Introduction
CS 111
Operating System Principles
Peter Reiher

Introduction to the Course

- Purpose of course and relationships to other courses
- Why study operating systems?
- Major themes & lessons in this course

What Will CS 111 Do?

- Build on concepts from other courses
  - Data structures, programming languages, assembly language programming, network protocols, computer architectures, ...
- Prepare you for advanced courses
  - Data bases and distributed computing
  - Security, fault-tolerance, high availability
  - Computer system modeling, queueing theory
- Provide you with foundation concepts
  - Processes, threads, virtual address space, files
  - Capabilities, synchronization, leases, deadlock

Why Study Operating Systems?

- Few of you will actually build OSs
- But many of you will:
  - Set up, configure, manage computer systems
  - Write programs that exploit OS features
  - Work with complex, distributed, parallel software
  - Work with abstracted services and resources
- Many hard problems have been solved in OS context
  - Synchronization, security, integrity, protocols, distributed computing, dynamic resource management, ...
  - In this class, we study these problems and their solutions
  - These approaches can be applied to other areas

Why Are Operating Systems Interesting?

- They are extremely complex
  - But try to appear simple enough for everyone to use
- They are very demanding
  - They require vision, imagination, and insight
  - They must have elegance and generality
  - They demand meticulous attention to detail
- They are held to very high standards
  - Performance, correctness, robustness,
  - Scalability, extensibility, reusability
- They are the base we all work from

Recurring OS Themes

- View services as objects and operations
  - Behind every object there is a data structure
- Separate policy from mechanism
  - Policy determines what can/should be done
  - Mechanism implements basic operations to do it
  - Mechanisms shouldn't dictate or limit policies
  - Must be able to change policies without changing mechanisms
- Parallelism and asynchrony are powerful and necessary
  - But dangerous when used carelessly

More Recurring Themes

- An interface specification is a contract
  - Specifies responsibilities of producers & consumers
  - Basis for product/release interoperability
- Interface vs. implementation
  - An implementation is not a specification
  - Many compliant implementations are possible
  - Inappropriate dependencies cause problems
- Modularity and functional encapsulation
  - Complexity hiding and appropriate abstraction

What Is An Operating System?

- Many possible definitions
- One is:
  - It is low level software . . .
  - That provides better abstractions of hardware below it
  - To allow easy, safe, fair use and sharing of those resources

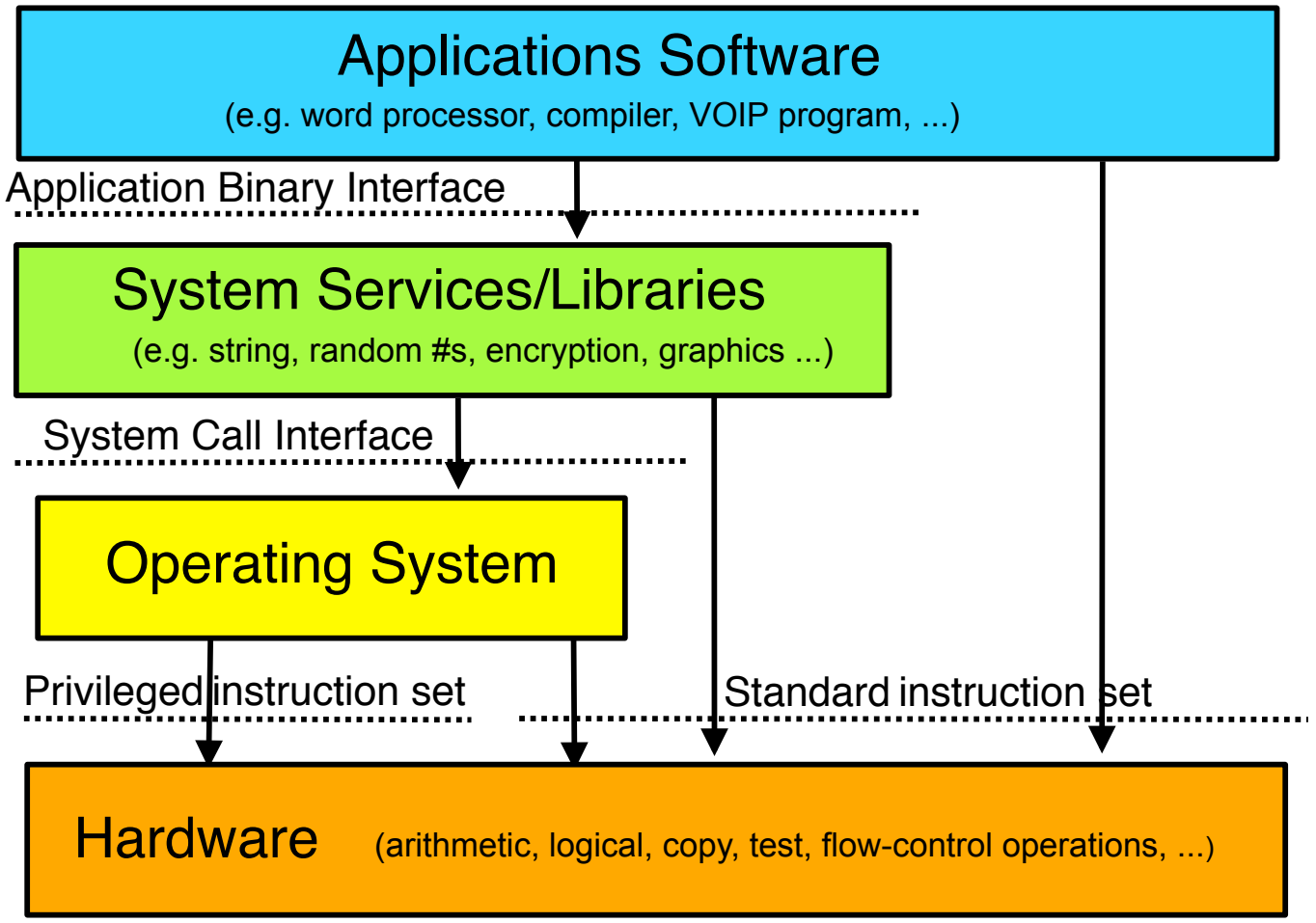
What Does an OS Do?

- It manages hardware for programs
  - Allocates hardware and manages its use
  - Enforces controlled sharing (and privacy)
  - Oversees execution and handles problems
- It abstracts the hardware
  - Makes it easier to use and improves s/w portability
  - Optimizes performance
- It provides new abstractions for applications
  - Powerful features beyond the bare hardware

What Does An OS Look Like?

- A set of management & abstraction services
  - Invisible, they happen behind the scenes
- Applications see objects and their services
  - CPU supports data-types and operations
    - Bytes, shorts, longs, floats, pointers, ...
    - Add, subtract, copy, compare, indirection, ...
  - So does an operating system, but at a higher level
    - Files, processes, threads, devices, ports, ...
    - Create, destroy, read, write, signal, ...
- An OS extends a computer
  - Creating a much richer virtual computing platform
    - Supporting richer objects, more powerful operations

Where Does the OS Fit In?



What's Special About the OS?

- It is always in control of the hardware
  - Automatically loaded when the machine boots
  - First software to have access to hardware
  - Continues running while apps come & go
- It alone has complete access to hardware
  - Privileged instruction set, all of memory & I/O
- It mediates applications' access to hardware
  - Block, permit, or modify application requests
- It is trusted
  - To store and manage critical data
  - To always act in good faith
- If the OS crashes, it takes everything else with it
  - So it better not crash . . .

What Functionality Is In the OS?

- As much as necessary, as little as possible
  - OS code is very expensive to develop and maintain
- Functionality must be in the OS if it ...
  - Requires the use of privileged instructions
  - Requires the manipulation of OS data structures
  - Must maintain security, trust, or resource integrity
- Functions should be in libraries if they ...
  - Are a service commonly needed by applications
  - Do not actually have to be implemented inside OS
- But there is also the performance excuse
  - Some things may be faster if done in the OS

Where To Offer a Service?

- Hardware, OS, library or application?
- Increasing requirements for stability as you move through these options
- Hardware services rarely change
- OS services can change, but it's a big deal
- Libraries a bit more dynamic
- Applications can change services much more readily

Another Reason For This Choice

- Who uses it?
- Things literally everyone uses belong lower in the hierarchy
  - Particularly if the same service needs to work the same for everyone
- Things used by fewer/more specialized parties belong higher
  - Particularly if each party requires a substantially different version of the service

The OS and Speed

- One reason operating systems get big is based on speed
- It's faster to offer a service in the OS than outside it
  - If it involves processes communicating, working at app level requires scheduling and swapping them
  - The OS has direct access to many pieces of state and system services
  - The OS can make direct use of privileged instructions
- Thus, there's a push to move services with strong performance requirements down to the OS

The OS and Abstraction

- One major function of an OS is to offer abstract versions of resources
  - As opposed to actual physical resources
- Essentially, the OS implements the abstract resources using the physical resources
  - E.g., processes (an abstraction) are implemented using the CPU and RAM (physical resources)
  - And files (an abstraction) are implemented using disks (a physical resource)

Why Abstract Resources?

- The abstractions are typically simpler and better suited for programmers and users
  - Easier to use than the original resources
    - E.g., don't need to worry about keeping track of disk interrupts
  - Compartmentalize/encapsulate complexity
    - E.g., need not be concerned about what other executing code is doing and how to stay out of its way
  - Eliminate behavior that is irrelevant to user
    - E.g., hide the sectors and tracks of the disk
  - Create more convenient behavior
    - E.g., make it look like you have the network interface entirely for your own use

Common Types of OS Resources

- Serially reusable resources
- Partitionable resources
- Sharable resources

Serially Reusable Resources

- Used by multiple clients, but only one at a time
  - Time multiplexing
- Require access control to ensure exclusive use
- Require graceful transitions from one user to the next
  - A switch that totally hides the fact that the resource used to belong to someone else
- Examples: printers, bathroom stalls

Partitionable Resources

- Divided into disjoint pieces for multiple clients
  - Spatial multiplexing
- Needs access control to ensure:
  - Containment: *you cannot access resources outside of your partition*
  - Privacy: *nobody else can access resources in your partition*
- Examples: disk space, dormitory rooms

Shareable Resources

- Usable by multiple concurrent clients
  - Clients do not have to "wait" for access to resource
  - Clients don't "own" a particular subset of resource
- May involve (effectively) limitless resources
  - Air in a room, shared by occupants
  - Copy of the operating system, shared by processes
- May involve under-the-covers multiplexing
  - Cell-phone channel (time and frequency multiplexed)
  - Shared network interface (time multiplexed)

General OS Trends

- They have grown larger and more sophisticated
- Their role has fundamentally changed
  - From shepherding the use of the hardware
  - To shielding the applications from the hardware
  - To providing powerful application computing platform
- They still sit between applications and hardware
- Best understood through services they provide
  - Capabilities they add
  - Applications they enable
  - Problems they eliminate

Another Important OS Trend

- Convergence
  - There are a handful of widely used OSs
  - New ones come along very rarely
- OSs in the same family (e.g., Windows or Linux) are used for vastly different purposes
  - Making things challenging for the OS designer
- Most OSs are based on pretty old models
  - Linux comes from Unix (1970s vintage)
  - Windows from the early 1980s

A Resulting OS Challenge

- We are basing the OS we use today on an architecture designed 30-40 years ago
- We can make some changes in the architecture
- But not too many
  - Due to compatibility
  - And fundamental characteristics of the architecture
- Requires OS designers and builders to shoehorn what's needed today into what made sense yesterday

Important OS Properties

- For real operating systems built and used by real people
- Differs depending on who you are talking about
  - Users
  - Service providers
  - Application developers
  - OS developers

For the End Users,

- Reliability
- Performance
- Upwards compatibility in releases
- Support for differing hardware
  - Currently available platforms
  - What's available in the future
- Availability of key applications
- Security

Reliability

- Your OS really should never crash
  - Since it takes everything else down with it
- But also need dependability in a different sense
  - The OS must be depended on to behave as it's specified
  - Nobody wants surprises from their operating system
  - Since the OS controls everything, unexpected behavior could be arbitrarily bad

Performance

- A loose goal
- The OS must perform well in critical situations
- But optimizing the performance of all OS operations not always critical
- Nothing can take too long
- But if something is “fast enough,” adding complexity to make it faster not worthwhile

Upward Compatibility

- People want new releases of an OS
  - New features, bug fixes, enhancements
  - Security patches to protect from malware
- People also fear new releases of an OS
  - OS changes can break old applications
- What makes the compatibility issue manageable?
  - Stable interfaces

Stable Interfaces

- Designers should start with well specified Application Interfaces
  - Must keep them stable from release to release
- Application developers should only use committed interfaces
  - Don't use undocumented features or erroneous side effects

APIs

- Application Program Interfaces
  - A source level interface, specifying:
    - Include files, data types, constants
    - Macros, routines and their parameters
- A basis for software portability
  - Recompile program for the desired architecture
  - Linkage edit with OS-specific libraries
  - Resulting binary runs on that architecture and OS
- An API compliant program will compile & run on any compliant system

ABIs

- Application Binary Interfaces
  - A binary interface, specifying
    - Dynamically loadable libraries (DLLs)
    - Data formats, calling sequences, linkage conventions
  - The binding of an API to a hardware architecture
- A basis for binary compatibility
  - One binary serves all customers for that hardware
    - E.g. all x86 Linux/BSD/MacOS/Solaris/…
    - May even run on Windows platforms
- An ABI compliant program will run (unmodified) on any compliant system

For the Service Providers,

- Reliability
- Performance
- Upwards compatibility in releases
- Platform support (wide range of platforms)
- Manageability
- Total cost of ownership
- Support (updates and bug fixes)
- Flexibility (in configurations and applications)
- Security

For the Application Developers,

- Reliability
- Performance
- Upwards compatibility in releases
- Standards conformance
- Functionality (current and roadmap)
- Middleware and tools
- Documentation
- Support (how to ...)

For the OS Developers,

- Reliability
- Performance
- Maintainability
- Low cost of development
  - Original and ongoing

Maintainability

- Operating systems have very long lives
  - Solaris, the "new kid on the block," came out in 1993
- Basic requirements will change many times
- Support costs will dwarf initial development
- This makes maintainability critical
- Aspects of maintainability:
  - Understandability
  - Modularity/modifiability
  - Testability

Critical OS Abstractions

- One of the main roles of an operating system is to provide abstract services
  - Services that are easier for programs and users to work with
- What are the important abstractions an OS provides?

Abstractions of Memory

- Many resources used by programs and people relate to data storage
  - Variables
  - Chunks of allocated memory
  - Files
  - Database records
  - Messages to be sent and received
- These all have some similar properties

The Basic Memory Operations

- Regardless of level or type, memory abstractions support a couple of operations
  - WRITE(name, value)
    - Put a value into a memory location specified by name
  - value <- READ(name)
    - Get a value out of a memory location specified by name
- Seems pretty simple
- But going from a nice abstraction to a physical implementation can be complex

An Example Memory Abstraction

- A typical file
- We can read or write the file
- We can read or write arbitrary amounts of data
- If we write the file, we expect our next read to reflect the results of the write
  - Coherence
- If there are several reads/writes to the file, we expect each to occur in some order
  - With respect to the others

Abstractions of Interpreters

- An interpreter is something that performs commands
- Basically, the element of a computer (abstract or physical) that gets things done
- At the physical level, we have a processor
- That level is not easy to use
- The OS provides us with higher level interpreter abstractions

Basic Interpreter Components

- An instruction reference
  - Tells the interpreter which instruction to do next
- A repertoire
  - The set of things the interpreter can do
- An environment reference
  - Describes the current state on which the next instruction should be performed
- Interrupts
  - Situations in which the instruction reference pointer is overriden

An Example Interpreter Abstraction

- A CPU
- It has a program counter register indicating where the next instruction can be found
  - An instruction reference
- It supports a set of instructions
  - Its repertoire
- It has contents in registers and RAM
  - Its environment

Abstractions of Communications Links

- A communication link allows one interpreter to talk to another
  - On the same or different machines
- At the physical level, wires and cables
- At more abstract levels, networks and interprocess communication mechanisms
- Some similarities to memory abstractions
  - But also differences

Basic Communication Link Operations

- SEND(link_name, outgoing_message_buffer)
  - Send some information contained in the buffer on the named link
- RECEIVE(link_name, incoming_message_buffer)
  - Read some information off the named link and put it into the buffer
- Like WRITE and READ, in some respects

An Example Communications Link Abstraction

- A Unix-style socket
- SEND interface:
  - send(int sockfd, const void *buf, size_t len, int flags)
  - The sockfd is the link name
  - The buf is the outgoing message buffer
- RECEIVE interface:
  - recv(int sockfd, void *buf, size_t len, int flags)
  - Same parameters as for send

Some Other Abstractions

- Actors
  - Users or other "active" entities
- Virtual machines
  - Collections of other abstractions
- Protection environments
  - Security related, usually
- Names
- Not a complete list
- Not everyone would agree on what's distinct

Hardware and the Operating System

- OS abstractions are built on the hardware, at the bottom
  - Everything ultimately relies on hardware
- 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
- 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

Processor Issues

- Execution mode
- Handling exceptions

Execution Modes

- Modern CPUs can usually 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

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

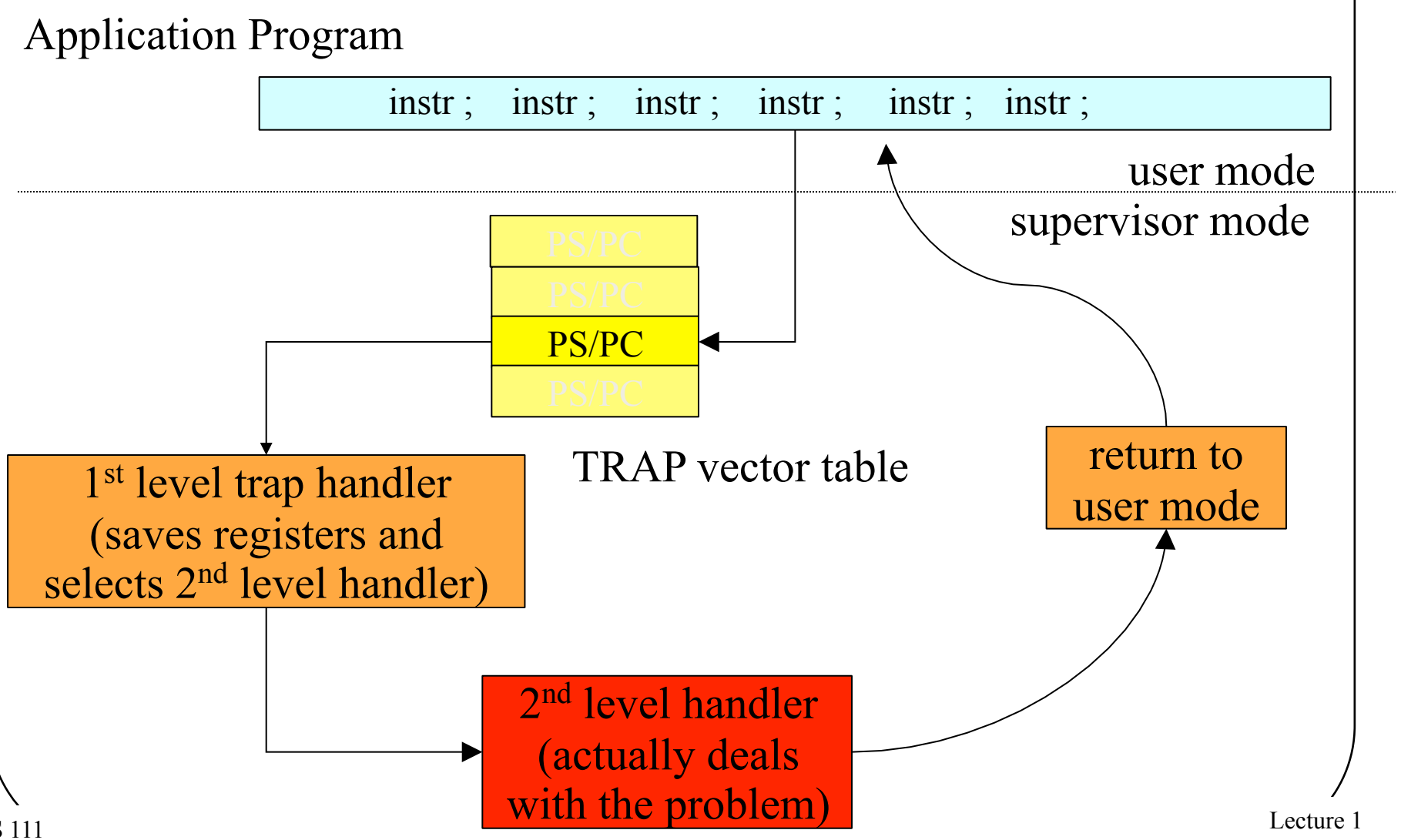
Asynchronous Exceptions and Handlers

- 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

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 managed by the OS
- Ordinary programs can't access these vectors
- The OS controls all supervisor mode transitions
  - By carefully controlling all of the trap/interrupt "gateways"
- Traps/interrupts can happen while in supervisor mode
  - Their handling is similar, but a little easier

Software Trap Handling



Dealing With the Cause of a Trap

- Some exceptions are handled by the OS
  - For example, 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