Introduction

Today
• Welcome to OS
• Administrivia
• OS overview and history
• Computer system overview

Next time
• OS components & structure
Why study operating systems?

- **Tangible reasons**
  - Build/modify one - OSs are everywhere
  - Administer and use them well
  - Tune your favorite application performance
  - Great capstone course

- **Intangible reasons**
  - Curiosity
  - Use/gain knowledge from other areas
  - Challenge of designing large, complex systems
A computer system - Where's the OS?

- Hardware provides basic computing resources
- Applications define ways in which resources are used to solve users' problems
- **OS controls & coordinates use of hardware by users’ applications**

A few vantage points
- End user
- Programmer
- OS Designer

Diagram:

```
+----------------------------------+
| User 1                          |
|                                 |
| compiler                        |
| text editor                     |
| systems and application programs|
| operating system                |
| machine language                |
| microarchitecture               |
| physical devices                |
```

End user

Programmer

OS Designer
What is an operating system?

- Extended machine – top-down/user-view
  - Hiding the messy details, presenting a virtual machine that's easier to program than the HW
- Resource manager – bottom-up/system-view
  - Everybody gets a fair-share of time/space from a resource (multiplexing in space/time)
  - A control program – to prevent errors & improper use (CP/M?)
- A bundle of helpful, commonly used things
- Goals
  - Convenience – make solving user problems easier
  - Efficiency – use hardware in an efficient manner ($$$ machines demand efficient use)
What's part of the OS?

- Trickier than you think: file system, device drivers, shells, window systems, browser, ...
- Everything a vendor ships with your order?
- The one program running at all times, or running in kernel mode
  - Everything else is either a system program (ships with the OS) or an application program
  - *Can the user change it?*

*Why does it matter? In 1998 the US Department of Justice filed suit against MS claiming its OS was too big*
The evolution of operating systems

- A brief history & a framework to introduce OS principles
- Early attempts – Babbage's (1702-1871)
  - Analytical Engine (Ada Lovelace – World's first programmer)
- 1945-55 – Vacuum tubes and plugboards
  - ABC, MARK 1, ENIAC
  - No programming languages, no OS
  - A big problem
    - Scheduling – signup sheet on the wall
Evolution ... Batch systems (1955)

- Transistors → machs. reliable enough to sell
  - Separation of builders & programmers

- Getting your program to run
  - Write it in paper (maybe in FORTRAN)
  - Punch it on cards & drop cards in input room
  - Operator may have to mount/dismount tapes, setting up card decks, ... setup time!

- Batch systems
  - Collect a tray of full jobs, read them all into tape with a cheap computer
  - Bring them to the main computer where the “OS” will go over each jobs one at a time
  - Print output offline
Evolution ... Spooling (1965)

- Disks much faster than card readers & printers
- Spool (Simultaneous Peripheral Operations On-Line)
  - While one job is executing, spool next one from card reader onto disk
    - Slow card reader I/O overlapped with CPU
  - Can even spool multiple programs onto disk
    - OS must choose which one to run next (job sched)
  - But CPU still idle when program interact with a peripheral during execution
  - Buffering, double buffering
Evolution ... Multiprogramming (1965)

- To increase system utilization
  - Keeps multiple runnable jobs loaded in memory at once
  - Overlap I/O of a job with computing of another
  - Needs asynchronous I/O devices
    - Some way to know when devices are done
      - Interrupt or polling
    - Goal- optimize system throughput
      - Cost on response time

- IBM OS/360 & the tar pit
Evolution ... Timesharing (1965)

- To support interactive use
  - Multiple terminals into one machine
  - Each user has the illusion of owning the entire machine

- Time-slicing
  - Dividing CPU equally among users
  - If jobs are truly interactive, CPU can jump between them without users noticing it
  - Recovers interactivity for the user (why do you care?)

- CTSS (Compatible Time Sharing System), MULTICS and UNIX
Evolution ... PCs (1980)

- Large-scale integration >> small & cheap machines
- 1974 – Intel's 8080 & Gary Kildall's CP/M
- Early 1980s – IBM PC, BASIC, CP/M & MS-DOS
- User interfaces, XEROX Altos, MACs and Windows
Evolution ... Distributed & pervasive

- Facilitate use of geographically distributed resources
  - Workstations on a LAN or across the Internet
- Support communication between programs
- Speed up is not really the issue, but access to resources
- Architectures
  - Client/servers
    - Mail server, print server, web server
  - Peer-to-peer
    - (Most) everybody is both, server and client
- Pervasive computing & embedded devices
“Ontogeny recapitulates phylogeny”* 

But new problems arise and others redefine themselves

The development of an embryo repeats the evolution of the species (* Ernst Haeckel)
Course overview ...

- Overall structure
  - Lectures
  - TA Sessions
    - Once a week and focused on projects
  - Homework (5)
    - Look at them as reading enforcers

<table>
<thead>
<tr>
<th>Course</th>
<th>Start Date</th>
<th>End Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Introduction to Operating Systems</td>
<td>09/25</td>
<td>10/02</td>
</tr>
<tr>
<td>2: Processes and Threads</td>
<td>10/16</td>
<td>10/23</td>
</tr>
<tr>
<td>3: Memory Management and Virtual Memory</td>
<td>11/06</td>
<td>11/13</td>
</tr>
<tr>
<td>4: I/O and File Systems</td>
<td>11/20</td>
<td>12/02</td>
</tr>
<tr>
<td>5: Research in Operating Systems</td>
<td>12/04</td>
<td>12/04 (in-class)</td>
</tr>
</tbody>
</table>
Course overview

- Overall structure
  - ...
  - Projects (4)
    - *First one out next Tuesday!*
  - Exams (2)

- Course book & other material
  - Read before class

- Other recommended sources
  - Stevens' book

- Grading, policies
Computer systems structure

- Abstract model of a simple computer

Diagram:
- CPU
- Memory
- Video controller
- Keyboard controller
- Floppy disk controller
- Hard disk drive

Connected by a Bus
Processor

- Basic operation cycle
  - Fetch next instruction
  - Decode it to determine type & operands
  - Execute it

- Set of instructions
  - Architecture specific - Pentium != SPARC
  - Includes: combine operands (ADD), control flow, data movement, etc

- Since memory access is slow ... registers
  - General registers to hold variables & temp. results
  - Special registers: Program Counter (PC), Stack Pointer (SP), Program Status Word (PSW)

- Moving away from basic operation cycle: pipeline architectures, superscalar, ...
Memory

- Ideally – fast, large & cheap
- Reality – storage hierarchy
  - Registers
    - Internal to the CPU & just as fast
    - 32x32 in a 32 bit machine
  - Cache
    - Split into cache lines
    - If word needs is in cache, get in ~2 cycles
  - Main memory
  - Hard disk
  - Magnetic tape
  - Coherency?

First core-based memory: IBM 405 Alphabetical Accounting Machine

Typical access time

<table>
<thead>
<tr>
<th>Typical capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1 KB</td>
</tr>
<tr>
<td>1 MB</td>
</tr>
<tr>
<td>64-512 MB</td>
</tr>
<tr>
<td>5-50 GB</td>
</tr>
<tr>
<td>20-100 GB</td>
</tr>
</tbody>
</table>

EECS 343 Operating Systems
Northwestern University
OS protection

- Multiprogramming & timesharing are useful but
  - How to protect programs from each other & kernel from all?
  - How to handle relocation?

- Some instructions are restricted to the OS
  - e.g. Directly access I/O devices
  - e.g. Manipulate memory state management

- How does the CPU know if a protected instructions should be executed?
  - Architecture must support 2+ mode of operation
  - Mode is set by status bit in a protected register (PSW)
    - User programs execute in user mode, OS in kernel mode

- Protected instructions can only be executed in kernel mode
Crossing protection boundaries

How can apps do something privileged?
- e.g. how do you write to disk if you can't do I/O?

User programs must call an OS procedure
- OS defines a sequence of system calls
- How does the user to kernel-mode transition happen?

There must be a system call instruction, which
- Causes an exception (throws a soft interrupt) which vector to a kernel handler
- Passes a parameter indicating which syscall is
- Saves caller's state so it can be restored
- OS must verify caller's parameters
- Must be a way to go back to user once done
Memory relocation

- Relocation simplest solution
  - Base (start) of program + limit registers
  - Solves both problems; cost 2 registers + cycle time incr

- Check and mapping to virtual address done by MMU (memory management unit)

- More sophisticated alternatives
  - 2 base and 2 limit registers for text & data; allow sharing program text
  - Paging, segmentation, virtual memory
I/O devices: magnetic disks

- 1+ platters rotating at >5,400 RPM
- Mechanical arm (arm assembly)
- Platter logically divided in tracks, sectors
- Cylinder – tack for a given head position
- Moving & transfer times
  - To next cylinder ~1msec
  - To random cylinder ~5-10msec
  - For sector to get under ~5-10msec
  - Transfer once in the right place 5-160MB/sec
I/O devices

I/O Device
- Device + Controller (simpler I/F to OS; think SCSI)
  - Read sector x from disk y → (disk, cylinder, sector, head), ...

Device driver – SW to talk to controller
- To use it, must be part of kernel: ways to include it
  - Re-link kernel with new driver and reboot (UNIX)
  - Make an entry in an OS file & reboot (OS finds it at boot time and loads it)
  - Dynamic load – OS takes new driver while running & installs it

I/O can be done in 3 different ways
- Busy waiting/synchronous
- Interrupt-based/asynchronous
- Direct Memory Access (DMA)
OSs are event driven
  – Once booted, all entry to kernel happens as result of an event (e.g. signal by an interrupt), which
    • Immediately stops current execution
    • Changes to kernel mode, event handler is called

Kernel defines handlers for each event type
  – Specific types are defined by the architecture
    • e.g. timer event, I/O interrupt, system call trap

Handling the interrupt
  – Push PC & PSW onto stack and switch to kernel mode
  – Device # is index in interrupt vector - get handler
  – Interrupt handler
    • Stores stack data
    • Handles interrupt
    • Returns to user program after restoring program state
Interrupts and exceptions

Three main types of events: interrupts & exceptions

- Exceptions/traps caused by SW executing instructions
  - e.g., the x86 ‘int’ instruction
  - e.g., a page fault, or an attempted write to a read-only page
  - An expected exception is a “trap”, unexpected is a “fault”
- Interrupts caused by HW devices
  - e.g., device finishes I/O, timer fires
Timers

- How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
  - Use a hardware timer that generates a periodic interrupt
  - Before it transfers to a user program, the OS loads the timer with a time to interrupt
  - When time's up, interrupt transfers control back to OS
    - OS decides which program to schedule next
    - Interesting policy question: 1+ class scheduled for that

- Should the timer be privileged?
  - for reading or for writing?
Synchronization

- **Issues with interrupts**
  - May occur any time, causing code to execute that interferes with the interrupted code
  - OS must be able to synchronize concurrent processes

- **Synchronization**
  - Guarantee that short instruction sequences (e.g. read-modify-write) execute atomically
  - Two methods
    - Turn off interrupts, execute sequence, reenable interrupts
    - Have special, complex atomic instructions – test-and-set

*Management of concurrency & asynchronous events is the biggest difference bet/ systems-level & traditional application programming.*
Summary

In this class you will learn
- Major components of an OS
- How are they structured
- The most important interfaces
- Policies typically used in an OS
- Algorithms used to implement those policies

Philosophy
- You many not ever build an OS, but
- As a CS/CE you need to understand the foundations
- Most importantly, OSs exemplify the sorts of engineering tradeoffs you'll need to make throughout your careers