

CS 140: operating systems and systems programming

Mendel Rosenblum
Stanford Computer Science Dept
Winter '07
Lecture 1: Introduction
(www.stanford.edu/class/cs140)

Today

- ◆ Course overview
- ◆ What's interesting about operating systems?
- ◆ What is an operating system?
- ◆ Principles of OS design
- ◆ Last 15-20 minutes:
group partners

Course overview

- ◆ Website: <http://www.stanford.edu/class/cs140>
- ◆ Email: cs140-win0708-staff@lists.stanford.edu
- ◆ Key People
 - Instructor:
Mendel Rosenblum (mendel@cs.stanford.edu)
 - CAs:
Megan Wachs
Varun Arora
Derrick Isaacson
- ◆ Key dates:
 - Lectures: MWF 10:00AM-10:50AM in Gates B03
 - Midterm: Friday, February 8th, 10:00AM-10:50AM in Gates B03
 - Final: Wednesday, March 19th, 8:30AM-11:30AM

Course Overview (Continued)

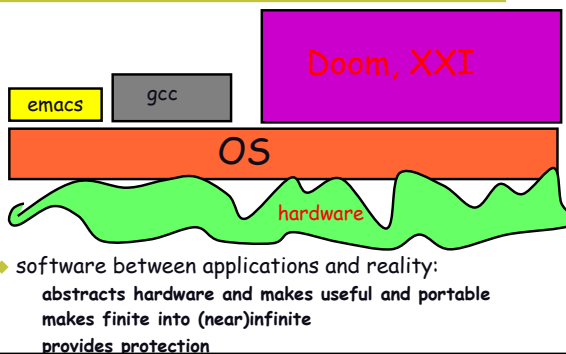
- ◆ Material:
 - Lecture notes - On website, copies given out in class.
 - Textbook - Silberschatz, Galvin, and Gagne, *Operating System Concepts* (Seventh Edition)
- ◆ Prerequisites:
 - Computer organization - (CS107 or EE182)
 - Concurrent programming - (CS107)
- ◆ Grading Policy
 - Programming assignments: 50%
 - Midterm Exam 17%
 - Final Exam: 33%

Give me a sign you learned the material.

Word of warning

- ◆ I have a firm belief in you learn by doing.
Learn OS concepts by coding them!
- ◆ Key Course Features
 - Workload is in the 99% of CS/EE courses
 - Most of the work comes from intense coding/debugging
 - You will need 1-2 good partners for the assignments
- ◆ Project grading:
 - 50% automatic tests (we give you access to the tests)
 - 50% design (code and design documentation)
 - if your code does not work, the TAs WILL NOT fix it.

What is an OS?

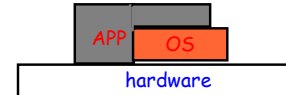


What's interesting here?

- ◆ OS = primal mud of computer system
Makes reality pretty
OS is magic to most people. This course rips open
- ◆ OS = extended example of a complex system
huge, parallel, not understood, insanely expensive to build
Win/NT/XP: 10 years, 1000s of people. Still doesn't work well
most interesting things are complex systems: internet, air traffic control, governments, weather, bf/gf, ...
- ◆ How to deal with complexity?
Abstraction + modularity + iteration
Fail early, fail often, grow from something that works
Unbelievably effective: `int main() { puts("hello"); }` = millions of lines of code! but don't have to think about it

OS evolution: step 0

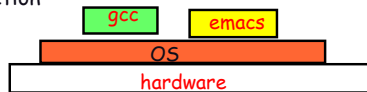
- ◆ Simple OS: One program, one user, one machine:
examples: early computers, early PCs, embedded controllers such as nintendo, cars, elevators, ...



- OS just a library of standard services. Examples: standard device drivers, interrupt handlers, I/O.
- ◆ Non-problems: No bad people. No bad programs. A minimum of complex interactions
- ◆ Problem: poor utilization, expensive

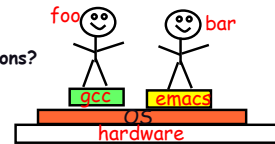
OS evolution: step 1 - Multitasking

- ◆ Simple OS is inefficient:
if process is waiting for something, machine sits wasted.
- ◆ (Seemingly) Simple hack:
run more than one process at once
when one process blocks, switch to another
- ◆ A couple of problems: what if a program infinite loops?
starts randomly scribbling on memory?
- ◆ OS adds protection
interposition
+ preemption
+ privilege



OS evolution: step 2 - Multitasking

- ◆ Simple OS is expensive:
one user = one computer (compare to Ieland)
- ◆ (Seemingly) Simple hack:
Allow more than user at once.
Does machine now run N times slower? Usually not! Key observation: users bursty. If one idle, give other resources.
- ◆ Couple of problems:
what if users are gluttons?
evil?
Or just too numerous?
- ◆ OS adds protection
(notice: as we try to utilize resources, complexity grows)



Protection at 50,000 feet

- ◆ Goal: isolate bad programs and people (security)
main things: preemption + interposition + privileged ops
- ◆ Pre-emption:
give application something, can always take it away
- ◆ Interposition:
OS between application and "stuff"
track all pieces that application allowed to use (usually in a table)
on every access, look in table to check that access legal
- ◆ Privileged/unprivileged mode
Applications unprivileged (peasant)
OS privileged (god)
protection operations can only be done in privileged mode

Wildly successful protection examples

- ◆ Protecting CPU: pre-emption
clock interrupt: hardware periodically "suspends" app, invokes OS
OS decides whether to take CPU away
Other times? Process blocks, I/O completes, system call
- ◆ Protecting memory: Address translation
Every load and store checked for legality
Typically use this machinery to translate to new value (why??)
(protecting disk memory similar)

Address translation

- ◆ Idea:
 - restrict what a program can do by restricting what it can touch!
- ◆ Definitions:
 - Address space: all addresses a program can touch
 - Virtual address: addresses in process' address space
 - Physical address: address of real memory
 - Translation: map virtual to physical addresses
- ◆ "Virtual memory"
 - Translation done using per-process tables (page table) done on every load and store, so uses hardware for speed
 - protection? If you don't want process to touch a piece of physical memory, don't put translation in table.

Quick example: Real systems have holes

- ◆ OSes protect some things, ignore others.
- ◆ Most will blow up if you run this simple program:

```
int main() {
    while(1)
        fork();
}
```

- common response: freeze (unfreeze = reboot) (if not, try allocating and touching memory too) assume stupid, but not malicious users
- ◆ Duality: solve problems technically or socially
 - technical: have process/memory quotas
 - social: yell at idiots that crash machines
 - another example: security: encryption vs laws

OS theme 1: fixed pie, infinite demand

- ◆ How to make pie go farther?
 - Key: resource usage is bursty! So give to others when idle
 - E.g., Waiting for web page? Give CPU to another process
 - 1000s of years old: Rather than one classroom, instructor, restaurant, etc. per person, share. Same issues.
- ◆ BUT, more utilization = more complexity.
 - How to manage? (E.g., 1 road per car vs freeway)
 - Abstraction (different lanes), synchronization (traffic lights), increase capacity (build more roads)
- ◆ BUT, more utilization = more contention. What to do when illusion breaks?
 - Refuse service (busy signal), give up (VM swapping), backoff and retry (ethernet), break (freeway)

Fixed pie, infinite demand (pt 2)

- ◆ How to divide pie?
 - User? Yeah, right.
 - Usually treat all apps same, then monitor and re-portion
- ◆ What's the best piece to take away?
 - OSes = last pure bastion of fascism
 - Use system feedback rather than blind fairness
- ◆ How to handle pigs?
 - Quotas (Iceland), ejection (swapping), buy more stuff (Microsoft products), break (ethernet, most real systems), laws (freeway)
 - A real problem: hard to distinguish responsible busy programs from selfish, stupid pigs.

OS theme 2: performance

- ◆ Trick 1: exploit bursty applications
 - take stuff from idle guy and give to busy. Both happy.
- ◆ Trick 2: exploit skew
 - 80% of time taken by 20% of code
 - 10% of memory absorbs 90% of references
 - basis behind cache: place 10% in fast memory, 90% in slow, seems like one big fast memory
- ◆ Trick 3: past predicts the future
 - what's the best cache entry to replace? If past = future, then the one that is least-recently-used works everywhere: past weather, stock market, ... ~ behavior today.

Course Topics

- ◆ Threads and Processes
- ◆ Concurrency and Synchronization
- ◆ CPU Scheduling
- ◆ Memory Allocation and Virtual Memory
- ◆ Disks, File Systems
- ◆ Protection and Security
- ◆ Networks
- ◆ Review - other types of Operating Systems

The present

- ◆ Today: Read Silberschatz/Galvin
 - Skim chapter 1 (history, tiny bits of today's lecture)
 - Skim chapter 2 (hardware overview)
- ◆ Next: threads and stupid thread tricks
 - Implementation and scheduling**
 - Synchronization, deadlocks, and communication**
 - 6th edition:
 - read chapter 4, skip 4.5 & 4.6
 - read chapter 5
 - 7th edition:
 - read chapter 3, skip 3.4, 3.5, 3.6
 - read chapter 4
- ◆ Now: Mix!