Modularity and Virtualization
CS 111
Operating Systems
Peter Reiher

Introduction

- Most useful abstractions an OS wants to offer can’t be directly realized by hardware
  - The hardware doesn’t do exactly what the abstraction requires
  - Multiple pieces of hardware are needed to achieve the abstraction
  - The hardware must be shared by multiple instances of the abstraction
- How do we provide the abstraction to users?

Virtualization and Modularity

- Use software to make the hardware we have look like the abstraction we want
  - That's virtualization
- Divide up the overall system you want into well-defined communicating pieces
  - That's modularity
- Using the two techniques allows us to build powerful systems from simple components
  - Without making the resulting system unmanageably complex

What Does An OS Do?

- At minimum, it enables one to run applications
- Preferably multiple applications on the same machine
- Preferably several at the same time
- At an 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)

What Have We Got To Work With?

- A processor
  - Maybe multicore
  - Maybe also some device controllers
- RAM
- Hard disks and other storage devices
- Busses and network hardware
- Other I/O devices

How to Get From What We've Got to What We Want?

- Build abstractions for what we want
- Out of the hardware we've actually got
- Use those abstractions to:
  - Hide messiness
  - Share resources
  - Simplify use
  - Provide safety and security
- From one point of view, that's what an operating system is all about

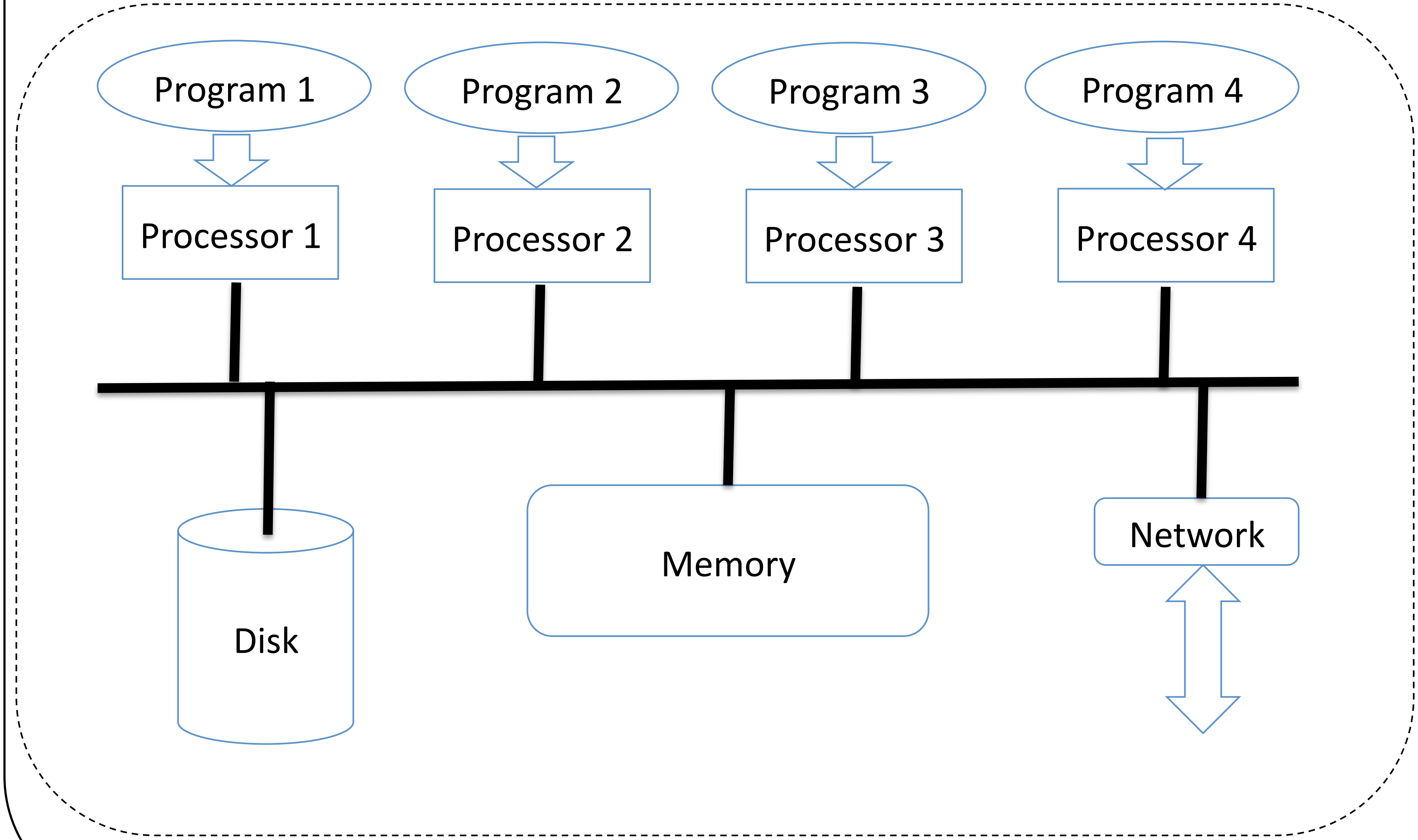
Real Hardware Vs. Desirable Abstractions

- In the last lecture, we looked at some real hardware issues
  - With relation to OS requirements
- Now let's see how those can be used to provide some useful OS abstractions

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

Things To Be Careful About

- Interference between different user tasks
- User task failure causing failure of other user tasks
  - Worse, causing failure of the overall system
- User tasks improperly overusing or misusing system resources
  - Need to be sure each task gets a fair share

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*

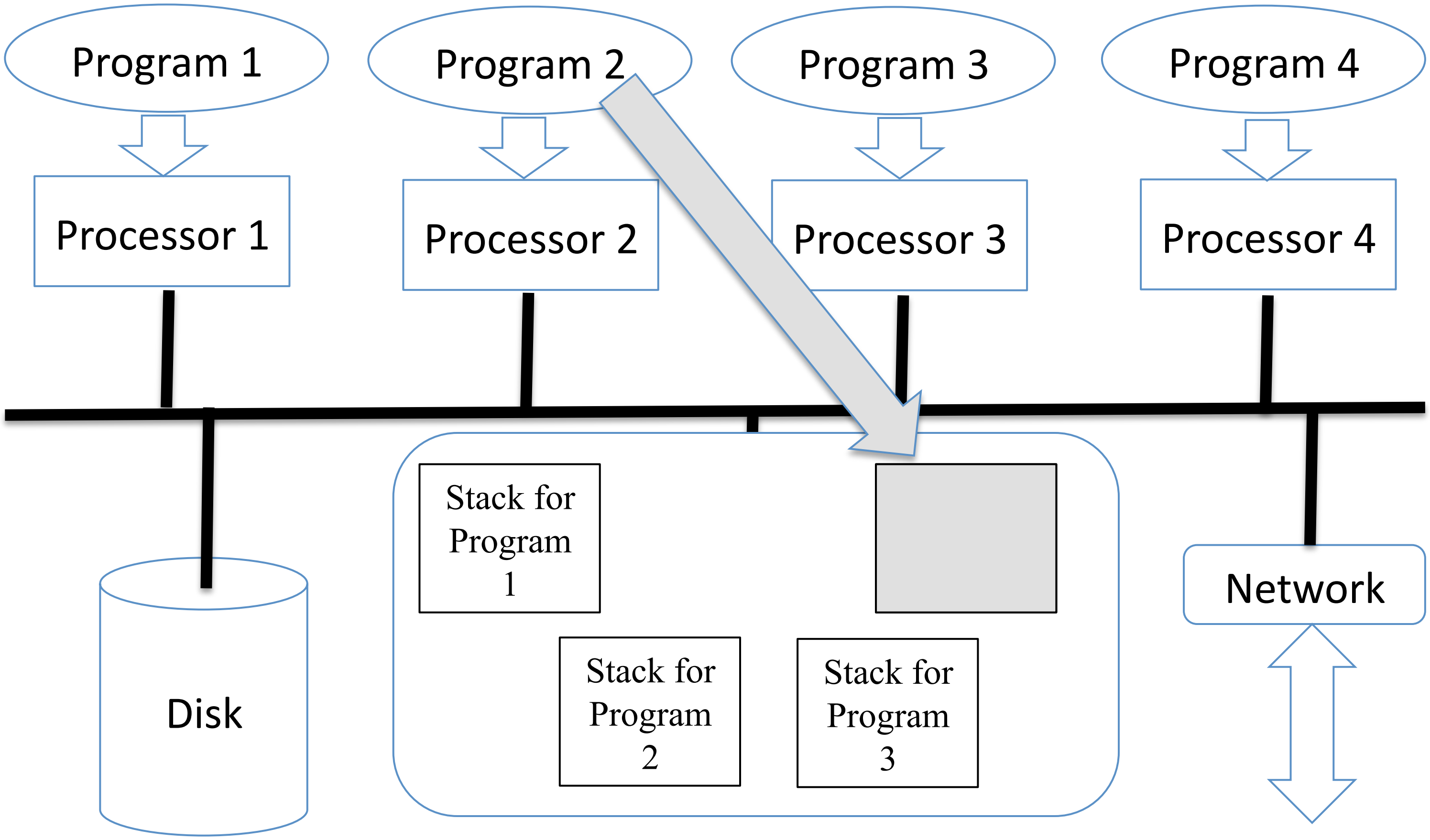
How Would This Work?

- Each program would be a self-contained set of subroutines
  - Subroutines in the program call each other
  - But not subroutines in other programs
- Shared services would be offered by other subroutines
  - Which any program can call
  - But which mostly don’t call programs
- Perhaps some “master routine” that calls subroutines in the various programs

What's Soft About This Modularity?

- Vital resources are shared
  - Like the RAM
- Proper behavior would prevent one program from treading on another's resources
- But no system or hardware features prevent it
- Maintaining module boundaries requires programs to all follow the rules
  - Even if they intend to, they might fail to do so because of programming errors

Illustrating the Problem



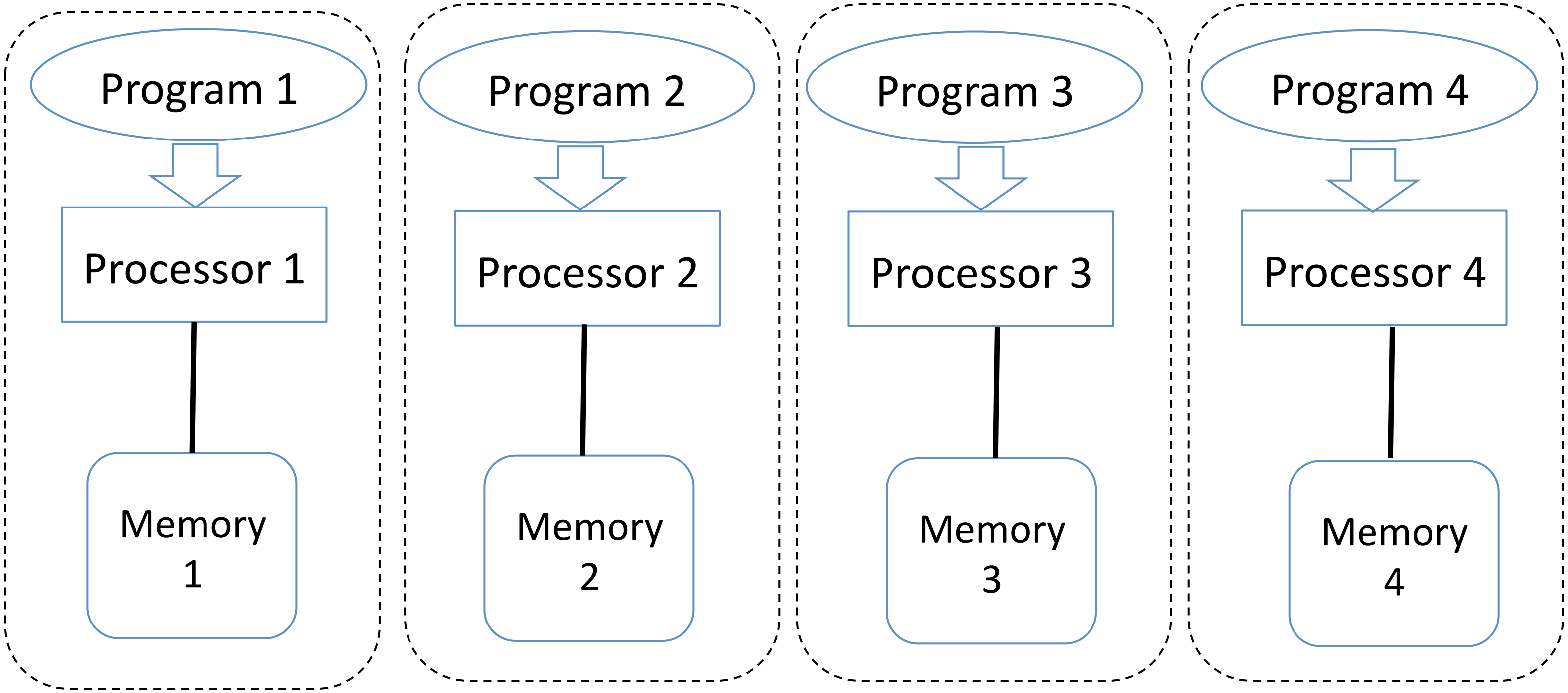
Now Program 4 is in trouble

Even though it did nothing wrong itself

Hardening the Modularity

- How can we more carefully separate the several competing programs?
- If each were on its own machine, the problem is easier
- No program can touch another's resources
  - Except via network messages
- Each program would have complete control over a full machine
  - No need to worry if some resource is yours or not

Illustrating Hard Modularity



Four separate machines

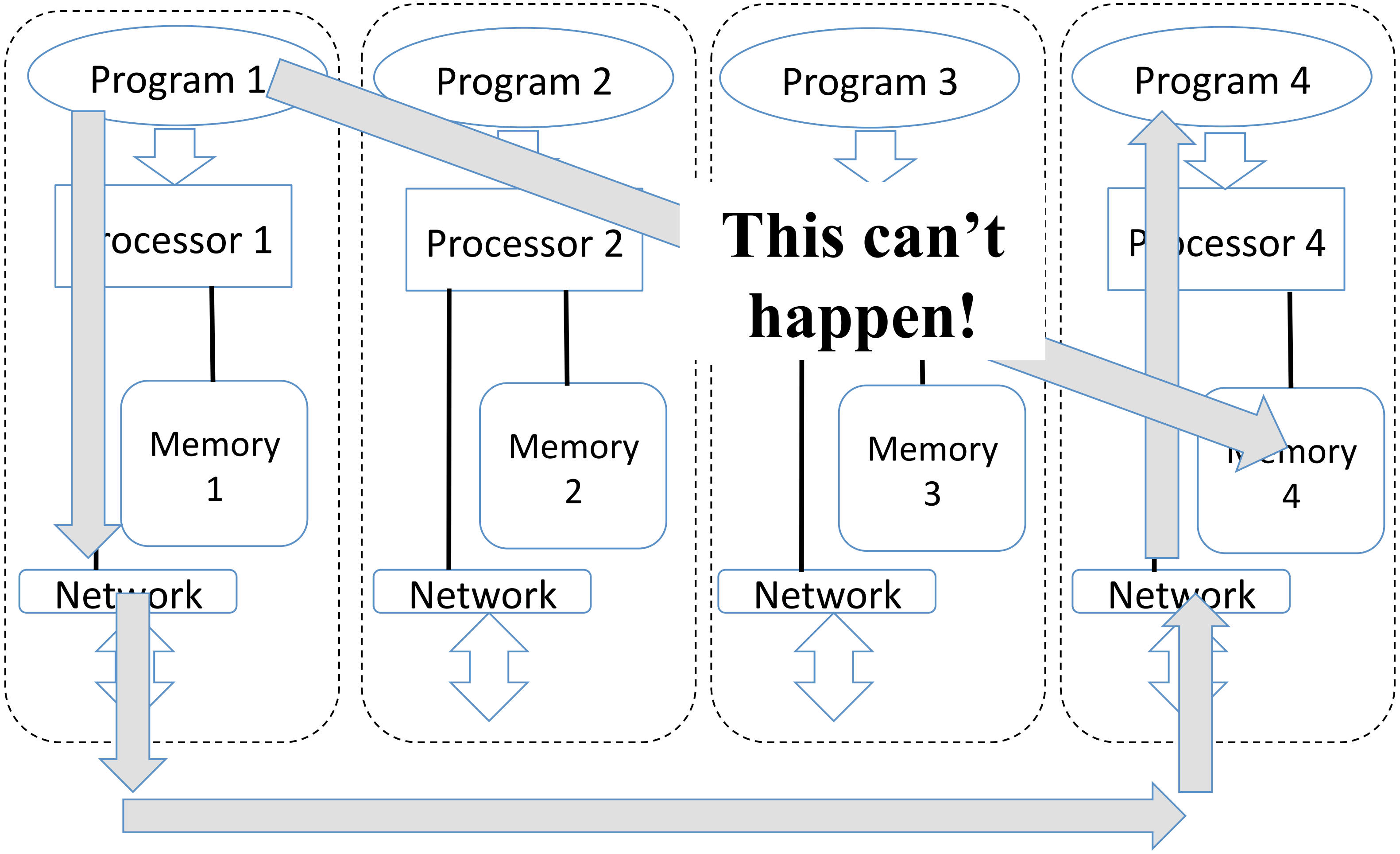
Perhaps in very different places

Each program has its own machine

Communications Across Machines

- Each machine would send messages to the others to communicate
- A machine receiving a message would take action as it saw fit
  - Typically doing what the sender requested
  - But with no opportunity for sender's own code to run
- Obvious opportunities for parallelism
  - And obvious dangers

Illustrating Communications



This can't happen!

If Program 1 needs to communicate with Program 4,

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 a message to the server to get help

A Storage Example

- A server keeps data persistently for all user programs
  - E.g., a file system
- User programs act as clients
  - Sending read/write messages to the server
- The server responds to reads with the requested data
- And to writes with acknowledgements of completion

Advantages of This Modularity For a Storage Subsystem

- Everyone easily sees the same persistent storage
- The server performs all actual data accesses
  - So no worries about concurrent writes or read/ write inconsistencies
- Server can ensure fair sharing
- Clients can't accidentally/intentionally corrupt the entire data store
  - Only things they are allowed to write

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 simple memory accesses
- If the boundary sits between components requiring fast interactions, possibly very bad
- A lot of what we do in operating systems involves this tradeoff

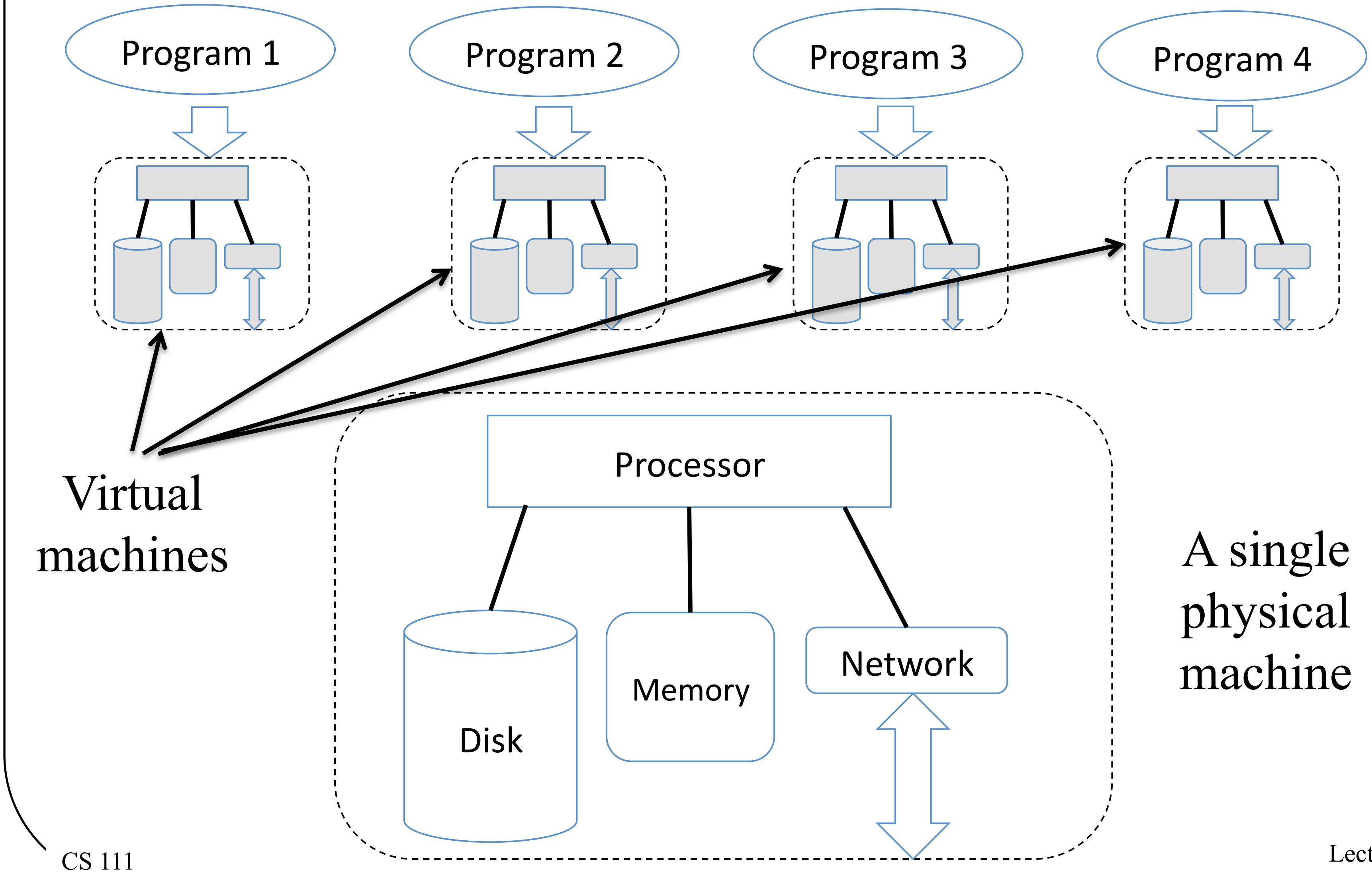
One Other Problem

- What if I don’t have enough hardware?
  - Not enough machines to give one to each client and server
  - Not enough memory, network capacity, etc.
- Am I forced to fall back on sharing machines and using soft modularity?

Virtualization

- A different alternative to providing harder modularity
- Provide the illusion of a complete machine to each program
- Use shared hardware to instantiate the various virtual machines
- System software (i.e., the operating system) and perhaps special hardware handle it

The Virtualization Concept



Program 1

Program 2

Program 3

Program 4

Virtual machines

Processor

Disk

Memory

Network

A single physical machine

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

Returning To Our Simple System

- We could build a system in which each program gets its own virtualized resources
- Providing stronger modularity than soft
  - But maybe not quite as hard as true separate hardware
- If we did that, what abstractions will our system need to support?
  - To provide the illusion of exclusive hardware

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

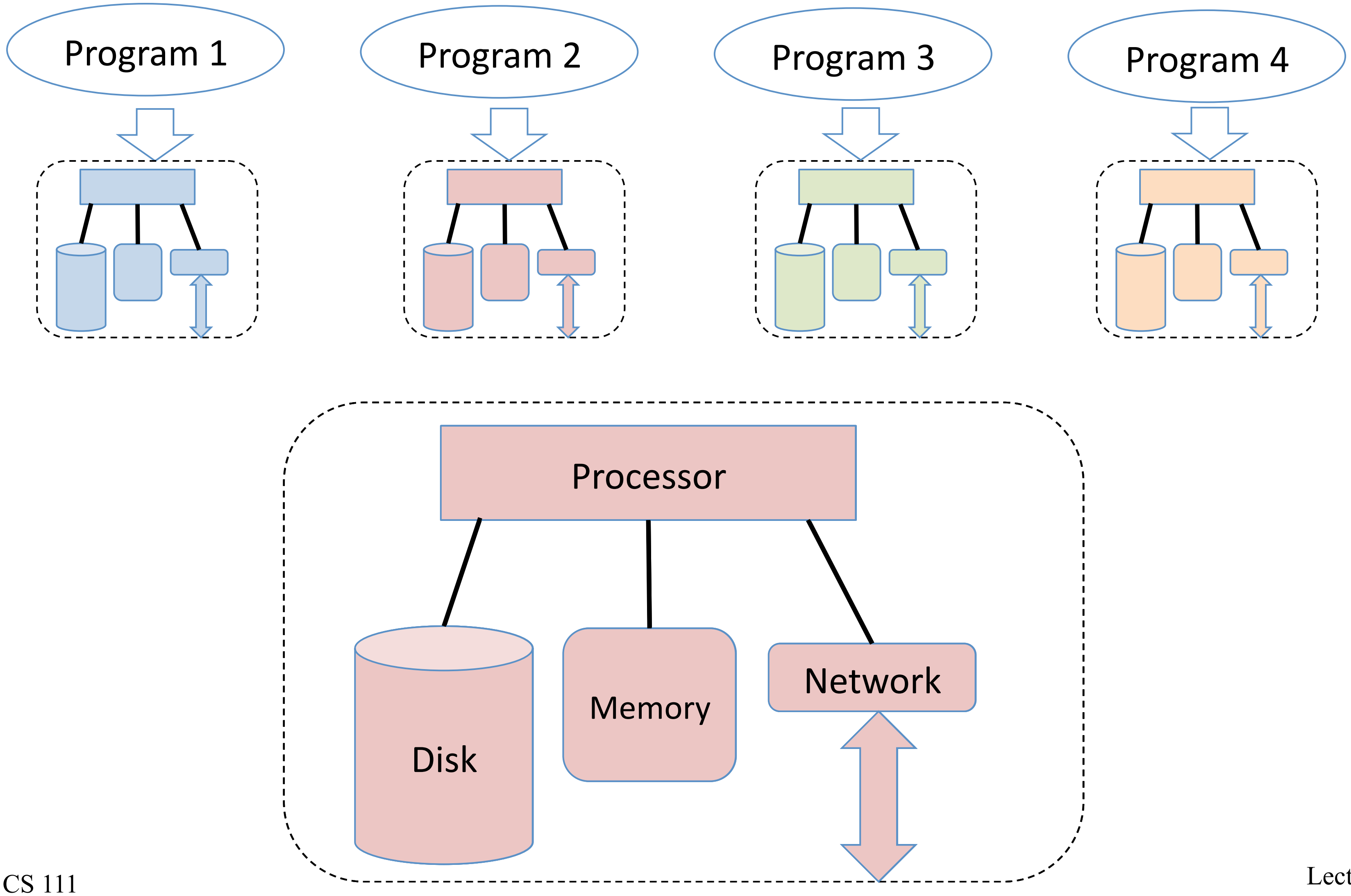
OS Handling of Threads

- There will be one (or more) threads for each program that is running
- The OS must choose which thread to run on which of its several processors
  - If more threads than processors, some threads will need to share processors
  - Which implies 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

Running One Thread

- The OS loads its executable code into memory
- The OS chooses a processor for the thread
- The OS creates control structures for the thread
  - A program counter to point to its first instruction
  - A stack to keep track of its various subroutine calls
  - Possibly other data areas for dynamic memory allocations
- The OS then transfers control of the processor to the thread

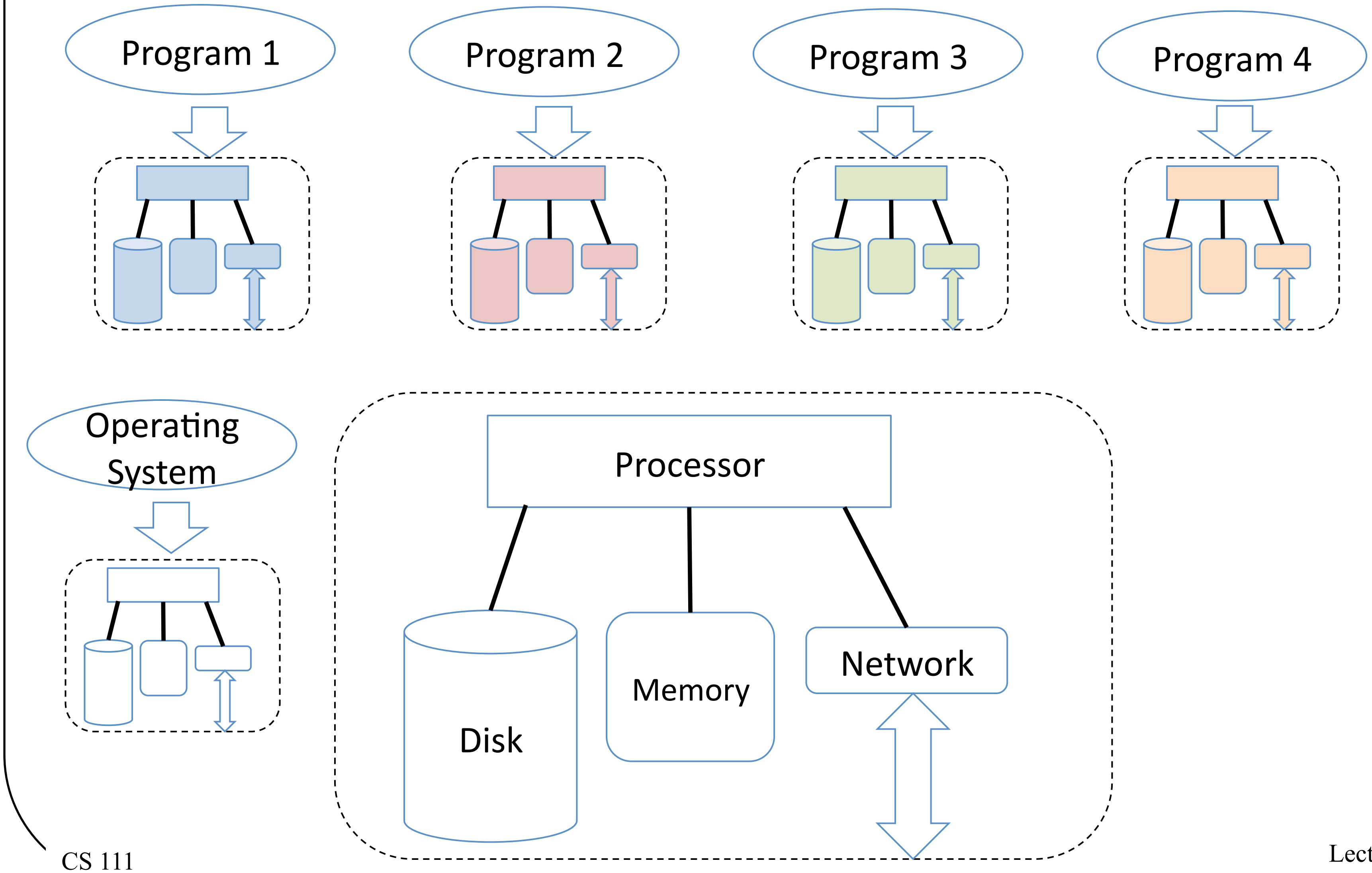
Time Slicing Virtualization



Wait a Minute . . .?

- How does the OS do all that?
- It’s just a program itself
  - Which implies it needs its own interpreter, memory, and communications
- It must use the same physical resources as all the other threads
- Basically, the OS itself is a thread
  - We’ll worry about where it comes from later
- It creates and manages other threads

The OS and Virtualization



Wait Another Minute . . .?

- Weren’t threads supposed to live in separate virtual machines?
  - Without interfering with each other?
- How can an OS thread set up and handle other threads if it can’t touch their virtual machines?
- It can’t
- The OS is a special thread, with special rights and responsibilities

Remember Supervisor Mode?

- From the last lecture
- One of modern processors' two modes
- Supervisor mode has special privileges
  - Which the other user mode does not
- Those privileges allow the OS thread to reach inside other threads' virtual machines
- Which allows the OS thread to set up and control them
  - That's why controlling who gets to be in supervisor mode is very important

The Thread Manager

- An OS component
- Its job is to handle the multiple current threads to be run
- Primary responsibilities:
  - Starting new threads
  - Ensuring each thread has its own contained environment
  - Ensuring fair treatment of all running threads

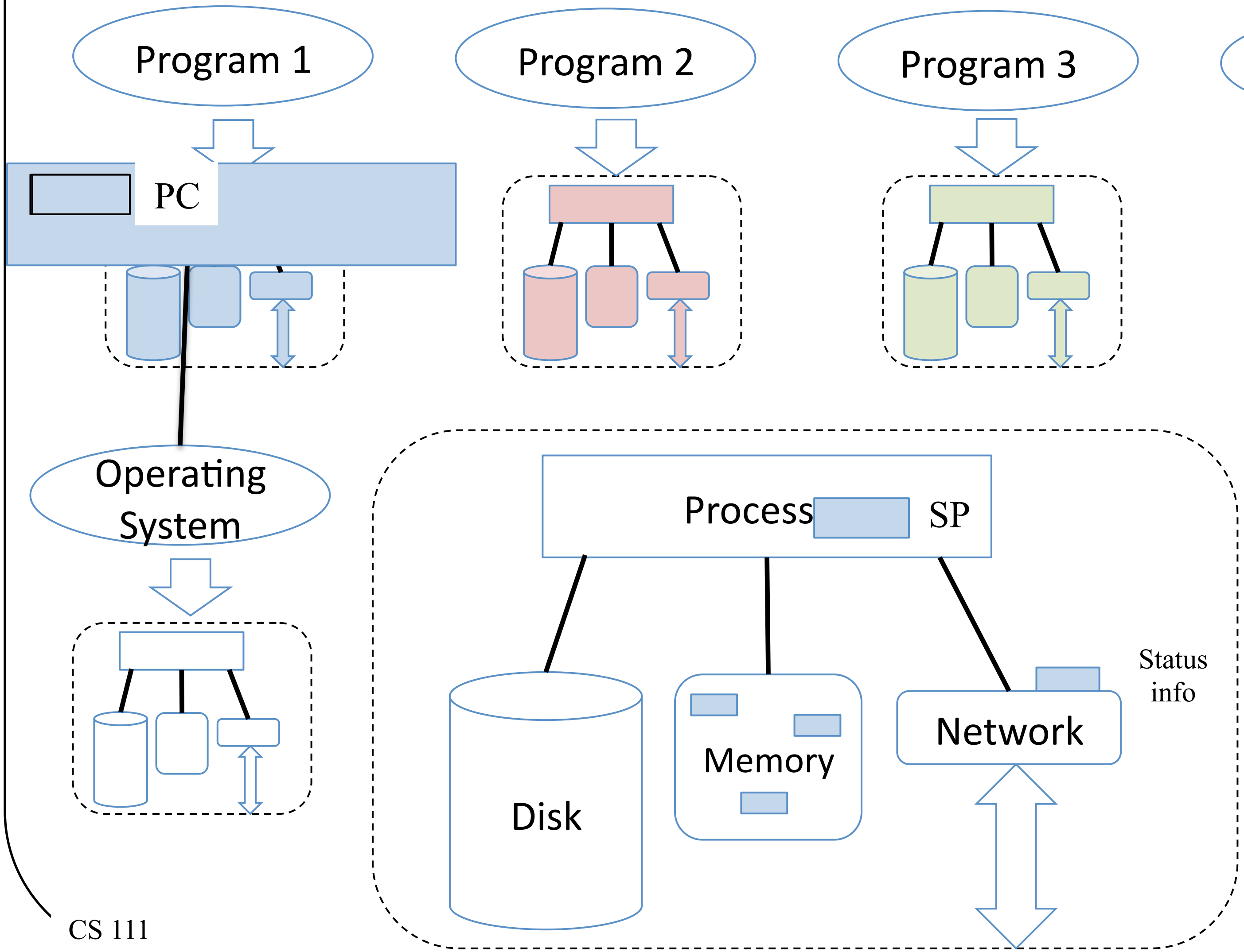
Providing Contained Environments

- 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



What about the disk?

That's handled differently, and we'll get to that later

Protecting Threads From Each Other

- Each thread is supposed to be independent
- Other threads should be unable to interfere with this one
  - And this one should not interfere with them
- Virtualization implies one or more forms of sharing of the hardware
  - Sharing makes interference more likely
- So how do we keep them safe from each other?

Protection via Execution Modes

- 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

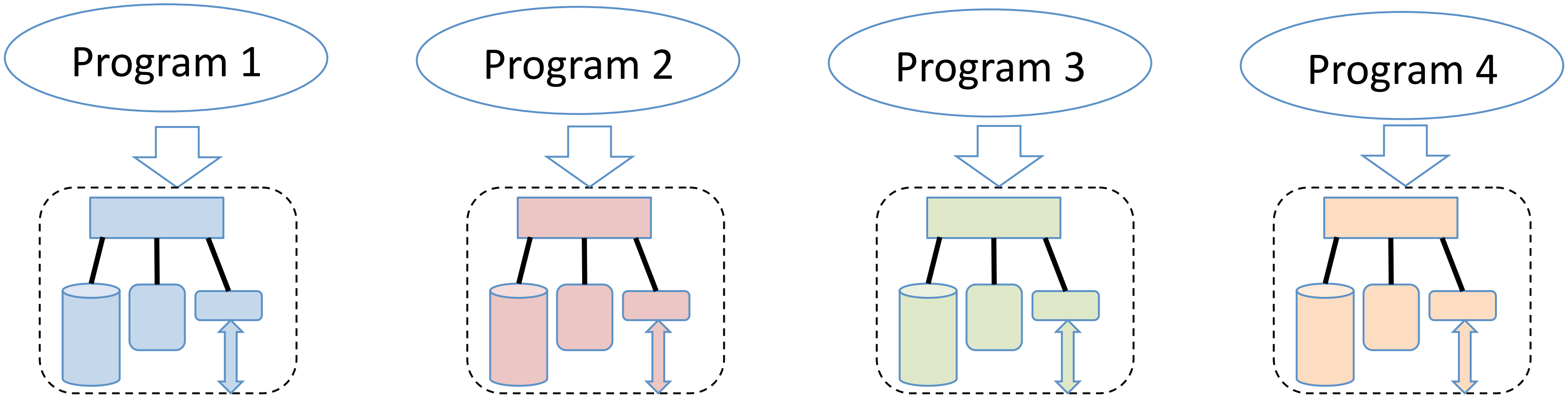
What's the Safety Issue?

- 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

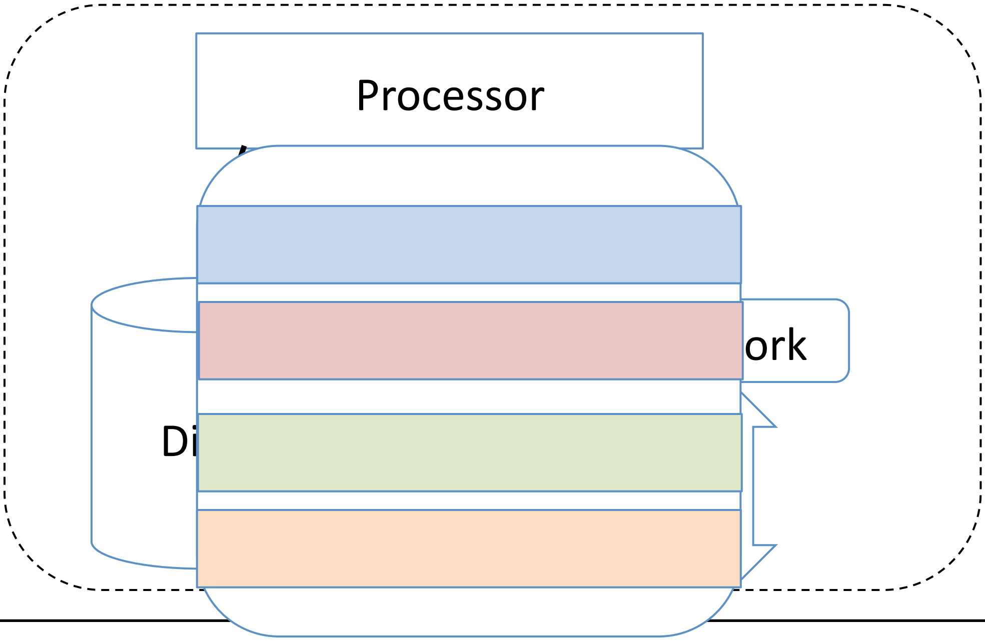
Domains

- 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 addresses in the thread’s domain should be honored
  - Issuing any other address should be caught as an error
- Can’t trust threads to police their own addresses
  - System must enforce that

Making It Work

- Generally requires hardware support
- In a simple way, a domain register
  - A processor has perhaps just one
  - It specifies the domain associated with the thread currently using the processor
  - By listing the low and high addresses that bound the domain
- OK, so we know what the thread's domain is
- Now what?

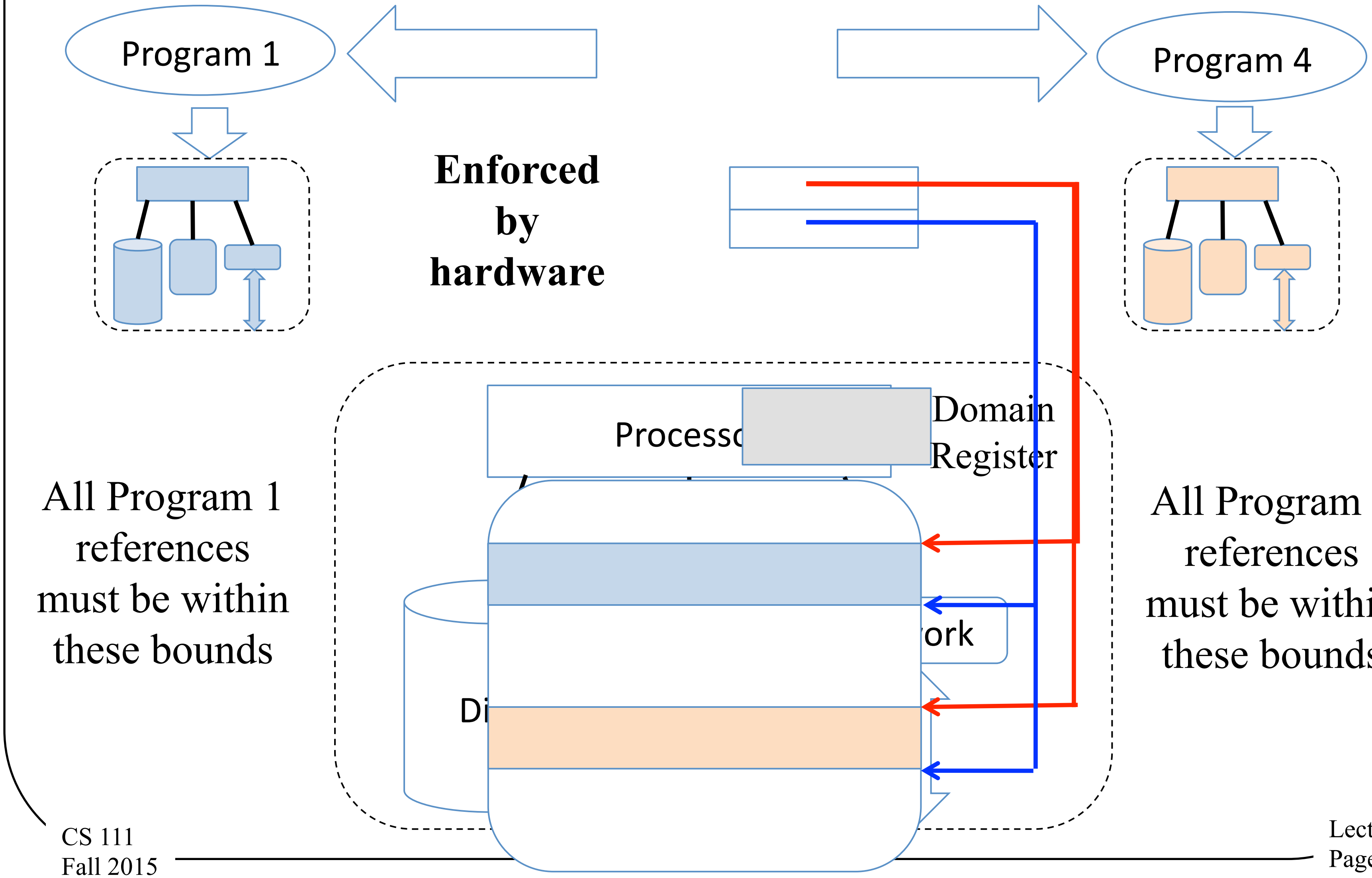
The Memory Manager

- 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 exception
- Only trusted code (i.e., the OS) can change the domain register

Illegal Memory Reference Exceptions

- The exception that gets thrown when a thread asks for memory not in its domain
  – Giving access might screw up another program
- What happens then?
- Trap to supervisor mode
  – To handle the problem safely

The Domain Register Concept



Enforced by hardware

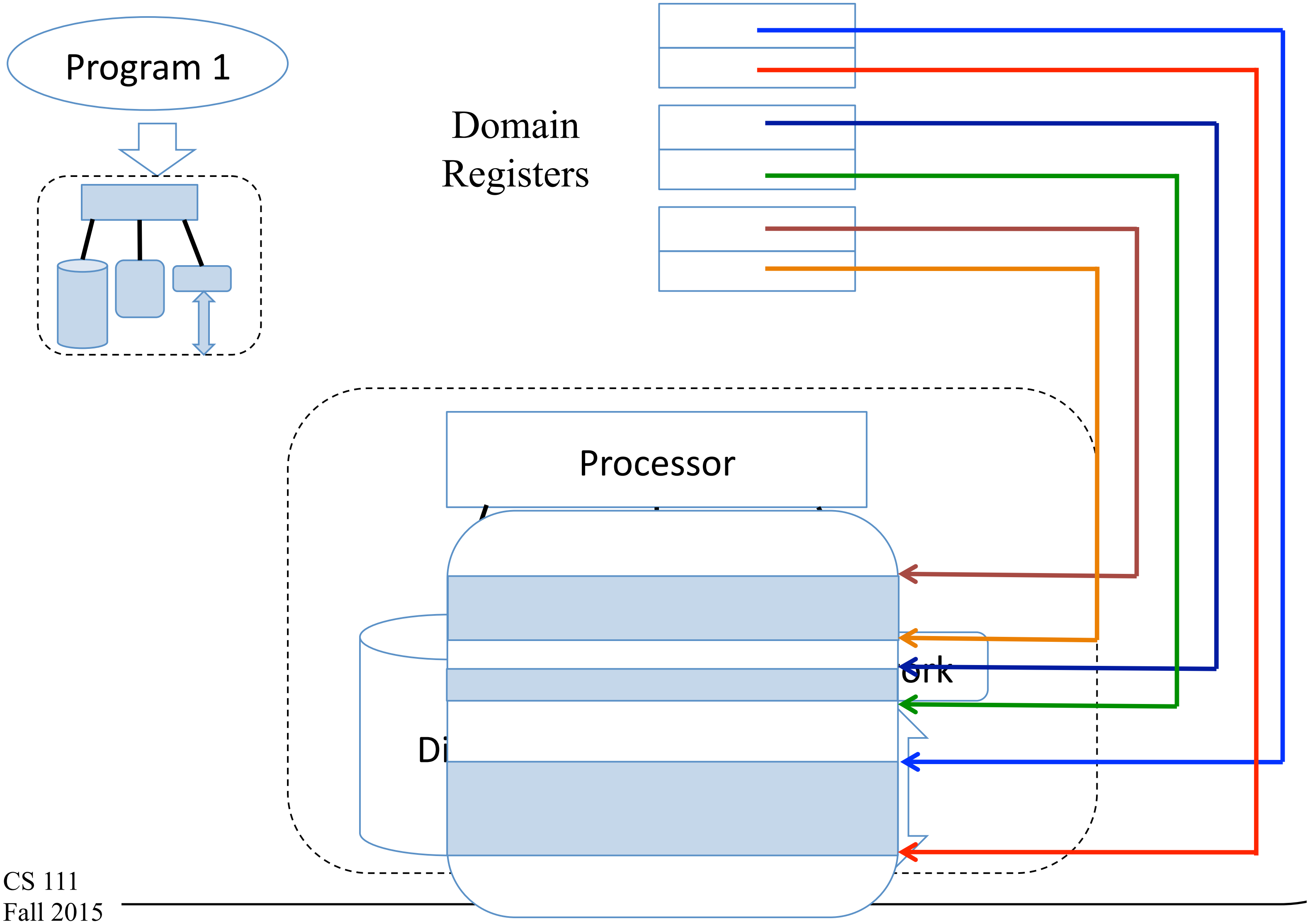
All Program 1 references must be within these bounds

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
  - Such as code to handle hardware traps
  - Such code must be in a domain accessible to the user thread
- 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)
  - Remember trapping to supervisor mode?