues

# Processor Issues

- Execution mode
- Handling exceptions

# Execution Modes

- Modern CPUs can execute in two different modes:
  - User mode
  - Supervisor mode
- User mode is to run ordinary programs
- Supervisor mode is for OS use
  - To perform overall control
  - To perform unsafe operations on the behalf of processes



# User Mode

- Allows use of all the “normal” instructions
  - Load and store general registers from/to memory
  - Arithmetic, logical, test, compare, data copying
  - Branches and subroutine calls
- Able to address some subset of memory
  - Controlled by a Memory Management Unit
- Not able to perform privileged operations
  - I/O operations, update the MMU
  - Enable interrupts, enter supervisor mode

# Why Only a Subset of Memory?

- Why do we limit user-mode execution to a sub-set of memory?
- What if a user mode process could access all of memory?
  - It could see or even potentially corrupt data belonging to other processes
  - It could even crash the operating system
- The subset it sees relates to its own data and program
  - So it can only screw itself

# Supervisor Mode

- Allows execution of privileged instructions
  - To perform I/O operations
  - Interrupt enable/disable/return, load PC
  - Instructions to change processor mode
- Can access privileged address spaces
  - Data structures inside the OS
  - Other process's address spaces
  - Can change and create address spaces
- May have alternate registers, alternate stack

# Controlling the Processor Mode

- Typically controlled by the *Processor Status Register* (AKA PS)
- PS also contains condition codes
  - Set by arithmetic/logical operations (0,+,-,ovflo)
  - Tested by conditional branch instructions
- Describes which interrupts are enabled
- May describe which address space to use
- May control other processor features/options
  - Word length, endian-ness, instruction set, ...

# How Do Modes Get Set?

- The computer boots up in supervisor mode
  - Used by bootstrap and OS to initialize the system
- Applications run in user mode
  - OS changes to user mode before running user code
    - User programs cannot do I/O, restricted address space
  - They can't arbitrarily enter supervisor mode
    - Because instructions to change the mode are privileged
- Re-entering supervisor mode is strictly controlled
  - Only in response to traps and interrupts

# So When Do We Go Back To Supervisor Mode?

- In several circumstances
- When a program needs OS services
  - Invokes system call that causes a trap
  - Which returns system to supervisor mode
- When an error occurs
  - Which requires OS to clean up
- When an interrupt occurs
  - Clock interrupts (often set by OS itself)
  - Device interrupts

# Asynchronous Exceptions and Handlers

- Most program errors can be handled “in-line”
  - Overflows may not be errors, noted in condition codes
  - If concerned, program can test for such conditions
- Some errors must interrupt program execution
  - Unable to execute last instruction (e.g. illegal op)
  - Last instruction produced non-results (e.g. divide by zero)
  - Problem unrelated to program (e.g. power failure)
- Most computers use traps to inform OS of problems
  - Define a well specified list of all possible exceptions
  - Provide means for OS to associate handler with each

# Why Not Check It All In User Mode?

- Can't my program handle all its own errors?
- Sometimes an instruction couldn't be executed at all
  - A failure of the virtual execution engine
- Can't check all possible errors after each and every instruction
  - Would require dozens of checks per instruction
  - When the failures are extremely rare, it makes more sense to raise an exception condition



# Control of Supervisor Mode Transitions

- All user-to-supervisor changes via traps/interrupts
  - These happen at unpredictable times
- There is a designated handler for each trap/interrupt
  - Its address is stored in a trap/interrupt vector table
  - The operating system sets up all of the handler vectors
- Ordinary programs can't access these vectors
  - Vectors are not in the process' address spaces
- The OS controls all supervisor mode transitions
  - By carefully controlling all of the trap/interrupt “gateways”

# Transition Into Supervisor Mode

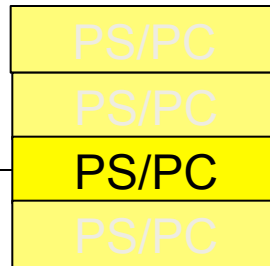
- Due to either hardware or software trap
- Hardware trap handling
  - Trap cause provides index into trap vector table
  - Load new processor status word, switch to supervisor mode
  - Push PC/PS of program that caused trap onto stack
  - Load new program counter from trap vector table entry
- Software trap handling
  - 1st level handler pushes all other registers onto stack
  - 1st level handler gathers info, selects 2nd level handler
  - 2nd level handler deals with the exception condition

# Software Trap Handling

Application Program

instr ; instr ; instr ; instr ; instr ; instr ;

user mode  
supervisor mode



TRAP vector table

1<sup>st</sup> level trap handler  
(saves registers and  
selects 2<sup>nd</sup> level handler)

2<sup>nd</sup> level handler  
(actually deals  
with the problem)

return to  
user mode

# Dealing With the Cause of a Trap

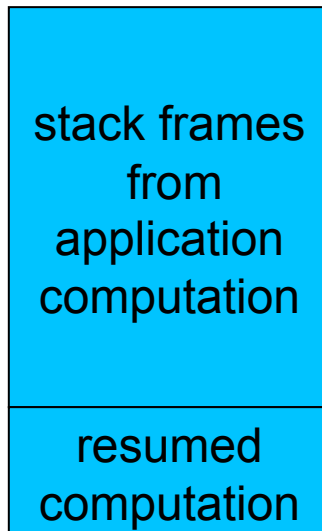
- Some exceptions are handled by the OS
  - E.g. page faults, alignment, floating point emulation
  - OS simulates expected behavior and returns
- Some exceptions may be fatal to running task
  - E.g. zero divide, illegal instruction, invalid address
  - OS reflects the failure back to the running process
- Some exceptions may be fatal to the system
  - E.g. power failure, cache parity, stack violation
  - OS cleanly shuts down the affected hardware

# Returning To User Mode

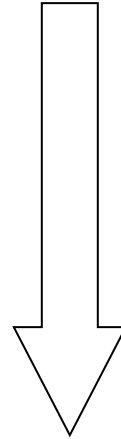
- Return is opposite of interrupt/trap entry
  - 2nd level handler returns to 1st level handler
  - 1st level handler restores all registers from stack
  - Use privileged return instruction to restore PC/PS
  - Resume user-mode execution after trapped instruction
- Saved registers can be changed before return
  - To set entry point for newly loaded programs
  - To deliver signals to user-mode processes
  - To set return codes from system calls

# Stacking and Unstacking a Trap

User-mode Stack

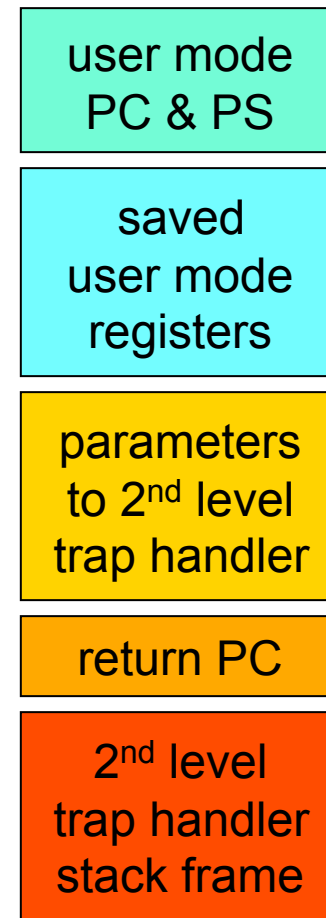


**TRAP!**



direction  
of growth

Supervisor-mode Stack



# Traps While In Supervisor Mode

- Nearly identical to traps while in user mode
  - Trap saves interrupted PC/PS on supervisor stack
  - Trap goes to same vector & 1st level handler
  - Same register saving, restoring, and return
- There are very few differences
  - Saved PS at interrupt time shows supervisor mode
  - 2nd level handler knows trap was from supervisor mode
  - May be more or less severe than the same trap from user mode

# Traps and Protection

- The OS is very careful in protecting trap vectors
- Why?
- The trap vector specifies the code and mode to be executed when an exception occurs
- If a user-mode program could change these vectors, it could execute arbitrary code
  - In supervisor mode
  - Bypassing all of the built-in protections



# I/O Architecture

- I/O is:
  - Varied
  - Complex
  - Error prone
- A bad place for the typical user to be wandering around
- The operating system really needs to make I/O a lot friendlier

# Important Elements of I/O Architecture

- Types of I/O devices
- Busses
  - Types, arbitration, bus-mastering
- Device controllers
  - Controller registers
  - A sample device
  - Direct I/O

# What Counts as an I/O Device?

- Storage devices (hard drives, flash drives, DVD/CD drives, tape drives)
- Displays (monitors and speakers)
- Input devices (keyboards, mice, microphones and cameras)
- Network devices (wired and wireless, including 802.11, Bluetooth, maybe infrared)
- Sensor devices (GPS, accelerometers, etc.)
- And sometimes exotic stuff

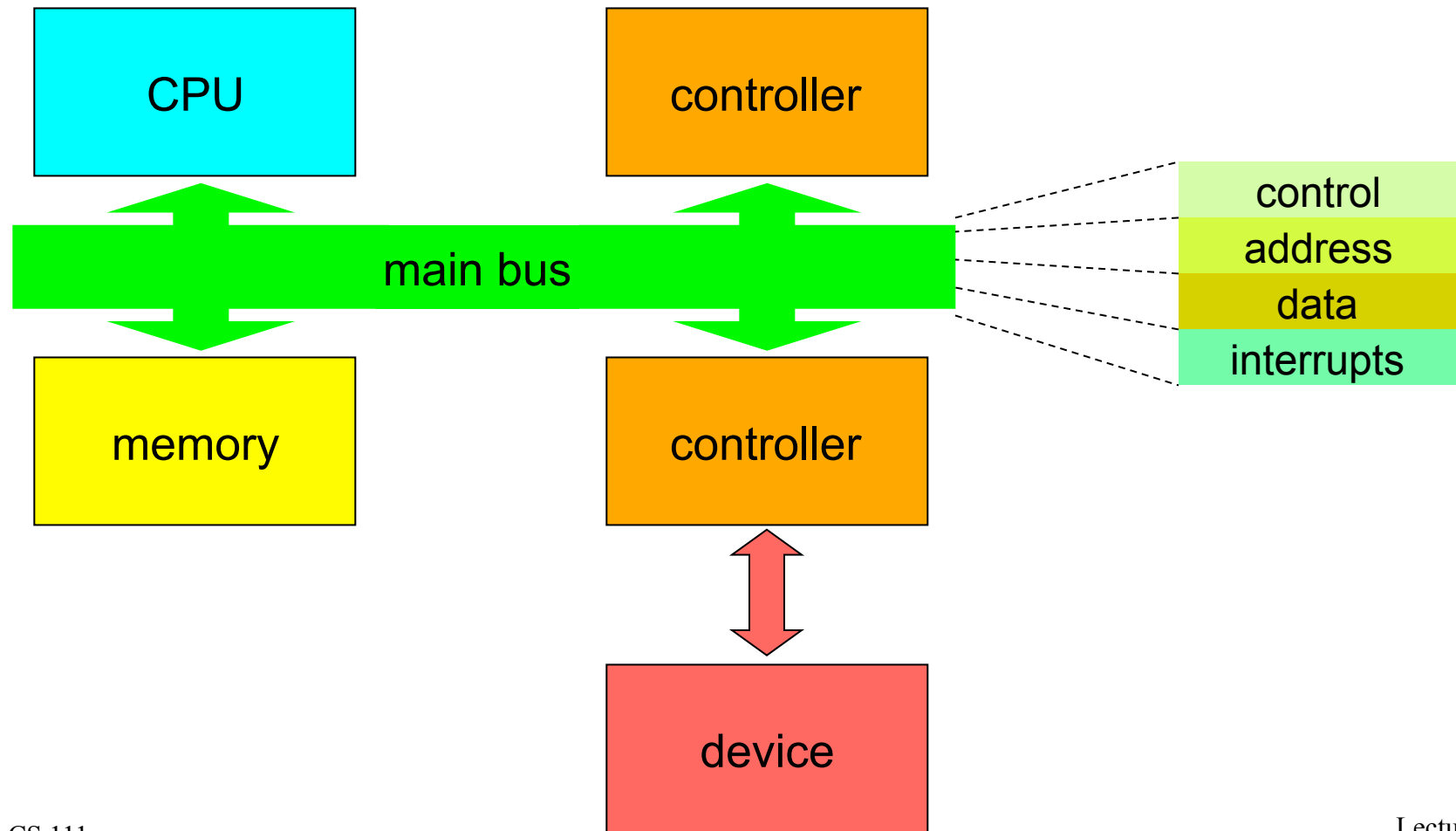
# Sequential vs. Random Access Devices

- Sequential access devices
  - Byte/block  $N$  must be read/written before byte/block  $N+1$
  - May be read/write once, or may be rewindable
  - Examples: magnetic tape, printer, keyboard
- Random access devices
  - Possible to directly request any desired byte/block
  - Getting to that byte/block may or may not be instantaneous
  - Examples: memory, magnetic disk, graphics adaptor
- They are used very differently
  - Requiring different handling by the OS

# Busses

- Something has to hook together the components of a computer
  - The CPU, memory, various devices
- Allowing data to flow between them
- That is a *bus*
- A type of communication link abstraction

# A Simple Bus



# Memory Type Busses

- Initially back-plane memory-to-CPU interconnects
  - A few “bus masters”, and many “slave devices”
  - Arbitrated multi-cycle bus transactions
    - Request, grant, address, respond, transfer, ack
    - Operations: read, write, read/modify/write, interrupt
- Originally most busses were of this sort
  - ISA, EISA, PCMCIA, PCI, cPCI, video busses, ...
  - Distinguished by
    - Form-factor, speed, data width, hot-plug, maximum length, ...
    - Bridging, self identifying, dynamic resource allocation, ...

# Bus Masters, Slaves, and Arbitration

- Bus master
  - Any device (or CPU) that can request the bus
  - One can also speak of the “current bus master”
- Bus slave
  - A device that can only respond to bus requests
- Bus arbitration
  - Process of deciding to whom to grant the bus
    - May be based on time, geography or priority
    - May also clock/choreograph steps of bus cycles
    - Bus arbitrator may be part of CPU or separate



# Network Type Busses

- Evolved as peripheral device interconnects
  - SCSI, USB, 1394 (Firewire), Infiniband, ...
  - Cables and connectors rather than back-planes
  - Designed for easy and dynamic extensibility
  - Originally slower than back-plane, but no longer
- Much more like a general purpose network
  - Packet switched, topology, routing, node identity
  - May be master/slave (USB) or peer-to-peer (1394)
  - May be implemented by controller or by host

# Devices and Controllers

- I/O devices
  - Peripheral devices that interface between the computer and other media
    - Disks, tapes, networks, serial ports, keyboards, displays, pointing devices, etc.
- Device controllers connect a device to a bus
  - Communicate control operations to device
  - Relay status information back to the bus
  - Manage DMA transfers for the device
  - Generate interrupts for the device
- Controller usually specific to a device and a bus

# Device Controller Registers

- Device controllers export registers to the bus
  - Registers in controller can be addressed from bus
  - Writing into registers controls device or sends data
  - Reading from registers obtains data/status
- Register access method varies with CPU type
  - May use special instructions (e.g., x86 IN/OUT)
    - Privileged instructions restricted to supervisor mode
  - May be mapped onto bus like memory
    - Accessed with normal (load/store) instructions
    - I/O address space not accessible to most processes

# Direct Polled I/O

- One way of moving data into/out of computer
  - Common in very old peripheral devices
- All transfers happen under direct control of CPU
  - CPU transfers data to/from device controller registers
  - Transfers are typically one byte or word at a time
  - May be accomplished with normal or I/O instructions
- CPU polls device until it is ready for data transfer
  - Received data is available to be read
  - Previously initiated write operations are completed
- Advantages
  - Very easy to implement (both hardware and software)

# Disadvantage of Direct Polled I/O

- CPU-intensive data transfers
  - Each byte/word requires multiple instructions
- CPU wasted while awaiting completion
  - Busy-wait polling ties up CPU until I/O is completed
- Devices are idle while we are running other tasks
  - I/O can only happen when an I/O task is running
- How can these problems be dealt with?
  - Let controller transfer data without attention from CPU
  - Let application block pending I/O completion
  - Let controller interrupt CPU when I/O is finally done
- Requires OS support

# Handling I/O Performance Issues

- Various techniques are possible
- Direct Memory Access (DMA)
  - Non-CPU bus-masters
  - Completion interrupts
  - Typical DMA programming
- Enhanced Techniques
  - Memory Mapped I/O
  - Smart Device Controllers
  - I/O Channel Controllers

# Direct Memory Access

- Essentially, use the bus without CPU control
  - Move data between memory and device controller
- Bus facilitates data flow in all directions between:
  - CPU, memory, and device controllers
- CPU can be the bus-master
  - Initiating data transfers with memory, device controllers
- But device controllers can also master the bus
  - CPU instructs controller what transfer is desired
    - What data to move to/from what part of memory
  - Device controller does transfer w/o CPU assistance
  - Device controller generates interrupt at end of transfer

# DMA Interrupts

- CPU usually needs to know when DMA is done
- Handled by sending interrupt on the bus
  - Devices signal controller when they are done/ready
  - When device finishes, controller puts interrupt on bus
- CPUs and interrupts
  - Interrupts look very much like traps
    - Traps come from CPU, interrupts are caused externally
  - Unlike traps, interrupts can be enabled/disabled
    - A device can be told it can or cannot generate interrupts
    - Special instructions can enable/disable interrupts to CPU

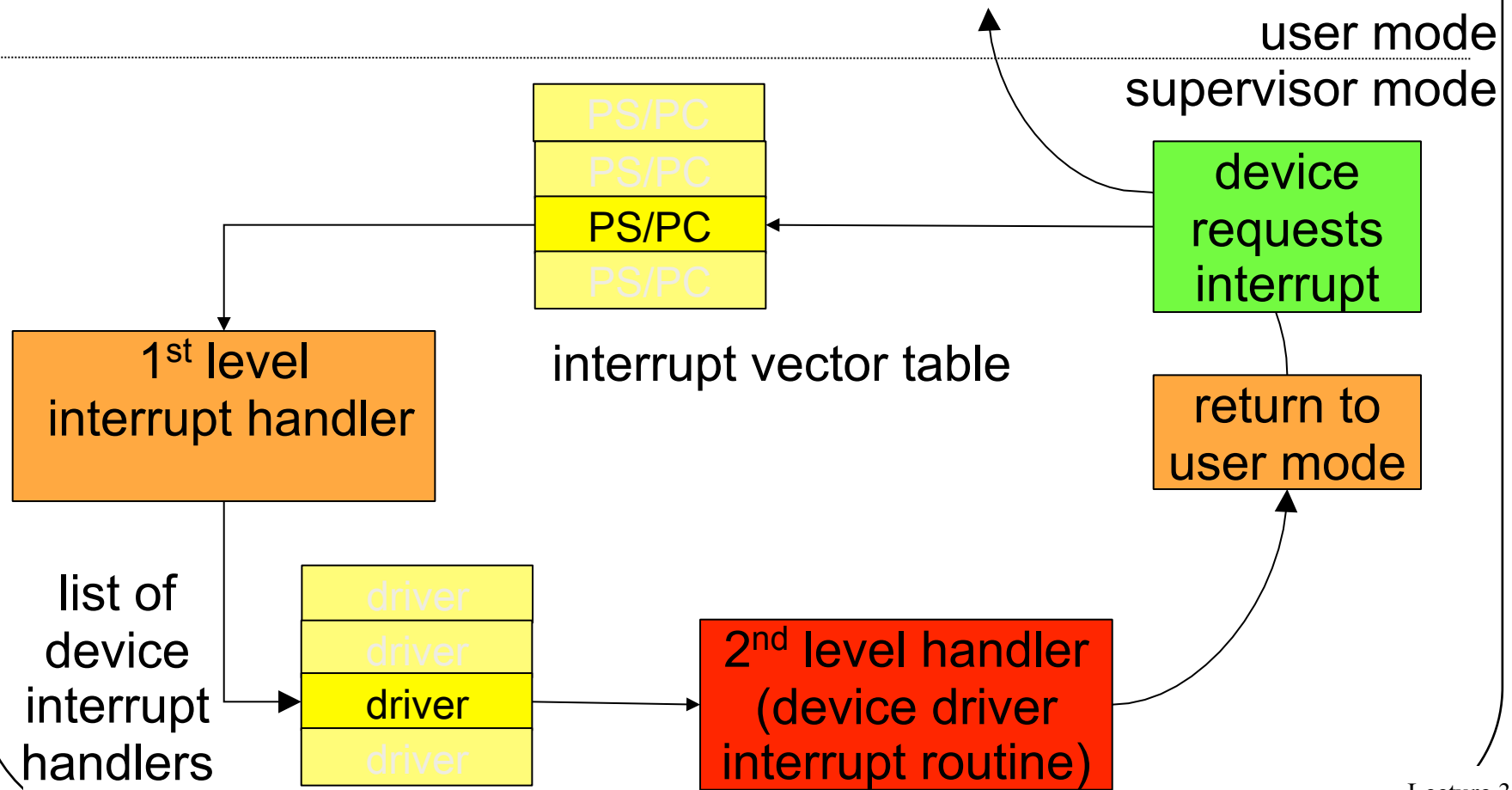


# Interrupt Handling

Application Program

instr ; instr ; instr ; instr ; instr ; instr ;

user mode  
supervisor mode



# Interrupts vs. Traps

- Most traps caused by an instantaneous condition
  - Triggered in response to illegal program actions
  - Related to something CPU was doing
- Interrupts are caused a device being in some state
  - Triggered when the device enters a particular state
    - E.g., device state changes from BUSY to DONE
  - They are asserted as long as device is in that state
    - E.g., until the device is BUSY again
- Once delivered, an interrupt must be disabled
  - CPU must ignore continuing request for that interrupt
  - Cause must be cleared, and interrupt acknowledged

# Performing I/O Using Interrupts

- Requesting process checks to see if device is busy
  - If idle, start the I/O operation, and await its completion
  - Meanwhile, CPU does something else (for this process or another one)
  - If busy, wait for the device to become idle
- I/O interrupt handler
  - Gathers completion information from the device
  - Awakes requester to handle the interrupt
- When current owner finishes using the device
  - Wake up the next requester
- We'll talk about waiting and waking up soon

# Problems With DMA

- DMA is designed for fairly large data transfers
- What if you want to move a rather small amount of data?
  - Frequently and efficiently
  - E.g., consider a video game display adaptor
  - Lots of data in the display, but maybe only a few bytes get updated
- DMA is rather heavyweight for that

# Memory Mapped I/O

- CPU treats control and data registers of I/O devices as if they were memory addresses
- Reads/writes to them just like memory
- Makes everything the processor works with look just like memory
  - No special instructions to read/write I/O devices
- Applications themselves can write to the memory locations
  - Avoiding traps to the OS

# A Memory Mapping Example

- A bit-mapped display adaptor
  - 1Mpixel display controller, on the CPU memory bus
  - Each word of display memory corresponds to one pixel
  - Application uses ordinary stores to update display
  - Device always has access to the data without interrupts or polling
- Low overhead per update, no interrupts
- Relatively easy to program

# Memory Mapping Devices and Security

- Memory mapped I/O from ordinary instructions gives user-mode processes direct access to an I/O device
- Isn't this a security problem?
  - Yes, but perhaps the device does not contain anybody else's data
    - E.g., the device is a graphics adaptor and the program is a video game
  - Memory mapping devices is a protected operation
  - OS controls which processes can use which devices when

# DMA vs. Memory Mapping

- DMA performs large transfers efficiently
  - Better utilization of both the devices and the CPU
    - Device doesn't have to wait for CPU to do transfers
  - But there is considerable per transfer overhead
    - Setting up the operation, processing completion interrupt
- Memory-mapped I/O has no start/finish overhead
  - But every byte is transferred by a CPU instruction
    - No waiting because device accepts data at memory speed
- DMA better for occasional large transfers
- Memory-mapped better for frequent small transfers
- Memory-mapped devices more difficult to share
- Memory mapping can be used to set up DMA



# Smart Device Controllers

- Smarter controllers can improve on basic DMA
- They can queue multiple input/output requests
  - When one finishes, automatically start next one
  - Reduce completion/start-up delays
  - Eliminate need for CPU to service interrupts
- They can relieve CPU of other I/O responsibilities
  - Request scheduling to improve performance
  - They can do automatic error handling & retries
- Abstract away details of underlying devices

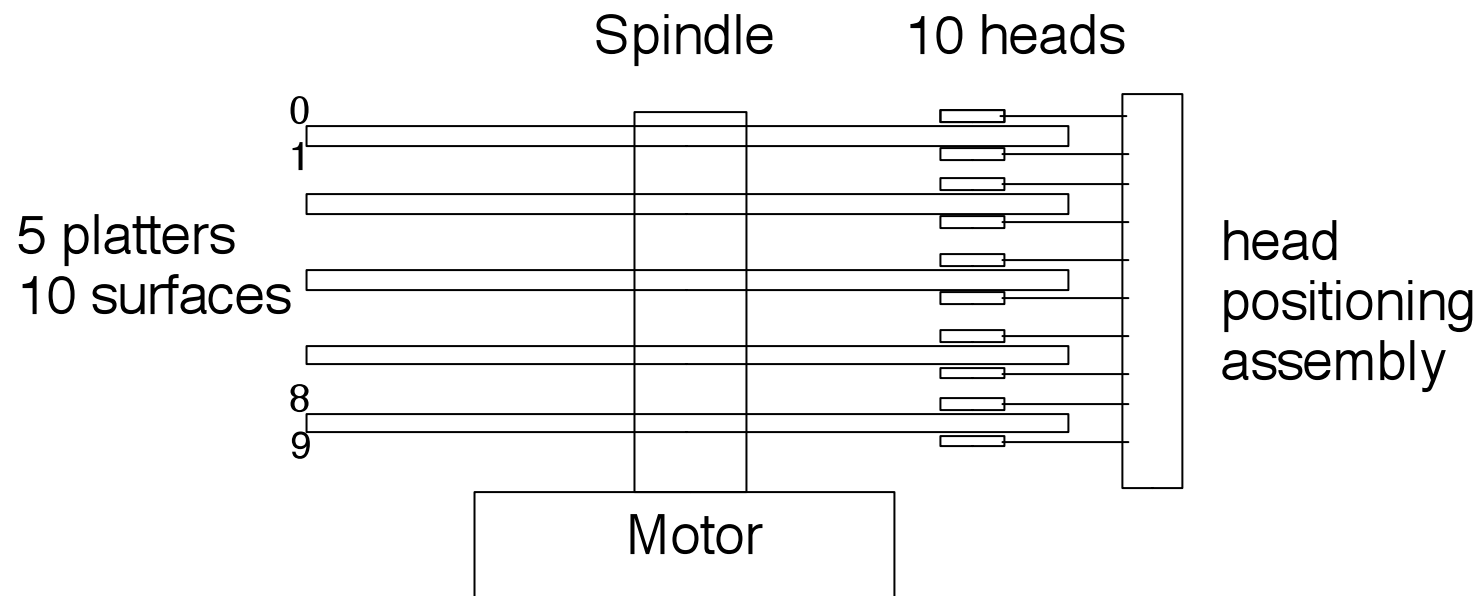
# Disk Drives

- An especially important and complex form of I/O device
- Still the primary method of providing stable storage
  - Storage meant to last beyond a single power cycle of the computer
- A place where physics meets computer science
  - Somewhat uncomfortably

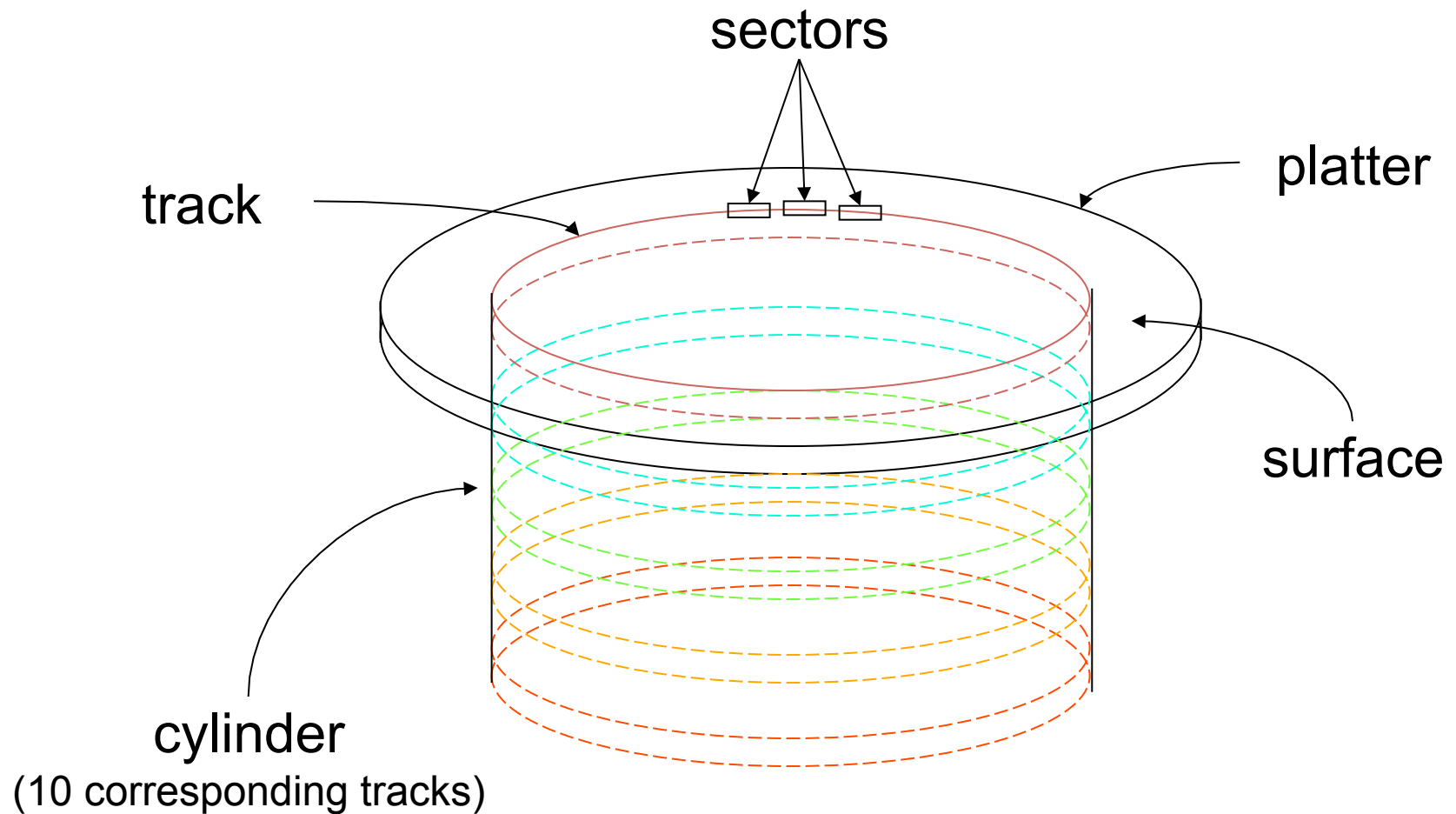
# Some Important Disk Characteristics

- Disks are random access devices (mostly . . .)
  - With complex usage, performance, and scheduling
- Key OS services depend on disk I/O
  - Program loading, file I/O, paging
  - Disk performance drives overall performance
- Disk I/O operations are subject to overhead
  - Higher overhead means fewer operations/second
  - Careful scheduling can reduce overhead
  - Clever scheduling can improve throughput, delay

# Disk Drives – A Physical View



# Disk Drives – A Logical View



# Disk Drive Terms

- *Spindle*
  - A mounted assembly of circular platters
- *Head assembly*
  - Read/write head per surface, all moving in unison
- *Track*
  - Ring of data readable by one head in one position
- *Cylinder*
  - Corresponding tracks on all platters
- *Sector*
  - Logical records written within tracks
- *Disk address* = <cylinder / head / sector >

# Seek Time

- At any moment, the heads are over some track
  - All heads move together, so all over the same track on different surfaces
- If you want to read another track, you must move the heads
- The time required to do that is seek time
- Seek time is not constant
  - Amount of time to move from one track to another depends on start and destination
  - Usually reported as an average

# Rotational Delay

- Once you have the heads over the right track, you need to get them to the right sector
- The head is over only one sector at a time
- If it isn't the right sector, you have to wait for the disk to rotate over that one
- Like seek time, not a constant
  - Depends on which sector you're over
  - And which sector you're looking for
  - Also usually reported as an average
- Also called *latency*



# Transfer Time

- Once you're on the correct track and the head's over the right sector, you need to transfer data
- You don't read/write an entire sector at a time
- There is some delay associated with reading every byte in the sector
- All sectors are usually the same size
- So transfer time is usually constant

# Disk Drives and Controllers

- The disk drive is not directly connected to the bus
- It is connected to a disk drive controller
  - Special hardware designed for this task
- There may be several disk drives attached to the same controller
  - Which then multiplexes its attention between them
- Many disks have their controller bundled with them (e.g., SCSI disks)

# Typical Disk Drive Performance

heads	10	platters	5
cylinders	17,000	tracks/inch	18,000
sectors/track	400	bytes/sector	512
RPM	7200	speed	196Mb/sec
seek time	0-15 ms	latency	0-8ms

## Time to read one 8192 byte block

	seek	rotate	transfer	total
best case	0ms	0ms	333us	333us
worst case	15ms	8ms	333us	23.3ms (70X)
average	9ms	4ms	333us	13.3ms (40X)

# Why Is This An Issue For the OS?

- When you go to disk, it could be fast or slow
  - If you go to disk a lot, that matters
- The OS can make choices that make it faster or slower
  - Deciding where to put a piece of data on disk
  - Deciding when to perform an I/O
  - Reordering multiple I/Os to minimize seek time and latency
  - Perhaps optimistically performing I/Os that haven't been requested

# Optimizing Disk I/O

- Don't start I/O until disk is on-cylinder or near sector
  - I/O ties up the controller, locking out other operations
  - Other drives seek while one drive is doing I/O
- Minimize head motion
  - Do all possible reads in current cylinder before moving
  - Make minimum number of trips in small increments
- Encourage efficient data requests
  - Have lots of requests to choose from
  - Encourage cylinder locality
  - Encourage largest possible block sizes
  - All by OS design choices, not influencing programs/users

# Algorithms to Control Head Movement

- First come, first served
  - Just do them in the order they happen
- Shortest seek time first
  - Always go with the request that's closest to the current head position
  - Since requests keep arriving, can cause starvation
- Scan/Look (AKA the Elevator Algorithm)
  - Service all requests in one direction, then go in the other direction

CS III No starvation, but may take longer

# Head Travel With Various Algorithms

## First Come First Served

76		124		17		269		201		29		137		12
48		107		252		68		172		108		125		
total head motion: 880 cylinders														

## Shortest Seek First

76		29		17		12		124		137		201		269
47		12		5		112		13		64		68		
total head motion: 321 cylinders														

## Scan/Look (elevator algorithm)

76		124		137		201		269		29		17		12
48		13		64		68		240		12		5		
total head motion: 450 cylinders														

# Disks as an Example of the Memory Abstraction

- They support the read and write operations
- But, unlike RAM, they are not word-addressable
  - You read and write sectors
- Also unlike RAM, they have variable delays in their operations
  - Not just because of queued operations, either
- Either the OS must expose these differences
  - Or work to hide them