OS Evolution and Architecture

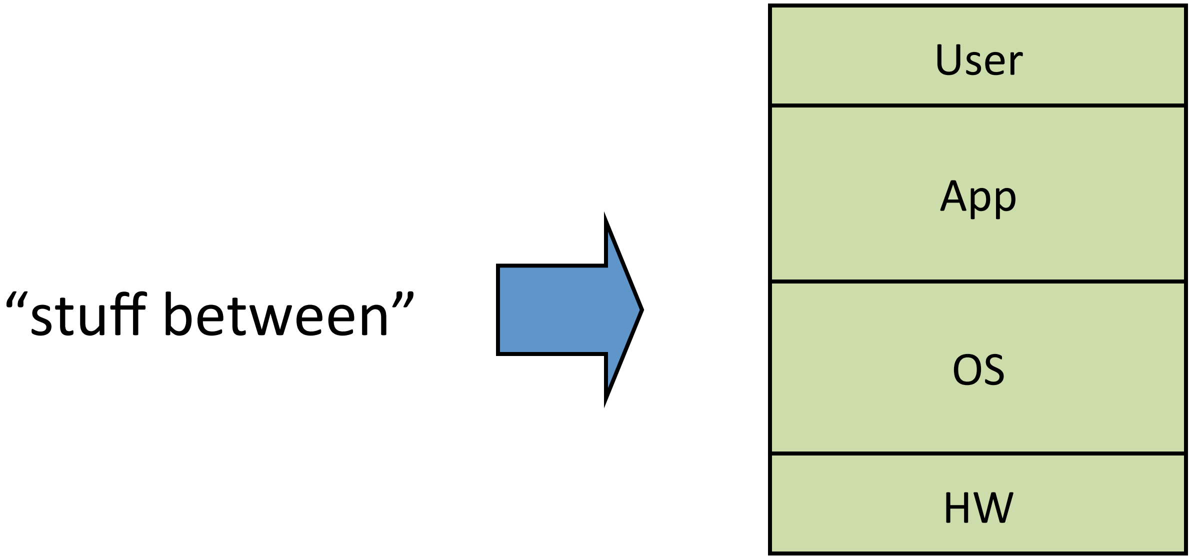
COMS W4118
Prof. Kaustubh R. Joshi
krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

What is an OS?

- “A program that acts as an intermediary between a user of a computer and the computer hardware.”



“stuff between”

User

App

OS

HW

Two popular definitions

- Bottom-up perspective: resource manager/ coordinator, manage your computer's resources
- Top-down perspective: hardware abstraction layer, turn hardware into something that applications can use

Outline

- Architecture review
- OS evolution
- Modern OS structures
- Modern OS abstractions

What does hardware provide?

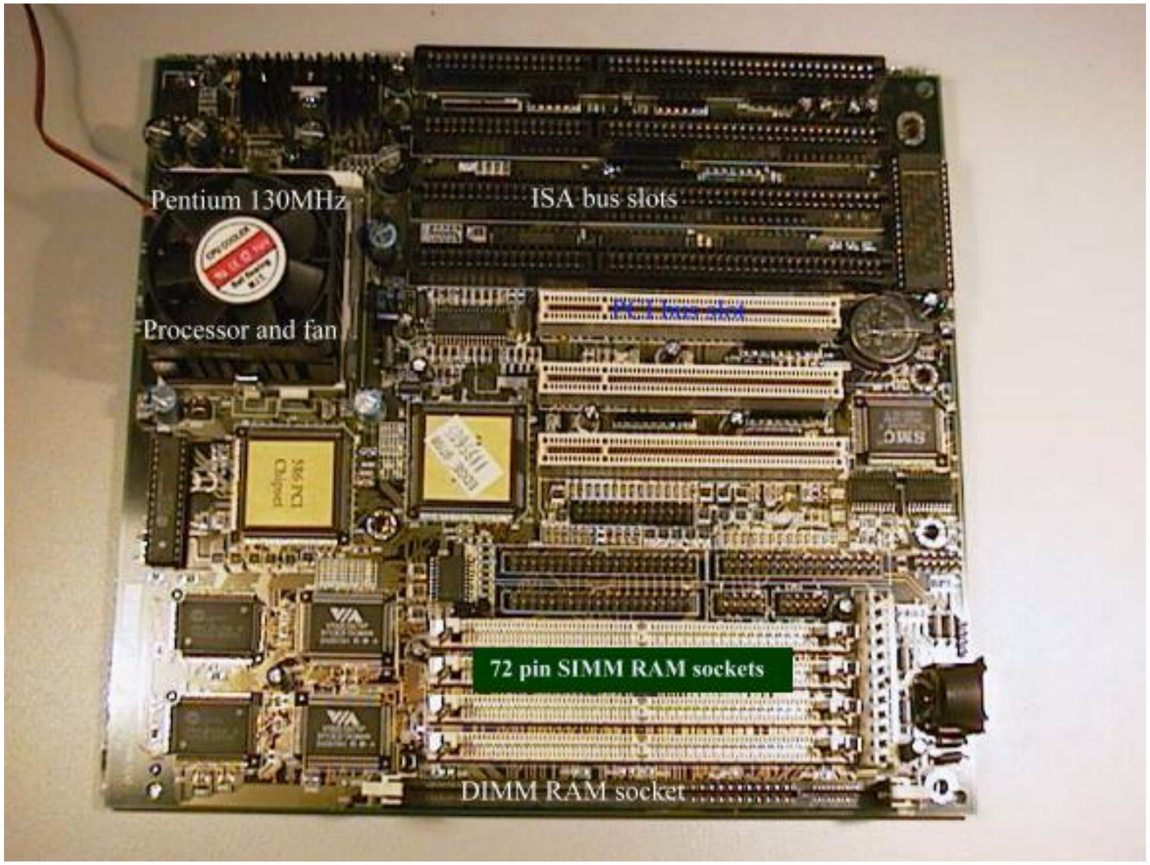
- Seen a glimpse of the functions OS provides
- But what hardware does it have to work with to provide those functions?
- Lets take a high level overview of how a typical computer system looks inside
- Different platforms have different chips: phone, PC, your DVD player, etc.
- But major concepts are the same

The x86 Platform

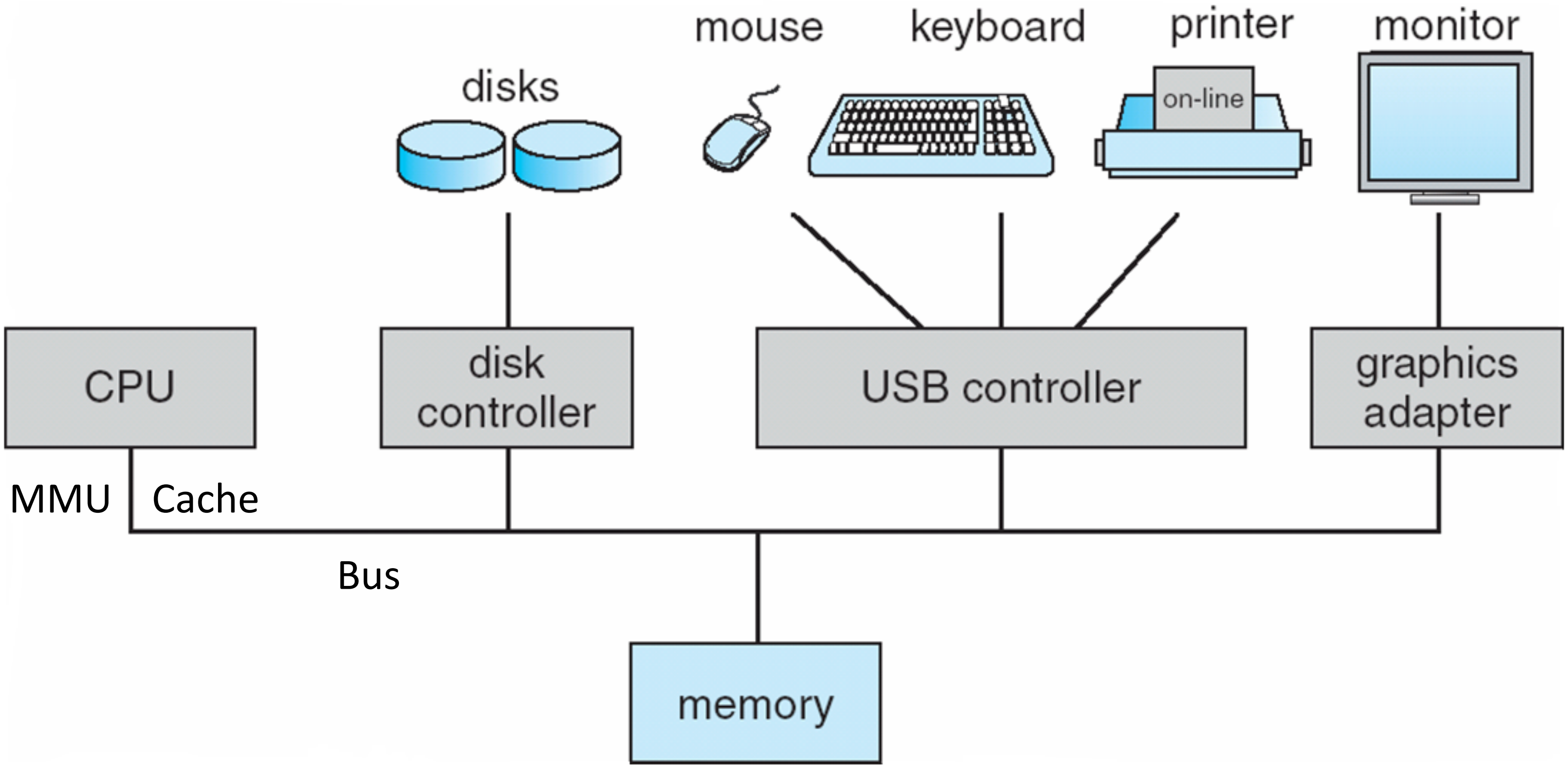
- Many processor architectures: Intel x86, ARM, Oracle Sparc, IBM Power, etc.
- We'll use x86 as an exemplar
  - Our Android emulator uses an x86 Atom image
  - Familiarity. Lots of online resources.
- Applies to most other architectures as well (with small differences)
- Besides, most of the OS code we will encounter is architecture agnostic

Ever assembled a PC?

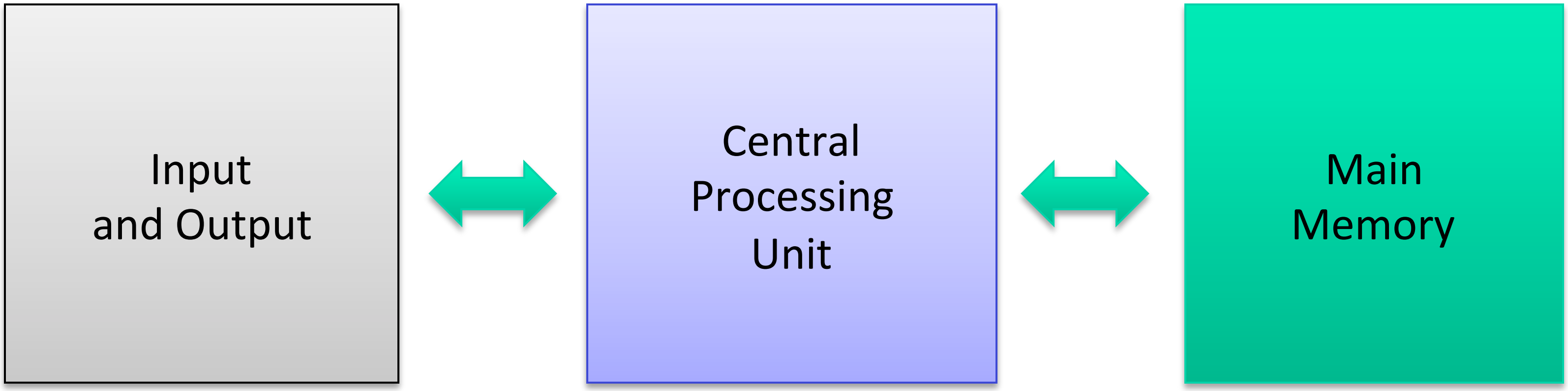
- One or more CPUs, memory, and device controllers connected through system bus



Computer organization

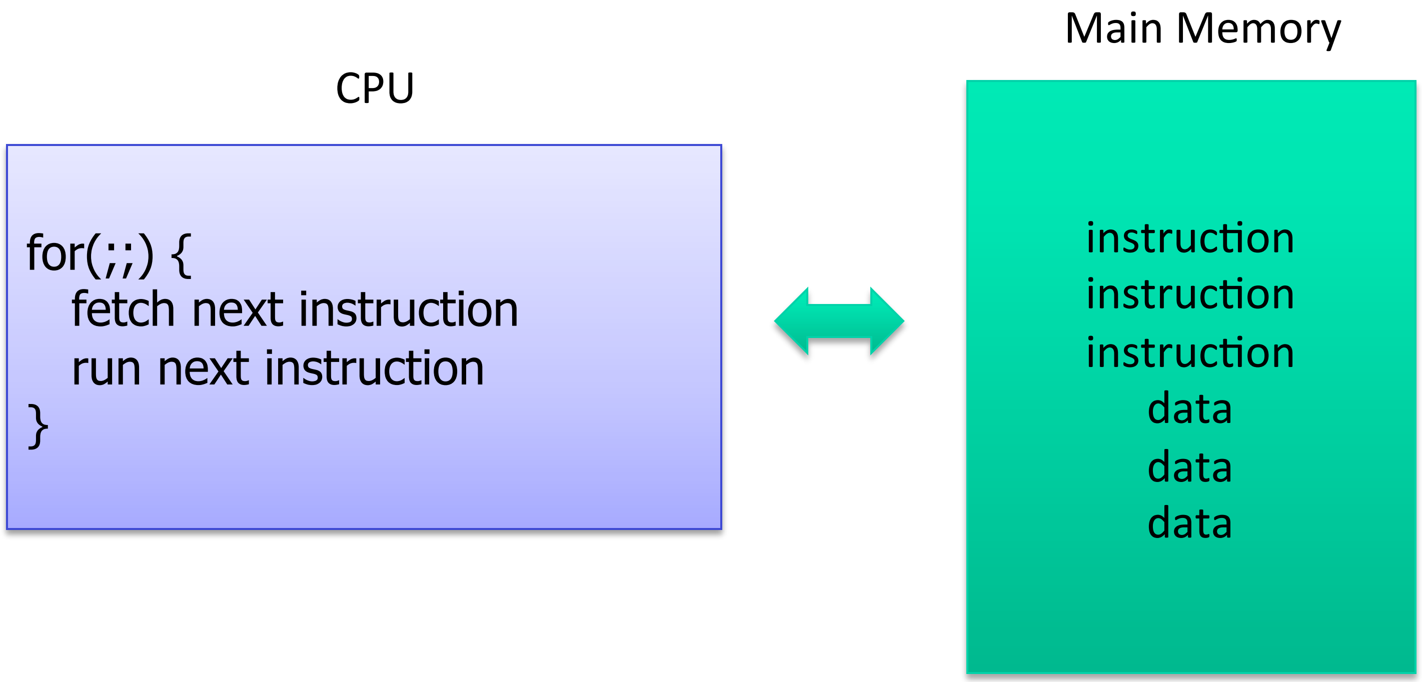


Abstract model



- I/O: communicating data to and from devices
- CPU: digital logic for doing computation
- Memory: N words of B bits

The stored program computer



CPU

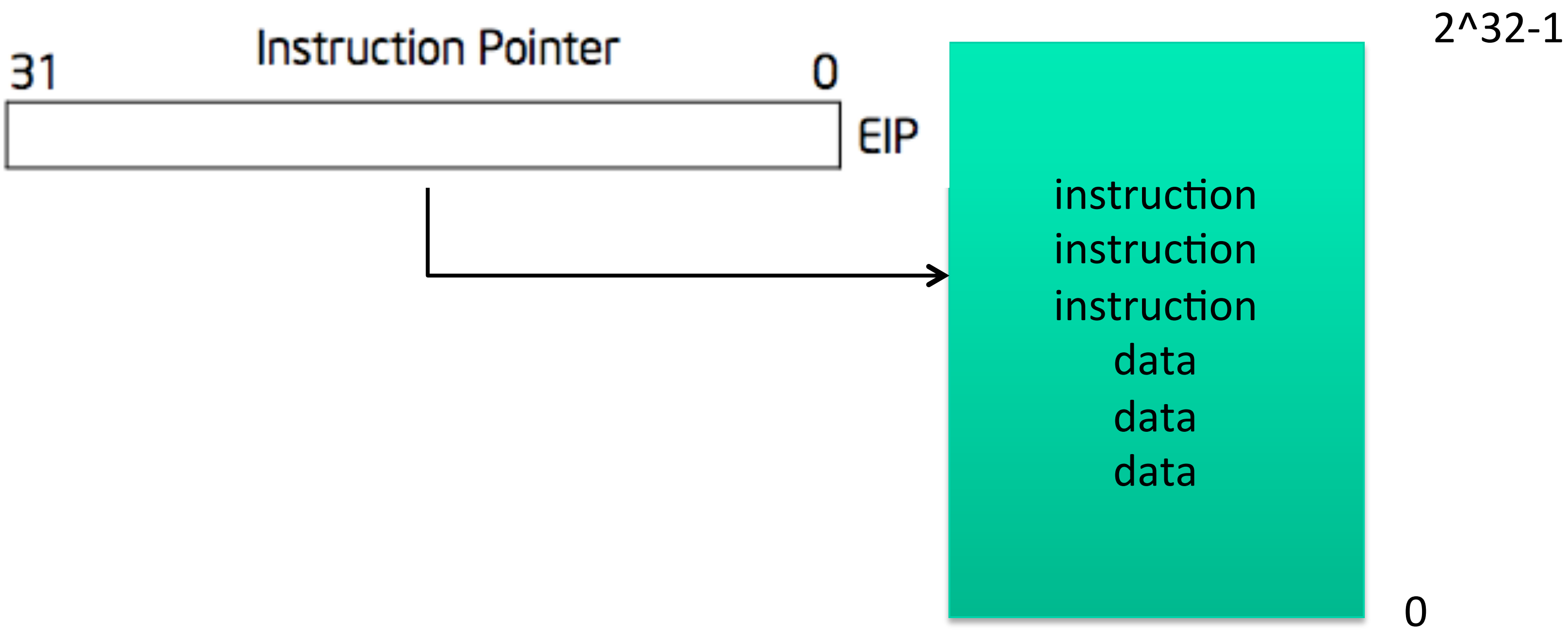
```
for(;;) {
   fetch next instruction
   run next instruction
}
```

Main Memory

instruction
instruction
instruction
data
data
data

- Often called the “Von Neumann” architecture
- Memory holds both *instructions* and *data*
- CPU interprets instructions
- Instructions read/write data

x86 implementation



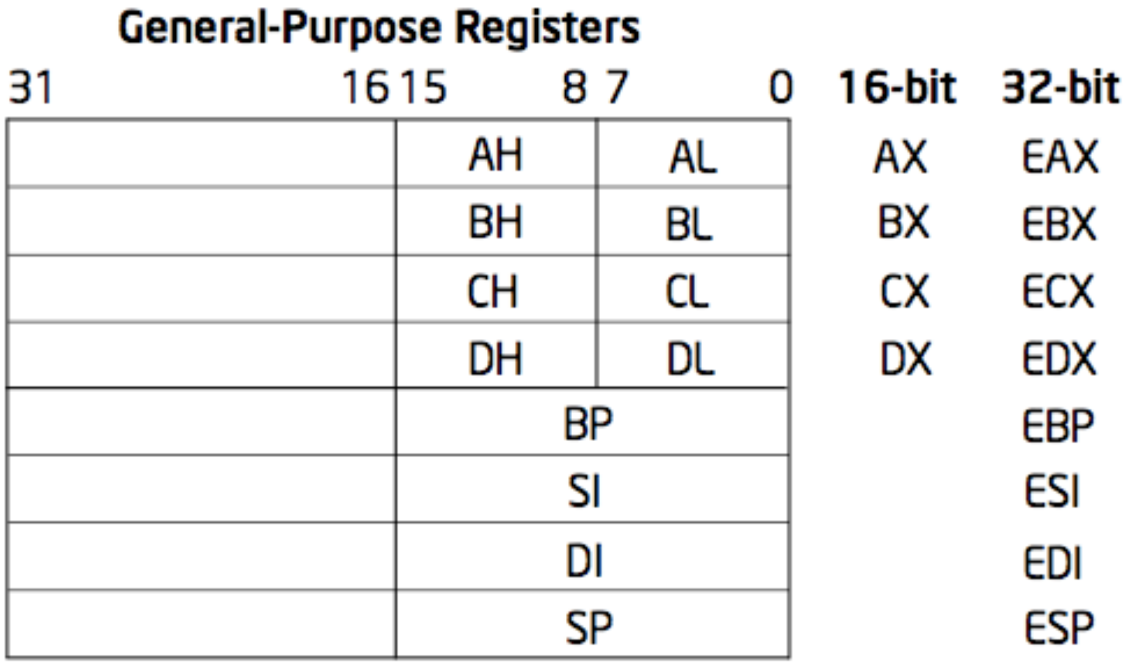
2^32-1

instruction
instruction
instruction
data
data
data

0

- On boot-up, EIP points to 0xFFFFFFF0
- Generally, BIOS (firmware) is mapped to that region
- EIP incremented after each instruction
- Variable length instructions
- EIP modified by CALL, RET, JMP, conditional JMP

Registers: work space



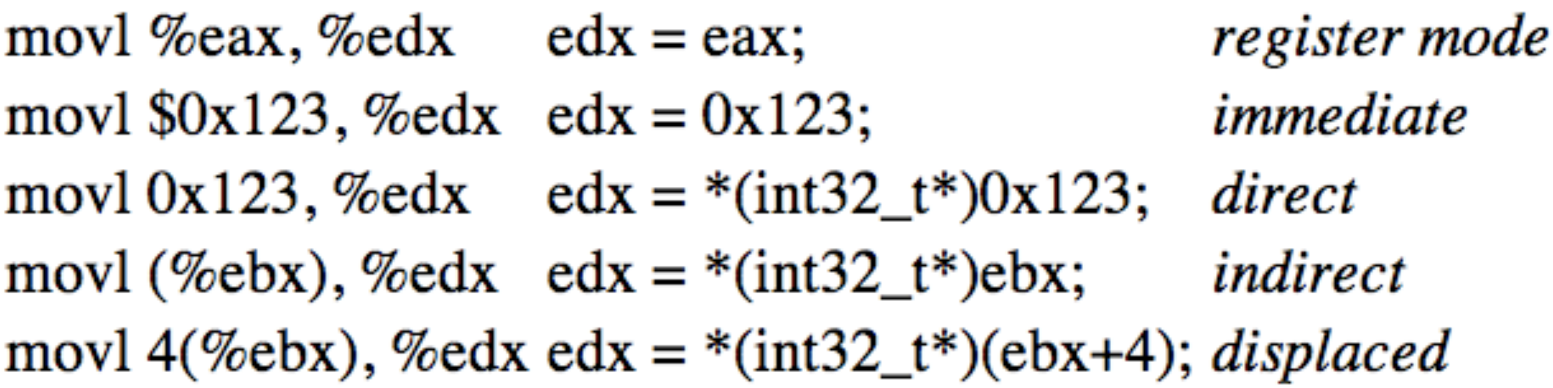
General-Purpose Registers

31 … 16	15 … 8	7 … 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

- ☐ 8, 16, and 32 bit versions
- ☐ Example: ADD EAX, 10
  - More: SUB, AND, etc
- ☐ By convention some for special purposes

ESP: stack pointer
EBP: frame base pointer
ESI: source index
EDI: destination index

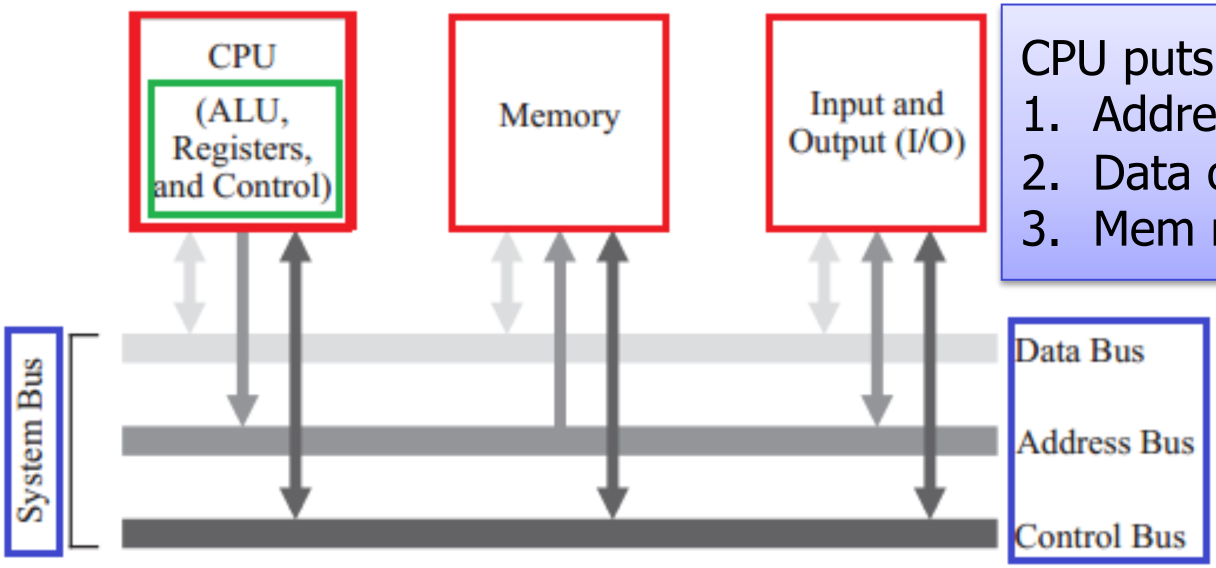
Memory: more work space



movl %eax, %edx	edx = eax;	*register mode*
movl $0x123, %edx	edx = 0x123;	*immediate*
movl 0x123, %edx	edx = *(int32_t*)0x123;	*direct*
movl (%ebx), %edx	edx = *(int32_t*)ebx;	*indirect*
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4);	*displaced*

- Memory instructions: MOV, PUSH, POP, etc
- Most instructions can take a memory address

Memory access

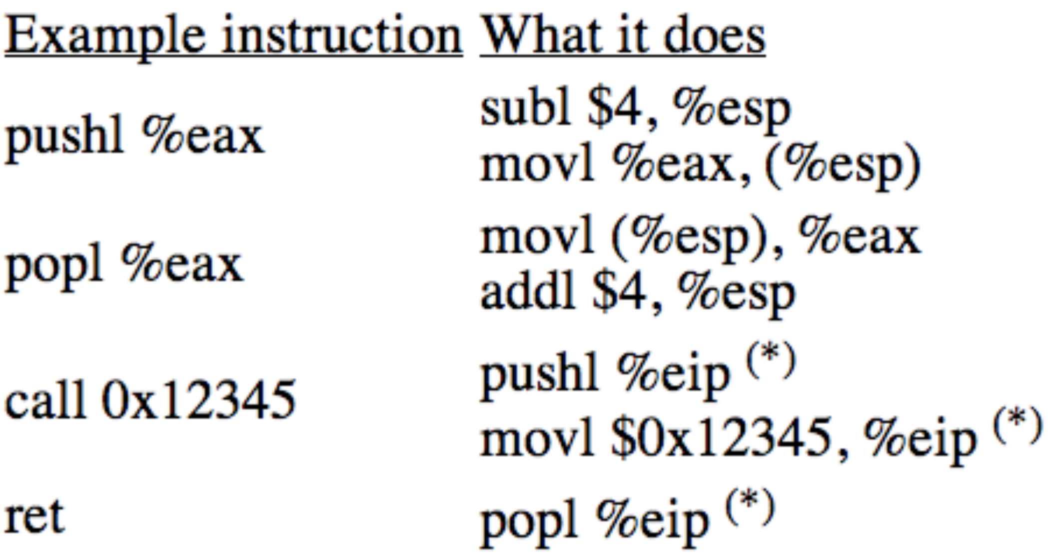


CPU puts:

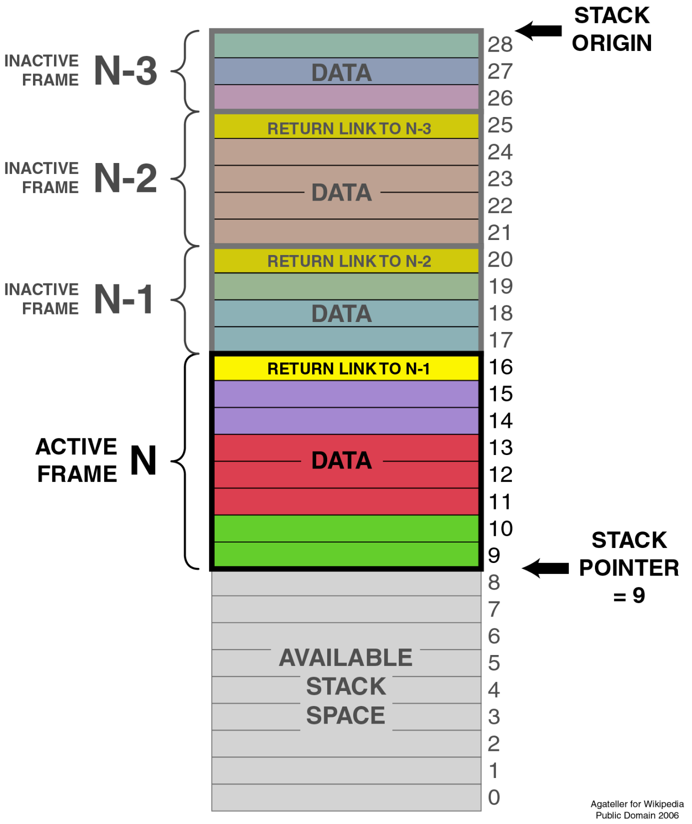
1. Address on address bus
2. Data on data bus
3. Mem read/write cmd on control bus

- Memory accesses are synchronous
- Instruction stalls while memory is fetched
- Caches are used to reduce the performance hit

The Stack



Example instruction	What it does
pushl %eax	subl $4, %esp movl %eax, (%esp)
popl %eax	movl (%esp), %eax addl $4, %esp
call 0x12345	pushl %eip (*) movl $0x12345, %eip (*)
ret	popl %eip (*)



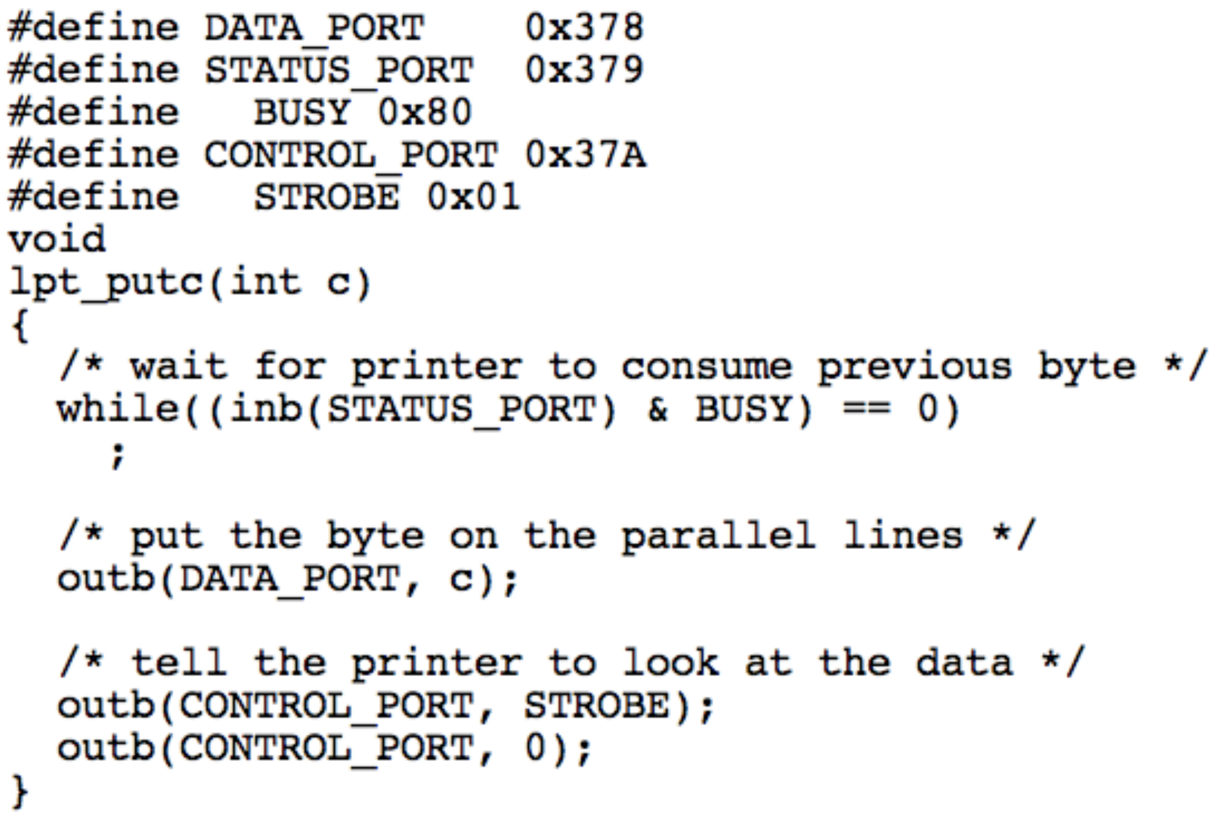
Agateller for Wikipedia Public Domain 2006

- For implementing function calls
- Temporary storage area
  - Saved register values, local variables, parameters
- Stack grows “down” on x86

Instruction classes

- Instruction classes
  - – Data movement: MOV, PUSH, POP, …
  - – Arithmetic: TEST, SHL, ADD, AND, …
  - – I/O: IN, OUT, …
  - – Control: JMP, JZ, JNZ, CALL, RET
  - – String: MOVSB, REP, …
  - – System: INT, IRET

I/O space and instructions



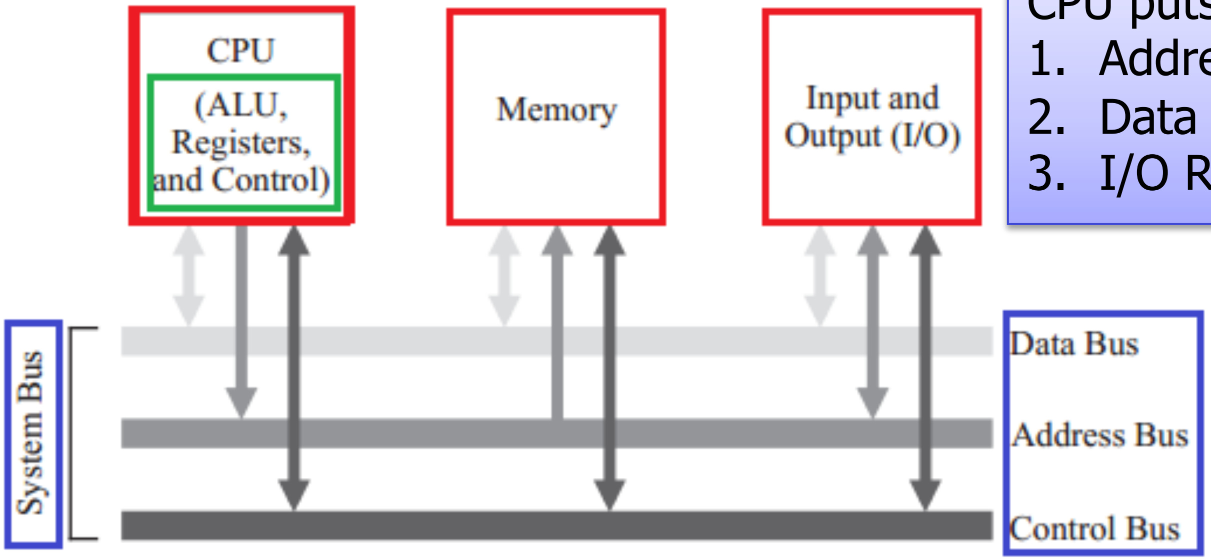
```
#define DATA_PORT    0x378
#define STATUS_PORT  0x379
#define   BUSY 0x80
#define CONTROL_PORT 0x37A
#define   STROBE 0x01
void
lpt_putc(int c)
{
  /* wait for printer to consume previous byte */
  while((inb(STATUS_PORT) & BUSY) == 0)
    ;

  /* put the byte on the parallel lines */
  outb(DATA_PORT, c);

  /* tell the printer to look at the data */
  outb(CONTROL_PORT, STROBE);
  outb(CONTROL_PORT, 0);
}
```

8086: 1024 ports, later processors, 16 bit addresses (65536 ports)

I/O Access



CPU puts:

1. Address on address bus
2. Data on data bus
3. I/O Read/write cmd on control bus

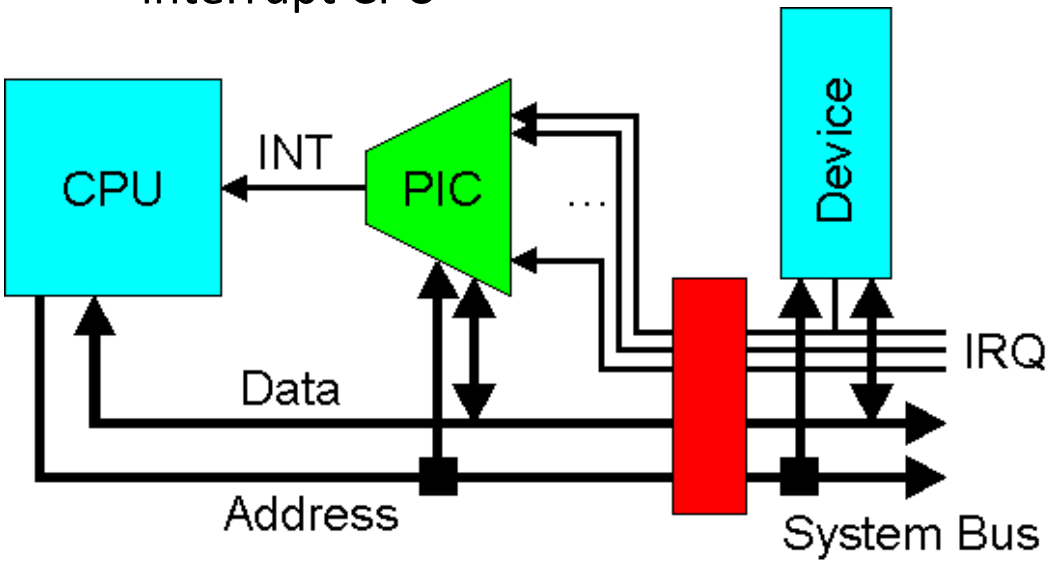
- Similar to memory access
- Except different control signals, so I/O devices know to respond (rather than memory)
- IN, OUT instructions are also synchronous

Memory-mapped I/O

- Use normal addresses for I/O
  - No special instructions
  - No 65536 limit
  - Hardware routes to device
- Works like "magic" memory
  - I/O device addressed and accessed like memory
  - However, reads and writes have "side effects"
  - Read result can change due to external events

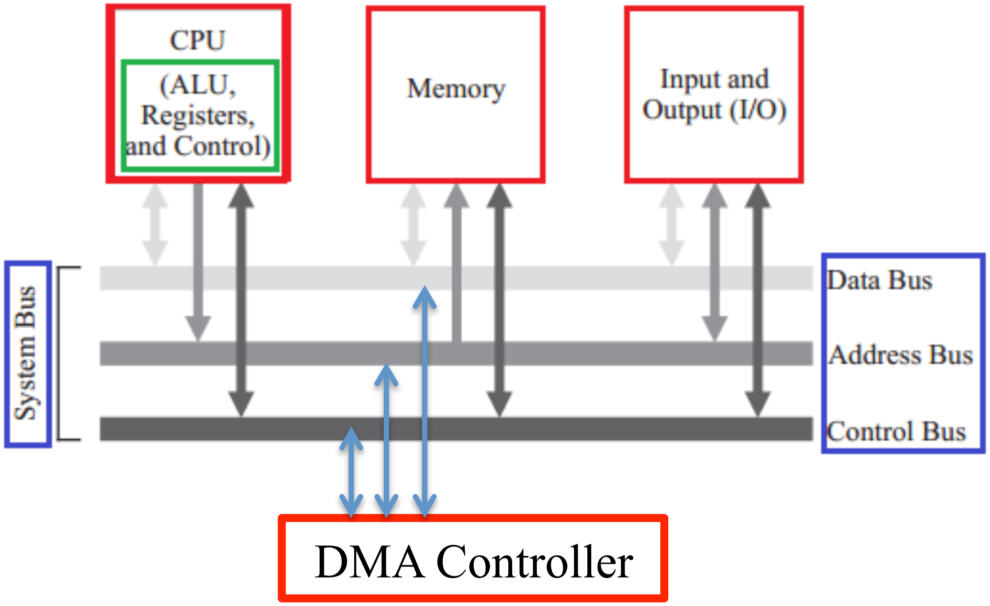
Interrupts

- I/O instructions synchronous, but hardware devices can be slow
  - Reading from disk (milliseconds), waiting for a keystroke (hours?)
  - Should CPU stay idle when waiting for device?
  - Interrupts allow CPU to multitask while waiting
  - Allows I/O to be asynchronous
  - Programmable Interrupt Controller (PIC) allows more than one device to interrupt CPU



```
for(;;) {
   if (interrupt) {
       n = get interrupt number
       call interrupt handler n
   }
   fetch next instruction
   run next instruction
}
```

Direct Memory Access (DMA)

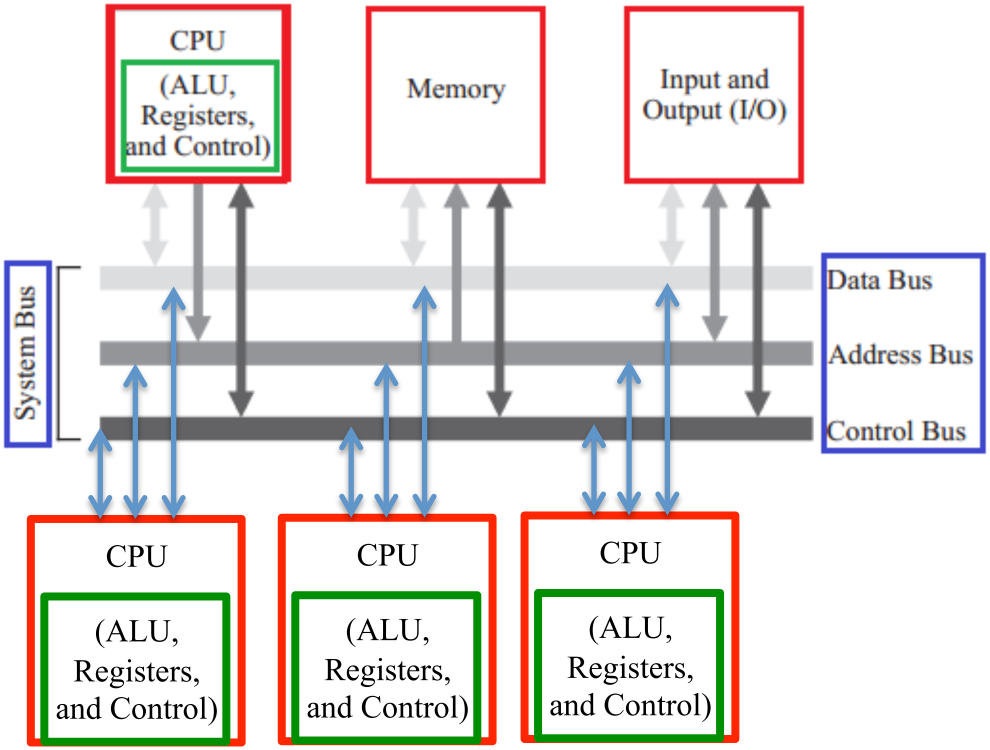


```
CPU:
Program DMA controller by setting
        mem address, length
Program I/O device to initiate transfer

DMA Controller:
for (length) {
        addr on address bus
        data on data bus
        mem read/write on control bus
    }
    Interrupt CPU when done
```

- DMA allows CPU to do useful work while transferring data between disk and memory
- CPU programs DMA controller (or device directly)
- DMA controller gets ownership of the bus
- DMA controller produces mem read/write bus signals

Symmetric Multiprocessors (SMP)



- Initially, one CPU starts executing instructions to initialize system
- Thereafter, each CPU executes independently
- Only sharing is through common memory (data structures)
- Each CPU receives interrupts independently through an APIC (Advanced PIC)

Outline

- Architecture overview

- OS evolution

- Modern OS structures

- Modern OS abstractions

OS evolution

- Many outside factors affect OS
- User needs + technology changes ➔ OS must evolve
  - New/better abstractions to users
  - New/better algorithms to implement abstractions
  - New/better low-level implementations (hw change)
- Current OS: evolution of these things

The Simplest OS

- If we’re running only one program and we don’t care about maximizing hardware utilization
  - Single purpose system
  - Don’t need much of an OS
  - Program executes instructions, wait for all I/O
- Running multiple programs, still one at a time
  - General purpose computers
  - OS provides facilities to load programs
  - Provides library of commonly used subroutines
  - So that everyone doesn’t need to rewrite
  - Earliest OS (monitors) were exactly like this

50-60s: Early mainframes

- Hardware:
  - Huge, $$$, slow
  - IO: punch card, line printer

- OS (Monitors)
  - simple library of device drivers (no resource coordination)
  - Human = OS: single programmer/operator programs, runs, debugs
  - One job at a time

- Problem: poor performance (utilization / throughput)

  Machine $$$, but idle most of the time because programmer slow

Batch Processing

- Batch: submit group of jobs together to machine
  - Operator collects, orders, runs (resource coordinator)

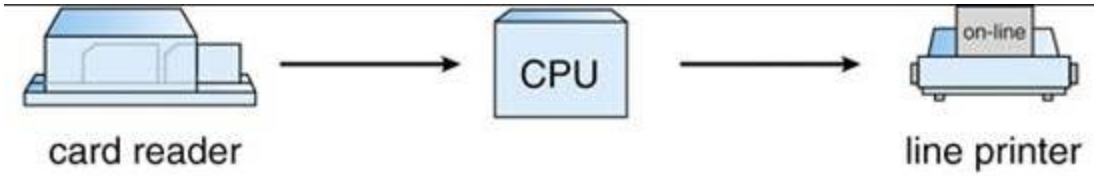
- Why good? can better optimize given more jobs
  - Cover setup overhead
  - Operator quite skilled at using machine
  - Machine busy more (programmers debugging offline)

- Why bad?
  - Must wait for results for long time

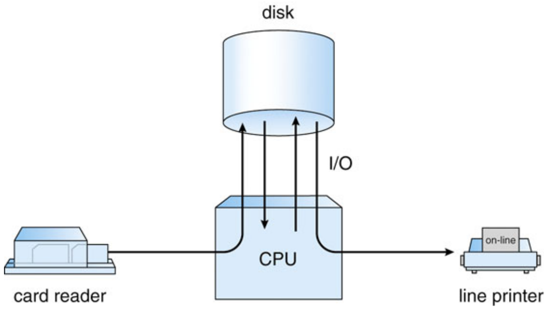
- Result: utilization increases, interactivity drops

Spooling

- Problem: slow I/O ties up fast CPU
  - Input ➔ Compute ➔ Output
  - Slow punch card reader and line printer



- Idea: overlap one job’s IO with other jobs’ compute



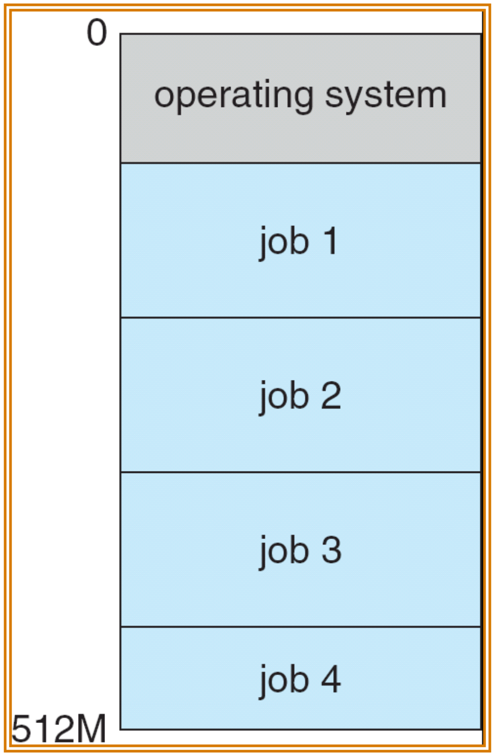
- OS functionality
  - buffering, DMA, interrupts
- Good: better utilization/throughput
- Bad: still not interactive

Multiprogramming

- Spooling → multiple jobs
- Multiprogramming
  - keep multiple jobs in memory, OS chooses which to run
  - When job waits for I/O, switch

- OS functionality
  - job scheduling, mechanism/policies
  - Memory management/protection

- Good: better throughput
- Bad: still not interactive



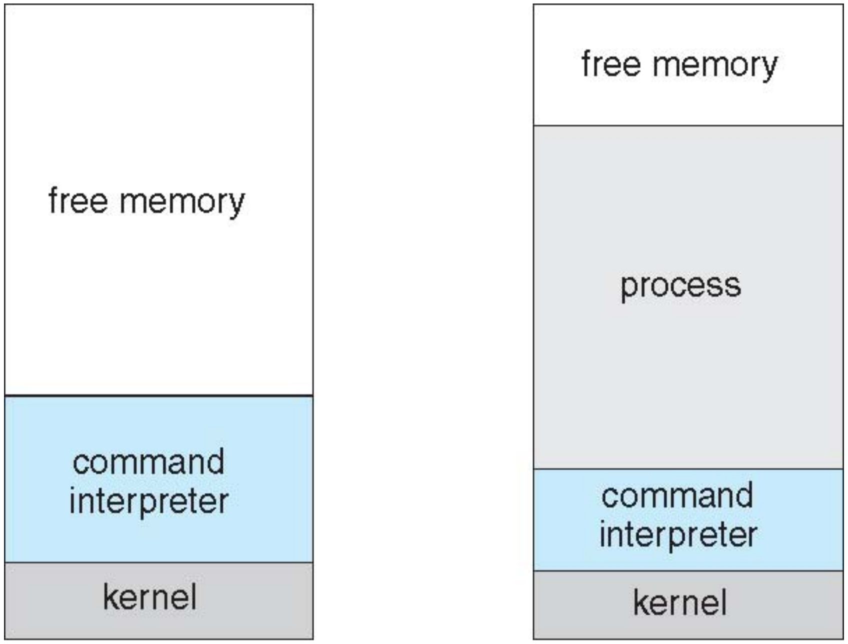
Late 60s, early 70s: Timesharing

- Many tasks needed better interactivity, short response time
- Concept: timesharing
  - Fast switch between jobs to give impression of dedicated machine
  - Compatible Time-Sharing System (CTSS) was the first time-sharing system prototype (1966)
  - Project MAC followed with MULTICS
  - UNIX followed shortly thereafter (1969-74). Still widely used.
- OS functionality:
  - More complex scheduling, memory management
  - Concurrency control, synchronization
- Good: immediate feedback to users

The 80s and 90s PC: turning back time

- Cheap personal computers
- Goal: ease of use, limited hardware
- Do not need a lot of stuff
- Example: DOS
  - No time-sharing, multiprogramming, protection, VM
  - One job at a time
  - OS is subroutine again

»



(a) (b)

Users + Hardware ➔ OS functionality

2000s: PCs come back to the future

- PC hardware gets more powerful and users expect more sophistication
  - Want to do many things at once (email, chat, browsing)
  - Don’t want bad programs to crash system
  - Want to support multiple family members to use (access control)
- Windows 95 and variants (Win 98, ME)
  - First Windows to use memory protection, poor local protection
- Windows 2000, XP, OS X
  - More robust isolation mechanisms
  - Return to local security
  - Multiple users, authentication, access control
- Relearn same lessons in the mobile world
  - Initially, poor security, no multitasking, no multiple users
  - Today, full fledged UNIX variant on most mobile devices

Operating System Range

- Mainframes — vast I/O bandwidth (Unix, VM/CMS)
- Clusters – extreme parallelism (Linux, AIX, Windows)
- Servers – utilization (Solaris, Linux, Windows Server)
- Desktops — premium on interaction: mouse, graphics (Windows, Linux, OS X)
- Mobile – premium on energy management, interactivity (iOS, Android)
- Embedded systems — sometimes lack memory protection (VxWorks, Symbian, QNX, Linux)

Major trends in History

- Hardware: cheaper and cheaper
- Computers/user: increases
- Functionality, capability/size: increases

- Timeline
  - 70s: mainframe, 1 / organization
  - 80s: minicomputer, 1 / group
  - 90s: PC, 1 / user
  - 00s: mobile, many / user

Current trends?

- Even larger systems
  - Can't make CPUs any faster (physics)
  - Make more of them in the same space
  - Multicore (100s of cores)
  - How to use efficiently?
- Even smaller systems: e.g. handheld, embedded devices
  - New, limited user interfaces
  - More sensors
  - Energy, battery life
- Reliability, Security
  - More features, more devices
  - Few errors in code, can recover from failures

Outline

- Architecture overview
- OS evolution
- Modern OS structures
- Modern OS abstractions

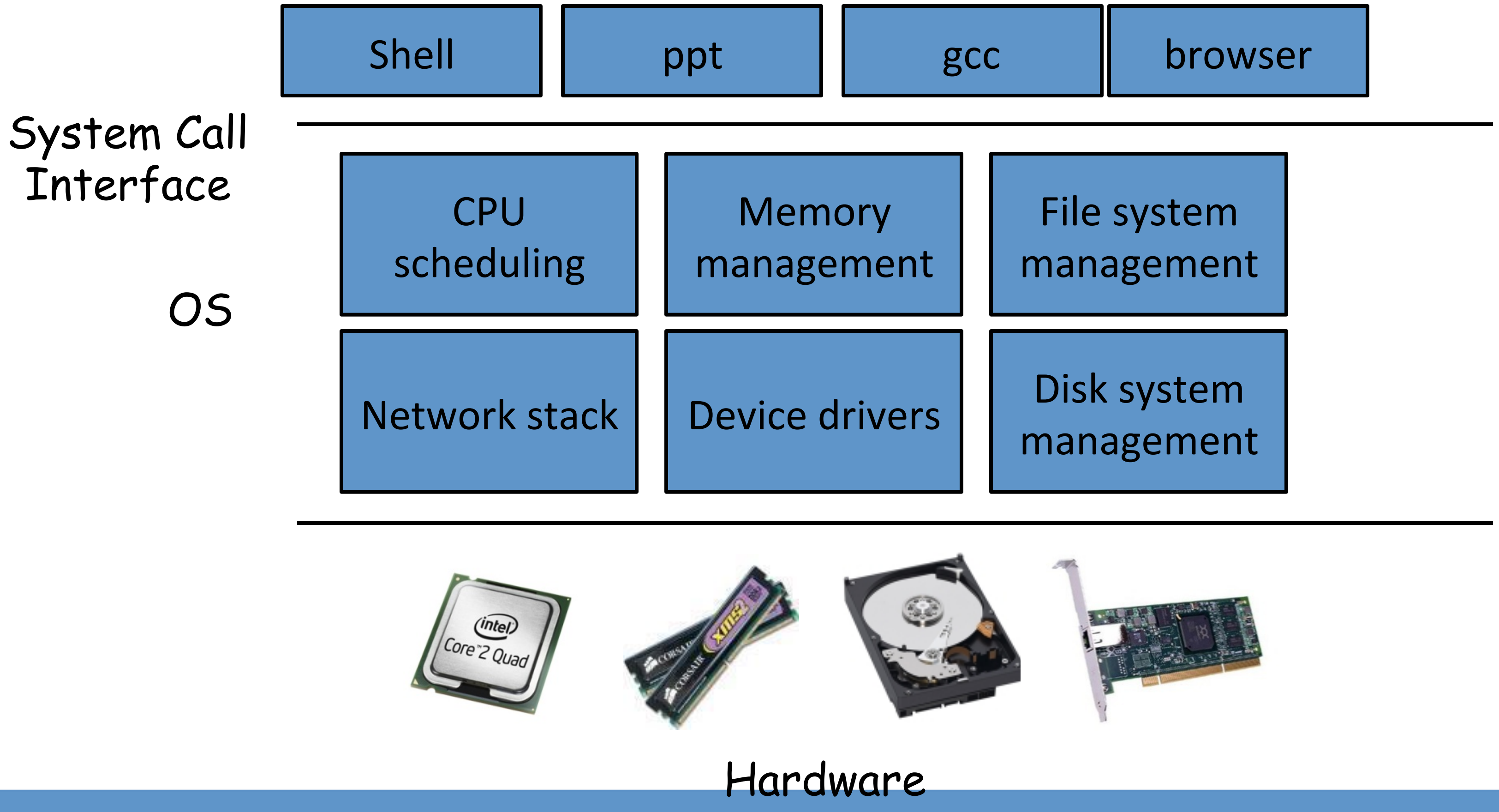
Two popular definitions

- Bottom-up perspective: resource manager/coordinator, manage your computer's resources

- Top-down perspective: hardware abstraction layer, turn hardware into something that applications can use

OS = resource manager/coordinator

- Computer has resources, OS must manage.
  - Resource = CPU, Memory, disk, device, bandwidth, …



Shell ppt gcc browser

System Call Interface

OS

CPU scheduling

Memory management

File system management

Network stack

Device drivers

Disk system management

Hardware

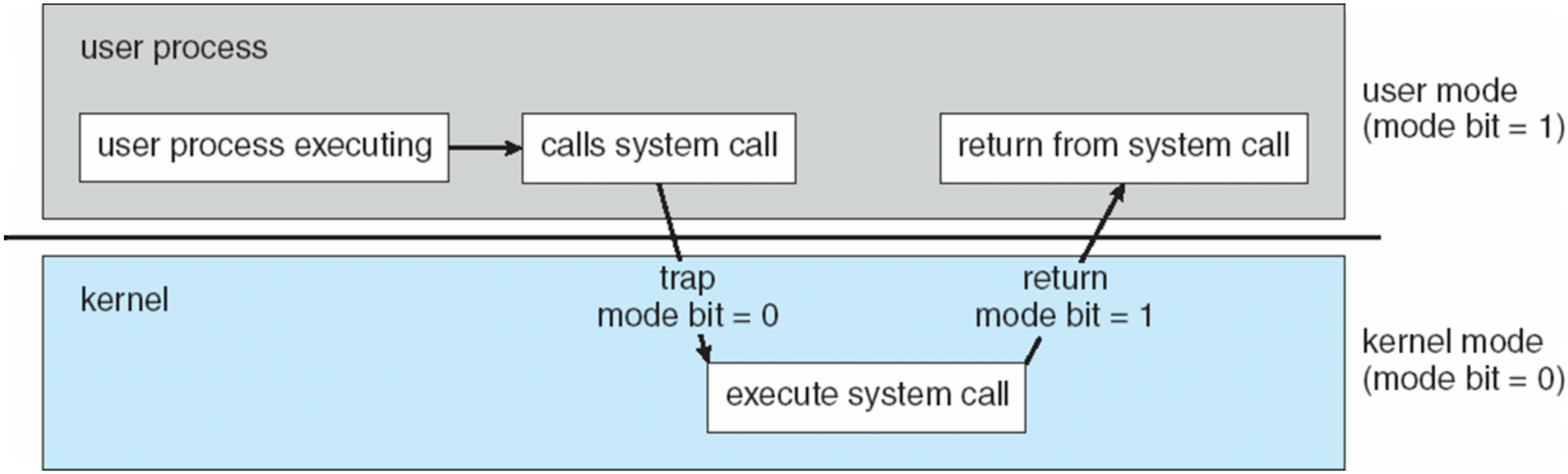
OS = resource manager/coordinator (cont.)

- Why good?
  - Sharing/Multiplexing: more than 1 app/user to use resource
  - Protection: protect apps from each other, OS from app
    - Who gets what when
  - Performance: efficient/fair access to resources
- Why hard? Mechanisms vs. policies
  - Mechanism: how to do things
  - Policy: what will be done
  - Ideal: general mechanisms, flexible policies
    - Difficult to design right

x86 Privileged Instructions

- Examples
  - IN, OUT: I/O instructions to protected ports
  - STI, CLI: enable, disable interrupts
  - SGDT, SLDT: change memory protection tables
  - LIDT: change interrupt handler table
  - Memory access to protected memory regions

Privileged Mode Operation



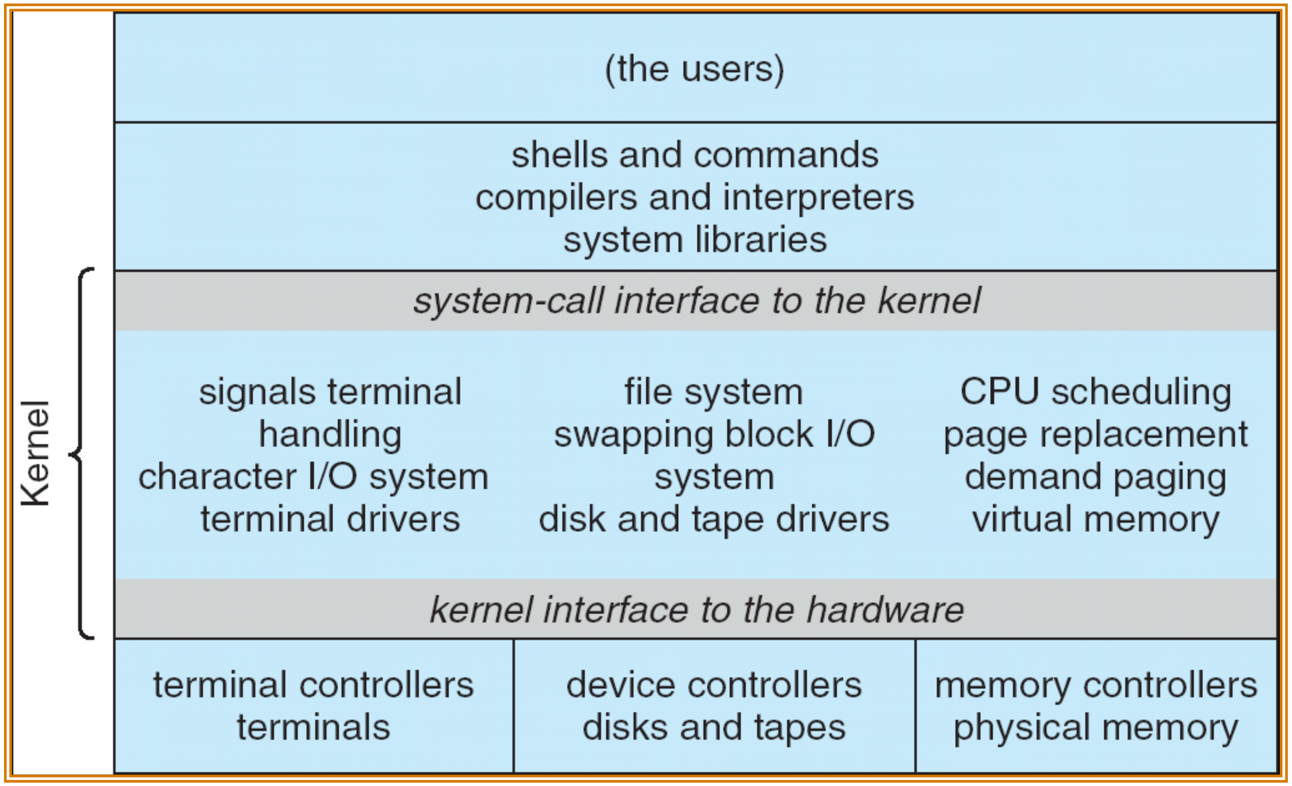
- Privileged mode can do everything
- User mode is restricted
  - I/O operations, changing CPU configuration tables, restricted memory access
- Privileged mode must delegate or implement functions on behalf of user mode

OS structure

- OS structure: what goes into the kernel?
  - Kernel: most interesting part of OS
    - Privileged; can do everything ➔ must be careful
    - Manages other parts of OS
- Different structures lead to different
  - Performance, functionality, ease of use, security, reliability, portability, extensibility, cost, …
- Tradeoffs depend on technology and workload

Monolithic

- Most traditional functionality in kernel



Kernel

(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

signals terminal handling
character I/O system
terminal drivers

file system
swapping block I/O system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

terminal controllers
terminals

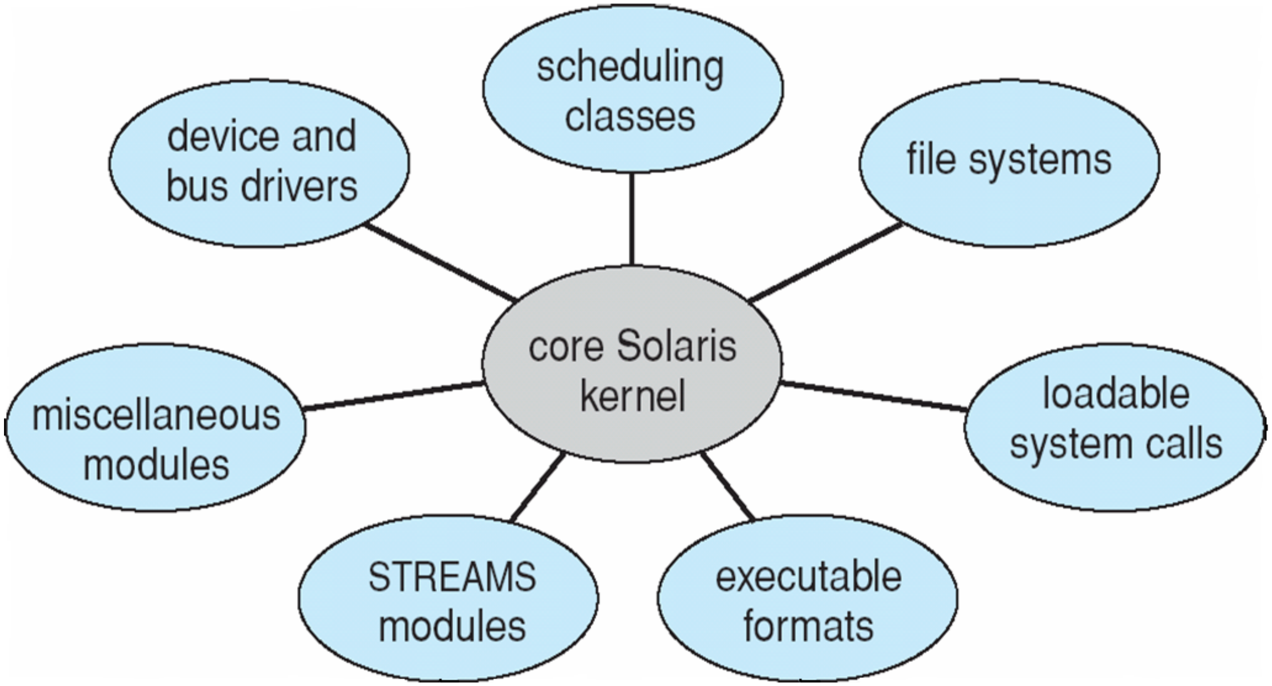
device controllers
disks and tapes

memory controllers
physical memory

Unix System Architecture

Modular kernels

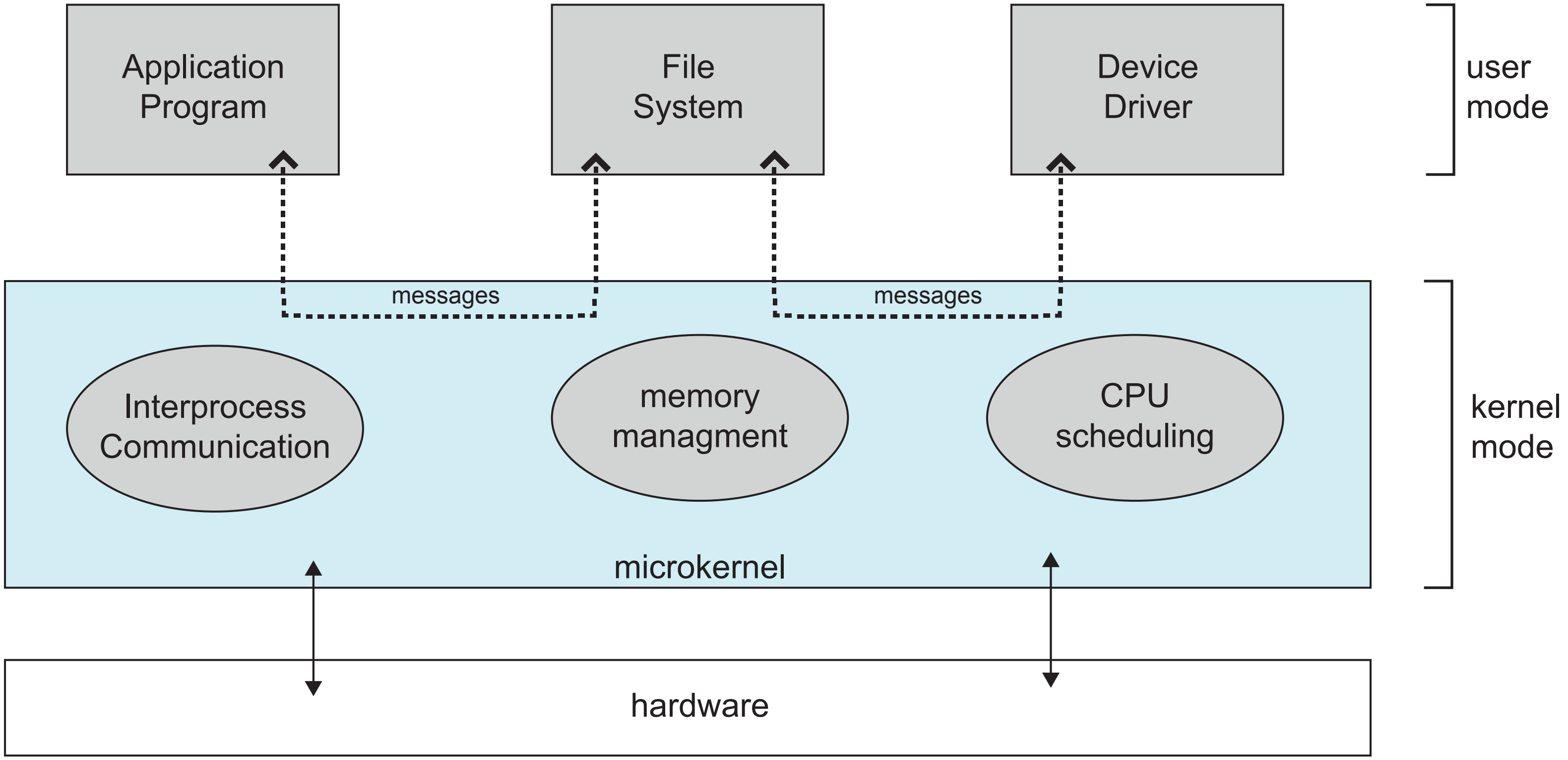
- Can dynamically add/change functionality



Solaris modules. Linux also supports kernel modules.

Microkernel

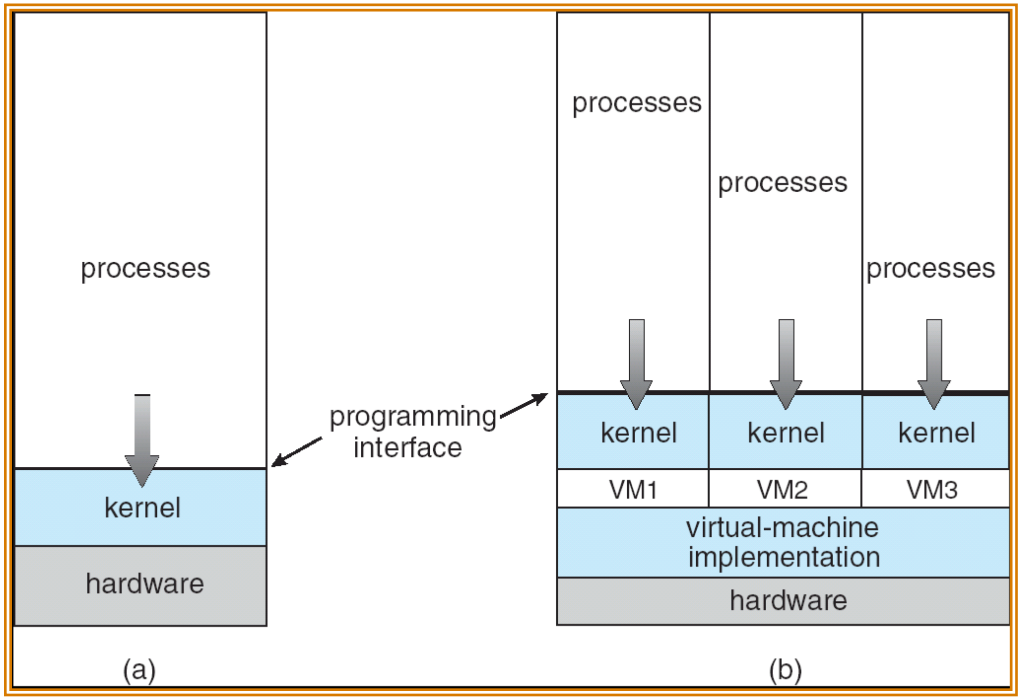
- Move functionality out of kernel



Microkernel handles interrupts, processing, scheduling, IPC

Virtual machine

- Export a fake hardware interface so that multiple OS can run on top



Non-virtual Machine

Virtual Machine

Outline

- Architecture overview
- OS evolution
- Modern OS structures
- Modern OS abstractions

Two popular definitions

- Bottom-up perspective: resource manager/ coordinator, manage your computer's resources

- Top-down perspective: hardware abstraction layer, turn hardware into something that applications can use

OS = hardware abstraction layer

- “standard library” “OS as virtual machine”
  - E.g. printf(“hello world”), shows up on screen
  - App issue system calls to use OS abstractions
- Why good?
  - Ease of use: higher level, easier to program
  - Reusability: provide common functionality for reuse
    - E.g. each app doesn’t have to write a graphics driver
  - Portability / Uniformity: stable, consistent interface, different OS/ver/hw look same
    - E.g. scsi/ide/flash disks
- Why hard?
  - What are the right abstractions ?

OS abstraction: process

- Running program, stream of running instructions + process state
  - A key OS abstraction: the applications you use are built of processes
    - Shell, powerpoint, gcc, browser, …
- Easy to use
  - Processes are protected from each other
    - process = address space
  - Hide details of CPU, when&where to run

OS abstraction: address space

- Contiguous array of bytes
  - Abstraction for RAM, not persistent across reboot
  - Easy way to store and access temporary data
  - Hide details of architecture, e.g., caches
  - Hide details of who else is sharing memory

- Everybody gets the same layout
  - Same code in same position in memory
  - Makes it easy to share library code

OS abstraction: file

- Array of bytes, persistent across reboot
  - Nice, clean way to read and write data
  - Hide the details of disk devices (hard disk, CDROM, flash …)

- Related abstraction: directory, collection of file entries

Process communication: pipe

- int pipe(int fds[2])
  - Creates a one way communication channel
  - fds[2] is used to return two file descriptors
  - Bytes written to fds[1] will be read from fds[0]
- Often used together with fork() to create a channel between parent and child

OS abstraction: thread

- “miniprocesses,” stream of instructions + thread state
  - Convenient abstraction to express concurrency in program execution and exploit parallel hardware

```
for(;;) {
    int fd = accept_client();
    create_thread(process_request, fd);
}
```

  - More efficient communication than processes

Implementing abstractions

- Indirection
  - Don’t access resource directly, but through a pointer
  - Can change pointer to point to a different resource
  - Can access only those things to which we have a pointer
  - E.g., address space translates to physical memory through a set of mapping “page” tables
  - May need hardware support for performance
- Resource management – managing pointers
  - If we can change resources that pointer points to…
  - Need to decide who gets what resource (allocators)
  - Need to track who has what resource (accounting)
  - Need to make pointer dereferencing efficient (e.g., caches)
  - This is where most of the complexity is

Next Class

- Our first OS abstraction
  - The process