Introduction to Operating Systems

1. What is an Operating System?
2. Review of OS-Related Hardware

What is an Operating System?

Software that provides useful abstractions to application programs and effectively manages computer resources

Operating System Goals

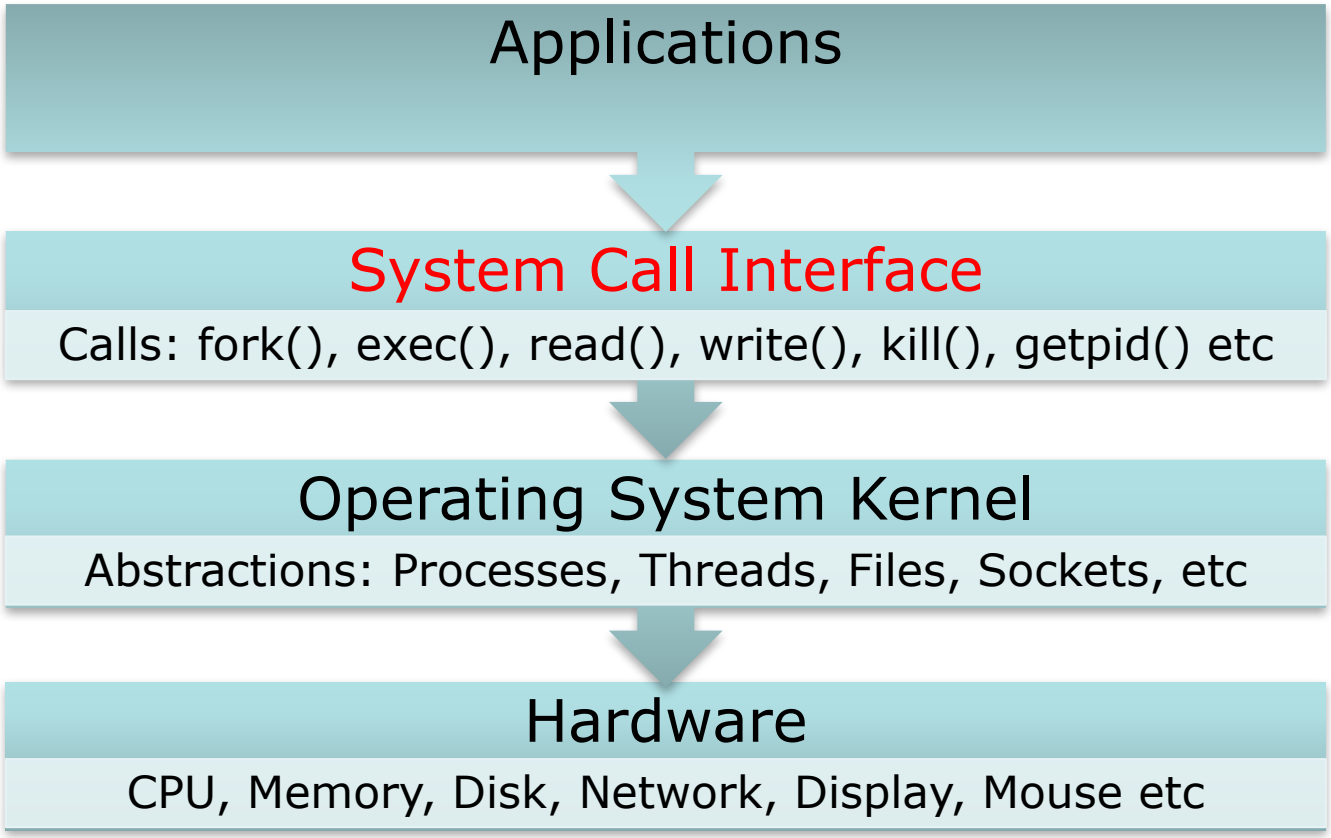
Useful Abstractions

- Hide complex details of the underlying hardware
- Provide interfaces to applications and services
- Simplify application writing

Effective Management of Resources

- Deciding what to hide and what to expose
- Defining suitable abstractions
- Efficient, secure, robust mapping to hardware
- Fair sharing among diverse applications and users

Operating System Structure



The Resource Manager Role

Allocating resources to applications
- time sharing resources
- space sharing resources

Making efficient use of limited resources
- improving utilization
- minimizing overhead
- improving throughput

Protection via enforcement of allocations

Enforcement by OS

The OS must protect itself from applications
It must protect applications from each other
It must prevent direct access to hardware
(Should it prevent direct access to hardware?)

But the OS is just software!
How can it do all this?

OS Needs Help from Hardware

The OS dilemma:

- The OS is just a program!
  - *When it is not running, it can't do anything!*
- The OS's goal is to run applications, not itself!

The OS needs help from the hardware in order to detect and prevent certain activities, and to enforce allocations

Brief Review of Hardware

Instruction sets define all that a CPU can do

- They differ among CPU architectures
- But all have load and store instructions to move data between memory and registers
- Many instructions for comparing and combining values in registers

Examine the x86 instruction set

Basic Anatomy of a CPU

Program Counter (PC)

Basic Anatomy of a CPU

Program Counter (PC)

- Holds the memory address of the next instruction

Basic Anatomy of a CPU

Program Counter (PC)

- Holds the memory address of the next instruction

Instruction Register

Basic Anatomy of a CPU

Program Counter (PC)

- Holds the memory address of the next instruction

Instruction Register

- Holds the instruction currently being executed

Basic Anatomy of a CPU

Program Counter (PC)

- Holds the memory address of the next instruction

Instruction Register

- holds the instruction currently being executed

General Purpose Registers

Basic Anatomy of a CPU

Program Counter (PC)

- Holds the memory address of the next instruction

Instruction Register

- holds the instruction currently being executed

General Purpose Registers

- hold variables and temporary results

Basic Anatomy of a CPU

Program Counter (PC)

- Holds the memory address of the next instruction

Instruction Register

- holds the instruction currently being executed

General Purpose Registers

- hold variables and temporary results

Arithmetic and Logic Unit (ALU)

Basic Anatomy of a CPU

Program Counter (PC)

- Holds the memory address of the next instruction

Instruction Register

- holds the instruction currently being executed

General Purpose Registers

- hold variables and temporary results

Arithmetic and Logic Unit (ALU)

- performs arithmetic functions and logic operations

Basic Anatomy of a CPU

Stack Pointer (SP)

Basic Anatomy of a CPU

Stack Pointer (SP)

- holds memory address of a stack top
- one frame for parameters & local variables of each active procedure

Basic Anatomy of a CPU

- Stack Pointer (SP)
  - holds memory address of a stack top
  - one frame for parameters & local variables of each active procedure
- Status Register

Basic Anatomy of a CPU

- Stack Pointer (SP)
  - holds memory address of a stack top
  - one frame for parameters & local variables of each active procedure
- Status Register
  - A word full of control flags/bits
  - Includes the mode bit to determine whether the CPU will execute privileged instructions

Program Execution

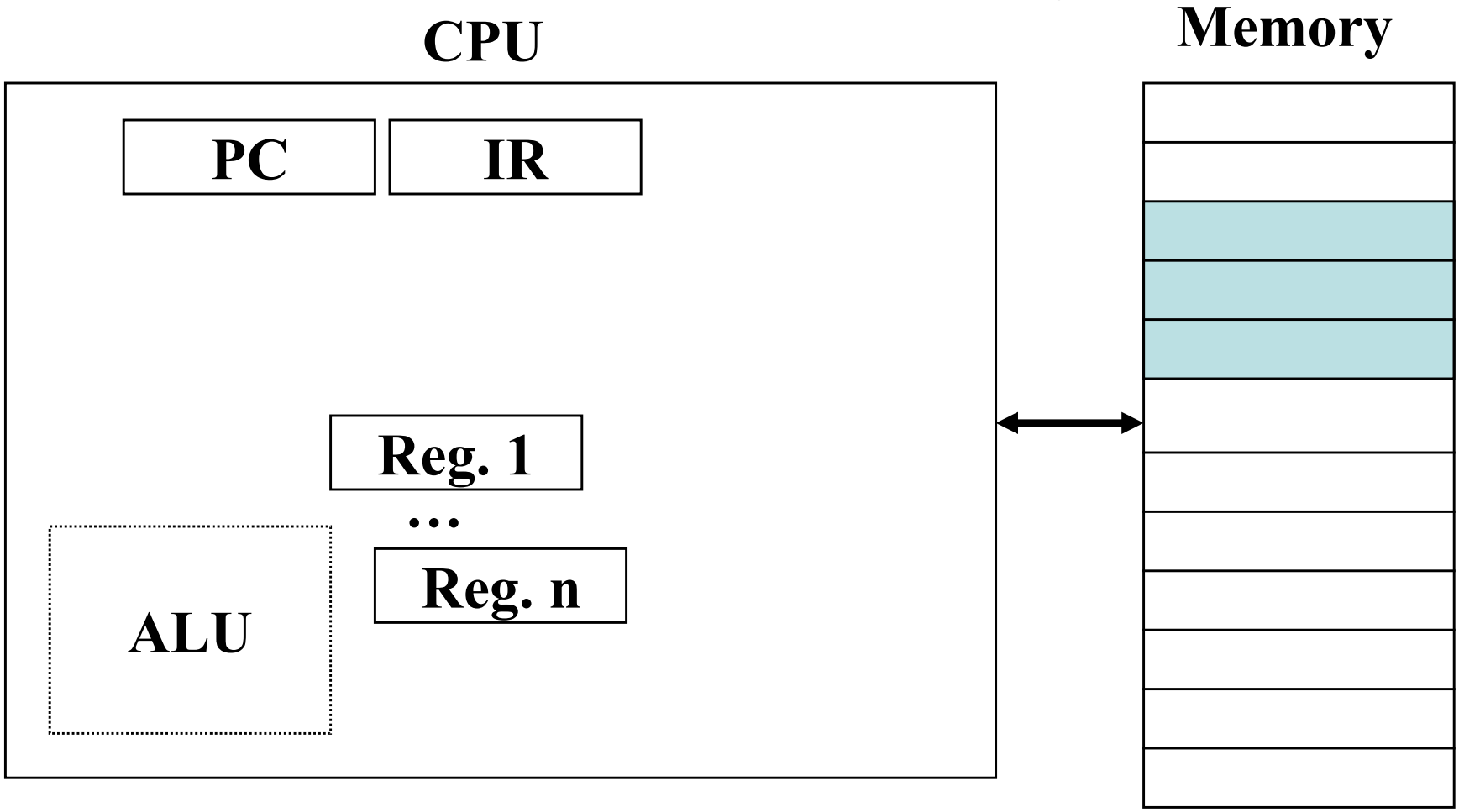
The Fetch/Decode/Execute cycle

- fetch next instruction pointed to by PC
- decode it to find its type and operands
- execute it
- repeat

At a fundamental level, this is all a CPU does -

- It does not know which program it is executing!

Fetch/Decode/Execute Cycle



Key Concept: Limited Direct Execution

1. We want user programs to use the CPU/hardware directly
2. But we want the OS to still be in control and to prevent disasters

The tension between these two goals drives a lot of OS design

The OS is Just a Program!

The OS is just a sequence of instructions that the CPU will fetch/decode/execute

How can the OS cause application programs to run?

How can applications cause the OS to run?

How Can an OS Run Applications?

How Can an OS Run Applications?

The OS must load the address of the application's starting instruction into the PC

One way to do it:

- OS loads application (executable file) into memory
- OS loads the address of the app's first instruction into the PC
- CPU fetches/decodes/executes the application's instructions

But will the OS Ever Run Again?

Yes! If the application calls a system call, then control will transfer to the OS.

Examples: open(), read(), write(), getpid(), close(), exit()

System calls look like C function calls but they are much more complicatedsites.google.com/pdx.edu/bruceirvin

How Can the OS Regain Control?

What if an application doesn't call any system calls and instead just hogs the CPU?

- OS needs something to *interrupt* the CPU and load OS instructions again
- interrupts can be generated from a timer device
- OS must register a future timer interrupt before handing control of the CPU over to an application
- When the timer interrupt goes off the hardware starts running the OS at a pre-specified location called an *interrupt handler*

What is an Interrupt?

Interrupt:

- Hardware event that loads an address of an instruction into the PC

Interrupt handler:

- An instruction sequence that has been associated with an interrupt
- Address of first instruction is registered with the interrupt hardware by filling in an interrupt vector

Interrupts

- Interrupts are a form of event-based programming
  - Also known as reactive programming
- Interrupt handlers are invoked in response to hardware events
  - Invocation order is not known a priori
  - Their execution is interleaved with existing execution sequences, unless prevented by disabling interrupts

Can Applications Cheat the Timer?

Can the application disable the future timer interrupt so that the OS can not take control back from it?

Can Applications Cheat the Timer?

Enabling and disabling interrupts must by a privileged instructions that are not executable by applications

ISAs – Instruction Set Architecture – typical defines many privileged instructions that can only be used when system is in privileged mode.

The CPU knows whether or not to allow privileged instructions based on the value of the mode bit in the status register

Privileged instructions are only executed if the mode bit is set

- attempted execution in non-privileged mode generally causes an interrupt (trap) to occur

Can Applications Set the Mode Bit?

How is the mode bit set?

- Via a privileged instruction?
- If so, is this instruction privileged?

Can Applications Set the Mode Bit?

The mode bit is set via interrupts, traps, faults, and exceptions

These are all hardware events that cause the address of a previously registered instruction to be loaded into the PC

- This instruction is the start of an interrupt/trap/fault handler that is part of the OS!

Are There Other Ways to Cheat?

What stops the running application from modifying the interrupt vector, or the handlers associated with various interrupts, traps and faults?

- eg. modifying the timer interrupt handler to jump control back to the application?

What Stops Applications From Modifying the Interrupt Vector?

What Stops Applications From Modifying the Interrupt Vector?

Doing so requires privileged instructions!

What Stops Applications From Modifying the OS?

i.e, modifying the interrupt, trap, fault handler instructions in memory?

What Stops Applications From Modifying the OS?

Memory protection!

Certain areas of memory are off-limits unless the correct values are loaded into memory protection registers

- ... and accessing those registers requires use of privileged instructions
- ... they can only be changed if the mode bit is set

How Can the OS Maintain Control?

Why must the OS fill in the interrupt vectors before handing control over to applications

Why must it register a future timer interrupt?

Why must it set memory protections?

Why must it clear the mode bit?

What if it forgets to do one of the above?

How Can Applications Invoke the OS?

Why not just set PC to an OS instruction address and transfer control that way?

How would the mode bit get set?

How Can Applications Invoke the OS?

Special trap instruction causes a kind of interrupt

- changes PC to point to a predetermined OS entry point instruction
- simultaneously sets the mode bit
- CPU is now running in privileged mode

Application calls a library procedure that includes the appropriate trap instruction

ID of system call is passed in a register (`%eax`)

fetch/decode/execute cycle begins at a pre-specified OS entry point called a system call handler

Trap instruction

On Intel x86 chips, the “trap” instruction is called “int”

(grrrr….)

Are Traps Interrupts?

Traps, like interrupts, are hardware events

But traps are synchronous whereas interrupts are asynchronous

- traps are caused by the executing program rather than by a device that is external to the CPU

Interrupts, Traps, and Faults

	Synchronous?	Intentional?	Error?
Interrupt	No	No	No
Trap	Yes	Yes	No
Fault	Yes	No	Yes

Key Concepts: processes

- user mode vs. kernel mode
- Trap (Intel INT)
- System call
- Trap table
- Return from trap (Intel IRET)
- Limited Direct Execution
- Timer interrupt
- Context Switch

Review Questions

Why do we need a timer device?
Why do we need an interrupt mechanism?
Why do we need two instruction types and a mode bit?
Why are system calls different from procedure calls?
How are system calls different from interrupts?
Why is memory protection necessary?