I/O, Modularity and Virtualization
CS 111
Operating System Principles
Peter Reiher

Outline

- The role of I/O in operating systems
- Organizing systems via modularity
- Virtualization and operating systems

I/O Architecture

- I/O is:
  - Varied
  - Complex
  - Error prone
- Bad place for the user to be wandering around
- The operating system must make I/O friendlier
- Oriented around handling many different *devices* via *busses* using *device drivers*

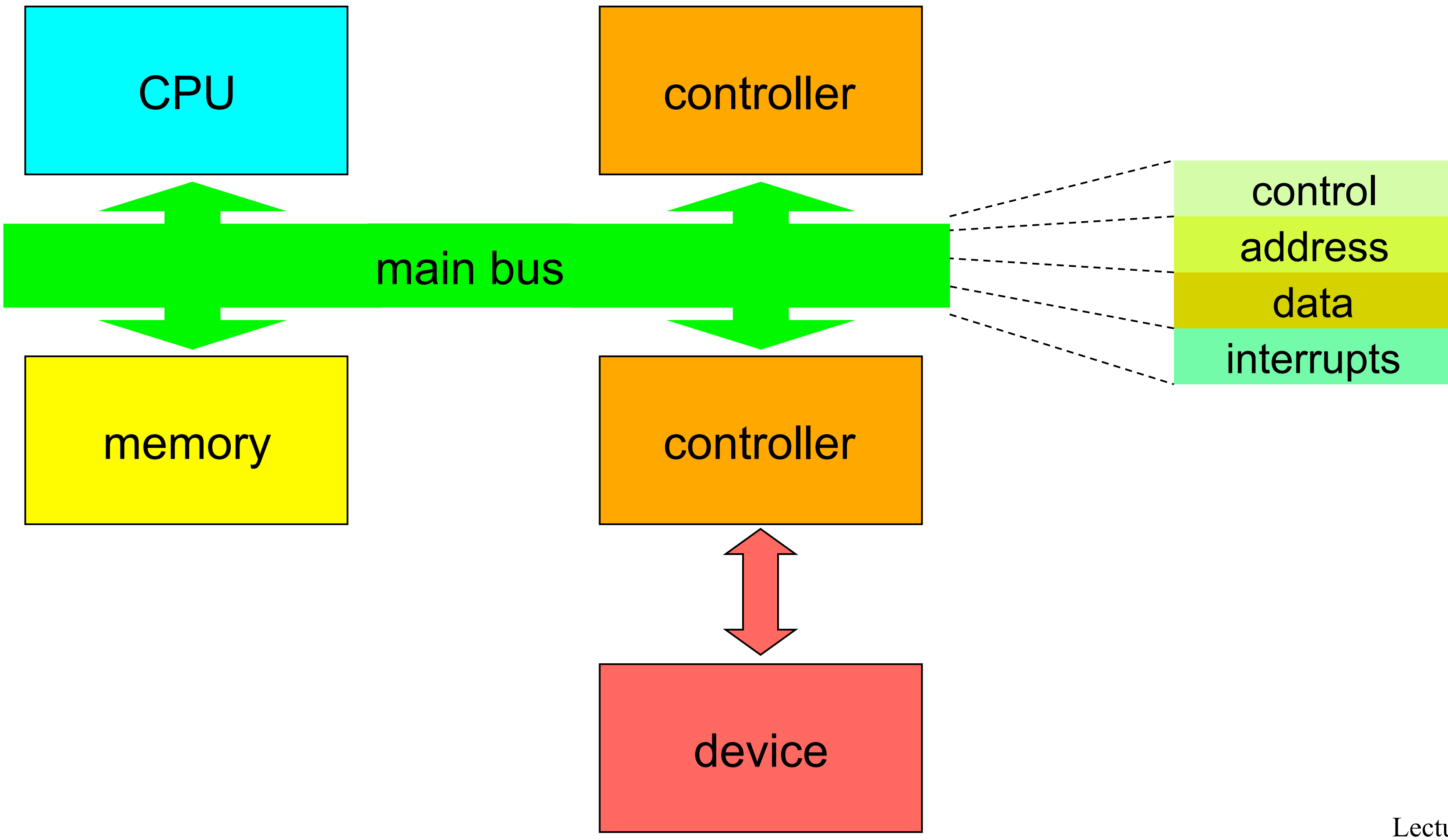
Sequential vs. Random Access Devices

- 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


CPU
controller
control
address
data
interrupts
main bus
memory
controller
device

Devices and Controllers

- Device controllers connect a device to a bus
  - Communicate control operations to device
  - Relay status information back to the bus, manage DMA, generate device interrupts
- Device controllers export registers to the 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)
  - May be mapped onto bus just like memory

Direct Polled I/O

- Method of accessing devices via direct CPU control
  - 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

\+ Very easy to implement (both hardware and software)

– CPU intensive, wastes CPU cycles on I/O control

– Leaves devices idle waiting for CPU when other tasks running

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
  - Device controller does transfer w/o CPU assistance
  - Device controller generates interrupt at end of transfer
- Interrupts tell CPU when DMA is done

Memory Issues

- Different types of memory handled in different ways
- Cache memory usually handled mostly by hardware
  - Often OS not involved at all
- RAM requires very special handling
  - To be discussed in detail later
- Disks and flash drives treated as devices
  - But often with extra OS support

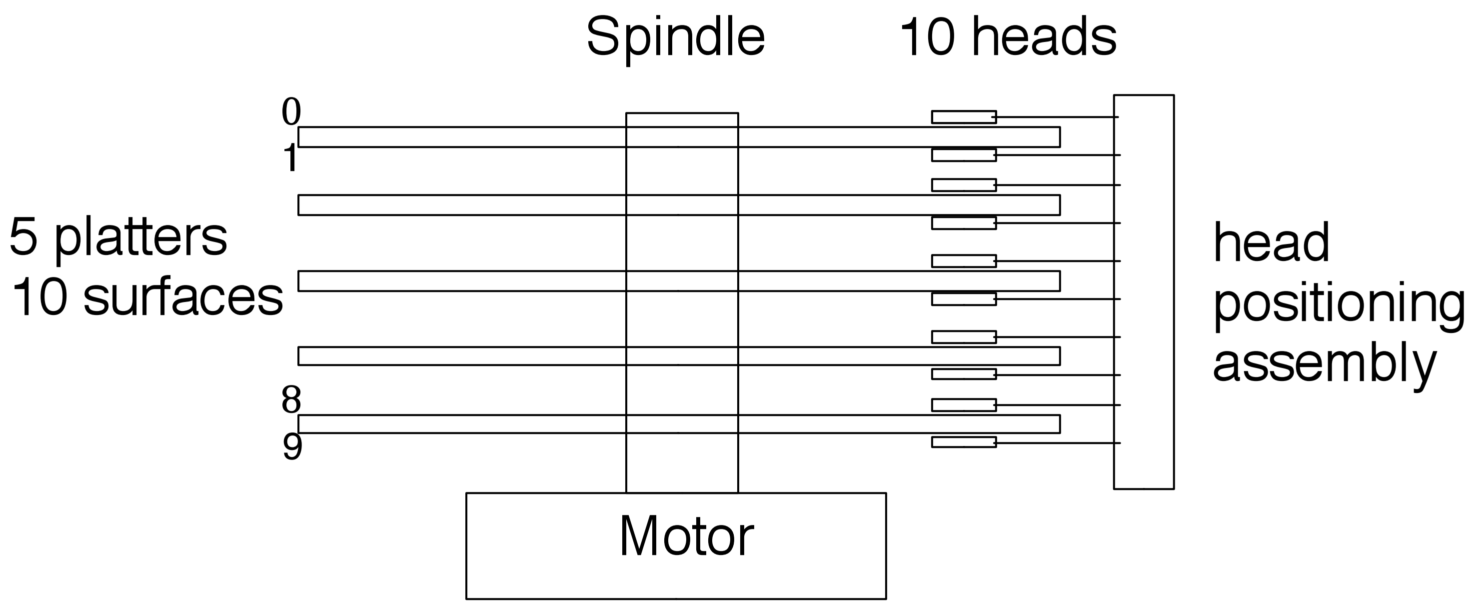
Disk Drives

- An especially important and complex form of I/O device
  - Gradually being replaced by SSDs
- 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

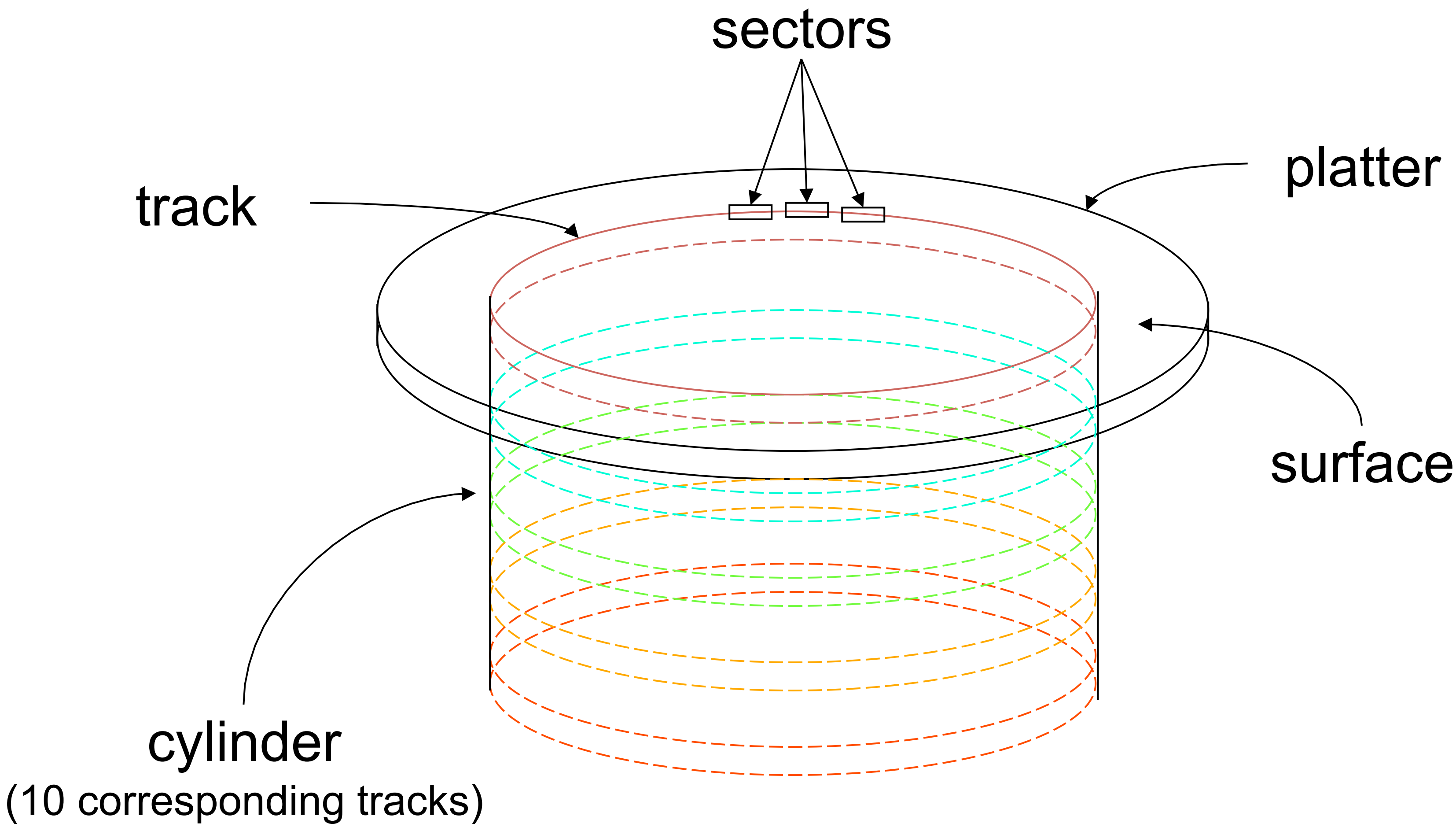
Some Important Disk Characteristics

- 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



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)

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
  - 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

Modularity

- Most useful abstractions an OS wants to offer can’t be directly realized by hardware
- Modularity is one technique the OS uses to provide better abstractions
- Divide up the overall system you want into well-defined communicating pieces
- Critical issues:
  - Which pieces to treat as modules
  - How to organize the modules
  - Interfaces to modules

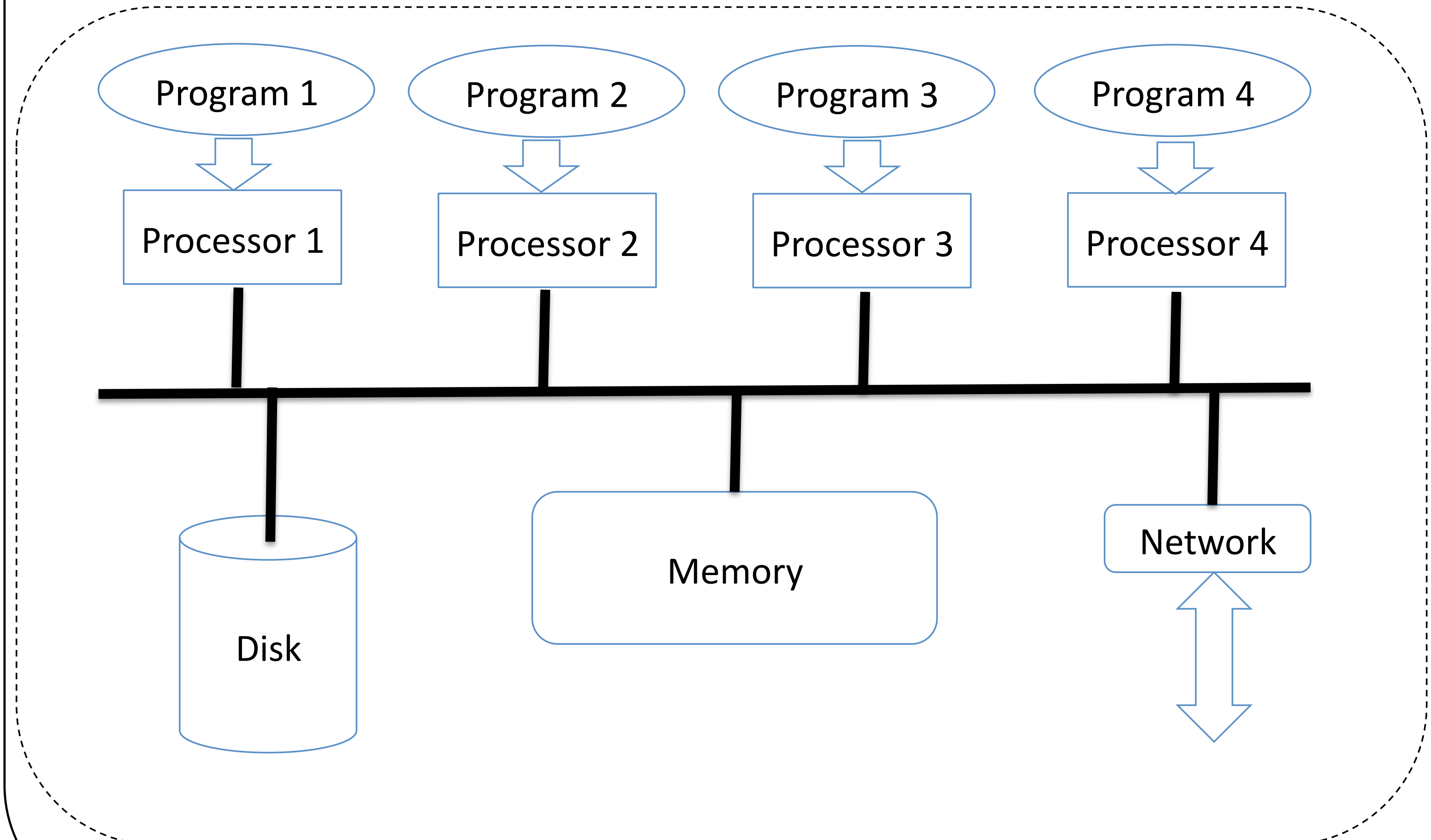
What Does An OS Do?

- At minimum, it enables one to run applications
  - Preferably several on the same machine
  - Preferably several at the same time
- At abstract level, what do we need to do that?
  - Interpreters (to run the code)
  - Memory (to store the code and data)
  - Communications links (to communicate between apps and pieces of the system)
- This suggests the kinds of modules we'll need

Starting Simple

- We want to run multiple programs
  - Without interference between them
  - Protecting one from the faults of another
- We've got a multicore processor to do so
  - More cores than programs
- We have RAM, a bus, a disk, other simple devices
- What abstractions should we build to ensure that things go well?

A Simple System



A machine boundary

Exploiting Modularity

- We'll obviously have several SW elements to support the different user programs
- Desirable for each to be modular and self-contained
  - With controlled interactions
- Gives cleaner organization
- Easier to prevent problems from spreading
- Easier to understand what's going on
- Easier to control each program's behavior

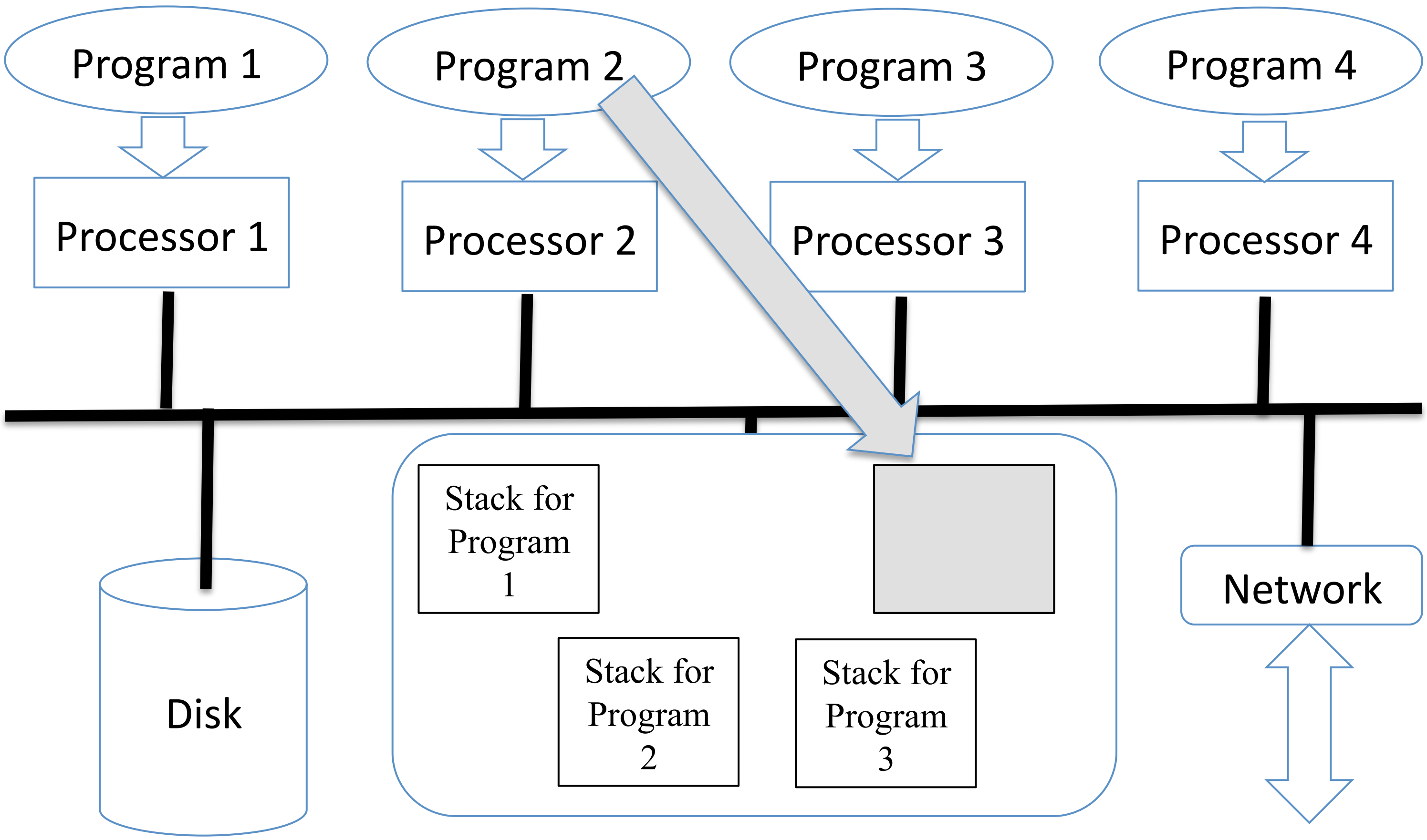
Subroutine Modularity

- Why not just organize the system as a set of subroutines?
  - All in the same address space
    - A simplifying assumption
    - Allowing easy in-memory communication
- System subroutines call user program subroutines as needed
  - And vice versa
- *Soft modularity*

How Would This Work?

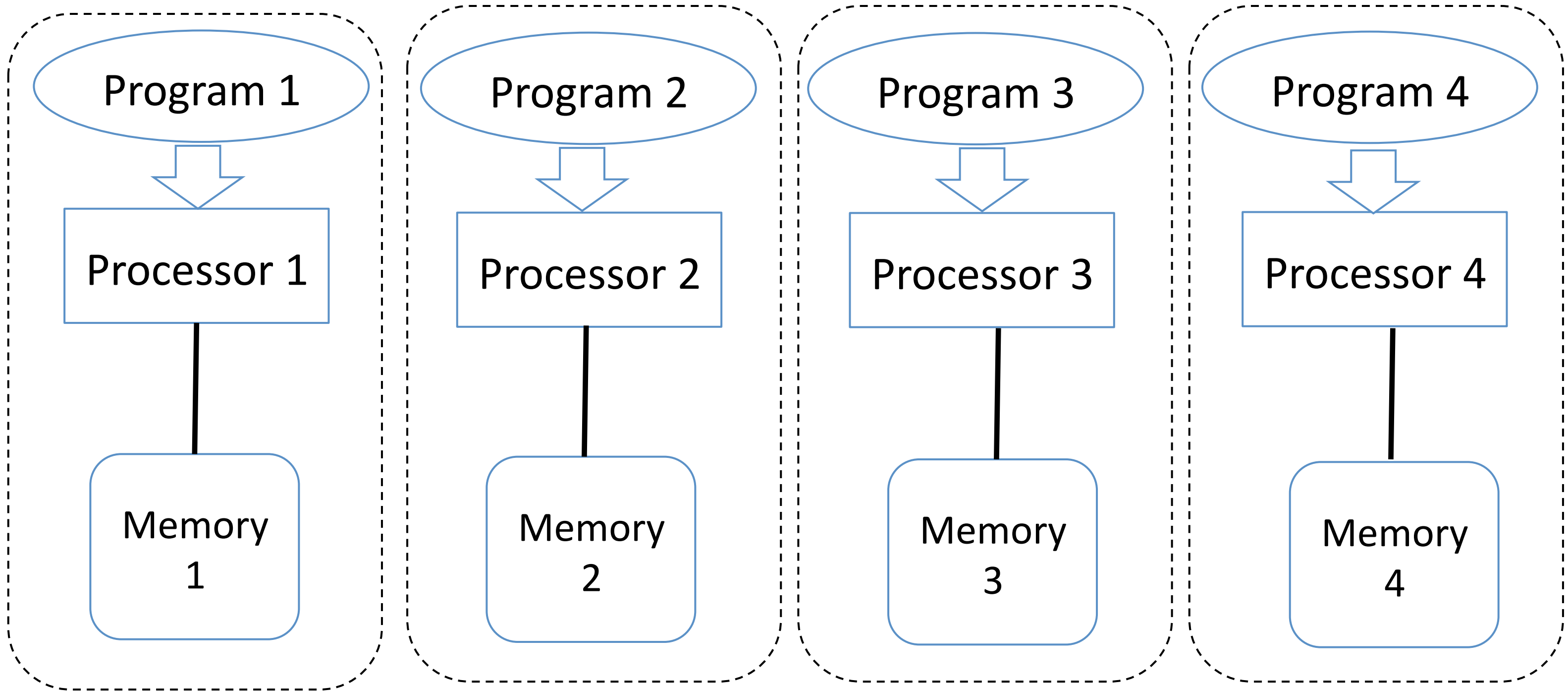
- Each program is a self-contained set of subroutines
  - Subroutines in the program call each other
  - But not subroutines in other programs
- Shared services offered by other subroutines
  - Which any program can call
- Perhaps some "master routine" that calls subroutines in the various programs
- Soft because no OS HW/SW enforces modularity
  - Important resources (like the stack) are shared
  - Only proper program behavior protects one program from the mistakes of another

Illustrating the Problem



Now Program 4 is in trouble
Even though it did nothing wrong itself

Hardening the Modularity



Four separate machines

Perhaps in very different places

Each program has its own machine

System Services In This Model

- Some activities are local to each program
- Other services are intended to be shared
  – Like a file system
- This functionality can be provided by a client/ server model
- The system services are provided by the server
- The user programs are clients
- The client sends message to server to get help
- OS uses HW/SW to enforce boundaries

Benefits of Hard Modularity

- With hard modularity, something beyond good behavior enforces module boundaries
- Here, the physical boundaries of the machine
- A client machine literally cannot touch the memory of the server
  - Or of another client machine
- No error or attack can change that
  - Though flaws in the server can cause problems
- Provides stronger guarantees all around

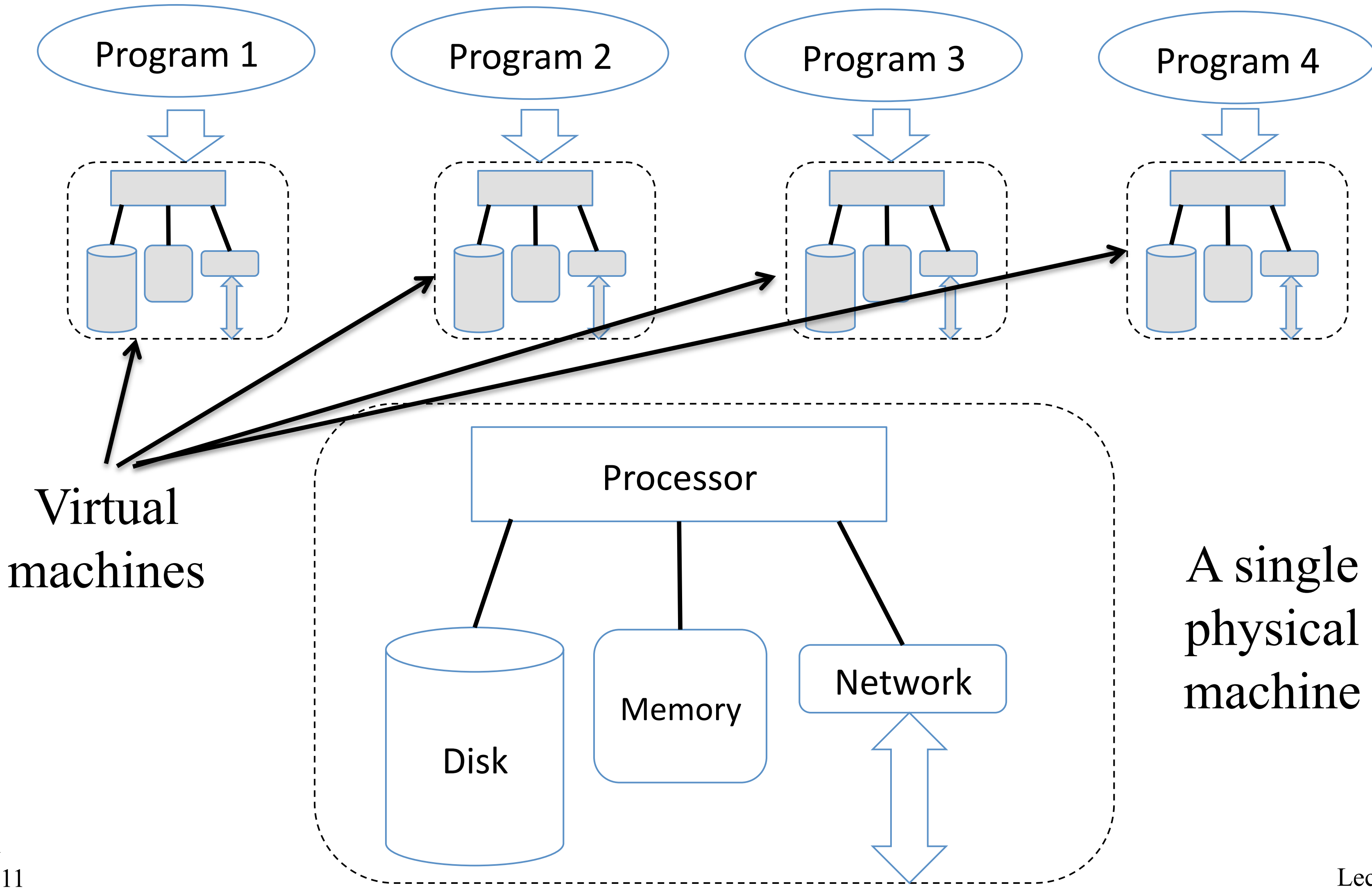
Downsides of Hard Modularity

- The hard boundaries prevent low-cost optimizations
- In client/server organizations, doing anything with another program requires messages
  - Inherently more expensive than memory accesses
- If the boundary sits between components requiring fast interactions, possibly very bad
- Must either give programs pieces of resources or time multiplex use of resources
  - More complexity to do this right

Virtualization

- Provide the illusion of a complete resource to each program that uses it
  - Hide hard modularity’s time/space divisions
- Possible to provide an entire virtual machine per process
- Use shared hardware to instantiate the various virtual devices or machines
- System software (i.e., the operating system) and perhaps special hardware handle it

The Virtualization Concept



The Trick in Virtualization

- All the virtual machines share the same physical hardware
- But each thinks it has its own machine
- Must be sure that one virtual machine doesn't affect behavior of the others
  - Intentionally or accidentally
- With the least possible performance penalty
  - Given that there will be a penalty merely for sharing at all

Performance and Virtualization

- To achieve good performance, can’t run many instructions “virtualized”
  - Most instructions must go directly to the processor
  - Rather than be mapped into multiple instructions via virtualization
- Similarly for access to other HW
  - Can’t afford to put lots of virtualization SW in the usual path
- The trick is to virtualize the minimal set of accesses

Abstractions for Virtualizing Computers

- Some kind of interpreter abstraction
  - A *thread*
- Some kind of communications abstraction
  - *Bounded buffers*
- Some kind of memory abstraction
  - *Virtual memory*
- For a virtualized architecture, the operating system provides these kinds of abstractions

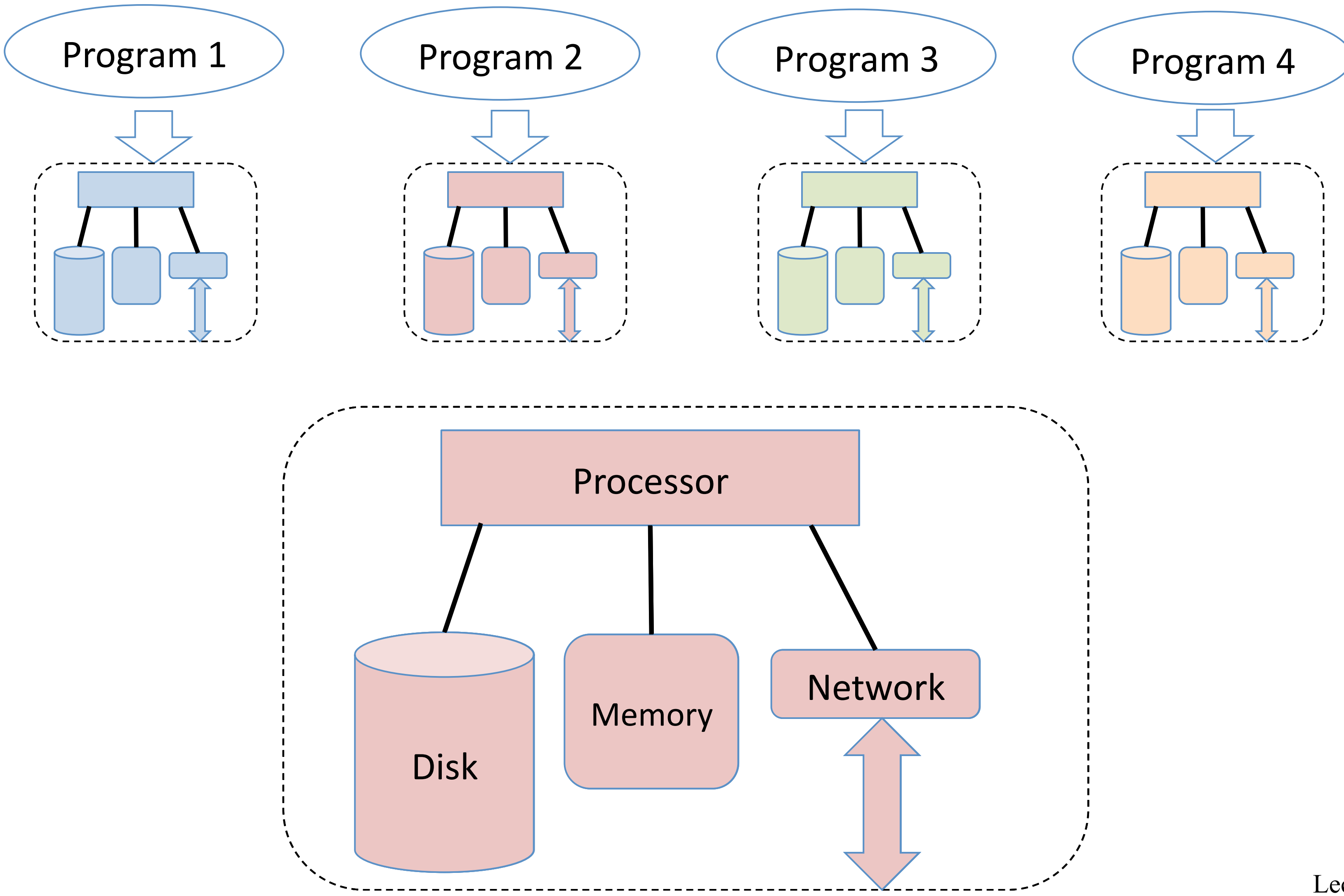
Threads

- Encapsulates the state of a running computation
- So what does it need?
  - Something that describes what computation is to be performed
  - Something that describes where it is in the computation
  - Something that maintains the state of the computation's data

OS Handling of Threads

- One (or more) threads per running program
- The OS chooses which thread to run
  - To share a processor, the OS must be able to cleanly stop and start threads
- While one thread is using a processor, no other thread should interfere with its use
- To run a thread, OS must:
  - Load its code and data into memory
  - Set up HW control structures (e.g., the PC)
  - Transfer control to the thread

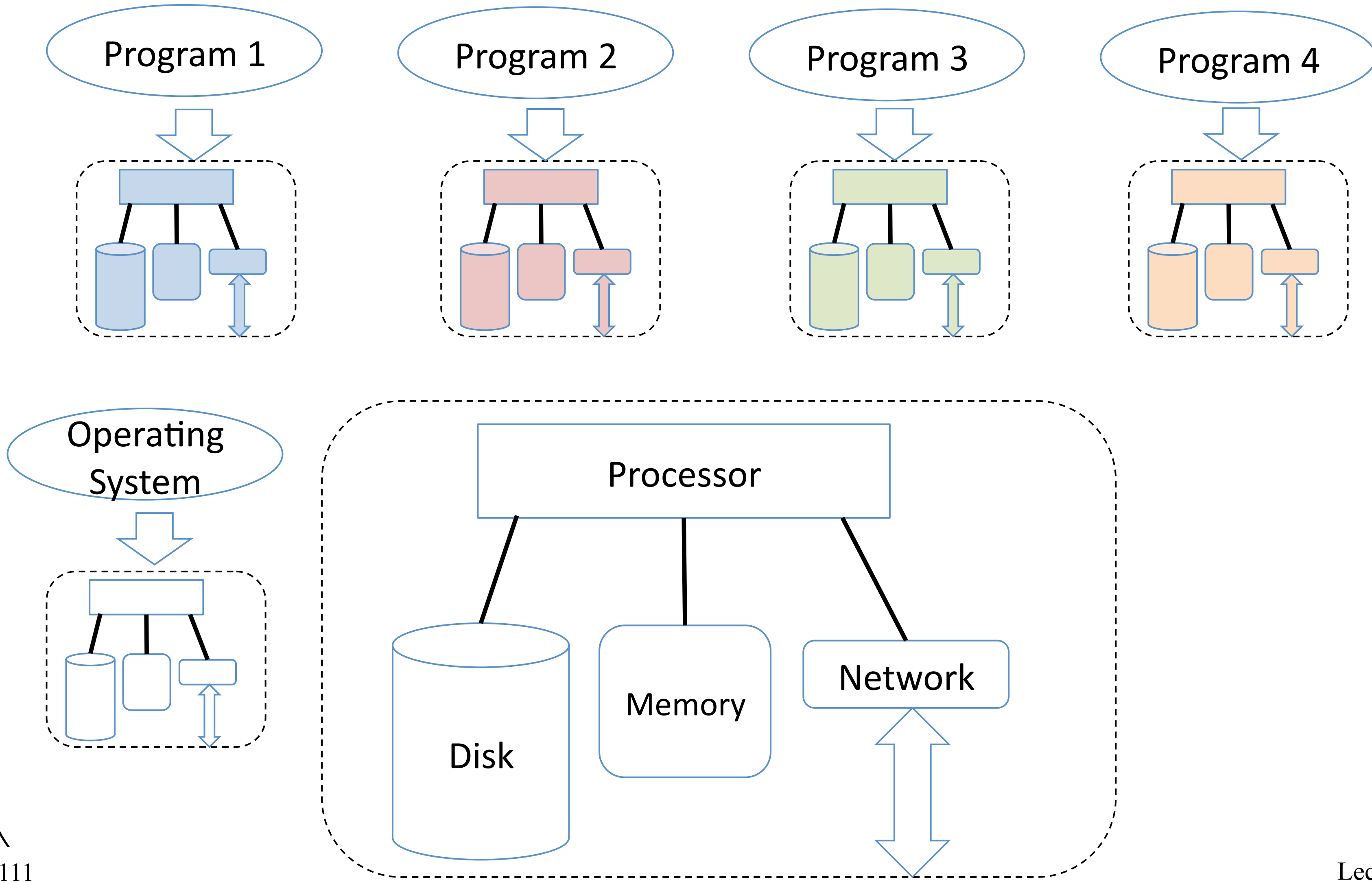
Time Slicing Virtualization



Wait a Minute . . .?

- How does the OS do all that?
- It's just a program itself
  - With its own interpreter, memory, etc.
- It must use the same physical resources as all the other threads
- Basically, the OS itself is a thread
- It creates and manages other threads
- Using privileged supervisor mode to safely and temporarily break virtualization boundaries

The OS and Virtualization



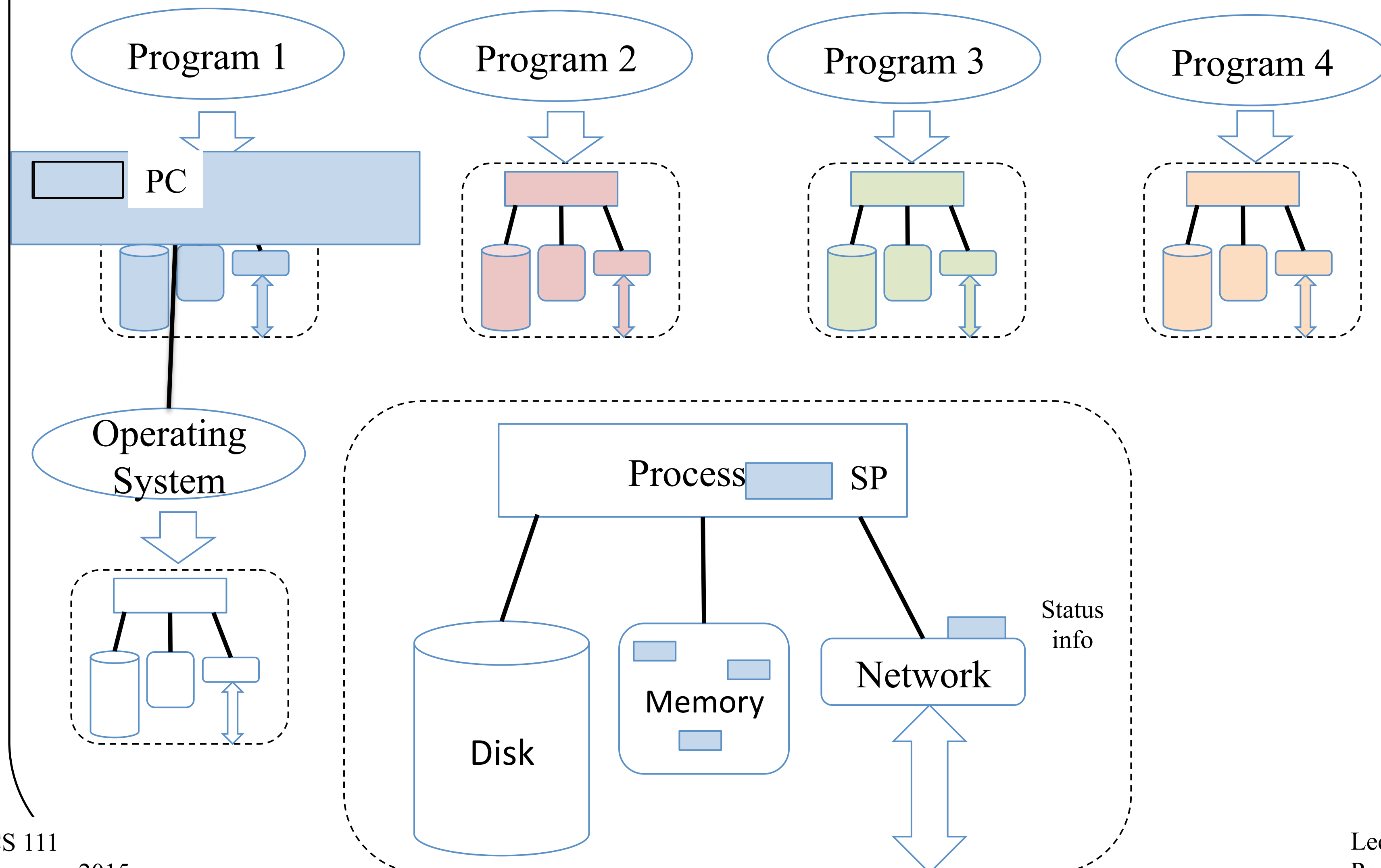
Providing Contained Environments

- What must a thread manager control to keep each thread isolated from the others?
- Well, what can each thread do?
  - Run instructions
    - Make sure it can only run its own
  - Access some memory
    - Make sure it can only access its own
  - Communicate to other threads
    - Make sure communication uses a safe abstraction

What Does This Boil Down To?

- Running threads have access to certain processor registers
  - Program counter, stack pointer, others
  - Thread manager must ensure those are all set correctly
- Running threads have access to some or all pieces of physical memory
  - Thread manager must ensure that a thread can only touch its own physical memory
- Running threads can request services (like communications)
  - Thread manager must provide safe access to those services

Setting Up a User-Level VM



Protecting Threads

- Normal threads usually run in user mode
- Which means they can't touch certain things
  - In particular, each others' stuff
- For certain kinds of resources, that's a problem
  - What if two processes both legitimately need to write to the screen?
  - Do we allow unrestricted writing and hope for the best?
  - Don't allow them to write at all?
- Instead, trap to supervisor mode

Trapping to Supervisor Mode

- To allow a program safe access to shared resources
- The trap goes to trusted code
  - Not under control of the program
- And performs well-defined actions
  - In ways that are safe
- E.g., program not allowed to write to the screen directly
  - But traps to OS code that writes it safely

Modularity and Memory

- Clearly, programs must have access to memory
- We need abstractions that give them the required access
  - But with appropriate safety
- What we've really got (typically) is RAM
- RAM is pretty nice
  - But it has few built-in protections
- So we want an abstraction that provides RAM with safety

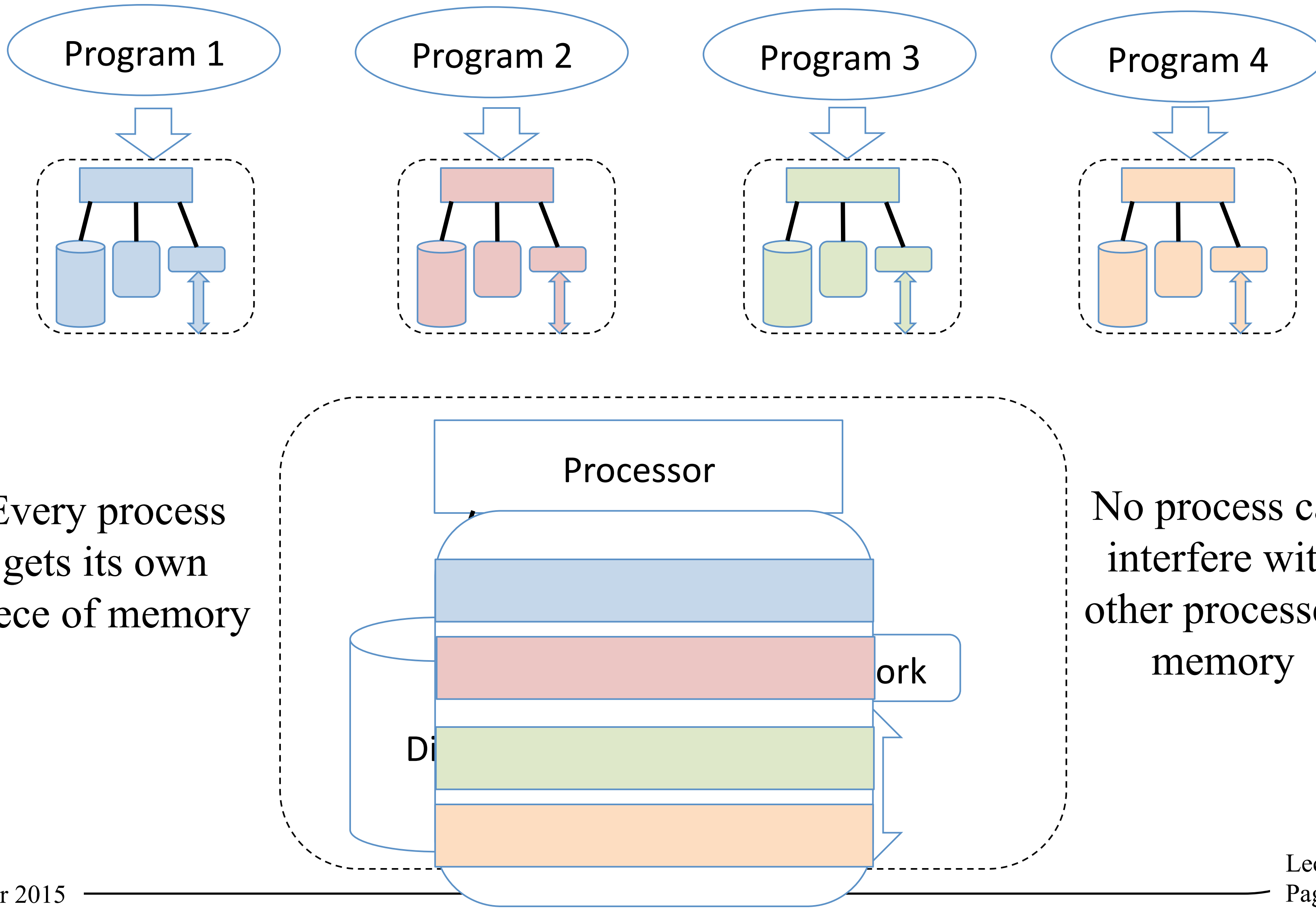
What's the Safety Issue?

- We have multiple threads running
- Each requires some memory
- Modern architectures typically have one big pool of RAM
- How can we share the same pool of RAM among multiple processes?
  - Giving each what it needs
  - Not allowing any to harm the others

Domains

- A simple memory abstraction
- Give each process access to some range of the physical memory
  - Its *domain*
  - Different domain for each process
- Allow process to read/write/execute memory in its domain
- And not touch any memory outside its domain

Mapping Domains



Every process gets its own piece of memory

No process can interfere with other processes' memory

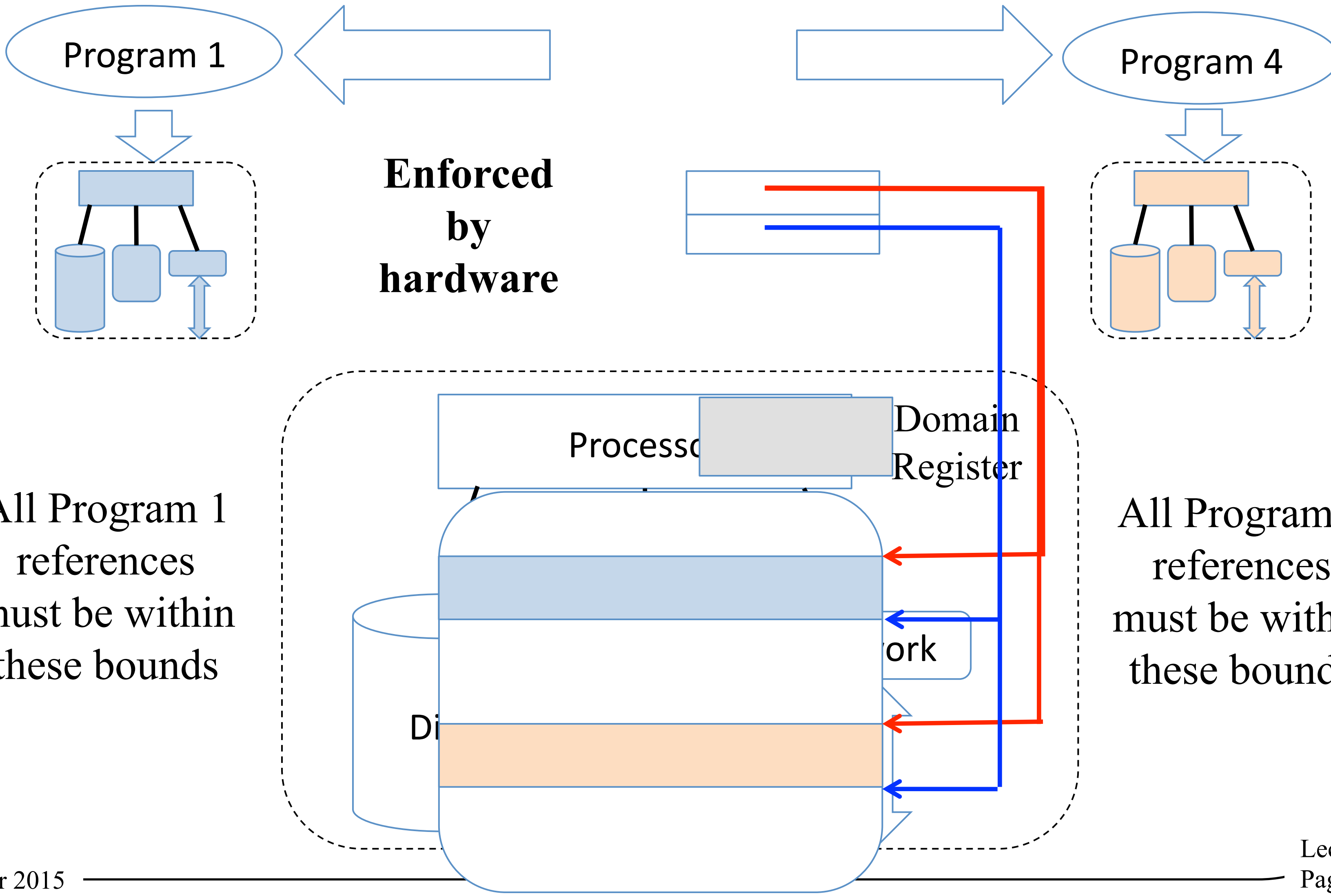
What Do Domains Require?

- Threads will issue instructions
  - Perhaps using arbitrary memory addresses
- Only honor addresses in the thread's domain
  - Any other address should be caught as an error
- Hard modularity here requires HW support
- E.g., a domain register
  - Specifies the domain associated with the thread currently using the processor
  - By listing the low and high addresses that bound the domain

The Memory Manager

- Hardware or software that enforces the bounds of the domain register
- When thread reads or writes an address, memory manager checks the domain register
- If within bounds, do the memory operation
- If not, throw an illegal memory reference exception
  - Trapping to supervisor mode
- Only trusted code (i.e., the OS) can change the domain register

The Domain Register Concept



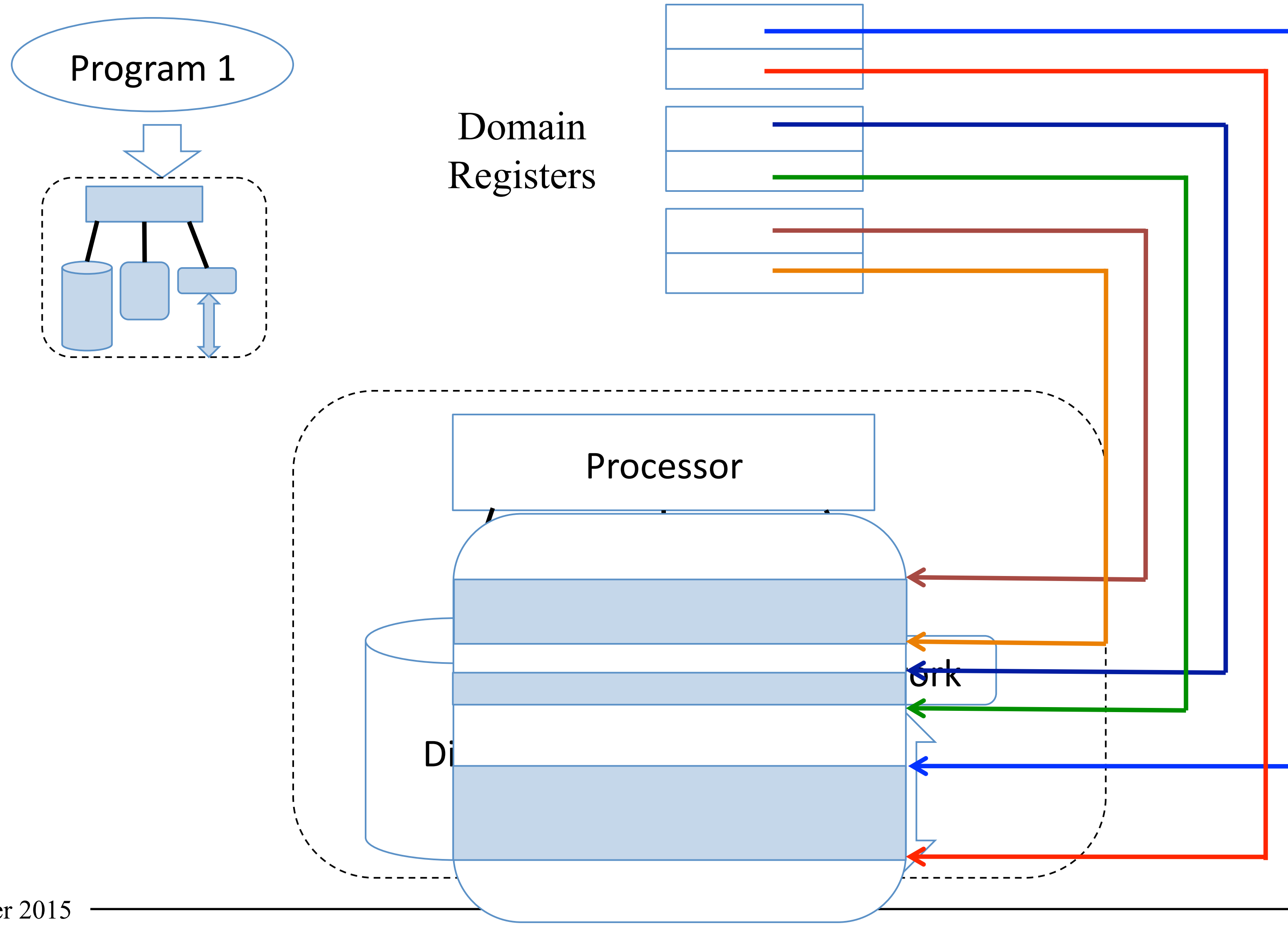
All Program 1 references must be within these bounds

All Program 4 references must be within these bounds

Multiple Domains

- Limiting a process to a single domain is not too convenient
- The concept is easy to extend
  - Simply allow multiple domains per process
- Obvious way to handle this is with multiple domain registers
  - One per allocated domain

The Multiple Domain Concept



Handling Multiple Domains

- Programs can request more domains
  - But the OS must set them up
- What does the program get to ask for?
  - A specific range of addresses?
  - Or a domain of a particular size?
- Latter is easier
  - What if requested set of addresses are already used by another program?
  - Memory manager can choose a range of addresses of requested size

Domains and Access Permissions

- One can typically do three types of things with a memory address
  - Read its contents
  - Write a new value to it
  - Execute an instruction located there
- System can provide useful effects if it does not allow all modes of use to all addresses
- Typically handled on a per-domain basis
  - E.g., read-only domains
- Requires extra bits in domain registers
- And other hardware support

What If Program Uses a Domain Improperly?

- E.g., it tries to write to a read-only domain
- A *permission error exception*
  - Different than an illegal memory reference exception
- But also handled by a similar mechanism
- Probably want it to be handled by somewhat different code in the OS
- Remember discussion of trap handling in previous lecture?

Do We Really Need to Switch Processes for OS Services?

- When we trap or make a request for a domain, must we change processes?
  - We lose context doing so
- Instead, run the OS code for the process
  - Which requires changing to supervisor mode
  - Context for process is still available
- But what about safety?
  - Use domain access modes to ensure safety
- We don’t do this for all OS services . . .

Domains in Kernel Mode

- Allow user threads to access certain privileged domains
  - Like code to handle hardware traps
  - Code must be in a user-accessible domain
- But can't allow arbitrary access to those privileged domains
- A supervisor (AKA *kernel*) mode access bit is set on such domains
  - So thread only accesses them when in kernel mode

How Does a Thread Get to Kernel Mode?

- Can't allow thread to arbitrarily put itself in kernel mode any time
  - Since it might do something unsafe
- Instead, allow entry to kernel mode only in specific ways
  - In particular, only at specific instructions
  - These are called *gates*
  - Typically implemented in hardware using instruction like SVC (supervisor call)