BIL 301 Operating Systems
Ammar Daskin
Administriva & Introduction

OS-intro content is mostly based on

- Chapter 1 slides of the book Operating System Concepts, Silberschatz, Galvin and Gagne: https://www.os-book.com/OS10/slide-dir/index.html
- https://www.scs.stanford.edu/24wi-cs212/notes/intro.pdf
- https://pages.cs.wisc.edu/~remzi/OSTEP/intro.pdf

BIL 222 vs BIL 301

BIL 222 System Programming

- How userspace apps interact with systems(kernel-space) by mostly using system calls.
- Unix system calls

BIL 301 OS

- Internal structure of OS-kernel-space
- Management of different resources
- And services provided to userspace
- HWs with Linux kernel

Attendance

Weekly in person lectures

- Slides are provided before the lectures
- Some things may not be on the slides or may not be explained fully
    - Take notes in the lecture
- joining the google classroom is mandatory
    - hws, announcements, etc are posted on the classroom!
- Remind me the break-times!

Assignments (Hw)

4-5 hw:

- Programming assignments (group projects)
  - kernel compile/build
  - writing kernel module
  - copying data from userspace to kernel-space and vice versa
  - Synchronization
  - security/hacking os
  - Coding standards
    - https://www.kernel.org/doc/html/latest/process/coding-style.html
    - or K&R
- Written problems
  - e.g. scheduling algorithms

Grading

30% homework (including programming and written assignments)

30% midterm exam

40% final exam

Discussion

piazza.com

https://piazza.com/istanbul_medeniyet_university/fall2024/bil301/home

classroom.google.com

class code:

- Do not post solutions or any significant part of an assignment.
- Do not post anything not related to the course.
- Ask a question when you would like some help with something
- Post something when you would like to help others with something.

Collaboration and Cheating Policy

- Any kind of plagiarism and cheating are prohibited (Please, refer to the university cheating policy).
- If you benefit from some work of others, list them as references (online references or books)
- Discussing the assignments or projects with your friends is allowed; but, all the submitted work should be yours alone. List your collaborators (if you discuss your homework with your friends) in your assignments.

Textbooks and Course Material

- Operating System Concepts, 10th Edition Abraham Silberschatz, Greg Gagne, Peter B. Galvin, https://www.wiley.com/en-us/Operating+System+Concepts%2C+10th+Edition-p-9781119320913

- **Lectures slides are based on the slides**

  - [https://www.scs.stanford.edu/24wi-cs212/notes/] (https://www.scs.stanford.edu/24wi-cs212/notes/)
  - https://www.os-book.com/OS10/slide-dir/index.html
  - and https://linux-kernel-labs.github.io/

  - weekly posted on classroom.google.com before the lecture.
- **Other resources**
  - Linux Kernel Documentation
  - intel CPU manual
  - OS security and more: https://www.ics.uci.edu/~goodrich/teach/cs201P/notes/
  - ebook for synchronization https://dl.acm.org/doi/book/10.5555/2385452
    - https://booksite.elsevier.com/9780123973375/
  - kernel source code
  - https://linux-kernel-labs.github.io/refs/heads/master/index.html
  - Testing Linux: https://linux-test-project.github.io/

Weekly topics

1. Chapter 1: Introduction
2. Chapter 2: Operating System Structures
3. Chapter 3: Processes
4. Chapter 4: IPC, Threads & Concurrency
5. Chapter 5: CPU Scheduling
6. Chapter 5, 6-7: CPU Scheduling-II, Intro to Synchronization Tools & Examples
7. (not included in the book) Synchronization II (atomic instructions, memory barriers (e.g., mb, fence, volatile), C11 atomic library (relaxed, acquire, release), lock free programming, cache coherency,)
8. Midterm exam
9. Synchronization review, Deadlocks
10. Chapter 9: Main Memory
11. Chapter 10: Virtual Memory
12. Chapter 11-12: I/O Systems
13. Chapter 12-13-14-15: File-System Interface, Implementation, and Internals
14. Protection, Security

Course goals

Introduce you to operating system concepts

- Hard to use a computer without interacting with OS
- Understanding the OS makes you a more effective programmer

Cover important systems concepts in general

- Caching, concurrency, memory management, I/O, protection
- Synchronization and many topics may help you write more efficient user apps

Teach you to deal with(code/compile/build/install) larger software systems

- Programming assignments much larger than many courses
- Compiling linux kernel may take a few hours

Programming assignments

Implement/edit parts of linux kernel

- writing new system call
- adding new module
- security

Writing/testing synchronization tools (userspace)

- memory barriers
- thread libraries

First homework (hw0):

- **Download and install linux on virtualbox**
- **Download source code, and compile it…**

Installing Linux on Windows

Windows:

with WSL

https://learn.microsoft.com/en-us/windows/wsl/install

choose any linux distro that are available in appstore

https://learn.microsoft.com/en-us/windows/wsl/use-custom-distro

GUI

https://learn.microsoft.com/en-us/windows/wsl/tutorials/gui-apps?source=recommendations

windows terminal:

https://learn.microsoft.com/en-us/windows/terminal/install

Using virtual machine

If you have linux or windows machine

You can install new OS through virtual machine (prefered method):

VirtualBox, VMware, Parallels

https://www.geeksforgeeks.org/how-to-install-ubuntu-on-virtualbox/

https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox#1-overview

You can also run linux containers on windows

https://ubuntu.com/tutorials/windows-ubuntu-hyperv-containers#1-overview

macOS

Download linux ARM64 (AArch64) version

https://www.makeuseof.com/how-to-install-virtualbox-apple-silicon-mac/   :

For m3 chips

https://wiki.qemu.org/Main_Page

https://www.qemu.org/download/

https://mac.getutm.app/

Linux test

https://linux-test-project.github.io/

Based on

Chapter 1 slides of the book Operating System Concepts, Silberschatz, Galvin and Gagne:
https://www.os-book.com/OS10/slide-dir/index.html
https://www.scs.stanford.edu/24wi-cs212/notes/intro.pdf
https://pages.cs.wisc.edu/~remzi/OSTEP/intro.pdf

Intro to OS

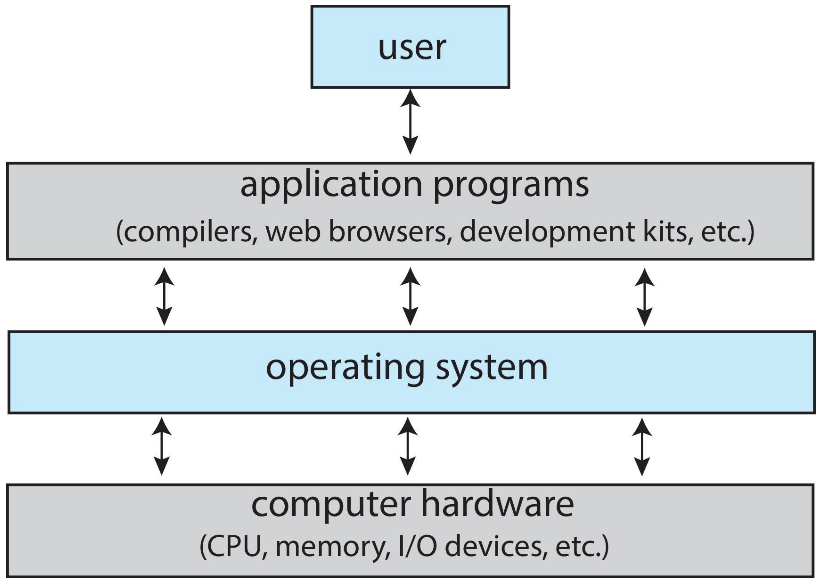
What Does the Term Operating System Mean?

- An operating system is “...”?
- What about:
  - Car
  - Airplane
  - Printer
  - Washing Machine
  - Toaster
  - Compiler
  - Etc.

Computer System Structure

- Computer system can be divided into four components:
  - **Hardware** – provides basic computing resources
    - CPU, memory, I/O devices
  - **Operating system**
    - Controls and coordinates use of hardware among various applications and users
  - **Application programs** – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - **Users**
    - People, machines, other computers

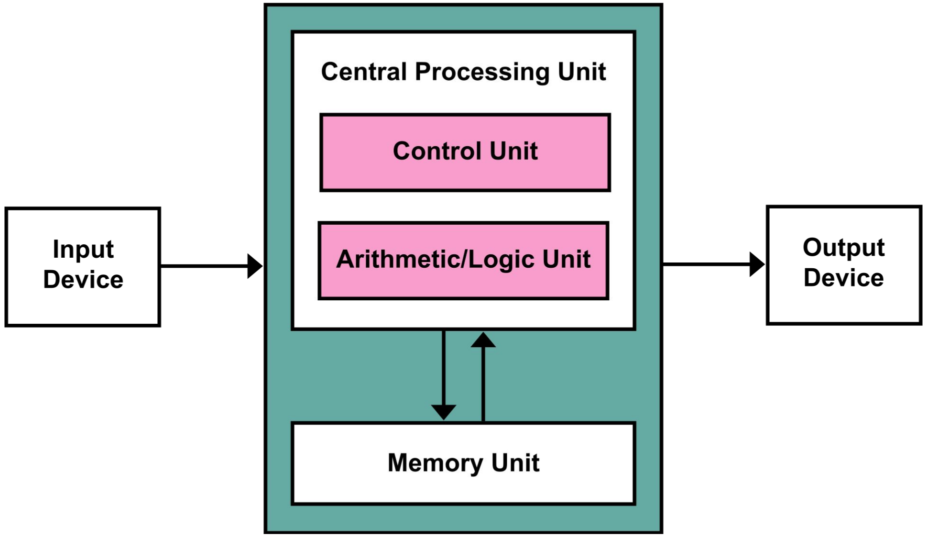
What is an Operating System?


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

- An OS is a program that acts as an intermediary between a user of a computer and the computer hardware

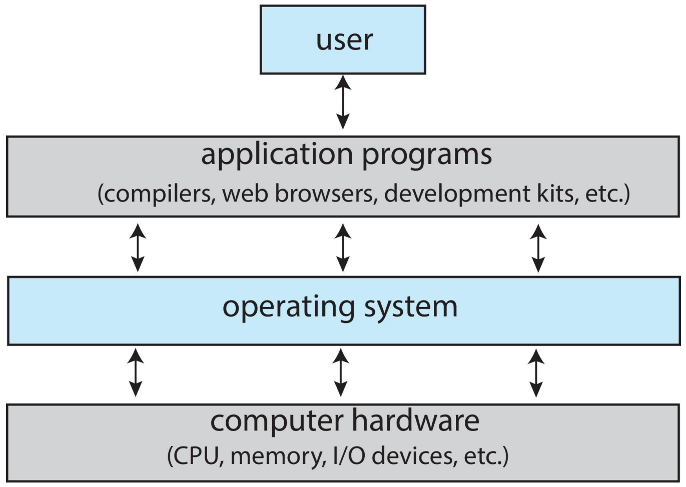
What happens when a program runs?

- it simply executes instructions:
  - von Neumann model of computing:
    - the processor fetches an instruction from memory, decodes it (figures out which instruction this is), and executes it.
    - After it is done with this instruction, the processor moves on to the next instruction, and so on, and so on, until the program finally completes
- billions of times every second,



https://en.wikipedia.org/wiki/Von_Neumann_architecture

What is an Operating System?


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

- Makes hardware useful to the programmer(goal):
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

```c
#include <stdio.h>
#include <stdlib.h>

#include "common.h"

int main(int argc, char *argv[]) {
   if (argc != 2) {
       fprintf(stderr, "usage: cpu <string>\n");
       exit(1);
   }
   char *str = argv[1];

   while (1) {
       printf("%s\n", str);
       Spin(1);
   }
   return 0;
}
```

```c
#ifndef __common_h__
#define __common_h__

#include <assert.h>
#include <sys/stat.h>
#include <sys/time.h>

double GetTime() {
   struct timeval t;
   int rc = gettimeofday(&t, NULL);
   assert(rc == 0);
   return (double)t.tv_sec + (double)t.tv_usec / 1e6;
}

void Spin(int howlong) {
   double t = GetTime();
   while ((GetTime() - t) < (double)howlong);  // do nothing in loop
}

#endif  // __common_h__
```

```
$ gcc -Wall -o cpu cpu.c
```
Running multiple programs at once

```
$ ./cpu "A" & ./cpu "B" & ./cpu "C"
```

——-----------------------------

Virtualizing CPU

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
   if (argc != 2) {
   fprintf(stderr, "usage: mem <value>\n");
   exit(1);
   }
   int *p;
   p = malloc(sizeof(int));
   assert(p != NULL);
   printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
   *p = atoi(argv[1]); // assign value to addr stored in p
   while (1) {
   Spin(1);
   *p = *p + 1;
   printf("(%d) value of p: %d\n", getpid(), *p);
   }
   return 0;
}
```

Running mem program multiple times
$./mem &; ./mem &

——-------------

Virtualizing Memory

```
volatile int counter = 0;
int loops;
void *worker(void *arg) {
   int i;
   for (i = 0; i < loops; i++) {
   counter++;
   }
   return NULL;
}
int main(int argc, char *argv[]) {
   if (argc != 2) {
   fprintf(stderr, "usage: threads <loops>\n");
   exit(1);
   }
   loops = atoi(argv[1]);
   pthread_t p1, p2;
   printf("Initial value : %d\n", counter);
   pthread_create(&p1, NULL, worker, NULL);
   pthread_create(&p2, NULL, worker, NULL);
   pthread_join(p1, NULL);
   pthread_join(p2, NULL);
   printf("Final value   : %d\n", counter);
   return 0;
}
```

What primitives needed from OS to build correct and efficient concurrent programs?

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    assert(fd >= 0);
    char buffer[20];
    sprintf(buffer, "hello world\n");
    int rc = write(fd, buffer, strlen(buffer));
    assert(rc == (strlen(buffer)));
    fsync(fd);
    close(fd);
    return 0;
}
```

HOW TO STORE DATA PERSISTENTLY

The file system is the part of the OS in charge of managing persistent data.

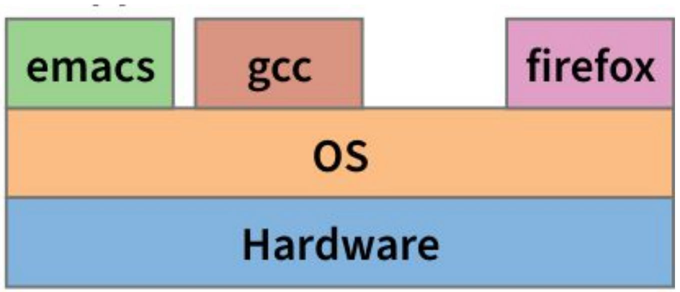
So far

OS **virtualizes** resources (CPU, memory, etc.), works as a **resource manager**

OS stores files **persistently**

OS helps tricky issues such as **concurrency**

What is an Operating System?



Design goals

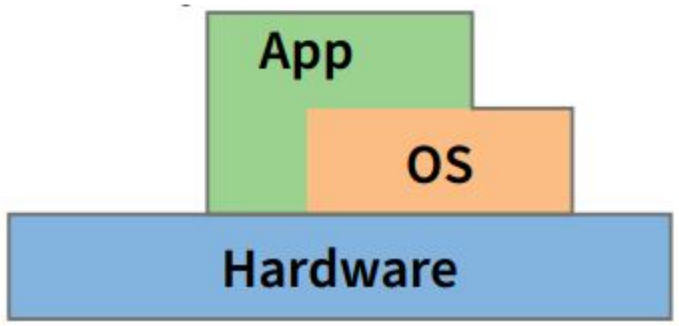
- [Usually] Provides **abstractions** for applications
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
  - makes the system convenient and easy to use.
- [Often] Provides **protection**
  - Prevents one process/user from clobbering another
- Other design goal(issues):
  - high performance
  - must also run non-stop(reliability)
  - security
  - energy-efficiency
  - mobility

Why study OS?

- Operating systems are a mature field
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- Still open questions in operating systems
  - Security
  - Hard to achieve security without a solid foundation
  - Scalability
  - How to adapt concepts when hardware scales 10× (fast networks, low service times, high core counts, big data. . .)
- High-performance servers are an OS issue
  - Face many of the same issues as OSes, sometimes bypass OS
  - Resource consumption is an OS issue
  - Battery life, radio spectrum, etc.
- New “smart” devices need new OSes
  - Robotics(ROS), IoTs

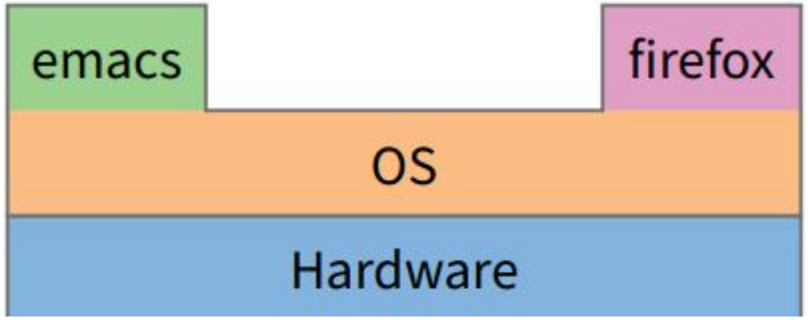
Early-primitive Operating Systems: Just Libraries

- Just a library of standard services **[no protection]**



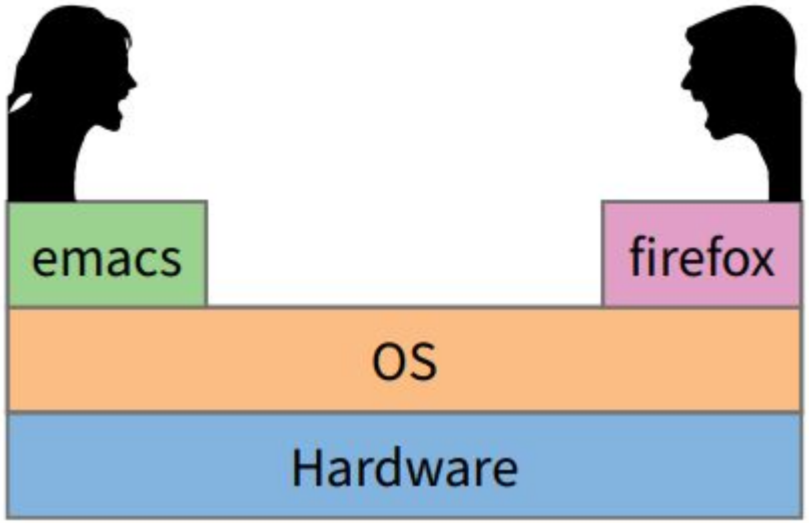
- Simplifying assumptions
  - System runs one program at a time (the program is chosen from a **batch**)
  - No bad users or programs (often bad assumption)
- Problem: Poor utilization
  - of hardware (e.g., CPU idle while waiting for disk)
  - of human user (must wait for each program to finish)

Beyond libraries: Multi-tasking & protection



- **Idea:** More than one process can be running at once
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem:** What can ill-behaved process do?
  - Go into infinite loop and never relinquish CPU
  - Scribble over other processes' memory to make them fail
- **OS provides mechanisms to address these problems**
  - Preemption – take CPU away from looping process
  - Memory protection – protect processes' memory from one another

Multi-user OS

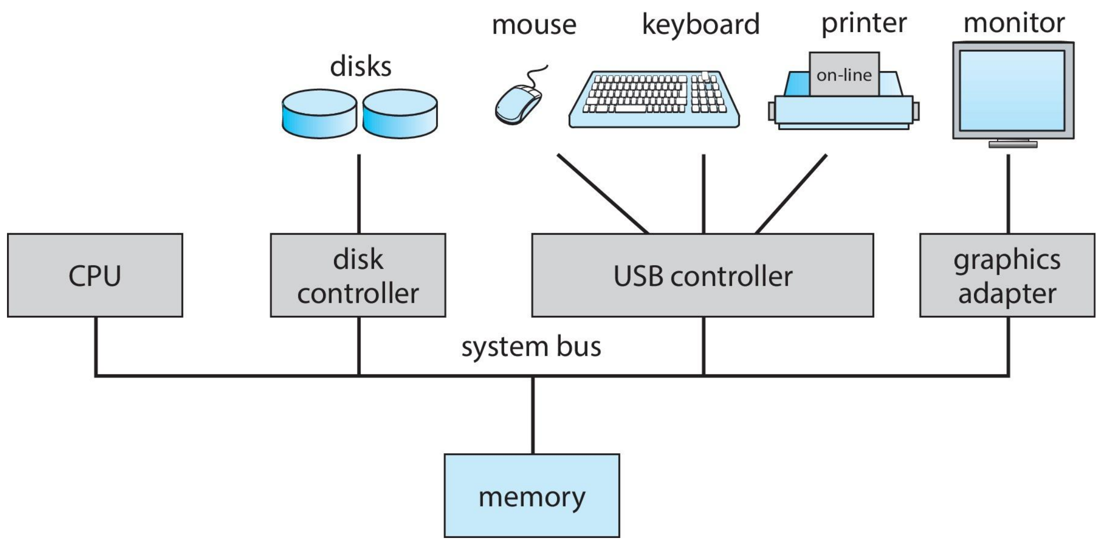


- Many OSes use **protection** to serve distrustful users/apps
- **Idea:** With N users, system not N times slower
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**
  - Users are gluttons, use too much CPU, etc. (need policies)
  - Total memory usage greater than machine's RAM (must virtualize)
  - Super-linear slowdown with increasing demand (thrashing)

Protection

- Mechanisms that **isolate** bad programs and people
    - also isolating processes from one another
- Pre-emption:
    - Give application a resource, take it away if needed elsewhere
- Interposition/mediation:
    - Place OS between application and “stuff”
    - Track all pieces that application allowed to use (e.g., in table)
    - On every access, look in table to check that access legal
- Privileged & unprivileged modes in CPUs:
    - Applications unprivileged (unprivileged user mode)
    - OS privileged (privileged supervisor/kernel mode)
    - Protection operations can only be done in privileged mode

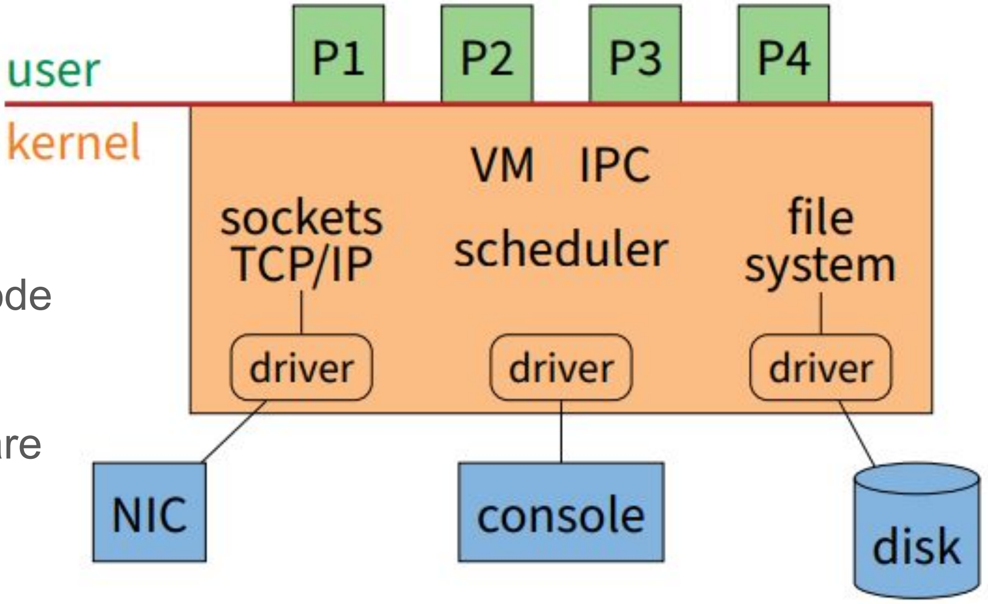
- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Typical OS structure

- Most software runs as **user-level processes (P[1-4])**
  - process ≈ instance of a program

- **OS kernel** runs in privileged mode (**orange)**
  - Creates/deletes processes
  - Provides access to hardware



System calls

Code run on behalf of the OS is special!

The idea of **a system call** was pioneered by the Atlas computing system.

Instead of providing OS routines as a library (where you just make a procedure call to access them),

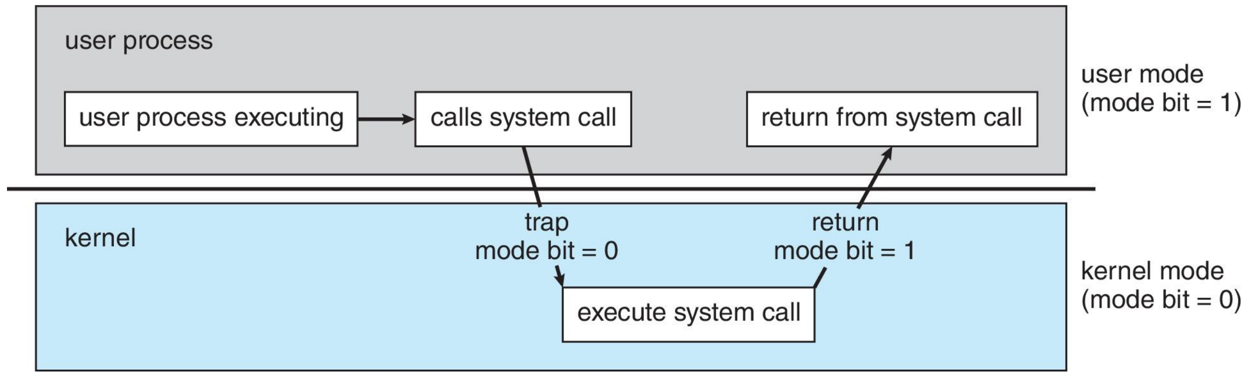
the idea here was to add **a special pair of hardware instructions** and hardware state to make the transition into the OS a more formal, controlled process.

Dual-mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
    - **User mode** and **kernel mode**
    - **Mode bit** provided by hardware
        - 1 mode bit is "user"
        - 0 mode bit is "kernel"

- How do we guarantee that user does not explicitly set the mode bit to "kernel"?
    - System call changes mode to kernel, return from call resets it to user
- Some instructions designated as **privileged**, only executable in kernel mode

Transition from User to Kernel Mode

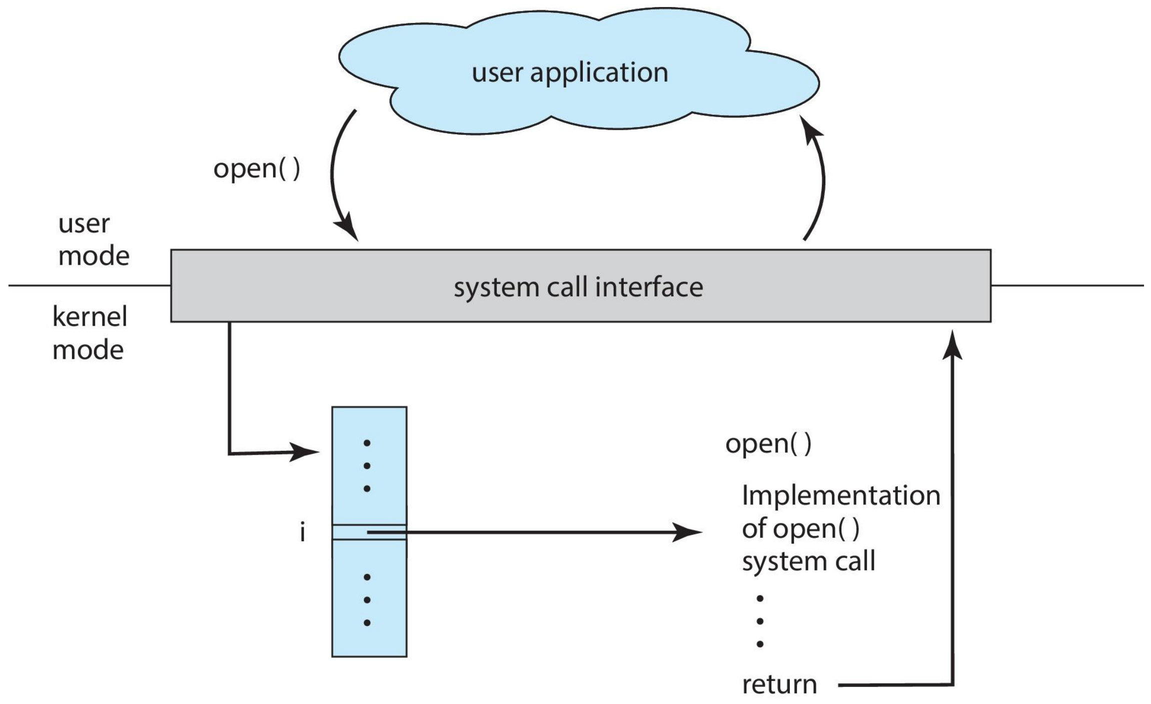


System calls

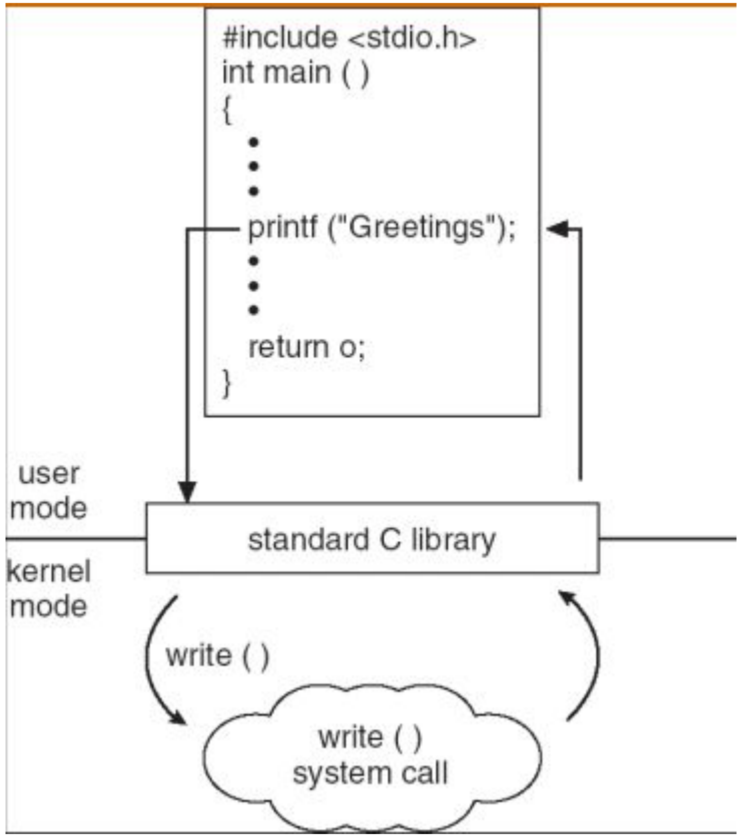
- Goal: Do things application can’t do in unprivileged mode
    - Like a library call, but into more privileged kernel code
- Kernel supplies well-defined system call interface
    - Applications set up syscall arguments and trap to kernel
    - Kernel performs operation and returns result
- Higher-level functions built on syscall interface
    - printf, scanf, fgets, etc. all user-level code
- Example: POSIX/UNIX interface
    - open, close, read, write, ...

System call example

- Applications can invoke kernel through system calls
  - Special instruction transfers control to kernel
  - . . . which dispatches to one of few hundred syscall handlers



- Standard library implemented in terms of syscalls
  - printf – in libc, has same privileges as application
  - calls write – in kernel, which can send bits out serial port



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return o;
}
```

UNIX file system calls

- Applications “open” files (or devices) by name
  - I/O happens through open files
- ***int open(char *path, int flags, /*int mode*/...);***
  - flags: O_RDONLY, O_WRONLY, O_RDWR
  - O_CREAT: create the file if non-existent
  - O_EXCL: (w. O_CREAT) create if file exists already
  - O_TRUNC: Truncate the file
  - O_APPEND: Start writing from end of file
  - mode: final argument with O_CREAT
- Returns file descriptor—used for all I/O to file

Error returns

- What if open fails? Returns -1 (invalid fd)

- **Most system calls return -1 on failure**
  - Specific kind of error in global int errno
  - In retrospect, bad design decision for threads/modularity

- #include <sys/errno.h> for possible values
  - 2 = ENOENT “No such file or directory”
  - 13 = EACCES “Permission Denied”
- perror function prints human-readable message
  - perror ("initfile");

  → “initfile: No such file or directory”

Operations on file descriptors

- int read (int fd, void *buf, int nbytes);
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- int write (int fd, const void *buf, int nbytes);
  - Returns number of bytes written, -1 on error
- off_t lseek (int fd, off_t pos, int whence);
  - whence: 0 – start, 1 – current, 2 – end
  - ▹ Returns previous file offset, or -1 on error
- int close (int fd);

File descriptor numbers

- File descriptors are inherited by processes
  - When one process spawns another, same fds by default
- Descriptors 0, 1, and 2 have special meaning
  - *0 – “standard input” (stdin in ANSI C)*
  - *1 – “standard output” (stdout, printf in ANSI C)*
  - *2 – “standard error” (stderr, perror in ANSI C)*
  - Normally all three attached to terminal
- Example: type.c
  - Prints the contents of a file to stdout

example

```
void typefile(char *filename) {
   int fd, nread;
   char buf[1024];
   fd = open(filename, O_RDONLY);
   if (fd == -1) {
       perror(filename);
       return;
   }
   while ((nread = read(fd, buf, sizeof(buf))) > 0)
       write(1, buf, nread);
   close(fd);
}
```

Can see system calls using strace utility (ktrace on BSD)

Protection example: CPU preemption

Protection mechanism to prevent monopolizing CPU

- E.g., kernel programs timer to interrupt every 10 ms
    - Must be in supervisor mode to write appropriate I/O registers
    - User code cannot re-program interval timer
- Kernel sets interrupt to vector back to kernel
    - Regains control whenever interval timer fires
    - Gives CPU to another process if someone else needs it
    - Note: must be in supervisor mode to set interrupt entry points
    - No way for user code to hijack interrupt handler
- Result: Cannot monopolize CPU with infinite loop
    - At worst get 1/N of CPU with N CPU-hungry processes

Protection is not security

How can you monopolize CPU?

- Use multiple processes
- For many years, could wedge most OSes with
  - **int main() { while(1) fork(); }**
  - Keeps creating more processes until system out of proc. slots
- Other techniques: use all memory (chill program)

Typically solved with technical/social combination

  - Technical solution: Limit processes per user
  - Social: Reboot and yell at annoying users
  - Social: Ban harmful apps from play store

Protection by Address translation

Protect memory of one program from actions of another

- Definitions
  - Address space: all memory locations a program can name
  - Virtual address: addresses in process’ address space
  - Physical address: address of real memory
  - Translation: map virtual to physical addresses
- Translation done on every load, store, and instruction fetch
  - Modern CPUs do this in hardware for speed
- Idea: If you can’t name it, you can’t touch it
  - Ensure one process’s translations don’t include any other process’s memory

More memory protection

- CPU allows kernel-only virtual addresses
    - Kernel typically part of all address spaces,
        - e.g., to handle system call in same address space
    - But must ensure apps can’t touch kernel memory
- CPU lets OS disable (invalidate) particular virtual addresses
    - Catch and halt buggy program that makes wild accesses
    - Make virtual memory seem bigger than physical
        - (e.g., bring a page in from disk only when accessed)
- CPU enforced read-only virtual addresses useful
    - E.g., allows sharing of code pages between processes
    - + many other optimizations
- CPU enforced execute disable of VAs
    - Makes certain code injection attacks harder

Different system contexts

At any point, a CPU (core) is in one of several contexts

- User-level
  - CPU in user mode running application
- Kernel process context
  - running kernel code on behalf of a particular process
    - E.g., performing system call, handling exception (memory fault, numeric exception, etc.)
  - Or executing a kernel-only process (e.g., network file server)
- Kernel code not associated with a process
  - Timer interrupt (hardclock)
  - Device interrupt
  - “Softirqs”, “Tasklets” (Linux-specific terms)
- Context switch code – change which process is running
  - Requires changing the current address space
- Idle – nothing to do (bzero pages, put CPU in low-power state)

Transitions between contexts

- User → kernel process context: syscall, page fault, . . .
- User/process context → interrupt handler: hardware
- Process context → user/context switch: return
- Process context → context switch: sleep
- Context switch → user/process context

Resource allocation & performance

Multitasking permits higher resource utilization

- Simple example:
    - Process downloading large file mostly waits for network
    - You play a game while downloading the file
    - Higher CPU utilization than if just downloading
- Complexity arises with cost of switching
- Example: Say disk 1,000 times slower than memory
    - 1 GiB memory in machine
    - 2 Processes want to run, each use 1 GiB
    - Can switch processes by swapping them out to disk
    - Faster to run one at a time than keep context switching

Useful properties to exploit

- Skew
  - 80% of time taken by 20% of code
  - 10% of memory absorbs 90% of references
  - Basis behind cache: place 10% in fast memory, 90% in slow, usually looks like one big fast memory
- Past predicts future (a.k.a. temporal locality)
  - What's the best cache entry to replace?
  - If past ≈ future, then least-recently-used entry
- Note conflict between fairness & throughput
  - Higher throughput (fewer cache misses, etc.) to keep running same process
  - But fairness says should periodically preempt CPU and give it to next process

(The rest is skipped in the lecture)

Overview of Computer System Structure

Storage Structure

Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - **Random access**
  - Typically **volatile**
  - Typically **random-access memory** in the form of **Dynamic Random-access Memory (DRAM)**
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity

Storage Structure (Cont.)

- **Hard Disk Drives** (**HDD**) – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer
- **Non-volatile memory** (**NVM**) devices– faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular as capacity and performance increases, price drops

Storage Definitions and Notation Review

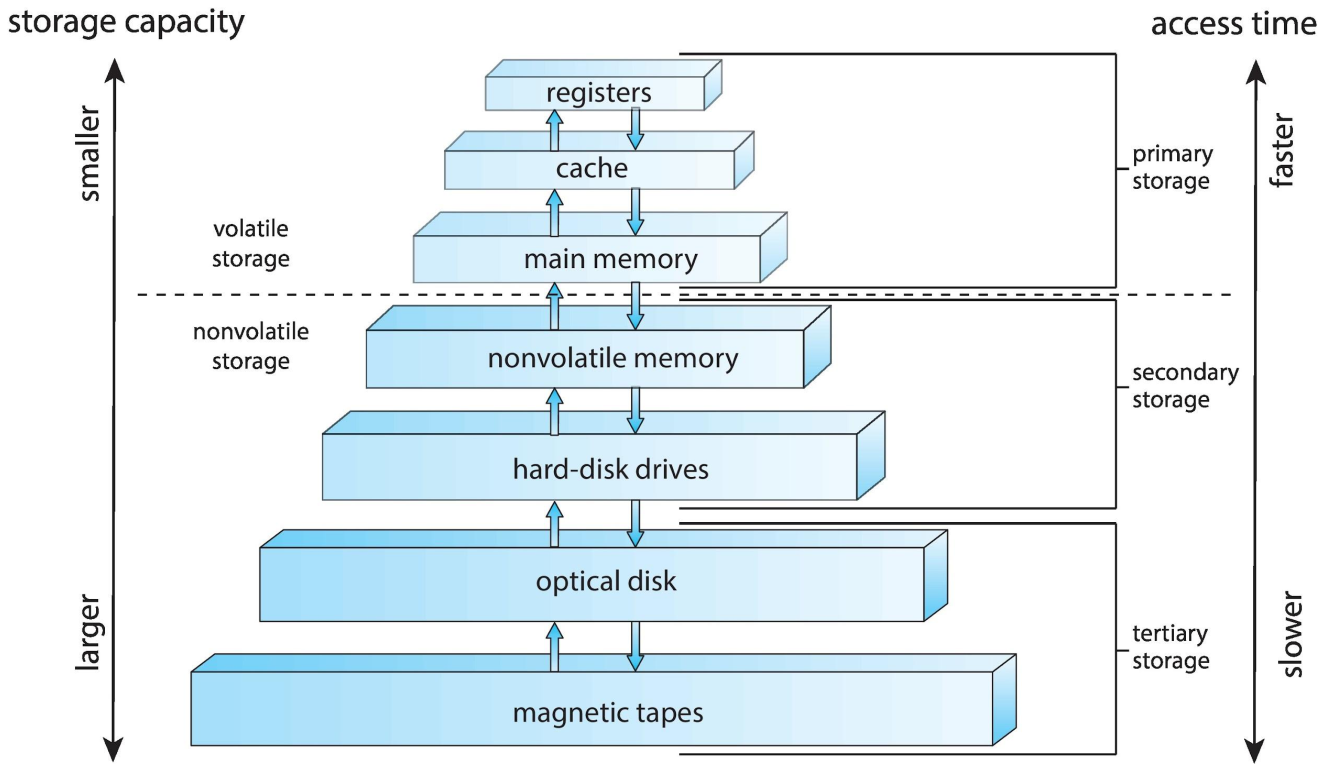
The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or KB , is 1,024 bytes; a **megabyte**, or **MB**, is $1{,}024^2$ bytes; a **gigabyte**, or **GB**, is $1{,}024^3$ bytes; a **terabyte**, or **TB**, is $1{,}024^4$ bytes; and a **petabyte**, or **PB**, is $1{,}024^5$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

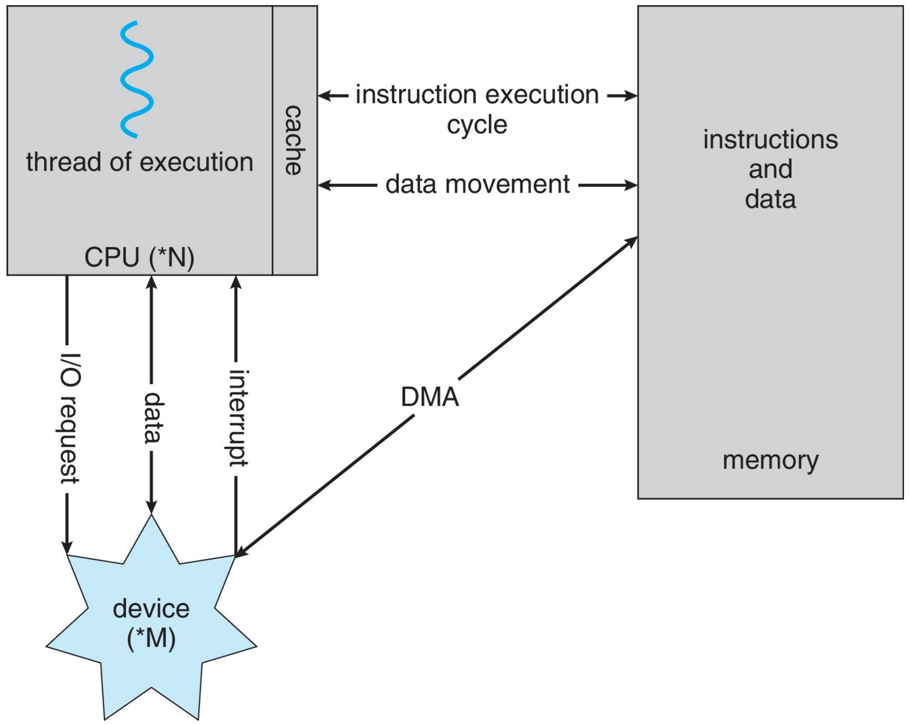
Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

Storage-Device Hierarchy



How a Modern Computer Works



A von Neumann architecture

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

Operating-System Operations

- Bootstrap program – simple code to initialize the system, load the kernel
- Kernel loads
- Starts **system daemons** (services provided outside of the kernel)
- Kernel **interrupt driven** (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (**exception** or **trap**):
    - Software error (e.g., division by zero)
    - Request for operating system service – **system call**
    - Other process problems include infinite loop, processes modifying each other or the operating system

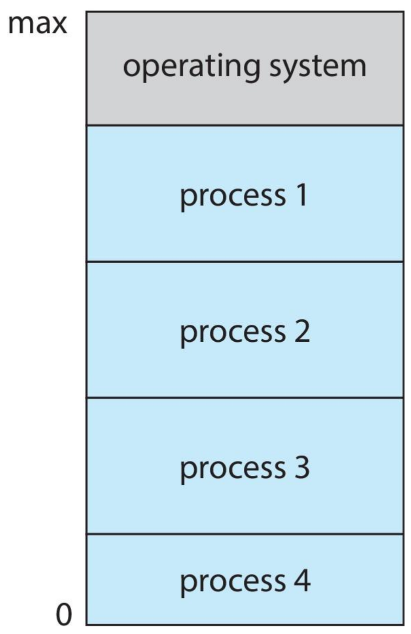
Multiprogramming (Batch system)

- Single user cannot always keep CPU and I/O devices busy
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute
- A subset of total jobs in system is kept in memory
- One job selected and run via **job scheduling**
- When job has to wait (for I/O for example), OS switches to another job

Multitasking (Timesharing)

- A logical extension of Batch systems– the CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory □ **process**
  - If several jobs ready to run at the same time □ **CPU scheduling**
  - If processes don’t fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

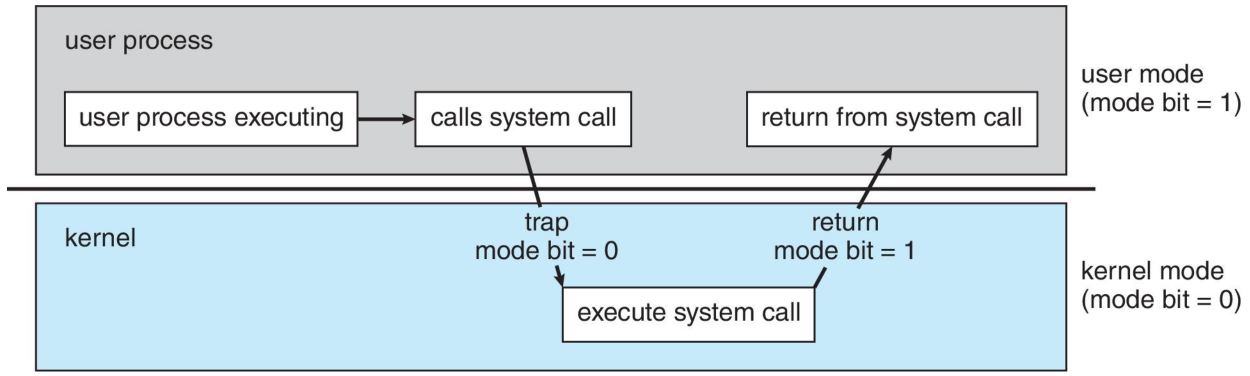
Memory Layout for Multiprogrammed System



Dual-mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
- **Mode bit** provided by hardware
  - Provides ability to distinguish when system is running user code or kernel code.
  - When a user is running □ mode bit is "user"
  - When kernel code is executing □ mode bit is "kernel"
- How do we guarantee that user does not explicitly set the mode bit to "kernel"?
  - System call changes mode to kernel, return from call resets it to user
- Some instructions designated as **privileged**, only executable in kernel mode

Transition from User to Kernel Mode



Timer

- Timer to prevent infinite loop (or process hogging resources)
    - Timer is set to interrupt the computer after some time period
    - Keep a counter that is decremented by the physical clock
    - Operating system set the counter (privileged instruction)
    - When counter zero generate an interrupt
    - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a ***passive entity;*** process is an ***active entity***.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory
- Memory management determines what is in memory and when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

File-system Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

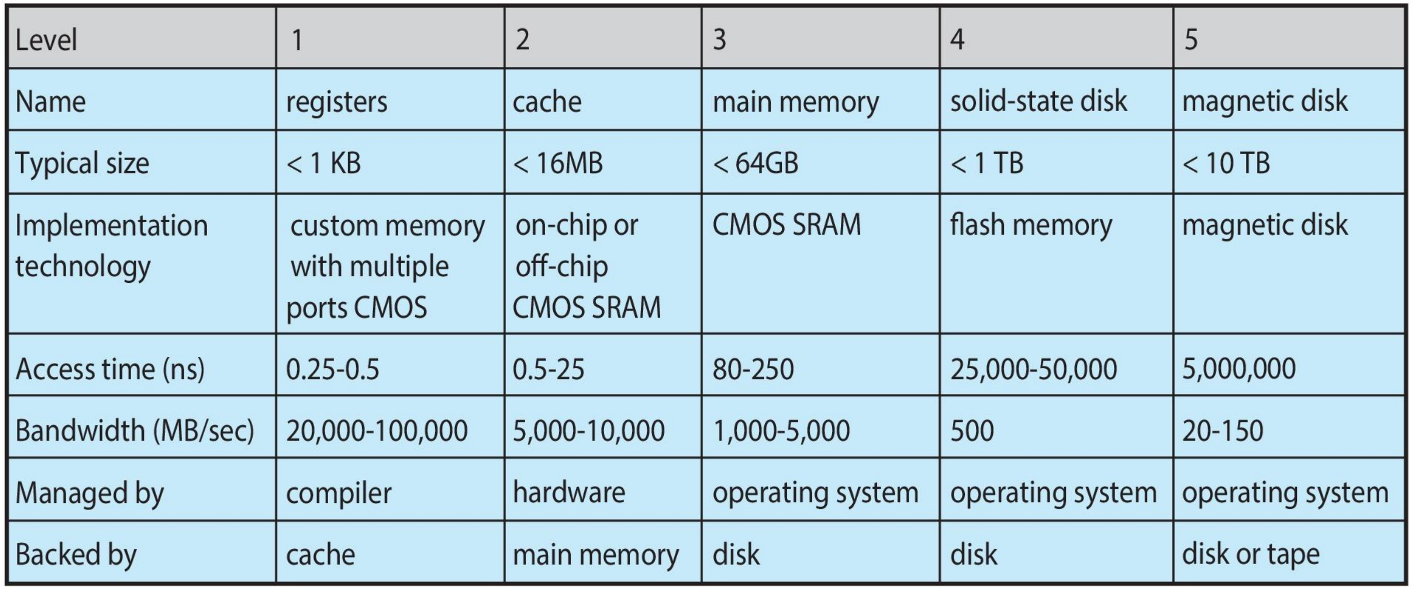
Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
  - Mounting and unmounting
  - Free-space management
  - Storage allocation
  - Disk scheduling
  - Partitioning
  - Protection

Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
    - If it is, information used directly from the cache (fast)
    - If not, data copied to cache and used there
- Cache smaller than storage being cached
    - Cache management important design problem
    - Cache size and replacement policy

Characteristics of Various Types of Storage

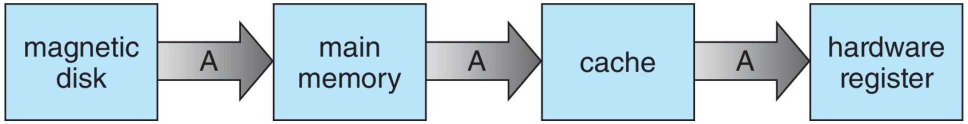


Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Movement between levels of storage hierarchy can be explicit or implicit

Migration of data “A” from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
    - Several copies of a datum can exist
    - Various solutions covered in Chapter 19

I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
    - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
    - General device-driver interface
    - Drivers for specific hardware devices

Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
    - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
    - User identities (**user IDs**, security IDs) include name and associated number, one per user
    - User ID then associated with all files, processes of that user to determine access control
    - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
    - **Privilege escalation** allows user to change to effective ID with more rights

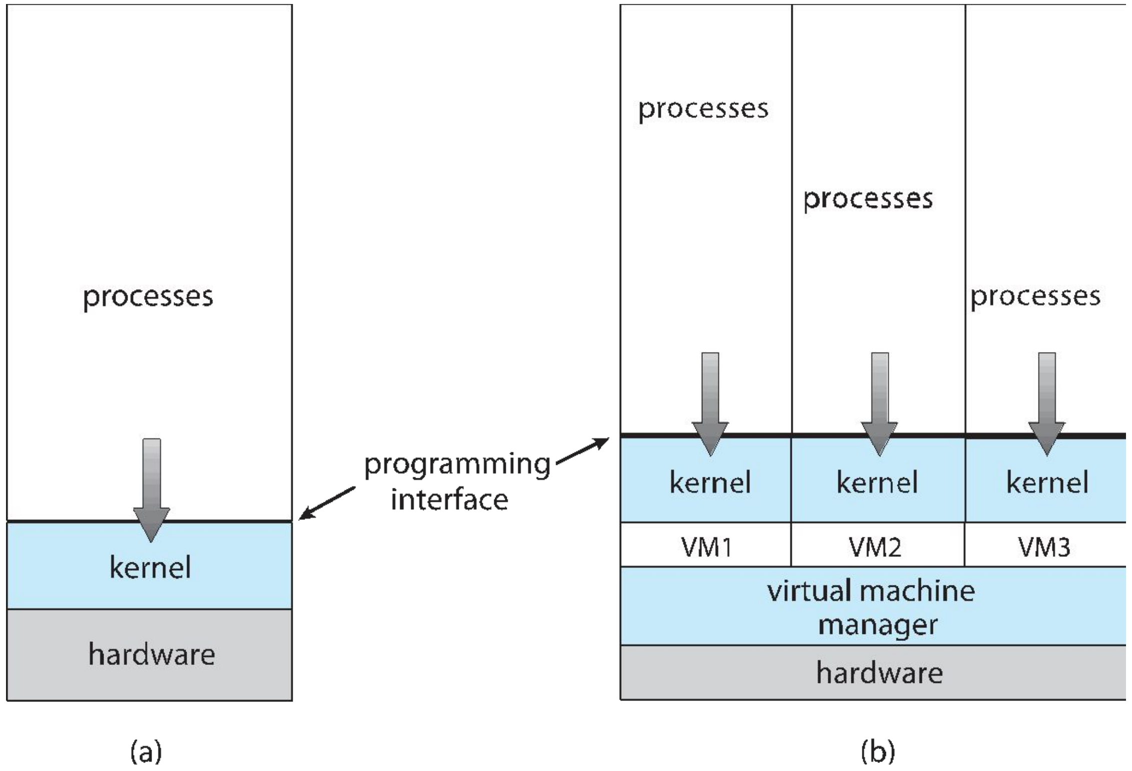
Virtualization

- Allows operating systems to run applications within other OSes
  - Vast and growing industry
- **Emulation** used when source CPU type different from target type (i.e. PowerPC to Intel x86)
  - Generally slowest method
  - When computer language not compiled to native code – **Interpretation**
- **Virtualization** – OS natively compiled for CPU, running **guest** OSes also natively compiled
  - Consider VMware running WinXP guests, each running applications, all on native WinXP **host** OS
  - **VMM** (virtual machine Manager) provides virtualization services

Virtualization (cont.)

- Use cases involve laptops and desktops running multiple OSes for exploration or compatibility
  - Apple laptop running Mac OS X host, Windows as a guest
  - Developing apps for multiple OSes without having multiple systems
  - Quality assurance testing applications without having multiple systems
  - Executing and managing compute environments within data centers
- VMM can run natively, in which case they are also the host
  - There is no general-purpose host then (VMware ESX and Citrix XenServer)

Computing Environments - Virtualization



Distributed Systems

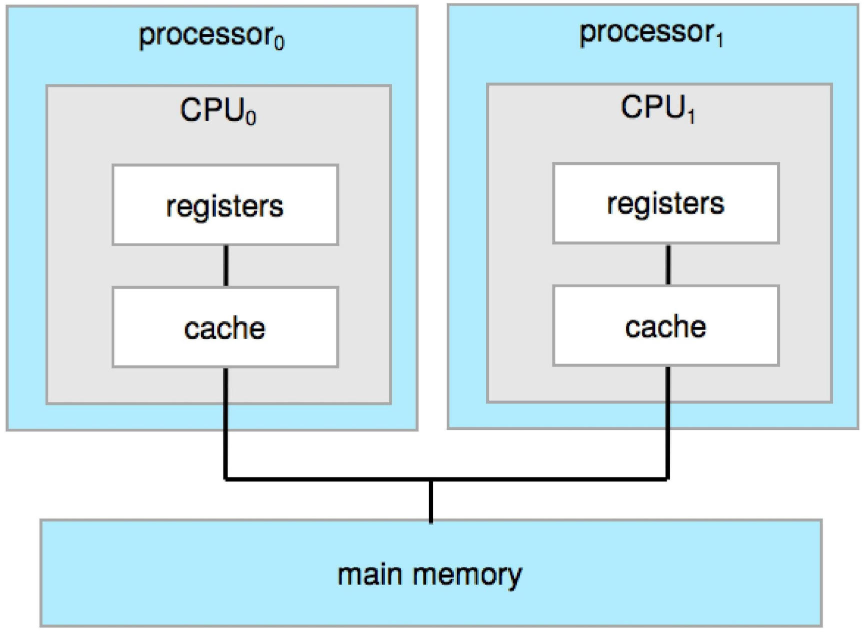
- Collection of separate, possibly heterogeneous, systems networked together
  - **Network** is a communications path, **TCP/IP** most common
    - **Local Area Network** (**LAN**)
    - **Wide Area Network** (**WAN**)
    - **Metropolitan Area Network** (**MAN**)
    - **Personal Area Network** (**PAN**)
- **Network Operating System** provides features between systems across network
  - Communication scheme allows systems to exchange messages
  - Illusion of a single system

Computer System Architecture

Computer-System Architecture

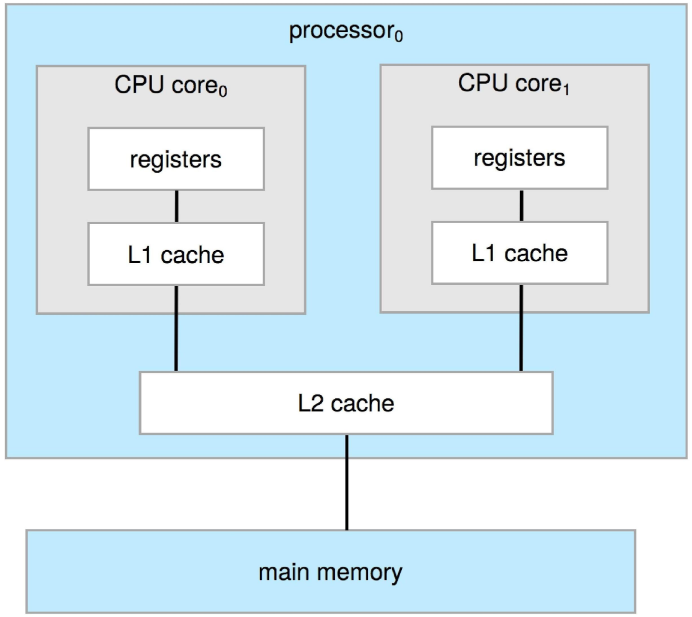
- Most systems use a single general-purpose processor
  - Most systems have special-purpose processors as well
- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    1. **Increased throughput**
    2. **Economy of scale**
    3. **Increased reliability** – graceful degradation or fault tolerance
  - Two types:
    1. **Asymmetric Multiprocessing** – each processor is assigned a specie task.
    2. **Symmetric Multiprocessing** – each processor performs all tasks

Symmetric Multiprocessing Architecture

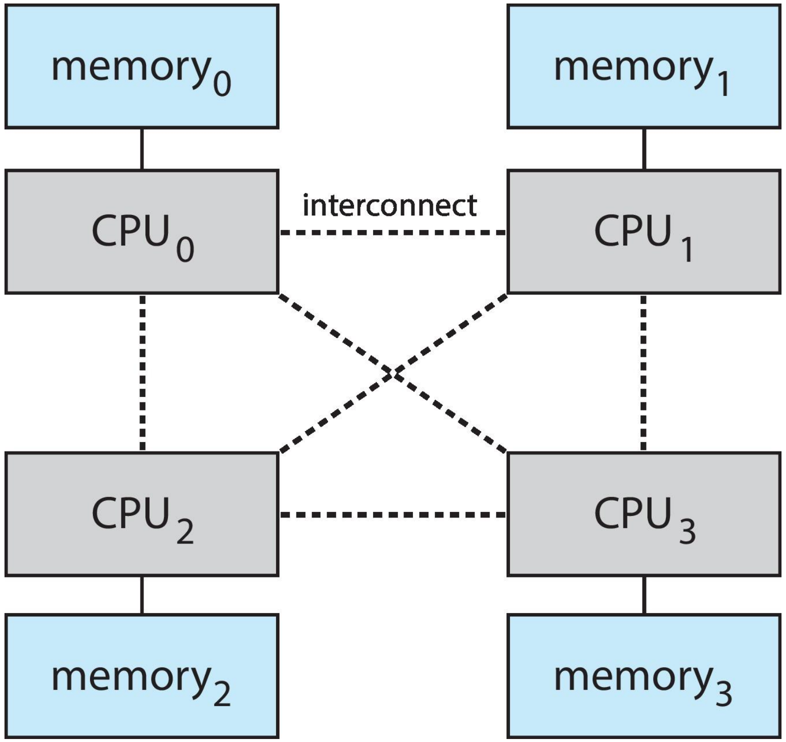


Dual-Core Design

- Multi-chip and **multicore**
- Systems containing all chips
    - Chassis containing multiple separate systems



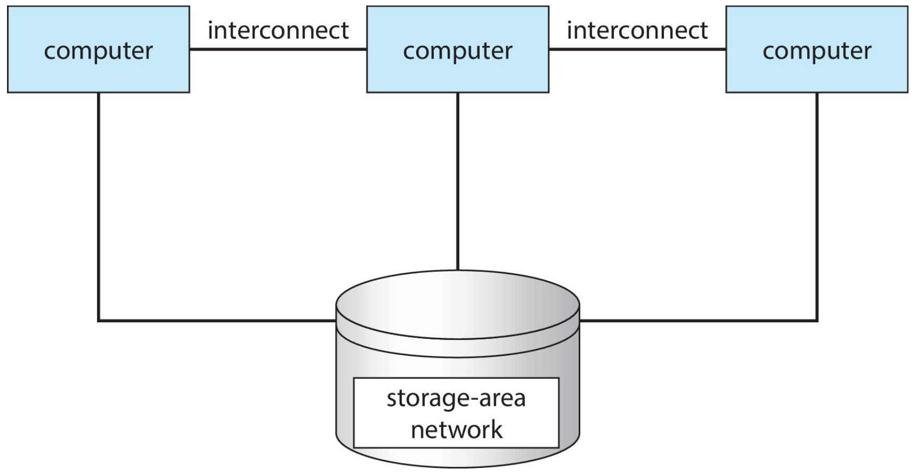
Non-Uniform Memory Access System



Clustered Systems

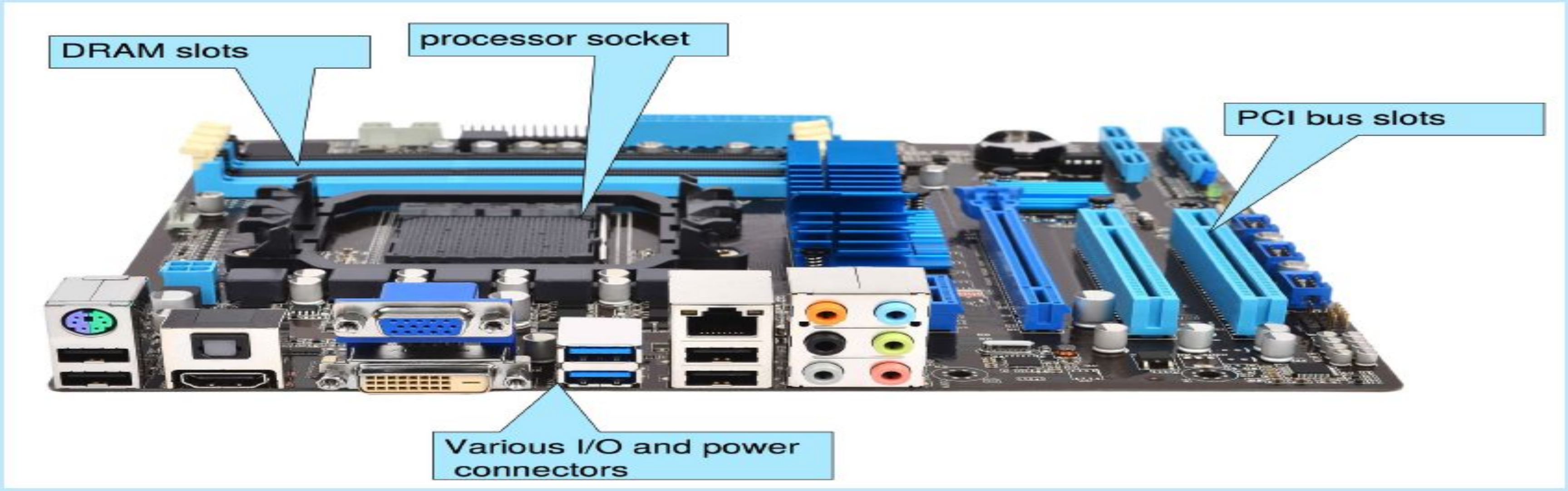
- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a **storage-area network** (**SAN**)
  - Provides a **high-availability** service which survives failures
    - **Asymmetric clustering** has one machine in hot-standby mode
    - **Symmetric clustering** has multiple nodes running applications, monitoring each other
  - Some clusters are for **high-performance computing** (**HPC**)
    - Applications must be written to use **parallelization**
  - Some have **distributed lock manager** (**DLM**) to avoid conflicting operations

Clustered Systems



PC Motherboard

Consider the desktop PC motherboard with a processor socket shown below:



This board is a fully-functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

Computer System Environments

Computing Environments

- Traditional
- Mobile
- Client Server
- Peer-to-Peer
- Cloud computing
- Real-time Embedded

Traditional

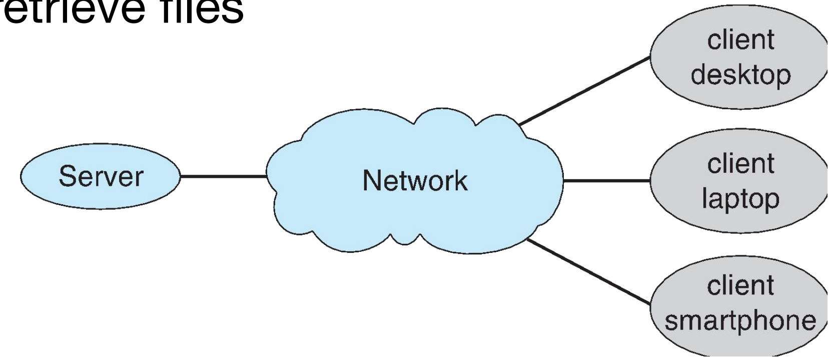
- Stand-alone general-purpose machines
- But blurred as most systems interconnect with others (i.e., the Internet)
- **Portals** provide web access to internal systems
- **Network computers** (**thin clients**) are like Web terminals
- Mobile computers interconnect via **wireless networks**
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks

Mobile

- Handheld smartphones, tablets, etc.
- What is the functional difference between them and a “traditional” laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**

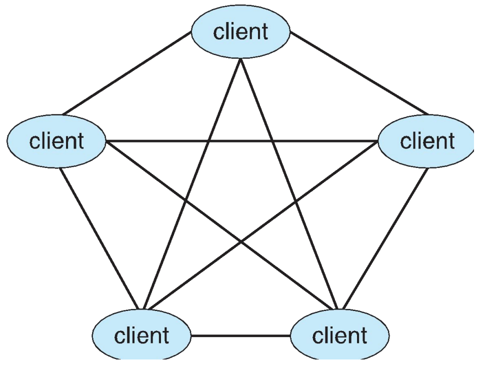
Client Server

- Client-Server Computing
  - Dumb terminals supplanted by smart PCs
  - Many systems now **servers**, responding to requests generated by **clients**
    - **Compute-server system** provides an interface to client to request services (i.e., database)
    - **File-server system** provides interface for clients to store and retrieve files



Peer-to-Peer

- Another model of distributed system
- P2P does not distinguish clients and servers
  - Instead all nodes are considered peers
  - May each act as client, server or both
  - Node must join P2P network
    - Registers its service with central lookup service on network, or
    - Broadcast request for service and respond to requests for service via ***discovery protocol***
  - Examples include Napster and Gnutella, **Voice over IP** (**VoIP**) such as Skype



Cloud Computing

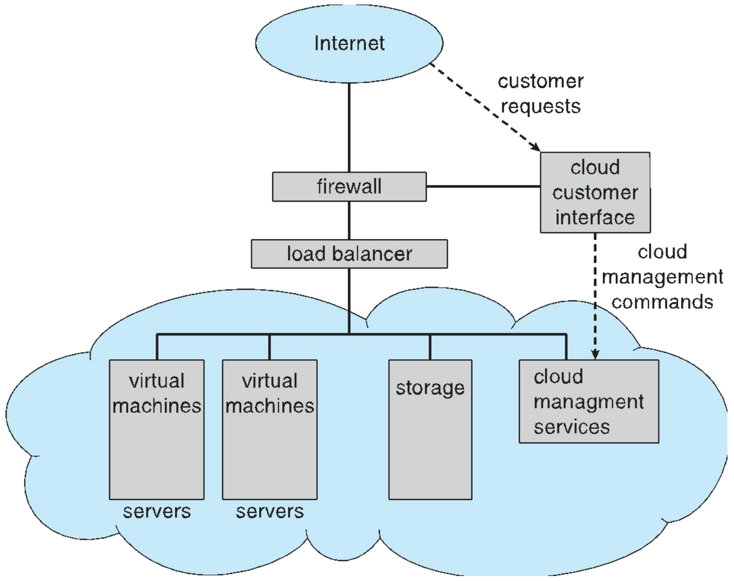
- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for it functionality.
    - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage

Cloud Computing (Cont.)

- Many types
  - **Public cloud** – available via Internet to anyone willing to pay
  - **Private cloud** – run by a company for the company's own use
  - **Hybrid cloud** – includes both public and private cloud components
  - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
  - Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
  - Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)

Cloud Computing (cont.)

- Cloud computing environments composed of traditional OSes, plus VMMs, plus cloud management tools
  - Internet connectivity requires security like firewalls
  - Load balancers spread traffic across multiple applications



Real-Time Embedded Systems

- Real-time embedded systems most prevalent form of computers
  - Vary considerable, special purpose, limited purpose OS, **real-time OS**
  - Use expanding
- Many other special computing environments as well
  - Some have OSes, some perform tasks without an OS
- Real-time OS has well-defined fixed time constraints
  - Processing ***must*** be done within constraint
  - Correct operation only if constraints met

Free and Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary **closed-source** and **proprietary**
- Counter to the **copy protection** and **Digital Rights Management** (**DRM**) movement
- Started by **Free Software Foundation** (**FSF**), which has “copyleft” **GNU Public License** (**GPL**)
  - Free software and open-source software are two different ideas championed by different groups of people
    - **https://www.gnu.org/philosophy/open-source-misses-the-point.en.html**
- Examples include **GNU/Linux** and **BSD UNIX** (including core of **Mac OS X**), and many more
- Can use VMM like VMware Player (Free on Windows), Virtualbox (open source and free on many platforms - http://www.virtualbox.com)
  - Use to run guest operating systems for exploration

The Study of Operating Systems

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BUSD UNIX, Solaris, and part of macOS.
The availability of source code allows us to study operating systems from the inside out.
Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources.
An extensive but incomplete list of open-source operating-system projects is available from https://curlie.org/Computers/Software/Operating_Systems/Open_Source/

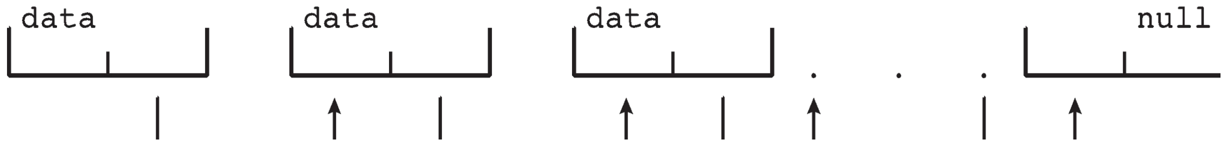
In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (http://www.vmware.com) provides a free “player” for Windows on which hundreds of free “virtual appliances” can run. Virtualbox (http://www.virtualbox.com) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Just a few years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.

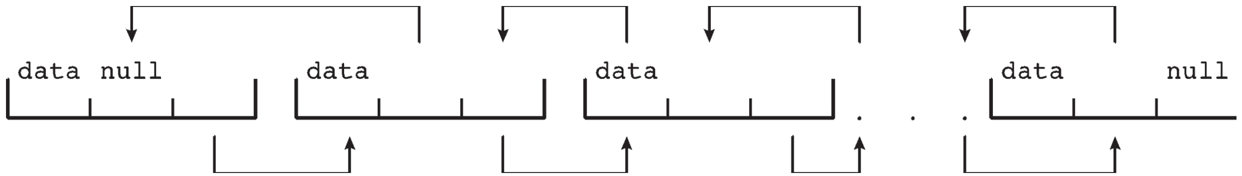
Kernel Data Structure

Kernel Data Structures

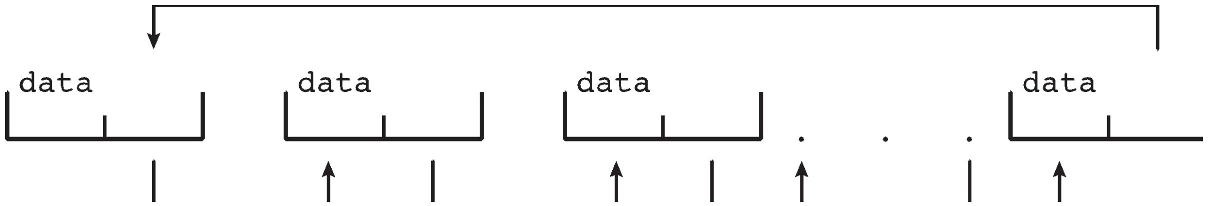
- Many similar to standard programming data structures
- ***Singly linked list***



- ***Doubly linked list***

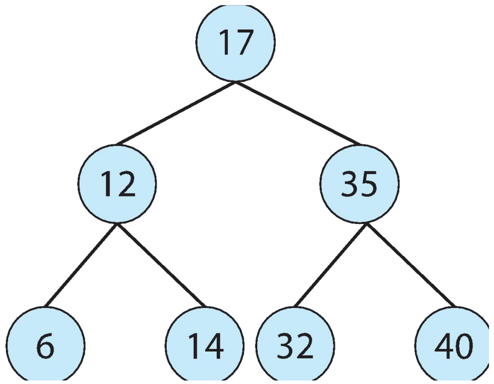


- ***Circular linked list***



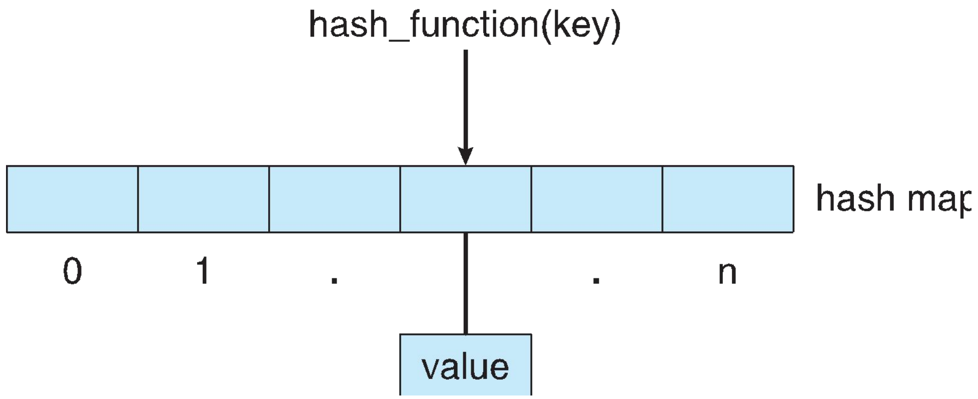
Kernel Data Structures

- **Binary search tree**
  left <= right
  - Search performance is *O(n)*
  - **Balanced binary search tree** is *O(lg n)*



Kernel Data Structures

- **Hash function** can create a **hash map**
- **Bitmap** – string of *n* binary digits representing the status of *n* items
- Linux data structures defined in ***include*** files `<linux/list.h>`, `<linux/kfifo.h>`, `<linux/rbtree.h>`



End of Chapter 1