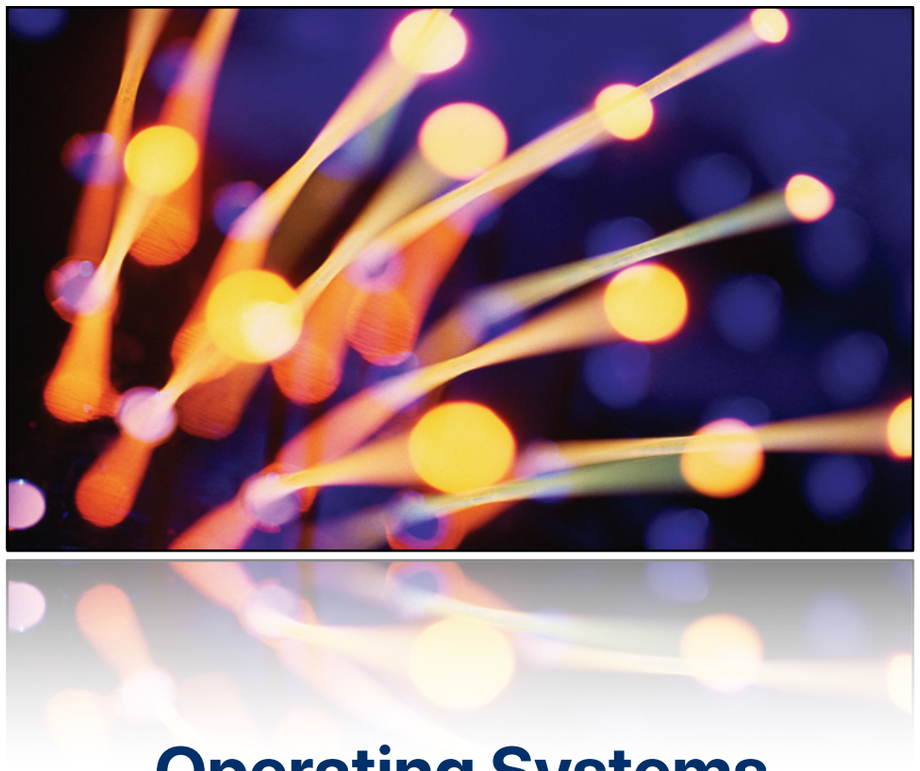
OS: a bird's-eye view



Operating Systems

Baochun Li
University of Toronto

Last time

A brief history of operating systems

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Last time

Operating system: a layer of software between applications and hardware

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Last time

The OS provides an API to the applications above, and manages shared resources

Reading

Required: Three Easy Pieces

Chapter 2: Introduction to Operating Systems

What is an operating system?

The application developer's (or user's) view: "top-down"

OS designed to provide an Application Programming Interface (API) to make using hardware resources easier (called *system calls*, or *syscalls*)

Hides details via good abstractions

The system's view: resource manager ("bottom-up")

OS manages possibly conflicting requests for resources, such as CPU cycles, memory, and storage

It virtualizes physical resources

OS as an API (a standard library)

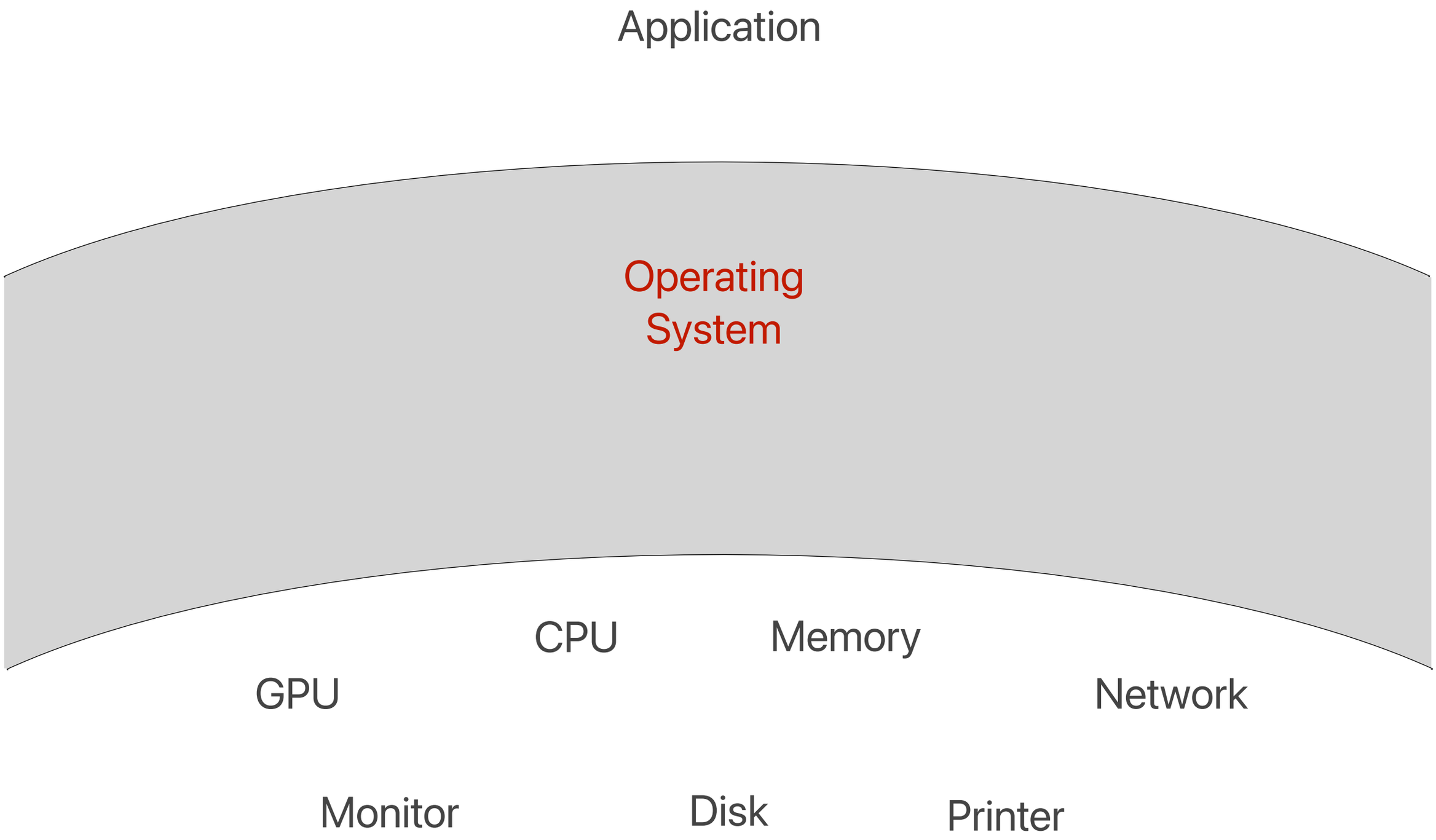
Why is such an abstraction important?

Otherwise, application writers must program all device accesses directly

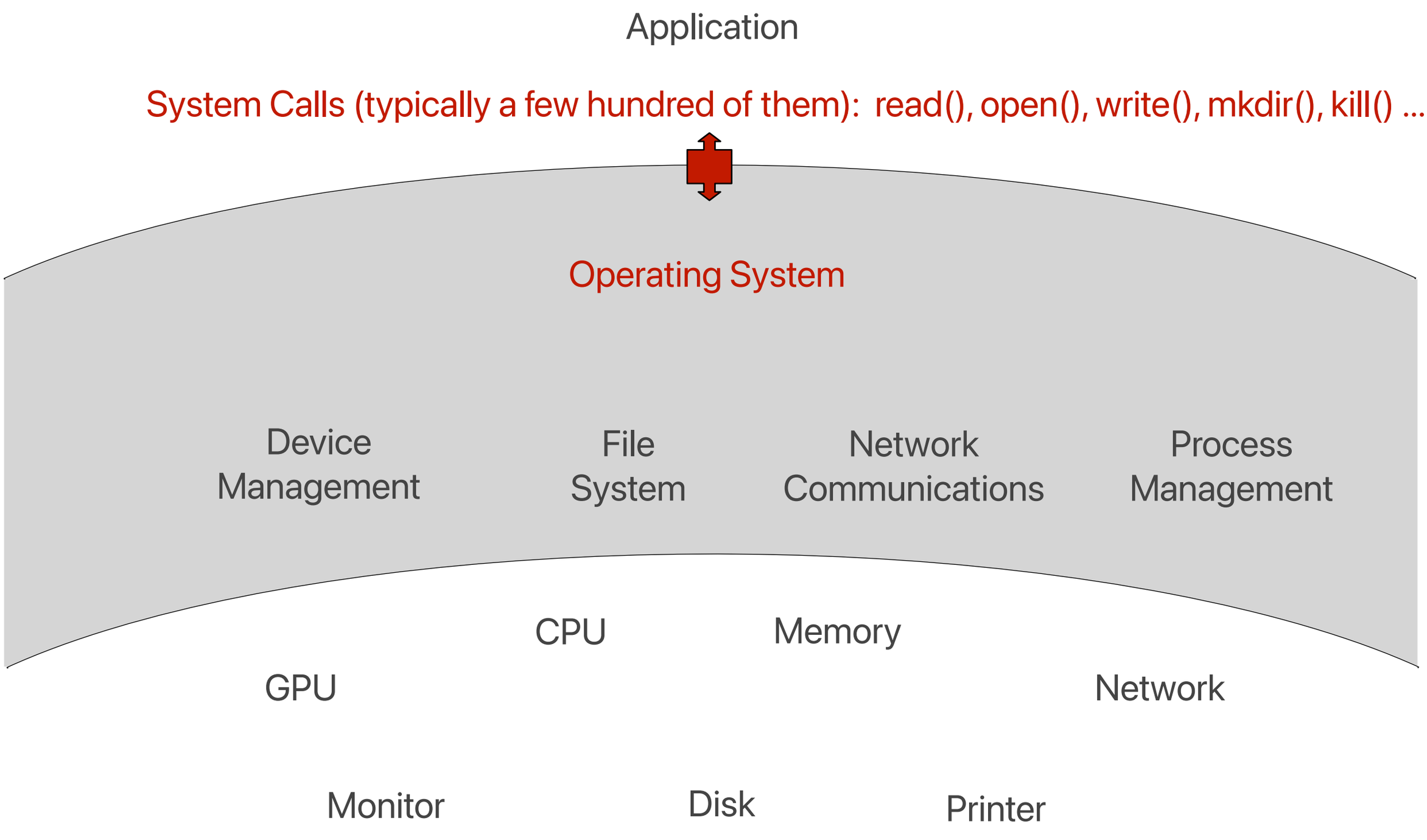
- Load device command codes into device registers
- Handle initialization and timing for physical devices
- Interpret return codes

Hard to maintain and upgrade code

Providing an API via system calls



Providing an API via system calls



OS as a resource manager

Shares resources across applications

- Sharing a resource (CPU) over time
- Sharing a resource (disk, memory) over space

Makes efficient use of a limited resource

- Improves utilization and performance
- Minimizes overhead

Protects applications from each other

- Enforces boundaries

The OS virtualizes resources — the OS takes a physical resource (CPU, memory, or a disk) and makes it easier to use.

Three “easy” pieces (main themes) in this course: virtualization, concurrency, and persistence

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Theme #1

Virtualization: The OS takes a physical resource (CPU and memory) and transforms it into an easy-to-use virtual form of itself

To the applications, the OS is a virtual machine

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

The big question: how does the OS virtualize resources? — mechanisms and policies

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Demo: Virtualizing the CPU

Theme #2

Concurrency: The host of problems that must be solved when working on many things concurrently in the same program

Demo: Concurrency with threads (functions that run in the same memory space as other functions)

Theme #3

Persistence: Storing data persistently using a file system

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Demo: I/O system calls

Ordering in this course

Virtualizing CPU
Concurrency
Virtualizing Memory
Persistence

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Up next

We start with a bird's eye view of what problems will arise and how they are solved when the OS virtualizes the CPU

Virtualizing the CPU

The OS needs to somehow share the physical CPU among many programs running seemingly at the same time

Basic idea: run one program for a little while, then switch to run another one, and so forth

Time-sharing the CPU — virtualization is achieved!

But we have to care about performance

To minimize the overhead of running a program, we use a technique called limited direct execution

Just run the program directly on the CPU!

Sounds quite simple

What may be a potential problem?

Two problems

If the OS directly runs the user program on the CPU, how can it make sure that the program doesn't do something it shouldn't be doing?

When a user program is running, how can the OS run? (It's just a software program itself!)

What we really need

The OS needs to be in control, yet we don’t want to compromise performance!

Getting started

Let’s start with a brief review of how programs are executed on the CPU

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

A CPU's instruction set

Different for different CPU architectures

All have load and store instructions for moving items between memory and registers

- Load a word located at an address in memory into a register
- Store the contents of a register to a word located at an address in memory

Many instructions for comparing and combining values in registers and putting result into a register

Basic anatomy on a CPU

Program Counter (PC)

Holds the memory address of the **next** instruction

Instruction Register (IR)

holds the instruction currently being executed

General Registers (R1 … Rn)

hold the environment of execution: temporary results

Arithmetic Logic Unit (ALU)

performs arithmetic functions and logic operations

Basic anatomy on a CPU

The Stack Pointer (SP)

holds the memory address of a stack, with a frame for each active function's parameters and local variables

The Program Status Word (PSW)

contains a few important control bits

Program execution

All the CPU does is Fetch/Decode/Execute

fetch next instruction pointed to by PC

decode it to find its type and operands

execute it

repeat

```
PC = <start address>
while (halt_flag not set)
    IR = memory[PC]
    decode_and_execute(IR)
    PC = PC + 1
```

The BLITZ architecture

The emulated CPU is a 32-bit architecture

16 general purpose integer registers with one word each

r0 -> r15

16 floating point registers with a double-precision floating point number each (64 bits)

f0 -> f15

69 different instructions

The Stack Pointer in BLITZ

Register r15 points to the top of the execution stack

The stack grows downwards from higher towards lower memory addresses

Two problems, revisited

If the OS directly runs the program on the CPU, how can it make sure that the program doesn't do something it shouldn't be doing?

When a program is running, how can the OS run? (It's just a software program itself!)

An intuitive solution

```
PC = <start address>
while (halt_flag not set)
    IR = memory[PC]
    if IR != special_instruction
        decode_and_execute(IR)
    else
        switch_to_the_OS_kernel()
    PC = PC + 1
```

The OS needs help from hardware to accomplish these tasks!

Processor Modes

CPUs have a mode bit in the PSW that defines the execution capability of a program

Kernel mode (mode bit **set**)

Executes any instruction

called “System mode” in the BLITZ documentation

User mode (mode bit **cleared**)

Executes a subset of instructions

Instructions that execute only in the kernel mode are called Privileged Instructions

What is an example of a privileged instruction?

I/O operations to the disk

The operation that sets the mode bit to 1!

Processor modes: a summary

OS operates in kernel mode

has all privileges to access devices and memory

Modules that run in kernel mode are called “the kernel”

Applications operate in user mode

Have limited privileges: limited access to memory, no access to I/O devices

Now the OS is in control

When applications need to run privileged instructions, they must enter the OS kernel

How can applications (user programs) enter the OS kernel?

Can an application just set the PC to the address of an OS instruction, and jump to that address?

How do we enforce that only the OS operates in the kernel mode?

How can applications enter the OS kernel?

Applications, running in user mode, need to perform a system call

To perform a system call, a user program needs to execute an instruction called a trap

The trap instruction

All it does is to jump into the kernel while simultaneously raising the privilege level to kernel mode

- the application calls a library procedure (in the standard C library) that includes the appropriate trap instruction
- fetch/decode/execute cycle then begins at a pre-specified OS kernel entry point for a system call
- CPU is now running in kernel mode

When the system call returns

We will need to return to the program making the system call

But at the same time, the mode bit will need to be cleared (reducing the privilege level back to user mode)

Again, the OS depends on some help from the CPU, by using another instruction, let's call it return-from-trap

What the hardware must do

When executing a trap, the hardware will need to make sure to save the caller's registers, the PC, and flags in the PSW

Because the hardware is responsible for returning correctly when the return-from-trap instruction is issued by the OS

One way to do this (Intel x86) is to save them onto a kernel stack

The return-from-trap instruction will pop these values off the stack and resume the execution of the program in the user mode

But there's one more important problem: How does the trap instruction know which code to run? (there are, after all, hundreds of system calls.)

Which code to run?

Can the calling program specify an address to jump to?

No.

The kernel must carefully control what code executes upon a trap.

Solution: The kernel sets up a trap table at boot time in kernel mode, and then let the hardware know where it is.

The trap table is also called the interrupt table, because it has entries for hardware interrupts and exceptions too!

It turns out that trap is only one of the entries in the trap table.

The actual type of the system call is called the syscall number, and it can be stored at a well-known location in the kernel stack.

Interrupts

Interrupts are asynchronous (comes in at any arbitrary time)

- Caused by hardware events
- Timer interrupts
- I/O (such as keyboard or disk) interrupts

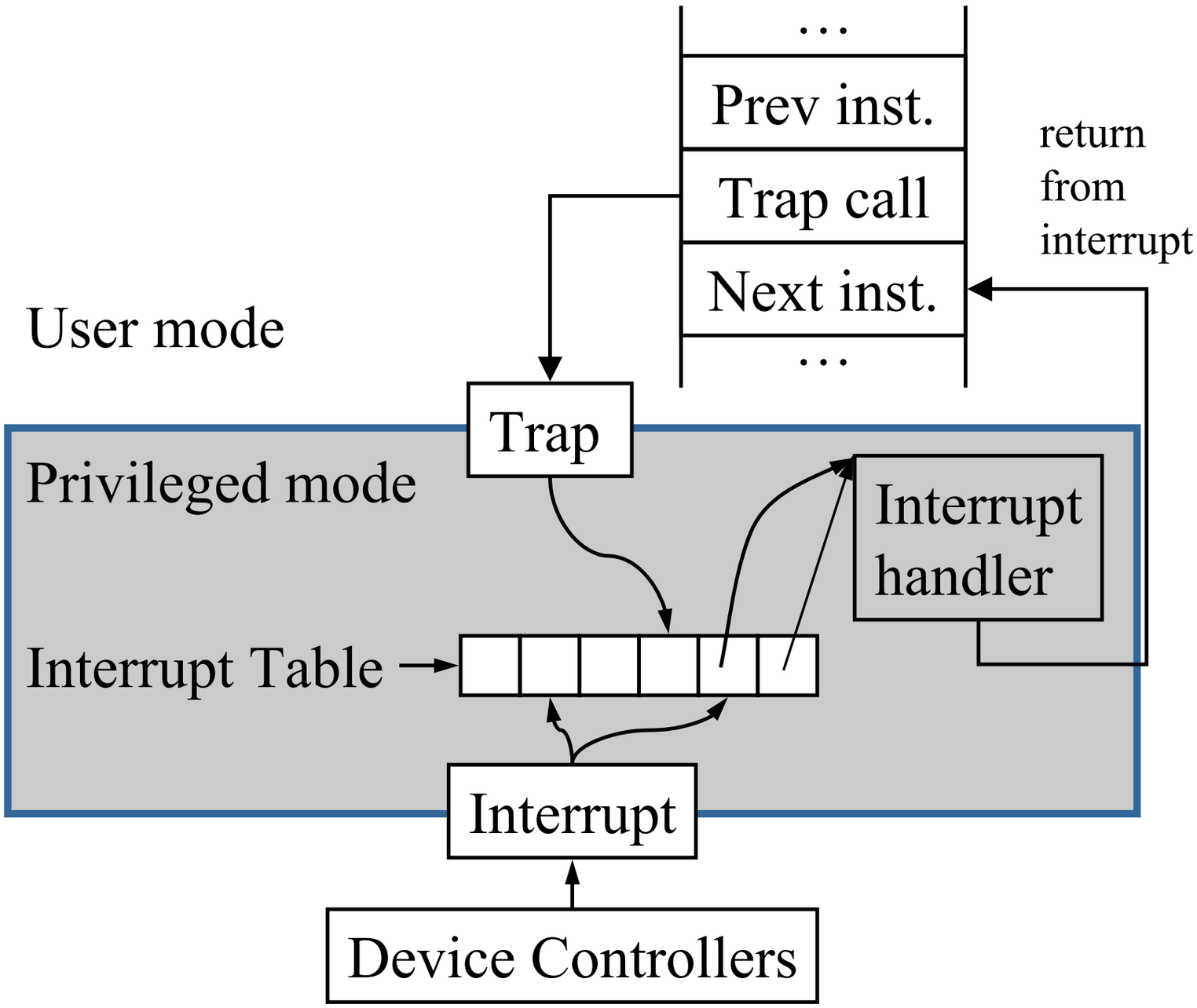
Exceptions

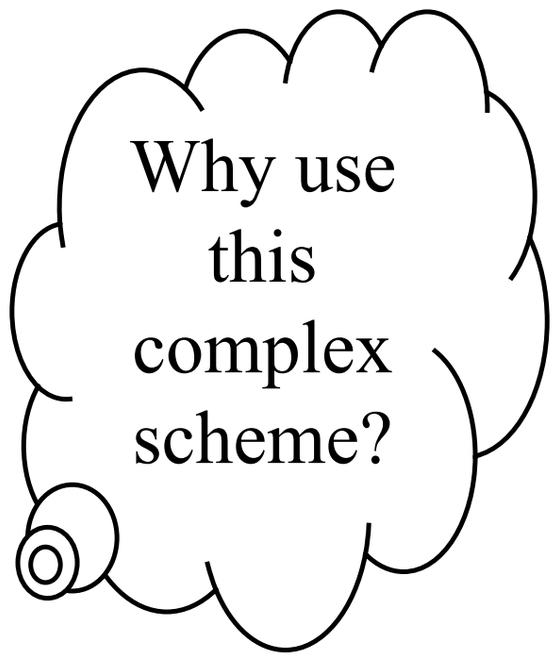
Caused by programming errors in a user-mode program

Examples

- an arithmetic exception (divide by zero)
- attempts to access memory that the program does not control
- attempts to execute a privileged instruction

The trap (interrupt) table: an illustration





Interrupt and Trap Processing in BLITZ ("The BLITZ Architecture," pages 22-25, 54)

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

The Program Status Word in BLITZ

Called the Status Register in BLITZ documentation

A special 32-bit register, but only 6 bits are used

Three condition codes (Z = Zero, V = Overflow, N = Negative) are set during certain arithmetic operations

Privileged bits (can only be changed by the kernel in the System Mode)

"I" bit: Interrupt Enabled (1) or Disabled (0)

"S" bit: System Mode (1) vs. User Mode (0)

"P" bit: Paging Enabled (1) or Disabled (0)

Interrupt Types in BLITZ

Asynchronous interrupts (hardware)

- Timer Interrupt
- Disk Interrupt

Synchronous interrupts (during execution)

- Exceptions
  - Privileged Instruction
  - Arithmetic
- Trap
  - **Syscall**

More about BLITZ registers

Two sets of integer registers: “system” (kernel) r0-r15 and “user” r0-r15

- When running in user mode, the user r0-r15 is used
- When running in kernel mode, the system r0-r15 is used
- In kernel mode, the kernel can read/write user registers

The use of two sets of integer registers allows kernel traps (such as system calls) to be executed quickly

- No registers need to be saved when trapping to the kernel
- But only one set of floating-point registers that are shared

When an interrupt occurs and is serviced

The current executing instruction is finished

- PC points to the next instruction for hardware interrupts
- PC points to the offending instruction for exceptions
- PC points to the instruction following the “syscall” trap for system calls

An “Exception Info Word” is pushed onto the system stack (system register r15 used)

- Syscall trap: system call number
- All zeros for most other types of interrupts

Status Register (PSW) is pushed onto the system stack

Program Counter (PC) is pushed onto the system stack

When a trap, interrupt, or exception occurs

Status Register changed

Switch to System Mode

Disable subsequent interrupts

Disable paging (address translation via the page table)

PC is loaded with the corresponding address of one of the interrupt vector entries

The entry contains a **jmp** instruction

which transfers control (again) to the interrupt handler

14 entries in the interrupt vector, in low memory

56 bytes in total

Some interrupts can be masked, some cannot

Don’t believe me?

Read blitz.c, singleStep() function, line 4997-6728

Returning from a trap (interrupt) handler

The **"Return from Interrupt"** **reti** instruction in BLITZ

First, restore the **PC** by popping the top value from the system stack and move it into the PC

Second, restore the **Status Register** to the old value, by popping the next value from the system stack into the Status Register

Third, pop and discard the next value from the system stack (the "**Exception Info Word**")

Instruction execution resumes in the interrupted program, as if nothing happened

None of the user registers are affected in an interrupted user program

The trap table (called Interrupt Vector in BLITZ)

```
Interrupt Vector in Low Memory
    Address  Description
================================
    000000   Power On Reset
    000004   Timer Interrupt
    000008   Disk Interrupt
    00000C   Serial Interrupt
    000010   Hardware Fault
    000014   Illegal Instruction
    000018   Arithmetic Exception
    00001C   Address Exception
    000020   Page Invalid Exception
    000024   Page Readonly Exception
    000028   Privileged Instruction
    00002C   Alignment Exception
    000030   Exception During Interrupt
    000034   Syscall Trap
```

Runtime.s in Lab 1

```
PowerOnReset:
        jmp     RuntimeStartup
TimerInterrupt:
        jmp     TimerInterruptHandler
DiskInterrupt:
        jmp     DiskInterruptHandler
SerialInterrupt:
        jmp     SerialInterruptHandler
HardwareFault:
        jmp     HardwareFaultHandler
IllegalInstruction:...
        jmp     IllegalInstructionHandler
ArithmeticException:
        jmp     ArithmeticExceptionHandler
AddressException:
        jmp     AddressExceptionHandler
PageInvalidException:
        jmp     PageInvalidExceptionHandler
PageReadonlyException:
        jmp     PageReadonlyExceptionHandler
PrivilegedInstruction:
        jmp     PrivilegedInstructionHandler
AlignmentException:
        jmp     AlignmentExceptionHandler
ExceptionDuringInterrupt:
        jmp     ExceptionDuringInterruptHandler
SyscallTrap:
        jmp     SyscallTrapHandler
```

With all these mechanisms, we still haven’t solved the second problem yet.

When a program is running, how can the OS run? (It's just a software program itself!)

Can we trust the programs to behave reasonably, and just wait for a "yield" system call?

yield(): transfer control to the OS

We call this a “cooperative” approach — used by the original MacOS and Windows.

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

This is, obviously, not ideal — a malicious or buggy program can take over the CPU indefinitely.

How does the OS gain control of the CPU without cooperation?

Baochun Li, Department of Electrical and Computer Engineering, University of Toronto

Again, the OS depends on help from the hardware.

The solution:
timer interrupts

What We've Covered So Far

Required reading in the textbook: Chapter 6: (Mechanism: Limited Direct Execution) 6.1, 6.2, 6.3 (before "Saving and Restoring Context")

Required reading in BLITZ:

An Overview of the BLITZ System (7 pages)

An Overview of the BLITZ Computer Hardware (8 pages)

The BLITZ Architecture (Page 1-9, 22-25, 54)

Optional reading:

Chapter 1 and 2 in Operating Systems Concepts

The remaining pages of "The BLITZ Architecture"

What We've Covered So Far

Required reading in the textbook: Chapter 6: (Mechanism: Limited Direct Execution) 6.1, 6.2, 6.3 (before "Saving and Restoring Context")

Required reading in BLITZ:

An Overview of the BLITZ System (7 pages)

An Overview of the BLITZ Computer Hardware (8 pages)

The BLITZ Architecture (Page 1-9, 22-25, 54)

Optional reading:

Chapter 1 and 2 in Operating Systems Concepts

The remaining pages of "The BLITZ Architecture"