CIS520 – Operating Systems
Handout 1
Overview and History

- What is an operating system? Hard to define precisely, because operating systems arose historically as people needed to solve problems associated with using computers.
- Much of operating system history driven by relative cost factors of hardware and people. Hardware started out fantastically expensive relative to people and the relative cost has been decreasing ever since. Relative costs drive the goals of the operating system.
  - In the beginning: **Expensive Hardware, Cheap People** Goal: maximize hardware utilization.
  - Now: **Cheap Hardware, Expensive People** Goal: make it easy for people to use computer.
- In the early days of computer use, computers were huge machines that are expensive to buy, run and maintain. Computer used in single user, interactive mode. Programmers interact with the machine at a very low level - flick console switches, dump cards into card reader, etc. The interface is basically the raw hardware.
  - Problem: Code to manipulate external I/O devices. Is very complex, and is a major source of programming difficulty.
  - Solution: Build a subroutine library (device drivers) to manage the interaction with the I/O devices. The library is loaded into the top of memory and stays there. This is the first example of something that would grow into an operating system.
- Because the machine is so expensive, it is important to keep it busy.
  - Problem: computer idles while programmer sets things up. Poor utilization of huge investment.
  - Solution: Hire a specialized person to do setup. Faster than programmer, but still a lot slower than the machine.
  - Solution: Build a batch monitor. Store jobs on a disk (spooling), have computer read them in one at a time and execute them. Big change in computer usage: debugging now done offline from print outs and memory dumps. No more instant feedback.
  - Problem: At any given time, job is actively using either the CPU or an I/O device, and the rest of the machine is idle and therefore unutilized.
  - Solution: Allow the job to overlap computation and I/O. Buffering and interrupt handling added to subroutine library.
  - Problem: one job can't keep both CPU and I/O devices busy. (Have compute-bound jobs that tend to use only the CPU and I/O-bound jobs that tend to use only the I/O devices.) Get poor utilization either of CPU or I/O devices.
  - Solution: multiprogramming - several jobs share system. Dynamically switch from one job to another when the running job does I/O. Big issue: protection. Don't want one job to affect the results of another. Memory protection and relocation added to hardware, OS must manage new hardware functionality. OS starts to become a significant software system. OS also starts to take up significant resources on its own.
- Phase shift: Computers become much cheaper. People costs become significant.

  - Issue: It becomes important to make computers easier to use and to improve the productivity of the people. One big productivity sink: having to wait for batch output (but is this really true?). So, it is important to run interactively. But computers are still so expensive that you can't buy one for every person. Solution: interactive timesharing.
  - Problem: Old batch schedulers were designed to run a job for as long as it was utilizing the CPU effectively (in practice, until it tried to do some I/O). But now, people need reasonable response time from the computer.
  - Solution: Preemptive scheduling.
  - Problem: People need to have their data and programs around while they use the computer.
  - Solution: Add file systems for quick access to data. Computer becomes a repository for data, and people don't have to use card decks or tapes to store their data.
  - Problem: The boss logs in and gets terrible response time because the machine is overloaded.
  - Solution: Prioritized scheduling. The boss gets more of the machine than the peons. But, CPU scheduling is just an example of resource allocation problems. The timeshared machine was full of limited resources (CPU time, disk space, physical memory space, etc.) and it became the responsibility of the OS to mediate the allocation of the resources. So, developed things like disk and physical memory quotas, etc.

  Overall, time sharing was a success. However, it was a limited success. In practical terms, every timeshared computer became overloaded and the response time dropped to annoying or unacceptable levels. Hard-core hackers compensated by working at night, and we developed a generation of pasty-looking, unhealthy insomniacs addicted to caffeine.

- Computers become even cheaper. It becomes practical to give one computer to each user. Initial cost is very important in market. Minimal hardware (no networking or hard disk, very slow microprocessors and almost no memory) shipped with minimal OS (MS-DOS). Protection, security less of an issue. OS resource consumption becomes a big issue (computer only has 640K of memory). OS back to a shared subroutine library.

- Hardware becomes cheaper and users more sophisticated. People need to share data and information with other people. Computers become more information transfer, manipulation and storage devices rather than machines that perform arithmetic operations. Networking becomes very important, and as sharing becomes an important part of the experience so does security. Operating systems become more sophisticated. Start putting back features present in the old time sharing systems (OS/2, Windows NT, even Unix).

- Rise of network. Internet is a huge popular phenomenon and drives new ways of thinking about computing. Operating system is no longer interface to the lower level machine - people structure systems to contain layers of middleware. So, a Java API or something similar may be the primary thing people need, not a set of system calls. In fact, what the operating system is may become irrelevant as long as it supports the right set of middleware.

- Network computer. Concept of a box that gets all of its resources over the network. No local file system, just network interfaces to acquire all outside data. So have a slimmer version of OS.

- In the future, computers will become physically small and portable. Operating systems will have to deal with issues like disconnected operation and mobility. People will also start using information with a psuedo-real time component like voice and video. Operating systems will have to adjust to deliver acceptable performance for these new forms of data.

- What does a modern operating system do?
  - **Provides Abstractions** Hardware has low-level physical resources with complicated, idiosyncratic interfaces. OS provides abstractions that present clean interfaces. Goal: make computer easier to use. Examples: Processes, Unbounded Memory, Files, Synchronization and Communication Mechanisms.
  - **Provides Standard Interface** Goal: portability. Unix runs on many very different computer systems. To a first approximation can port programs across systems with little effort.

  - **Mediates Resource Usage** Goal: allow multiple users to share resources fairly, efficiently, safely and securely. Examples:
    * Multiple processes share one processor. (preemptable resource)
    * Multiple programs share one physical memory (preemptable resource).
    * Multiple users and files share one disk. (non-preemptable resource)
    * Multiple programs share a given amount of disk and network bandwidth (preemptable resource).
  - **Consumes Resources** Solaris takes up about 8Mbytes physical memory (or about $400).

- Abstractions often work well - for example, timesharing, virtual memory and hierarchical and networked file systems. But, may break down if stressed. Timesharing gives poor performance if too many users run compute-intensive jobs. Virtual memory breaks down if working set is too large (thrashing), or if there are too many large processes (machine runs out of swap space). Abstractions often fail for performance reasons.

- Abstractions also fail because they prevent programmer from controlling machine at desired level. Example: database systems often want to control movement of information between disk and physical memory, and the paging system can get in the way. More recently, existing OS schedulers fail to adequately support multimedia and parallel processing needs, causing poor performance.

- Concurrency and asynchrony make operating systems very complicated pieces of software. Operating systems are fundamentally non-deterministic and event driven. Can be difficult to construct (hundreds of person-years of effort) and impossible to completely debug. Examples of concurrency and asynchrony:
  - I/O devices run concurrently with CPU, interrupting CPU when done.
  - On a multiprocessor multiple user processes execute in parallel.
  - Multiple workstations execute concurrently and communicate by sending messages over a network. Protocol processing takes place asynchronously.

  Operating systems are so large no one person understands whole system. Outlives any of its original builders.

- The major problem facing computer science today is how to build large, reliable software systems. Operating systems are one of very few examples of existing large software systems, and by studying operating systems we may learn lessons applicable to the construction of larger systems.