CS 370: OPERATING SYSTEMS [INTRODUCTION]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Lecture slides: where are they? http://www.cs.colostate.edu/~cs370
  - Where is the schedule of topics?
- Term project: Complexity, requirements, etc.
- Kernel vs the OS
  - What is a kernel? What does it do? What does the rest of the OS do?
- How does the kernel create this illusion of multitasking?
- User-mode/kernel mode
- Can you go deeper into memory management?
- Fault isolation?
- Is Unix/Linux better or Windows?

Frequently asked questions from the previous class survey

- Tests
  - Will the tests have concepts not taught in class … e.g. from textbook? NO!
  - How do I study for the quizzes/exams?

Topics covered in this lecture

- Caches and main memory
- Secondary storage
- Relative speeds of the memory hierarchy
- The Kernel Abstraction
- Buses

Memory hierarchy: Cache memory

- Mostly *controlled by hardware*
- Main memory divvied up into **cache lines**
  - Usually 64 bytes
  - Addresses 0-63 in cache line 1, 64-127 in cache line 2, and so on
- Most heavily used cache lines are stored in high-speed cache **close** to the CPU

When a program needs to read a memory word

- Cache hardware checks if the needed line is in the cache
- If it is, that's a **cache hit**
  - Request satisfied from cache in about *2 clock cycles*
  - No memory access needed
- If needed line is not present in cache
  - **Cache miss**, and must access memory
  - **Substantial** time penalty

Caching is a powerful concept used elsewhere too. Let's see when …

① Large resource *can be divided* into pieces

② Some pieces *used more heavily* than others

- OS caching examples:
  - Pieces of heavily used files in main memory
    - Reduce disk accesses
  - Conversion of file names to disk addresses
  - Addresses of Web pages (URLs) as hosts

CPUs usually have a couple of caches

- **L1 cache** is inside the CPU
  - Typically in the order of 16 KB
  - No access delay
- **L2 cache** holds several MB of data
  - Access delay of 1-2 clock cycles

Main Memory

- Usually called **RAM** (Random Access Memory)
- Cache misses go to the main memory
- **Volatile**
  - Contents lost when power is turned off
- Memory size is of the order of several GB in most modern desktops

Computers run most of their programs from (rewriteable) main memory

- Typically implemented in a technology called DRAM (dynamic random access memory)
- Ideal Scenario: Programs and data reside permanently in main memory. BUT …
  - Space is *limited*
  - Main memory is *volatile* storage

Secondary storage is needed to hold large quantities of data permanently

- Programs use the disk as the source and destination of processing
- Seek time 7 ms
- SPIN: 7200 – 15000 RPM
- Transfer rate
  - Disk-to-buffer: 70 MB/sec (SATA)
  - Buffer-to-Computer: 300 MB/sec
- Mean time between failures
  - 600,000 hours
- 1 TB capacity for less than $100

Improvements in hard disk capacity

- 1980 - 5 MB
- 1991 - 100 MB
- 1995 - 2 GB
- 1997 - 10 GB

Improvements in hard disk capacity

- 2002 - 128 GB addressing space barrier [28 bits]
  - Old IDE/ATA interface: 28-bit addressing
  - $2^{28} \times 512 = 2^{28} \times 2^{9} = 2^{37} = 128$ GB = 137,438,953,472 bytes
- 2003 – Serial ATA introduced
  - Bus interface providing connections to mass storage devices
- 2005 - 500 GB hard drives
- 2008 - 1 TB hard drives

Characteristics of peripheral devices & their speed relative to the CPU

Item	time	Scaled time in human terms (2 billion times slower)
Processor cycle	0.5 ns (2 GHz)	1 second
Cache access	1 ns (1 GHz)	2 seconds
Memory access	70 ns	140 seconds
Context switch	5,000 ns (5 μs)	167 minutes
Disk access	7,000,000 ns (7 ms)	162 days
Quantum	100,000,000 ns (100 ms)	6.3 years

Mechanical nature of disks limits their performance

- Disk access times *have not* decreased exponentially
  - Processor speeds are growing exponentially
- Disparity between processor and disk access times continues to grow
  - 1:14,000,000

RELATIVE SPEEDS OF THE MEMORY HIERARCHY

Since caches have limited size, cache management is critical

Level	**1**	**2**	**3**	**4**
Name	registers	cache	Main memory	Disk Storage
Typical Size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation Technology	Custom memory, CMOS	On/off chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access times	0.25 ns	0.5-25 ns	80-250 ns	> 5 ms
Bandwidth (MB/sec)	20,000 – 100,000	5000-10,000	1000-5000	80-300
Managed by	compiler	hardware	OS	OS
Backed by	cache	Main memory	Disk	CD/Tape

ONTOGENY RECAPITULATES PHYLOGENY

After Charles Darwin's book ON THE ORIGIN OF SPECIES was published

- German zoologist Ernst Haeckl stated
  - Ontogeny recapitulates Phylogeny

Development of an embryo *repeats* *the evolution of the species*

- i.e. human egg goes through stages of being a fish, … , before becoming human baby
- Modern biologists think this is a gross simplification!

Something vaguely similar has happened in the computer industry

- Each new species (*type of computer*) goes through the development its ancestors did
  - Both in hardware and software
  - Mainframe, mini computers, PC, handheld, etc

Much of what happens in computing and other fields is technology driven

- Ancient Romans lacked cars not because they liked walking
  - It is because they didn't know to build cars
- PCs exist not because people have a centuries-old pent-up desire to own one
  - It is now possible to manufacture them cheaply

Technology affects our view of systems

- A **change** in technology renders some idea *obsolete*
  - Another change could *revive* it
- Especially true when change has to do with **relative performance**
  - Of different parts of the system

Let’s look at this relative performance

- When CPUs become faster than memories?
  - Caches become important to speed-up slow memory
- If new memory technology makes memories much faster than CPUs?
  - Caches will vanish!
- In biology extinction is forever
  - In computer science, it is sometimes only for a few years

Historical developments Large Memories

- IBM 7090/7094 1959-1964
  - 128 KB of memory
  - Programmed in assembly language (even the OS)
  - With time FORTRAN/COBOL and assembly was dead
- PDP-1 had only 4096 18-bit words of memory
  - Assembly is back!
  - Over time memory increases, assembly is out
- Microcomputers in 1980s
  - 4 KB memory and assembly is back again

Other places where such a cycle has gone on?

- Protection hardware
- Disks
- Virtual memory

- What may seem dated ideas on PCs
  - May soon come back on embedded computers

PERFORMANCE

There are two approaches to improving performance

- Determine component **bottlenecks**
  - Replicate
  - Improve

To replicate or improve?

"*If one ox could not do the job, they* [pioneers] *did not grow a bigger ox, but used two oxen.*"

-- Admiral Grace Murray Hopper
Computer Software pioneer

"*If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?*"

-- Seymour Cray
Computer Hardware pioneer

Multiprocessor systems have 2-or-more processors in close communications

- The processors **share** the bus, and *may* share clock, memory and peripheral devices
- Advantages:
  - Increased throughput
  - Reliability

Multiprocessor systems fall in two categories based on control

- Asymmetric multiprocessing:
  - Controller processor manages the system
  - Workers **rely on controller** for instructions
- Symmetric multiprocessing
  - Processors are **peers** and perform all OS tasks
  - Have own set of registers and local cache
    - Share physical memory
  - **Supported by virtually all modern OS**

Recent trend has been towards adding multiple cores

- *Raison d'être*
  - On chip communications are much faster
  - Uses less power than multiple single-core chips
  - Cope with heat dissipations
  - Improve Thread level parallelism
- Number of cores doubling every year
  - Each core also gets more execution pipelines
  - Gartner Projection: 1024 cores soon!
- Challenge: Re-engineering programs daunting

Good fences make good neighbors
17th century proverb

THE KERNEL ABSTRACTION

A central role of the OS is **protection** — the isolation of potentially misbehaving applications and users

- Implementing protection is the job of the OS **kernel**
- The kernel has full access to all of the machine hardware
  - The lowest level of software running on the system
  - Necessarily trusted to do anything with the hardware
- Everything other than the kernel — that is, the untrusted software running on the system — is run in a restricted environment
  - Less than complete access to the full power of the hardware

What hardware is needed to let the kernel provide isolation?

- At a minimum, the hardware must support **three** things:
- **Privileged Instructions**: All potentially unsafe instructions are prohibited when executing in user mode
- **Memory Protection**: All memory accesses outside of a process's valid memory region are prohibited when executing in user mode
- **Timer Interrupts**: Regardless of what the process does, the kernel must have a way to periodically regain control from the current process

Conceptually, the kernel/user mode is a one-bit register

- When set to 1, the processor is in kernel mode and can do anything
- When set to 0, the processor is in user mode and is restricted

Privileged Instructions

- Instructions available in kernel mode, but not in user mode, are called **privileged instructions**
- To do its work, the kernel must be able to execute these instructions
  - Change privilege levels, adjust memory access, and disable and enable interrupts, set/reset timers
- If these instructions were available to applications?
  - A rogue application would in effect have the power of the kernel

Making memory sharing safe

- The kernel must be able to configure the hardware so that each application process can read and write **only its own memory**
  - Not the memory of the operating system or any other application
- While it might seem that read-only access to memory is harmless, the OS needs to provide both security and privacy.
  - For example, user passwords may be stored in kernel memory while they are being verified

Hardware timers

- Timers can be set to **interrupt** the processor after a specified delay
  - Either in time or after some number of instructions have been executed
- Each timer interrupts one processor … separate timer for each CPU
  - The kernel might set each timer to expire every few milliseconds
- Resetting the timer is a privileged operation
  - User-level process cannot inadvertently or maliciously disable the timer
- How does the kernel know if an application is in an infinite loop?
  - It doesn't!

Mode transitions

- The kernel places a user process in a carefully constructed sandbox
  - The next question is how to safely transition from executing a user process to executing the kernel, and vice versa
- These transitions are **not rare events**
  - E.g.: A web server might switch between user mode and kernel mode thousands of times per second
- Transitions must be both fast and safe

There are three reasons for the kernel to take control from a user process

- Reasons: interrupts, processor exceptions, and system calls
- Asynchronous events
  - **Interrupts** are triggered by an *external event* and can cause a transfer to kernel mode after any user-mode instruction
- Synchronous events
  - **Processor exceptions** and **system calls** are triggered by *process execution*
  - The term **trap** refers to any synchronous transfer of control from user mode to the kernel

Interrupts are also used to inform the kernel of the completion of I/O requests

- Mouse device hardware triggers an interrupt every time the user moves or clicks on the mouse
  - The kernel, in turn, notifies the appropriate user process — the one the user was "mousing" across
- Virtually **every I/O device generates an interrupt** whenever some input arrives for the processor and whenever a request completes
  - E.g.: the Ethernet, WiFi, hard disk, thumb drive, keyboard, mouse, etc.

As the processor executes instructions, it checks for whether an interrupt has arrived

- If so, it completes or stalls any instructions that are in progress
  - Instead of fetching the next instruction, the processor hardware saves the current execution state
  - Starts executing at a specially designated interrupt handler in the kernel

Processor exceptions

- A processor exception is a **hardware event** caused by user program behavior that causes a transfer of control to the kernel
- A processor exception occurs whenever a process
  - Attempts to perform a privileged instruction]
  - Accesses memory outside of its own memory region
  - Causes an arithmetic overflow. E.g. divide-by-zero
  - Accesses a word of memory with a non-aligned address
  - Attempts to write to read-only memory

User processes can also transition into the kernel voluntarily

- To request that the kernel perform an operation on the user's behalf
- A **system call** is any procedure provided by the kernel that can be called from user level
  - Examples include system calls to establish a connection to a web server, to send or receive packets over the network, to create or delete files, to read or write data into files, and to create a new user process.

To protect the kernel from misbehaving user programs

- It is key that the hardware transfers control on a system call to a pre-defined address
  - User processes cannot be allowed to jump to arbitrary places in the kernel
- The kernel handles the details of:
  - Checking and copying arguments
  - Performing the operation, and
  - Copying return values back into the process's memory

System calls provide the illusion that the kernel is simply a set of library routines available to users

- Implementing system calls requires the operating system to define a **calling convention**
- Once the arguments are in the correct format, the user-level program can issue a system call by executing the trap instruction to transfer control to the kernel
- The kernel implement its system calls in a way that protects itself from all errors and attacks that might be launched
  - Extreme version of defensive programming: always assume that system call parameters are intentionally designed to be as malicious as possible.

When an interrupt, processor exception or system call trap occurs …

- How does the processor know what code to run?
- The processor has a special register that points to an area of kernel memory called the **interrupt vector table**
- The hardware determines which device caused the interrupt, if the trap instruction was executed, or what exception condition occurred
  - Thus, the hardware can select the right entry from the interrupt vector table and invoke the appropriate handler
- The format of the interrupt vector table is processor-specific

The interrupt vector table on the x86

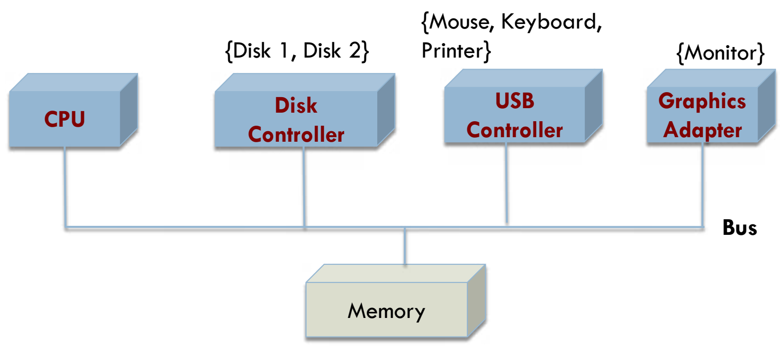
- Entries 0 – 31: are for different types of processor exceptions
  - E.g: divide-by-zero anything related to arithmetic overflow
- Entries 32 – 255 are for different types of interrupts
  - Timer, keyboard, etc.
  - By convention, entry 64 points to the system call trap handler

What about kernel to user mode transitions?

- New process
- Resume after an interrupt, processor exception, or system call
- Switch to a different process
- User-level upcall
  - Most OS provide user programs with the ability to receive asynchronous notification of events

BUSES

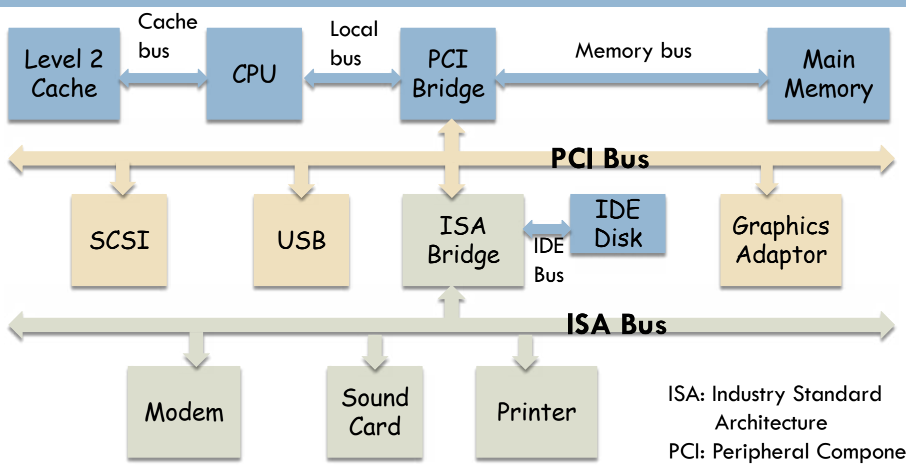
A simple bus-based structure



Limitations of the bus structure from the earlier slide

- As processors and memories got faster
  - Ability of a single bus to handle *all traffic* strained considerably
- Result?
  - Additional buses were added
  - For faster I/O devices and CPU-memory traffic

What a modern bus architecture looks like



ISA: Industry Standard Architecture
PCI: Peripheral Component Interconnect

There are two main BUS standards

- Original IBM PC ISA (Industry Standard Architecture)
- PCI (Peripheral Component Interconnect)
  - From Intel

The IBM PC ISA bus

- Runs at 8.33 MHz
- Transfers 2 bytes at once
- Maximum speed = 16.67 MB/sec
- Included for backward compatibility
  - Older and slower I/O cards

The PCI bus

- Can run at 66 MHz
- Transfer 8 bytes at once
- Data transfer rate: 528 MB/sec
- Most high-speed I/O devices use PCI
- Newer computers have an updated version of PCI
  - **PCI Express**

Other specialized buses: IDE (Integrated Drive Electronics) bus

- For attaching peripheral devices
  - CD-ROMs and Disks
- Grew out of the disk controller interface

Other specialized buses: USB (Universal Serial Bus)

- Attach **slow** I/O devices to the computer
  - Keyboard, mouse etc
- Uses a small **4-wire** connector
  - **Two** supply electrical power to the USB devices
- Centralized bus
  - Root device polls I/O devices every millisecond
    - Check if they have any traffic

Some more information about USB

- All USB devices share a **single** USB device driver
  - *No need to install* a driver for each device
  - Can be added to computer *without need to reboot*
- USB 1.0 has a transfer rate of 1.5 MB/sec
- USB 2.0 goes up to 60 MB/sec
- USB 3.0
  - Specification ready on 17 November 2008
  - Theoretical signaling rate: 600 MB/sec (4.8 Gbps)
  - USB 3.1: Jan 2013 goes to 10 Gbps
  - US 3.2 released in September 2017 transfer modes 10 and 20 Gbps

Other buses

- SCSI (Small Computer System Interface)
  - High performance bus
  - For devices that need high bandwidth
    - Fast disks, scanners
  - Up to 320 MB/sec
- IEEE 1394
  - Sometimes called FireWire (used by Apple)
  - Transfer speeds of up to 100 MB/sec
    - Camcorders and similar multimedia devices
  - No need for a central controller (unlike USB)

The contents of this slide-set are based on the following references

- *Thomas Anderson and Michael Dahlin. Operating Systems: Principles and Practice, 2nd Edition. Recursive Books. ISBN: 0985673524/978-0985673529. [Chapter 2]*
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 1]*
- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 1, 2]
- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2.* [Chapter 1]

In this setting the OS must know which devices are connected & how to configure them

- Led Intel and Microsoft to design **plug-and-play**
  - Similar concept had been implemented in the Mac

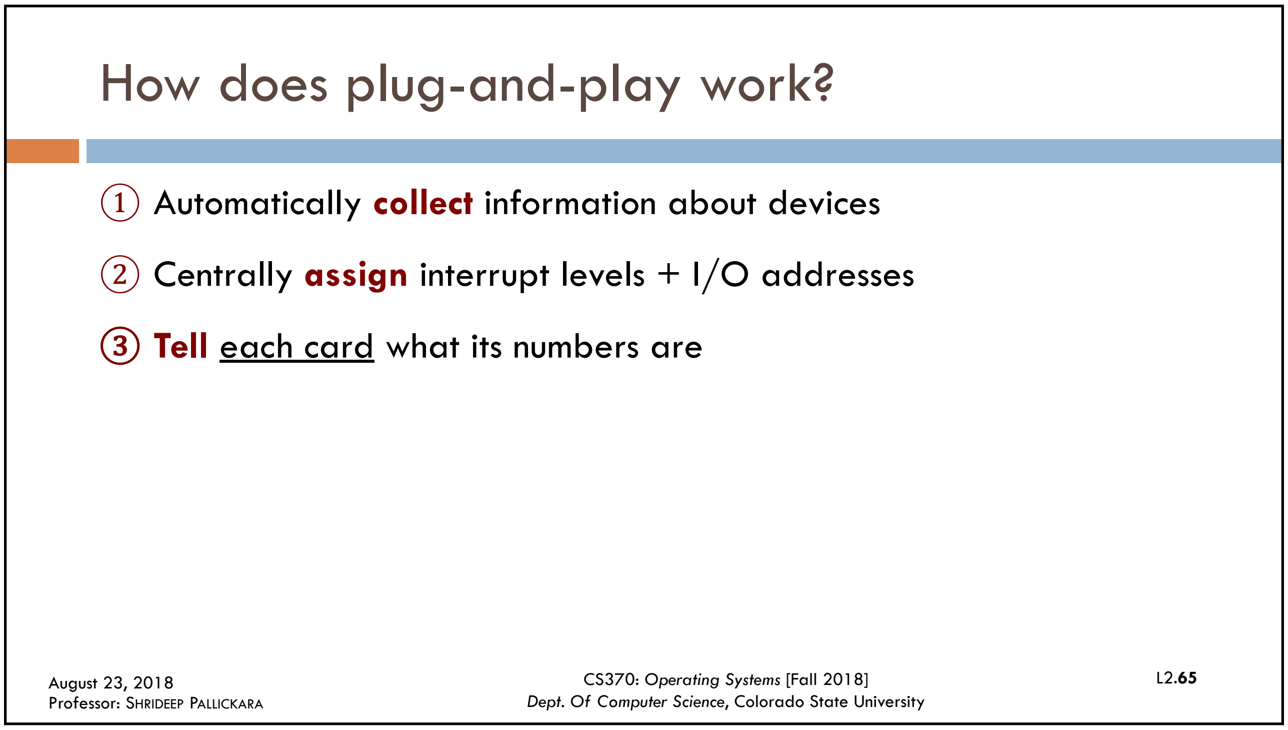
How things were before plug-and-play

- Each I/O card had a **fixed interrupt level**
  - Fixed addresses for its I/O registers

Device	Interrupt/I/O addresses
Keyboards	Interrupt 1, I/O addresses: 0x60-0x64
Floppy disk controller	Interrupt 6, I/O addresses: 0x3F0-0x3F7
Printer	Interrupt 7, I/O addresses: 0x378-0x37A

How things were before plug-and-play

- What if someone bought a sound card and a modem which happened to use interrupt 4?
  - Conflict
  - Would not work together
- Solution:
  - Use DIP (dual in-line package) switches or jumpers on every I/O card
  - Ask user to select interrupt level and I/O device addresses for the device
  - Tedious!



How does plug-and-play work?

① Automatically **collect** information about devices

② Centrally **assign** interrupt levels + I/O addresses

③ **Tell** <u>each card</u> what its numbers are

August 23, 2018
Professor: SHRIDEEP PALLICKARA

CS370: *Operating Systems* [Fall 2018]
Dept. Of Computer Science, Colorado State University

L2.**65**