# G22.2250-001
# Operating Systems


| | |
|---|---|
| Instructor: | Vijay Karamcheti (vijayk@cs.nyu.edu) |
| Lectures: | Tuesdays,<br>5:00pm-6:50pm, 109 CIWW |
| Office Hours: | Tuesdays, 7:00pm – 8:00pm<br>715 Broadway, Room 704, 8-3496 |
| Mailing List: | g22_2250_001_fa07@cs.nyu.edu |


http://www.cs.nyu.edu/courses/fall07/G22.2250-001/index.htm

## Outline

- Target audience for course
- Introduction
  - what is an operating system?
  - why you should care?
- Course organization, policies and guidelines
  - topics
  - workload and expectations
  - collaboration policy
- Overview of operating system functionality
  - A brief history of operating systems
- Computer system structures
- Operating system structures

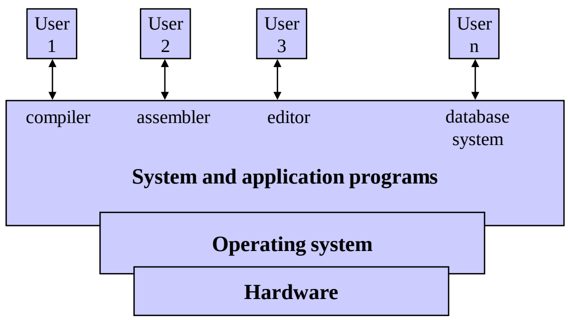
*[ Silberschatz/Galvin/Gagne: Chapters 1, 2, 23]*

## Target Audience for Course

- This is an introductory Operating Systems (OS) course
- Suitable for students who …
  - … have not taken an OS course in a Computer Science department so far
    - High-level courses focusing on “use” of an OS do not count
  - … have not had exposure to hands-on implementation of OS concepts
- Students with prior exposure to key OS concepts and their implementation may wish to consider more advanced courses:
  - G22.3250-001: Honors Operating Systems (usually offered in the Spring semester)
  - G22.2620-001: Networks and Distributed Systems



## What is an Operating System?




- An operating system is
  - a *government:* legislates/enforces proper use of system resources
  - a *resource allocator*
  - a *control program:* prevents errors and improper use

## Reasons for an Operating System

- An operating system provides
  - *convenience* for the user
  - *efficiency*
    - particularly important for large, shared multi-user systems
    - important even in dedicated single-user systems
      - to balance the needs of different kinds of tasks
  - a simple, more powerful *virtual machine*
    - convenient abstractions for hardware resources (e.g., disks)
  - *sharing* of resources
  - *isolation/protection* among user programs



## Why Study Operating Systems?

- Arguments against:
  - “very few OS designers/implementers needed”
  - “all I need to know is in the manual pages”
  - “everybody is going to run Windows anyway”
- Arguments for:
  - need to know about (*large*) system design in general
    OSes include several important design/optimization problems
    - resource sharing and management
    - protection and security
    - flexibility, robustness, and performance
    - design of good interfaces
  - growing need for OSes
    - embedded systems
    - several large applications contain mini-OSes
  - crucial for understanding application-hardware interactions

## What This Course is About

- Understanding the *general principles* of OS design
  - focus on general-purpose, multi-user, uniprocessor systems
  - emphasis on *widely applicable concepts*, rather than the features of any specific OS
    - protected kernels
    - processes and threads
    - concurrency and synchronization
    - memory management and virtual memory
    - file systems
- Understanding *problems*, *solutions*, and *design choices*
- Understanding *implementations of these concepts* in a non-trivial instructional OS (Nachos)



## What This Course Does Not Cover

- Specific features of commercial OS products
  - "how do I do X in operating system Y?"
- Topics deferred to advanced courses
  - Networking, Network-accessible File Systems (e.g., NFS)
  - Analytical modeling
  - Transactions and Database OSes
  - Distributed Oses
- That said, time permitting we will cover some advanced topics:
  - Virtualization technologies
  - Log-based file systems and other technologies suitable for use with new storage technologies

## Tentative Course Schedule

Lectures 1-2: Overview

Lectures 2-7: Process management

processes and threads, scheduling, synchronization, deadlocks

Lectures 8-12: Storage management

memory management, virtual memory, file systems

Lectures 12-14: I/O systems

advanced topics: VMs, log-structured file systems

Lectures 14-15: Protection and security

Final Exam: December 18, 2007



## Assessment of Student Background

- Programming languages
  - Java
  - C/C++
- UNIX environments
  - *commands*: ls, cat, mkdir, cd
  - *editors*: vi, emacs
  - *program development environments:*
    - compilers (gcc) and makefiles
    - debuggers (dbx, gdb)
- Computer systems organization
  - CPU, RAM, disk, cache
  - interrupts, DMA

## Workload and Expectations

- You should plan on putting in 6-8 hours of effort/week
  - Classes and assigned readings
  - Six programming projects: **50%**
    - each due approximately 2 weeks after it is handed out
    - additional info on next few slides
  - Final exam **50%**

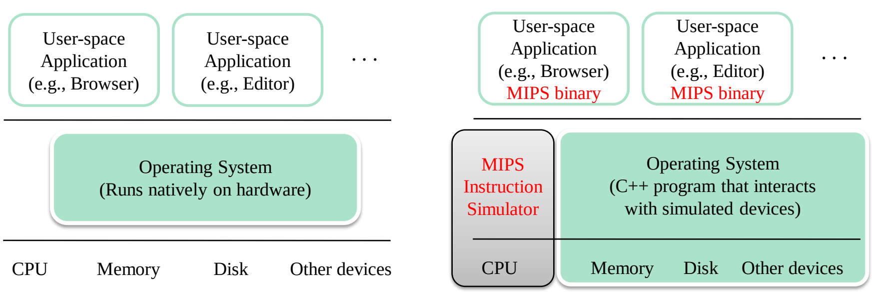
- Other expectations
  - Basic familiarity with
    - programming in C/C++
    - UNIX tools and development environment
      - command familiarity
      - editors (vi, emacs), compilers (gcc), debuggers (gdb), makefiles (GNU make)
  - Expect you to pick up necessary background on your own
    - course web page has links to online tutorials



## Nachos

- An “instructional” operating system
  - developed by Thomas Anderson and others at U.C. Berkeley
  - has seen widespread use in undergraduate/graduate classes since 1995
  - what is it: a user program that runs on a standard OS (Solaris, Linux)
    - all the features of a real OS, but **much** simpler
    - ~8000 lines in a restricted subset of C++

*Nachos Stack*



## Nachos Projects

- Course programming projects
  - flesh out the baseline implementation of various OS modules
    - *protected kernels*, *threads and synchronization*, *multiprogramming support*, *I/O (files)*, and *virtual memory*
  - each project builds upon what you have already done
    - at the end, you will be able to see execution of multiple user programs on shared hardware in a protected fashion
  - effectively, *you would have built a non-trivial OS*

- Nachos project guide (also available on the course web site)
  - Lab details, grading policies, etc.

- Computing resources
  - On department Sparc/Solaris machine: access*.cims.nyu.edu
    - Follow instructions on the course web site
    - Support programs are pre-installed
  - Any Linux machine that you have access to
    - Download Nachos code (for Linux) from the course web site



## Nachos Projects (cont'd)

- Nachos projects and documentation borrow heavily from resources developed by Jeff Chase and others at Duke University
  - Appropriately customized for NYU

Some suggestions for making your life less stressful …

- Allocate time for reading the Nachos code
  - Read the overview documents and the project guide before starting
  - The project guide tells you which directories/files you need to look at

- Carefully design your solution before you start coding
  - Getting the logic right is as hard as (if not harder than) coding/debugging

- Start early
  - If you plan to work only the last 1-2 days, you will not have enough time to complete the project

## Policy on Collaboration

- I expect you to adhere to the department's policies/guidelines on Academic Integrity
  - http://www.cs.nyu.edu/web/Academic/Graduate/academic_integrity.html
- Collaboration encouraged on every aspect of the class, except the final exam
  - Okay to discuss projects/approaches with others in class and co-develop high-level strategy for solution …
  - … but what you hand in must reflect your own effort



## Course Resources

- Text book(s)

  (required) Silberschatz/Galvin/Gagne, *Operating System Concepts*, 7th Ed.
  - Can use 5th or 6th Edition as well
- Nachos project guide
- Course web page:
  - http://www.cs.nyu.edu/courses/fall07/G22.2250-001/index.htm
- Class mailing list: g22_2250_001_fa07@cs.nyu.edu
  - send questions of general interest here
- E-mail: vijayk@cs.nyu.edu

## Outline

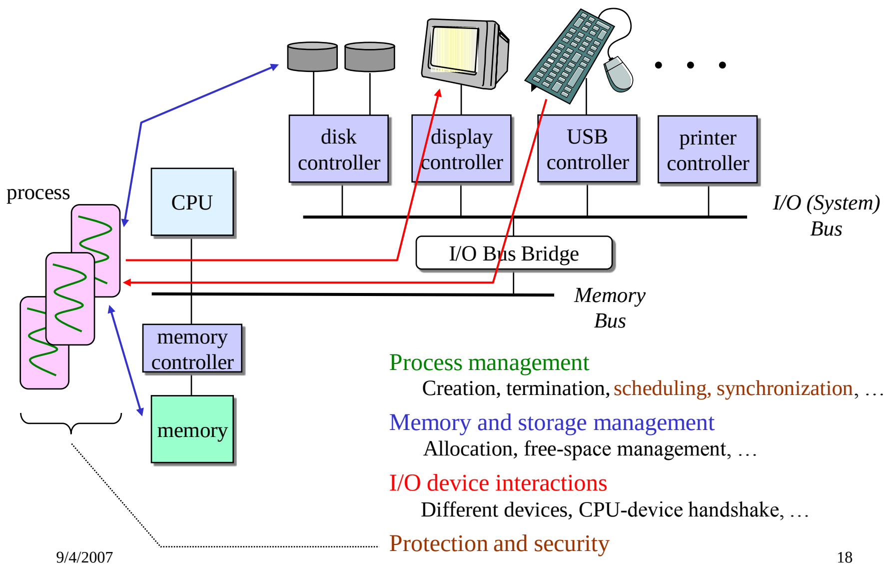
- Target audience for course
- Introduction
  - what is an operating system?
  - why you should care?
- Course organization, policies and guidelines
  - topics
  - workload and expectations
  - collaboration policy

- Overview of operating system functionality
  - A brief history of operating systems
- Computer system structures
- Operating system structures

*[ Silberschatz/Galvin/Gagne: Chapters 1, 2, 23]*



## Overview of Operating System Functionality




Process management
Creation, termination, scheduling, synchronization, …

Memory and storage management
Allocation, free-space management, …

I/O device interactions
Different devices, CPU-device handshake, …

Protection and security

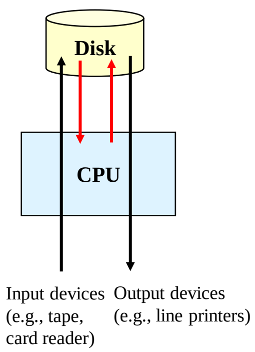
## A Brief History of OSes

- 1950's: No OS
  - Bare machine, single user
    - a button which executed a bootstrap loader
    - input via paper tape, or punched cards
  - Main perceived problems
    - human actions were slow; inefficient use of expensive hardware
- Early 1960's: Batch systems
  - Reduce set-up time by batching jobs with similar requirements
    - load jobs (punched cards) onto magnetic tape
    - process jobs on tape serially
    - output to tape
    - print output tape
  - Main perceived problem
    - turn-around time of several days
    - CPU often underutilized
      - most of the time spent reading and writing from tape



## Two Innovations

- Resident monitor for automatic job sequencing
  - "control cards" that eliminated operator involvement
    - mount this tape, compile, run
  - First instance of a primitive operating system
    - Example: IBM's Fortran Monitor System
- Spooling to improve CPU utilization
  - Use of *disks* to buffer input/output to tapes
    - disks are random-access I/O devices
  - Overlapped I/O and computation
    - one job's I/O can be overlapped with another's computation
  - Need for independent I/O controllers
    - CPU: starts I/O operation; continues computation
    - Controller: does I/O; interrupts CPU



## A Brief History of OSes (cont'd)

- Mid-1960's: Multiprogrammed systems
  - Many programs simultaneously in memory
    - objective: to keep CPU busy
    - OS switches between user processes
  - How to ensure that these programs do not interfere with each other?
    - memory protection
    - privileged instructions
- Mid-to-late 1960's: Time sharing and interactive systems
  - Programs could wait for I/O for an arbitrary time
    - CPU switched to another job
  - However, resident jobs took up valuable memory
    - needed to be swapped out to disk
    - technique that was developed to support this: virtual memory
- Several research OSes: CTSS, MULTICS at MIT, Atlas at Manchester U.



## OS Requirements and Solutions in the Late 1960s

Requirements

- Multiprogramming
  - memory allocation and protection
  - I/O operations were responsibility of OS
- Interactive systems
  - scheduling issues
  - swapping, or virtual memory
- Users wanted permanent files
  - hierarchical directory systems

Solutions

- OS structure specialized to hardware
- Examples
  - IBM: OS/360
  - CDC: Sipros, Chippewa, NOS
- Large in size, very complex
- WHY?

## UNIX (early 1970s)

- Originally developed at Bell Labs for the PDP-7
  - Ken Thompson
  - Dennis Ritchie
- Smaller and simpler
  - process spawn and control
    - each command creates a new process (activity)
  - simple inter-process communication
  - command interpreter (shell) not built in: runs as another process
  - files were streams of bytes
  - hierarchical file system
- Advantages
  - written in a high-level language
  - distributed in source form
  - powerful OS primitives on an inexpensive platform



## A Brief History of OSes (cont'd)

- 1980's: Personal computers
  - Originally: Single-user, simplified OSes (MSDOS)
    - no memory protection
  - Now run sophisticated OSes (Windows NT/2000/XP, Linux)
  - Innovation: Windowing systems
    - Graphical interface, mouse control
- 1990's: Multiprocessors, Networks of workstations
  - High-speed network connections
  - Client-server systems
    - File systems
    - Remote windowing systems
  - Differentiation based on workload
    - Client versus server
    - General processing versus transaction, multimedia processing

## (2000s and) The Future

- Distributed systems
  - network is invisible
- Micro-kernel and extensible OSes
  - support multiple OS flavors
    - e.g., Mach, Amoeba, Windows NT
  - allow insertion of application-specific functionality
  - Hypervisors and virtual machines
- Embedded devices and network computers
  - computer runs a very thin OS (Java Virtual Machine)
- Web Operating Systems
  - standard protocols (HTTP, SOAP)
  - container environments (J2EE, .NET)

- Unfortunately, we will not talk about these in this course
  - Talk to CS systems faculty for research opportunities to learn more
    - Grimm, Subramanian, Li, Kedem, Gottlieb, and me



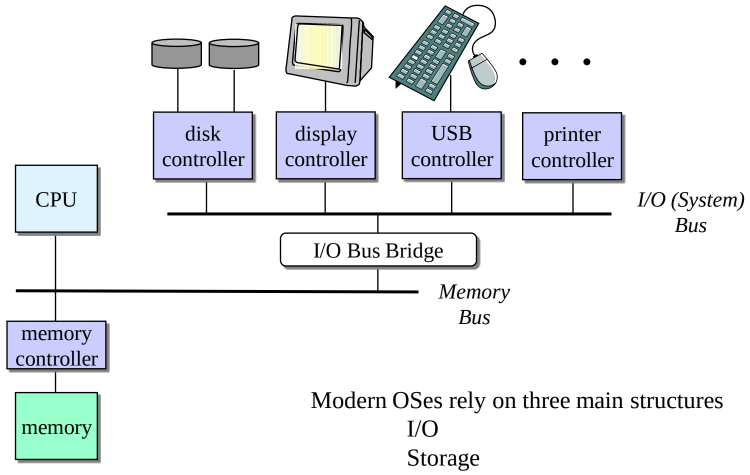
## Outline

- Target audience for course
- Introduction
  - what is an operating system?
  - why you should care?
- Course organization, policies and guidelines
  - topics
  - workload and expectations
  - collaboration policy

- Overview of operating system functionality
  - A brief history of operating systems
- Computer system structures
- Operating system structures

*[ Silberschatz/Galvin/Gagne: Chapters 1, 2, 23]*

## The Hardware of a Modern Computer System




Modern OSes rely on three main structures

I/O

Storage

Protection



## Computer-System Structures (1): Input/Output

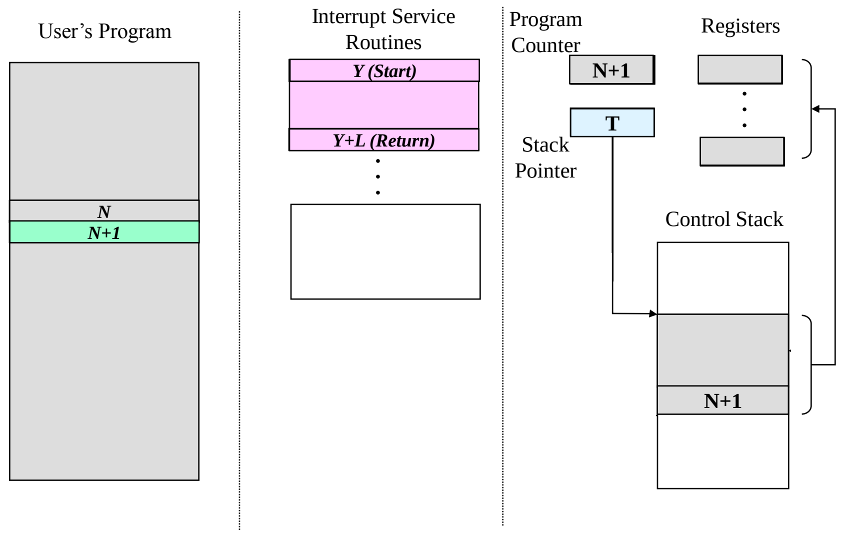
- Device controllers
  - special-purpose *processors*
  - local buffer *storage*
  - controllers contain *registers*
    - control (write-only)
    - data (read-write)
    - status (read-only)
- How do the CPU and the device controllers communicate?
  - instructions
    - read/write I/O addresses (e.g., video memory)
    - registers in I/O controllers addressed as memory
  - interrupts
    - device controllers can interrupt the CPU

## Interrupt Handling

- Interrupts are "asynchronous requests for service"
  - signal on a wire connecting the devices
- When an interrupt occurs, the CPU
  - preserves the present CPU state
    - this includes its registers and program counter
  - forces execution of code at an interrupt address
    - this may be dependent on the source of the interrupt
    - typically, table-driven: a table stores addresses of *interrupt handlers*
      - indexed by the interrupt number (ISR)
  - interrupt handlers
    - perform the requested service
    - selective processing of other interrupts
      - e.g., only higher-priority interrupts may be handled
  - resumes the interrupted program
- Most modern OSes are interrupt-driven

9/4/2007 29

## Interrupt Handling (contd.)




9/4/2007 30

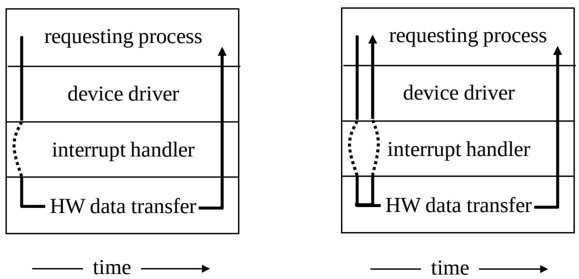
# Interrupts vs. Traps

- Interrupts
  - asynchronous
  - triggered by devices outside the CPU
- Traps
  - synchronous
  - triggered by special instructions in user program
- Other than the above, handling of interrupts and traps is identical
- Traps are the hardware mechanism for implementing system calls



# I/O Operation

- Two approaches: Synchronous and Asynchronous




- Problem with the above schemes: CPU handles all I/O
  - it can spend all its time doing interrupt processing
    - disk I/O , network I/O, video I/O

## Solution: Direct Memory Access (DMA)

- The main idea: add a *special device* to "intervene" between the device controller and the system's memory

- Operation
  - the CPU tells this DMA controller
    - the "chunk"' size to be transferred
      - e.g., 128 - 4096 bytes (sectors) for disks
    - the starting address in memory where this chunk ought to be stored
  - the DMA controller
    - accesses the secondary device via its controller
    - transfers the chunk from the device to system memory (and vice-versa)

- Benefit: Interrupts are now less frequent
  - at the level of chunks of data: only to indicate completion
  - hence, CPU can do a lot of work between interrupts



## Memory-Mapped I/O

- Traditionally, the CPU could directly access only main memory
  - all other devices are handled via controllers
  - accessed using special I/O instructions

- In most recent systems
  - controller's registers are mapped into RAM space
    - results in uniform treatment of the I/O devices
      - can be handled via memory management procedures
      - all addressing is to RAM space
      - DMA access, interrupt handling, polling, …
  - controller's buffers are mapped into RAM space
    - makes sense if the I/O is to a device that is particularly fast
    - e.g., a CRT screen where each pixel is an addressable location in RAM

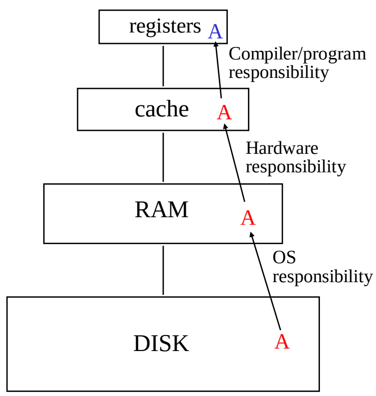
## Computer-System Structures (2): Storage

- Primary storage: *Main memory* (volatile)
  - accessed directly using load/store instructions
    - 1 cycle (registers), 2-5 cycles (cache), 20-50 cycles (RAM)
    - *before*: only one outstanding memory operation, CPU waits for completion
    - *now*: several outstanding operations
- Secondary storage: *Disks* (non-volatile)
  - accessed using a disk controller
  - supports random access but with non-uniform cost
- Tertiary storage: *Tapes, Optical disks* (non-volatile)
  - typically used only for backup
  - very inefficient support for random access
- Organized as a hierarchy
  - small amount of faster, more expensive storage closer to the CPU
  - larger amounts of slower, less expensive storage further away



## Storage Hierarchy

- Rationale
  - keep CPU busy: lots of fast memory
  - keep system cost down
- How does it work
  - *caching:* upon access, move datum or instruction and its neighbors into higher levels of the hierarchy
  - *replacement* when a level fills up
  - copies need to be kept coherent
- Why does it work
  - Real programs demonstrate *locality*
    - e.g.: rows and columns of a matrix
    - e.g.: sequential instructions
  - once a *datum* or *instruction* is used, things "near" them are likely to be used "soon"



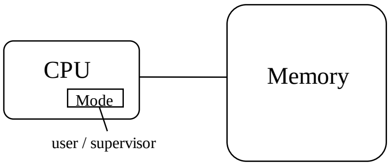
## Computer-System Structures (3): Protection

- Goal: Prevent user processes from accidentally/maliciously damaging
  - the OS structures
  - parts of other process's memory space
  - other user's I/O devices
- Mechanisms address different ways in which protection breaks down
  1. dual-mode operation
     - Prevent user process taking over part of the OS and using this to overwrite other processes or even modify the OS itself (as in MS-DOS)
  2. privileged instructions
     - Prevent user process intervening in I/O of another process via control of the I/O handlers and indirectly causing damage
  3. memory protection
     - Prevent user process directly accessing another user process' storage
  4. CPU protection via timers
     - Prevent hanging the OS -- e.g., via an infinite loop



## Protection Mechanisms (1): Dual-mode Operation and Privileged Instructions

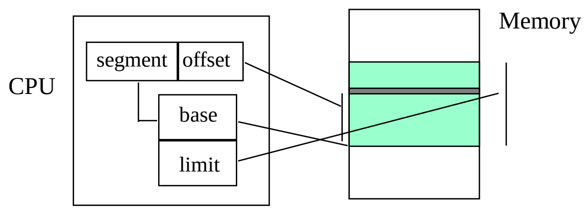
- Dual-mode operation
  - *supervisor* and *user* modes
  - system starts off in supervisor mode and reenters it for interrupt processing
  - operating system gains control in supervisor mode




- Privileged instructions
  - restrict use of certain instructions to supervisor mode
    - I/O, including interrupt control
      - exception is instructions which generate interrupts
      - may be done by memory mapping
    - affect memory mapping
    - affect CPU mode (user/supervisor)
  - hardware support crucial for performance and for atomicity

## Protection Mechanisms (2): Memory Protection

- Basic method: Memory is divided into segments




- Furthermore
  - logical addresses are mapped to physical addresses
    - provides sharing, etc.
  - hardware support for address mapping
  - a memory protection violation is detected
    - user process *traps* to (interrupts) the OS



## Protection Mechanisms (3): Timers

- OS code can enforce policies only if it gets a chance to run

- Timers maintain a count of elapsed (system) clock ticks
  - when timer expires, the CPU is interrupted → run the OS code

- Used for
  - interrupting hung processes
  - context switching in time-shared systems

- Access to timers is (usually) privileged
  - WHY?

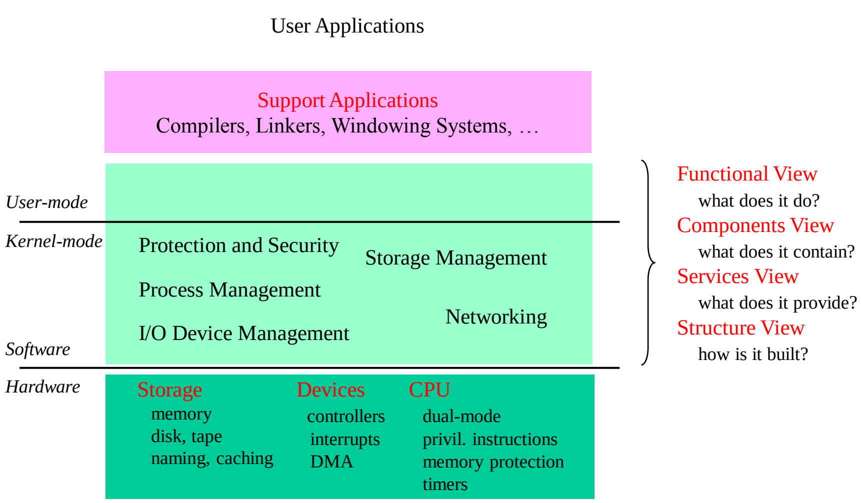
# Outline

- Target audience for course
- Introduction
  - what is an operating system?
  - why you should care?
- Course organization, policies and guidelines
  - topics
  - workload and expectations
  - collaboration policy
- Overview of operating system functionality
  - A brief history of operating systems
- Computer system structures
- Operating system structures

*[ Silberschatz/Galvin/Gagne: Chapters 1, 2, 23]*



# Hardware and OS Structures



## OS Views (1): Functional View

- What are the functions performed by an OS?

- Explicit operations
  - program execution and handling
  - I/O operations
  - file-system management
  - inter-process communication
  - exception detection and handling
    - e.g., notifying user that printer is out of paper

- Implicit operations
  - resource allocation
  - accounting
  - protection
    - e.g., maintaining data integrity, logging invalid login attempts



## OS Views (2): Components View

- Processes: run-time representations of user programs — Lectures 2-9
  - create, terminate, suspend, resume
  - access to shared resources (e.g., printers)
- Storage
  - allocation of memory among resident processes
  - disk management (e.g., scheduling of disk accesses)
- I/O
  - device drivers, handling of device interrupts
  - files and directories

  (Storage and I/O: Lectures 10-13)
- Protection — Lectures 14-15
  - user access to system resources

► Course organization follows this view

## OS Views (3): Services View

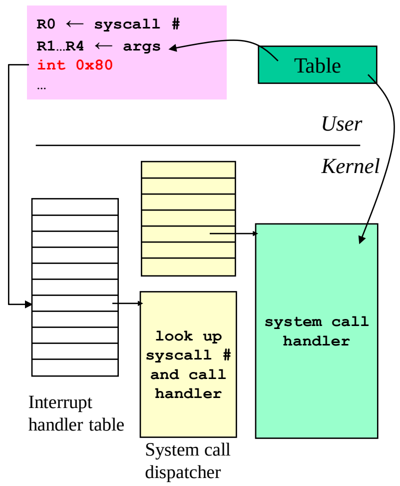
- Two issues
  - What services does an OS provide? (same as functional view)
  - How do users and user programs access these services?
- Interface between the user and the OS: Command Interpreter
  - typical commands
    - process creation and (implicitly) destruction
    - I/O handling and file system manipulation
    - communication: interact with remote devices
    - protection management: changing file/directory access control, etc.
  - different varieties
    - the interpreter contains the code for the requested command (e.g., **delete**)
    - the interpreter calls a system routine to handle the request
    - the interpreter spawns new process(es) to handle the request
      - process lookup through some general procedure

▶ you will implement a simple shell in Nachos Lab 5



## OS Views (3): Services View (contd.)

- Interface between a user program and the OS: System Calls
  - arguments passed in registers, a memory block, or on the stack
  - entry into the kernel using the *trap* mechanism
- Standard system calls
  - process control
  - file manipulation
  - device manipulation
  - information maintenance
    - *get/set* system data (time, memory/cpu usage), process and device attributes
  - communications



## OS Views (4): Structure View

- How to structure OS functionality
  - Layering
  - Microkernels
  - Virtual machines
- Designing and implementing an OS

- Read Sections 2.6-2.9, Silberschatz, Galvin, and Gagne
- Look at Nachos source code
  - Thomas Narten's roadmap



## Next Lecture

- Processes
  - The process concept
  - Processes vs. threads
  - Process states and scheduling
  - Process synchronization

- Reading
  - Silberschatz/Galvin/Gagne: Chapters 3-4

- Nachos Lab 1 is due September 18th, 2007