**Midterm Review – CS4440 Operating Systems**

**Chapter 1 : Introduction**

**Operating System** : A program that acts as an intermediary between a user of a computer and the computer hardware

**Goals of an OS:**

Execute user programs and make solving user problems easier

Make the computer system convenient to use

Use the computer hardware in an efficient manner

**4 Components of an OS**

* + - 1. Hardware
      2. Operating System
      3. Application Programs
      4. Users

**Kernel** : The one program running at all times ( Core of OS)

**Volatile Memory:** ( RAM, or Random Access Memory)

- Released on Power down/Shut Down

**Nonvolatile** **Memory:** ( ROM, or Read-Only Memory)

- Saved between boot ups

**System call** : request to the OS to allow user to wait for I/O completion

**Multiprocessing** : Multiple *Physical* Processors/CPUs

**Multiprogramming** : Multiple *Software* Processes on a single core

**Degree of Multiprogramming** : Number of processes that can be executed on a single CPU.

**Dual core** : Two Cores in a single chip/CPU

each with its own register and cache

**Trap/Exception** : Software-Driven Interrupt, usually due to an error

**Dual-Mode** (aka Kernel/User Mode, System Mode, Privileged Mode)

has a user mode where user programs do not have access to hardware

has a privileged mode

**Storage (KB ,MB ,GB ,TB ,PB )**

1 bit = “1”/”0”

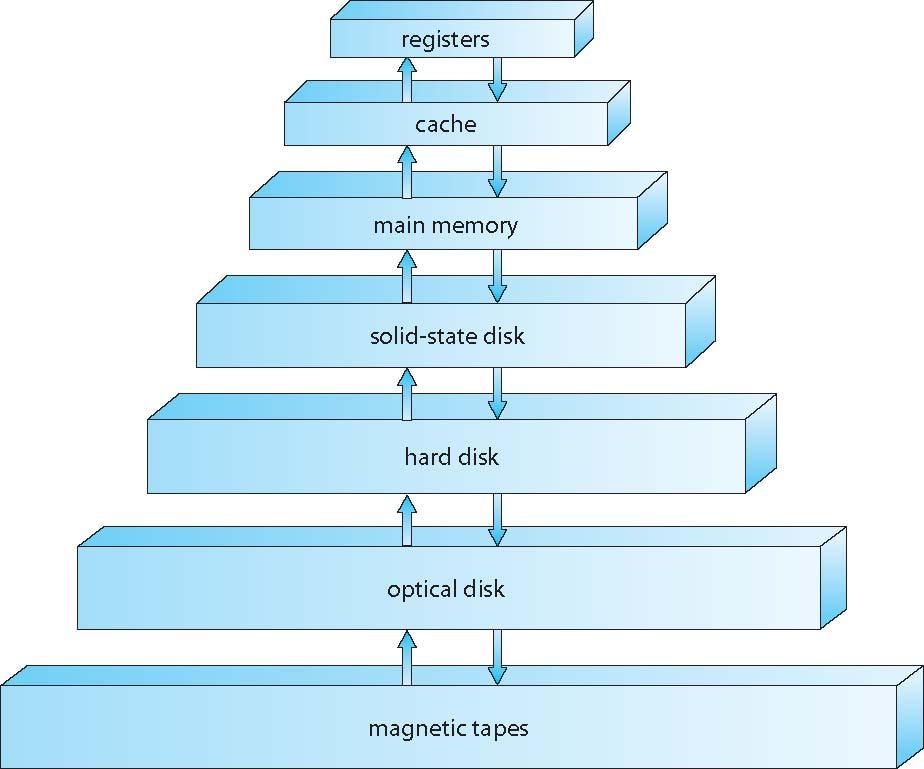
1 byte = 8 bits

1 KB = 1024 bytes

1 MB= 10242 bytes

1 GB= 10243 bytes

**Storage Hierarchy** ( Top 3 are volatile - Registers, Cache, Main)



**Busy Waiting** : Running on the CPU but waiting for a condition

**Bitmaps** : Used to represent the availability of a resource:

e.g. – 1 0 1 0 0 0 0 0 , where the ith element represents the availability of a resource ( 0 = not , 1 = is )

**Mobile OSes** :

**Android** (open-source)

**iOS** (proprietary)

**Open Source OSes** : Linux, FreeBSD, UNIX, Solaris (Sun Microsystems) w/ green threads

**Real Time Embedded System**: serves one a function without GUI. Ex: microwave, fridge

**Chapter 2 : Operating System Structures**

**Services of an Operating System**

User Interface, Program Execution, I/O Operations, File-System Manipulation, Communications, Error Detection, Resource Allocation, Accounting, Protection and Security

**Command Interpreter/CLI** = shell ( takes commands from the user)

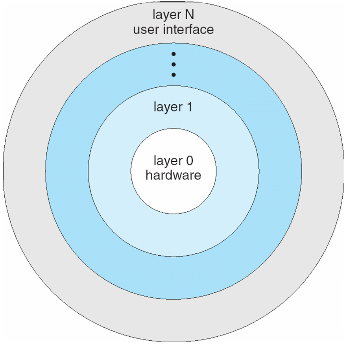
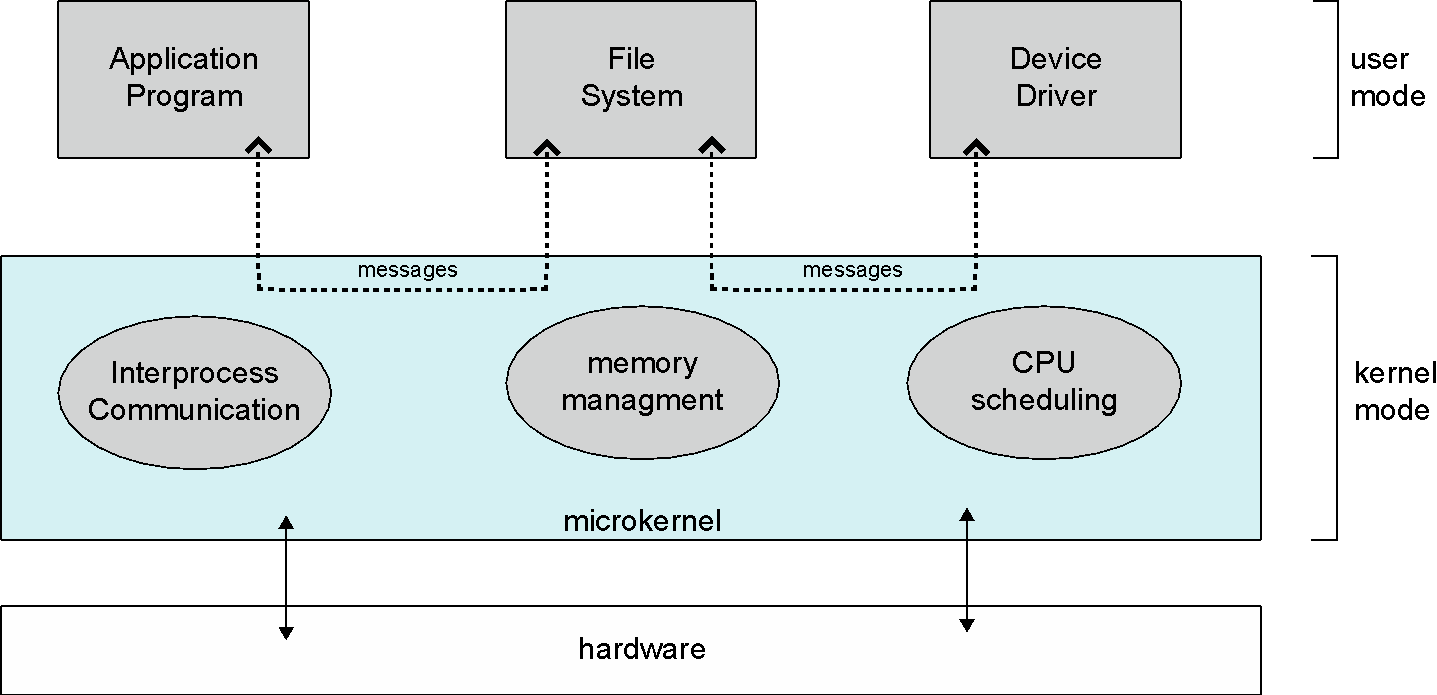
e.g BASH ( Bourne Again Shell )

System Call

* calling OS from program
  + via API (interface)
* library needed for fork()
  + #include <unistd.h>
* when fork() ends, it returns to library unistd

**Microkernels** ( Solaris, Windows, MacOS, Linux)

A smaller, simplified kernel with less code and overhead

**Layered Approach** **Modular Approach**

**Layered:** Everything is in the system, so it’s faster, but there’s too many layers

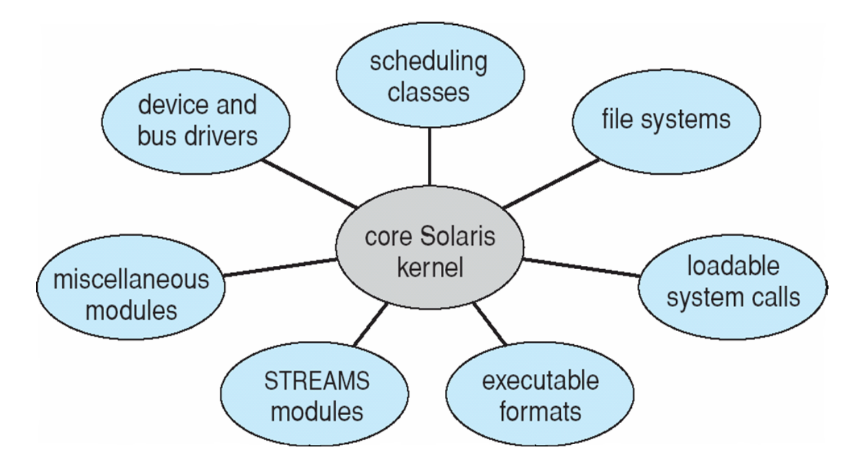
Layers: hardware-> OS -> I/O -> File System -> Command Interpreter

**Modular:** Can change user mode applications without messing with the kernel code

**Darwin** is the MacOS kernel.

**Bootstrap program/Firmware** : Software tied to hardware, loaded on power-up, stored in ROM/EPROM (Erasable Programmable)

Solaris:



**Chapter 3 : Processes**

**Aqua** = MacOS interface

**API** = Application Program Interface

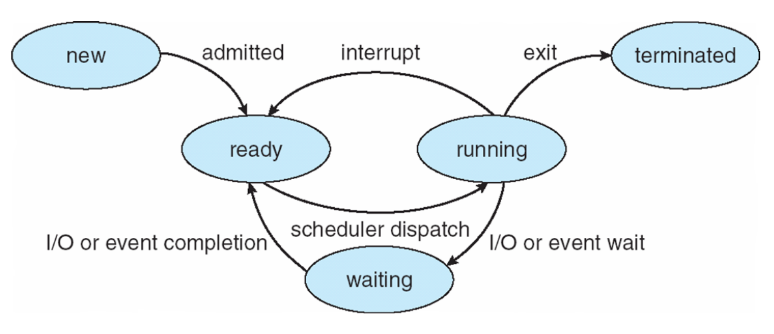
**#include <unistd.h>** :: for fork(), I/O operations

it’s an interface/API from the C library

**Heap** : Dynamic Memory

**Stack** : Static Memory

**Process State Diagram**

* + - 

**Process Control Block (PCB)** – Contains information about each process

**Context Switch** : Switching Processes:

1. Process information is stored
2. … //TODO

**Multiprocessing in Mobile OSes**

Apple/iOS : has the power to, but limits it

Android: no limits

**init()** = first process to run

//TODO

**fork()** = 2n calls, where n = number of times forked

**Daemon** : process that runs in the background

**Inter-Process Communication (IPC)**

*Shared Memory:* Share a location in memory = easier to synchronize

*Message Passing:* Send updates via messages = harder to sync

**Chrome Browser Multiprocess** (1 Process Per Webpage)

Sandboxed? Yes.

**Pipes**

**Named Pipes** – Retains Info

**Ordinary Pipes** – Process, disappears after process is done

**Sockets (Port Numbers)**

< 1024 = Well known/Reserved (CANNOT BE USED)

>1024 = Host numbers

***>>End Of Chapter Exercises (?)***

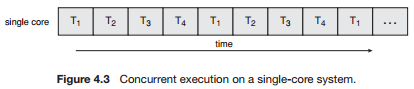
**Chapter 4 : Threads**

**Thread –** Shared memory

**Process -** Has it’s own memory

**4.10** *In Chapter 3, we discussed Google’s Chrome browser and its practice of opening each new website in a separate process. Would the same benefits have been achieved if instead Chrome had been designed to open each new website in a separate thread? Explain.*

**Answer:** If it was designed to be a thread per website browser, one webpage crashing will result in all the webpages failing.

**Formula:** (pg. 166)

**Concurrent Processes –** Take turns running each program

**//TODO**

**Parallel Processes –** Both processes run at once

**//TODO**

**Exercise: Can a program have concurrency but not parallelism?**

**Answer:** Yes. (e.g. above example)

**P- Thread** (POSIX Thread)

**POSIX** (Portable OS Interface) – Standard for Operating Systems

Thread Join = Process Wait

**Java Threads** managed by JVM/Host machine(pg. 176)

**Implicit Threading** – The creation of threads during runtime, done by library (NOT HANDLED BY PROGRAMMER)

Examples: Thread Pools, openMP

**Thread Pools** – Create a number of threads in a pool where they await work

**Pragma** (OpenMP) (pg. 181)

Runs parallel threads

#pragma omp parallel

Create as many threads as there are cores

#pragma omp parallel for

for(i=0;i<N;i++) {

c[i] = a[i] + b[i];

}

Run for loop in parallel

**Grand Central Dispatch** ( for Apple)

uses caret ^{} for parallel threads

two types of queue: serial (main queue) and concurrent (low, default, high queue)

**Signal** : Signals the end of the wait, or when the program is out of the critical section

**Critical Section** : Part of the shared memory

**Lightweight Process** (LWP) (pg. 187)

Data Structure

Thread Library

Virtual Process

**Upcall** : Call made by the kernel

the kernel must inform an application about certain events

**Linux Threads**

Clone() -> creates threads

**Chapter 5 : Introduction**

**Producer + Consumer** (pg. 206)

Producer adds to the buffer

Consumer consumes from the buffer

**Race Condition** : If the processes are not synchronized, the process that finishes last gets to change the value of the critical section.

**Solution to Critical Section Problem**

**Mutual Exclusion** (MutEx)

**Progress**

**Bounded Waiting**

**Peterson’s Solution** (uses Boolean flag and Int Turn )

***P***i (process)

do {

flag[i] = true;

turn = j;

while (flag[j] && turn = = j);

critical section

flag[i] = false;

remainder section

} while (true);

**Semaphores** // TODO

signal()/ wait()

creates deadlocks/starvations pg. 217

Deadlock - situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

starvation - happens when low priority never executes because higher priorities always come

**Priority Inversion** ( L -> M -> H)

Low priority processes pushed back

(e.g. Code that didn’t print date for 6 years )

**Dining Philosophers/Classic Read – Writers problems**

Problems it can create

**Chapter 6 : CPU Scheduling**

**Cpu Scheduling/Short Term Scheduling**

Schedules tasks

**Goal of CPU Scheduling**

**Maxmizing:** CPU Usage

CPU Throughput

**Minimizing:** Turnaround time

Waiting time

Response Time

**Calculate Turnaround + Waiting Times //TODO**

**Avg Waiting Time Schedule //TODO**

**Problems with**

First Come First Serve (FCFS)

problem is length of process

Round Robin (RR)

* treats processes same as First-Come-First-Serve (FCFS)
* example of dynamic RR
  + if process is short -> goes to short quantum
  + if need more memory -> goes to higher quantum
  + else it goes to background -> FCFS

Shortest Job First (SJF)

* + how do you know how long is each process?
    - we don’t know
  + issue is when many short burst processes
    - long process will be skipped frequently

Shortest Remaining Time (SRT)

* “Preemptive” = Priority
* P1 gets bypassed when P2 arrives, because P2 has shorter process time
* Remainder of P1 is less than P3, so P3 is put on hold until P1 is done

Priority Scheduling

**Multilevel Queue** -> Process has a predefined priority

* background
  + for batches
  + FCFS
* fore ground
  + interactive
  + RR

**Multilevel Feedback** -> Processes can switch from one priority to another (e.g. if quantum exceeds an amount)

**Aging** - As data age, it moves up in priority