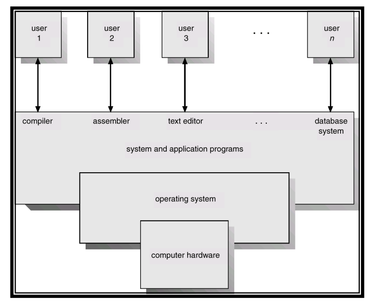
Why Study Operating Systems?

- Abstraction — how do you give the users the illusion of infinite resources (CPU time, memory, file space)?
- System design —tradeoffs between:
  - performance and convenience of these abstractions
  - performance and simplicity of OS
  - functionality in hardware or software
- Primary intersection point — OS is the point where hardware, software, programming languages, data structures, and algorithms all come together
- Curiosity — “look under the hood”
- “Operating systems are among the most complex pieces of software yet developed”, William Stallings, 1994

The Operating System (OS) in Context

- Components of a Computer System
  - *Hardware* – provides basic computing resources (CPU, memory, I/O devices)
  - ***Operating system*** – controls and coordinates the use of the hardware among the various application programs for the various users
  - *Applications programs* – define the ways in which the system resources are used to solve the computing problems of the users (compilers, databases, video games, business programs)
  - *Users* (people, machines, computers)



What is an Operating System?

- An *operating system* (OS) is the interface between the user and the hardware
  - It implements a virtual machine that is easier to program than bare hardware
- An OS provides standard **services** (functionality) which are implemented on the hardware, including:
  - Processes, CPU scheduling, memory management, file system, networking
- The OS **coordinates** multiple applications and users (multiple processes) in a fair and efficient manner

↪ The goal in OS development is to make the machine both **convenient** to use (a software engineering problem) as well as **efficient** (a system and engineering problem)

Modern OS Functionality

- Textbook talks about OS as a:
  - Control program — manages the execution of user programs, prevents errors and improper use of the computer
  - Resource allocator — CPU time, memory space, file space, I/O devices
- OS must provide:
  - Processes & CPU scheduling
    - Multiple processes active concurrently
    - Processes may need to communicate
    - Processes may require mutually-exclusive access to some resource
  - Memory management — must allocate memory to processes, move processes between disk and memory
  - File system — must allocate space for storage of programs and data on disk

History of Operating Systems

- Phase 0 — hardware is a very expensive experiment; no operating systems exist
  1. One user at console
     - One function at a time (computation, I/O, user think/response)
     - Program loaded via card deck
       - Libraries of device drivers (for I/O)
     - User debugs at console
- Phase 1 — hardware is expensive, humans are cheap
  2. Simple batch processing: load program, run, print results, dump, repeat
     - User gives program (cards or tape) to the operator, who schedules the jobs
     - *Resident monitor* automatically loads, runs, dumps user jobs
     - Requires memory management (relocation) and protection
     - More efficient use of hardware, but debugging is more difficult (from dumps)

History of Operating Systems (cont.)

- Phase 1 — hardware is expensive, humans are cheap
  3. Overlapped CPU & I/O operations
     - First: buffer slow I/O onto fast tape drives connected to CPU, replicate I/O devices
     - Later: *spool* data to disk
  4. Multiprogrammed batch systems
     - Multiple jobs are on the disk,waiting to run
     - *Multiprogramming* — run **several** programs at the "same" time
       - Pick some jobs to run (*scheduling*), and put them in memory (*memory management*)
       - Run one job; when it waits on something (tape to be mounted, key to be pressed), switch to another job in memory
     - First big failures:
       - MULTICS announced in 1963, not released until 1969
       - IBM's OS/360 released with 1000 known bugs
     - OS design should be a science, not an art

History of Operating Systems (cont.)

- Phase 2 — hardware is less expensive than before, humans are expensive
  5. Interactive *timesharing*
     - Lots of cheap terminals, one computer
       - All users interact with system at once
       - Debugging is much easier
     - Disks are cheap, so put programs and data online
       - 1 punch card = 100 bytes
       - 1MB = 10K cards
       - OS/360 was several feet of cards
     - New problems:
       - Need *preemptive scheduling* to maintain adequate *response time*
       - Need to avoid *thrashing* (swapping programs in and out of memory too often)
       - Need to provide adequate security measures
     - Success: UNIX developed at Bell Labs so a couple of computer nerds (Thompson, Ritchie) could play Star Trek on an unused PDP-7 minicomputer

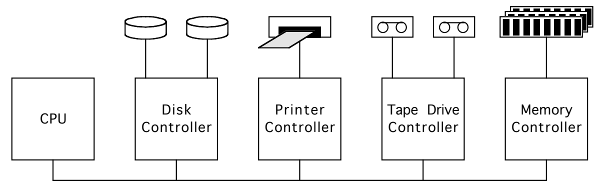
History of Operating Systems (cont.)

- Phase 3 — hardware is cheap, humans are expensive
  6. Personal computing
     - CPUs are cheap enough to put one in each terminal, yet powerful enough to be useful
       - Computers for the masses!
     - Return to simplicity; make OS simpler by getting rid of support for multiprogramming, concurrency, and protection
  7. Parallel systems
     - User multiple CPUs with a shared memory in close communication
       - Increased throughput
     - Mostly MIMD hardware, some SIMD
     - Symmetric multiprocessing (SMP)
       - Each processor runs an identical copy of the OS, multiple processes running at once

History of Operating Systems (cont.)

- Phase 4 — hardware is cheap, ubiquitous, and pervasive
  6. Distributed systems
     - Distribute the computation among several (possibly different) physical processors, each with its own memory, in loose communication
       - Resource sharing
       - Increased throughput
       - Reliability
     - Client-server computing
       - Server provides specified services (e.g., file service, directory service, print service) to a set of clients
  7. Embedded / handheld systems
     - PDAs, cell phones, CD players, etc.
     - Small OS "footprint" (limited memory)
     - Slow processors
     - Small displays
     - Power consumption is a primary consideration

Input / Output (I/O)



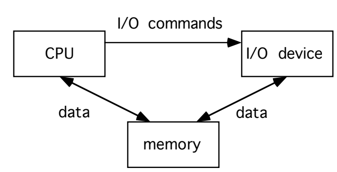
- CPU and device controllers all use a common bus for communication
- Software-polling synchronous I/O
  - CPU starts an I/O operation, and continuously *polls* (checks) that device until the I/O operation finishes
  - Device controller contains registers for communication with that device
    - Input register, output register — for data
    - Control register — to tell it what to do
    - Status register — to see what it's done
  - Why not connect all I/O devices directly to CPU? Why not… memory…?

Input / Output (I/O) (cont.)

- Terminology
  - *Synchronous* I/O — CPU execution waits while I/O proceeds
  - *Asynchronous* I/O — I/O proceeds concurrently with CPU execution
- Interrupt-based asynchronous I/O
  - Device controller has its own processor, and executes asynchronously with CPU
    - Device controller puts an interrupt signal on the bus when it needs CPU's attention
  - When CPU receives an interrupt:
    1. It saves the CPU state and invokes the appropriate interrupt handler using the *interrupt vector* (addresses of OS routines to handle various events)
    2. Handler must save and restore software state (e.g., registers it will modify)
    3. CPU restores CPU state

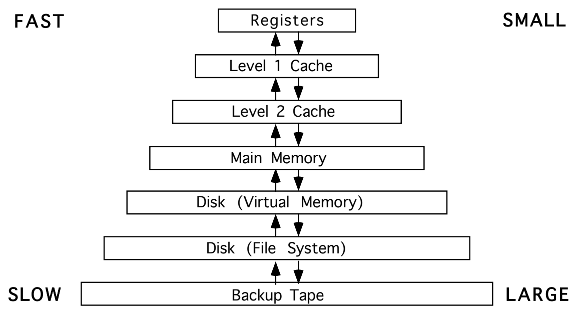
Input / Output (I/O) (cont.)

- Memory-mapped I/O
  - Uses direct memory access (DMA) — I/O device can transfer block of data to / from memory without going through CPU



  - OS allocates buffer in memory, tells I/O device to use that buffer
  - I/O device operates asynchronously with CPU, interrupts CPU when finished
  - Used for most high-speed I/O devices (e.g., disks, communication interfaces)

Storage Structures



- At a given level, memory may not be as big or as fast as you'd like it be
  - Tradeoffs between size and speed
- *Principle of Locality of Reference* leads to *caching*
  - When info is needed, look on this level
  - If it's not on this level, get it from the next slower level, and save a copy here in case it's needed again sometime soon

Magnetic Disks

- Provide secondary storage for system (after main memory)
- Technology
  - Covered with magnetic material
  - Read / write head "floats" just above surface of disk
  - Hierarchically organized as platters, tracks, sectors (blocks)
- Devices
  - Hard (moving-head) disk — one or more platters, head moves across tracks
  - Floppy disk — disk covered with hard surface, read / write head sits on disk, slower, smaller, removable, rugged
  - CDROM — uses laser, read-only, high-density
    - Optical —read / write

Protection

- Multiprogramming and timesharing require that the memory and I/O of the OS and user processes be **protected** against each other
  - Note that most PCs do not support this kind of protection
- Provide protection via two modes of CPU execution: *user mode* and *kernel mode*
  - In kernel / privileged / supervisor / monitor mode, *privileged instructions* can:
    - Access I/O devices, control interrupts
    - Manipulate the state of the memory (page table, TLB, etc.)
    - Halt the machine
    - Change the mode
  - Requires architectural support:
    - Mode bit in a protected register
    - Privileged instructions, which can only be executed in kernel mode

I/O Protection

- To prevent illegal I/O, or simultaneous I/O requests from multiple processes, perform all I/O via privileged instructions
  - User programs must make a *system call* to the OS to perform I/O
- When user process makes a system call:
  - A *trap* (software-generated interrupt) occurs, which causes:
    - The appropriate trap handler to be invoked using the trap vector
    - Kernel mode to be set
  - Trap handler:
    - Saves state
    - Performs requested I/O (if appropriate)
    - Restores state, sets user mode, and returns to calling program

Memory Protection

- Must protect OS's memory from user programs (can't overwrite, can't access)
  - Must protect memory of one process from another process
  - Must not protect memory of user process from OS
- Simplest and most common technique:
  - *Base register* —smallest legal address
  - *Limit register* — size of address range
  - Base and limit registers are loaded by OS before running a particular process
  - CPU checks each address (instruction & data) generated in user mode
    - Any attempt to access memory outside the legal range results in a trap to the OS
- Additional hardware support is provided for virtual memory

CPU Protection

- Use a timer to prevent CPU from being hogged by one process (either maliciously, or due to an infinite loop)
  - Set timer to interrupt OS after a specified period (small fraction of a second)
  - When interrupt occurs, control transfers to OS, which decides which process to execute for next time interval (maybe the same process, maybe another one)
- Also use timer to implement time sharing
  - At end of each time interval, OS switches to another process
  - *Context switch* = save state of that process, update Process Control Block for each of the two processes, restore state of next process

Computer Architecture & OS

- Need for OS services often drives inclusion of architectural features in CPU:

OS Service	Hardware Support
I/O	interrupts memory-mapped I/O caching
Data access	memory hierarchies file systems
Protection	system calls kernel & user mode privileged instructions interrupts & traps base & limit registers
Scheduling & Error recovery	timers