CS 140: Operating Systems and Systems Programming

Bob Lantz
Stanford University, Summer '08

Lecture 1: Introduction

cs140.stanford.edu

(lecture notes: Lantz/Rosenblum/Mazières)

Today

- Course overview
- What's interesting about operating systems?
- What is an operating system?
- Evolution of operating systems
- Principles/themes of OS design
- Last 15-20 minutes:
  - **Pick group partners (groups of 1-3 people for projects)**

Course overview

- Web site: **http://cs140.stanford.edu**
- Newsgroup: **su.class.cs140** (post here first!)
- E-mail: cs140-sum0708-staff@lists.stanford.edu
- Key People
  - **Instructor: Bob Lantz (rlantz@cs)**
  - **CA: Akbar Mehdi (samehdi@stanford)**
- Key Dates:
  - **Lectures: MWF 1:15-2:30 p.m. in (spacious)Terman Auditorium**
  - **Project/Problem sessions: periodic, TBA**
  - **Midterm: scheduled for Friday, July 18th, in class**
  - **Final: scheduled for Saturday, August 16th, 12:15-3:15 p.m.**

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 EE108b)**
  - **Concurrent programming – (CS107)**
- Grading Policy

Programming assignments:	**50%**
Midterm Exam	**17%**
Final Exam:	**33%**

Give me a sign you learned the material.

Course is also on Video

- Lectures are televised/available on SCPD web site
  - **can watch to review**
  - **but more fun/discussion if you attend class in person!**
  - **also better to watch live via SCPD - questions**
  - **avoid temptation to skip and watch them all at once!**
- SCPD students encouraged/welcome to attend in person
  - **lots of room in Terman Auditorium**
- Other notes for SCPD students
  - **please come in person to exams if possible**
  - **feel free to use newsgroup to find project partners**

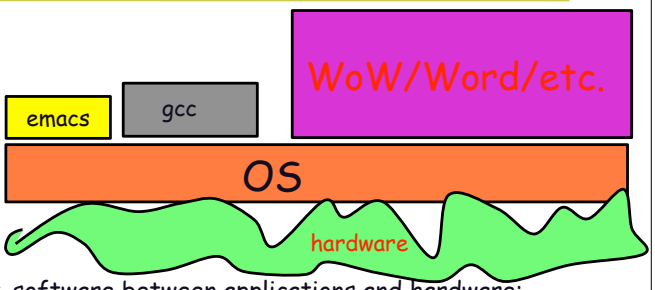
Great things about CS140!

- Fabulous kernel-hacking programming projects!
  - **The Pintos teaching OS is your new friend!**
  - **As are your project partners!**
  - **Implement threads, user processes, virtual memory, filesystem!**
- Learn more (?) than other, lesser CS classes
  - **learn how Unix/Linux/OS X/Windows/etc. work!**
    "non-trivial" projects = more fun/learning!
  - **Become a kernel hacker/OS expert**
  - **...and a better programmer/computer scientist!**
  - **Prepare for other classes (CS240), research, etc.!**
    Put it on your resumé or C.V. - get a job!

Words of warning about CS140!

- Learning by doing = more learning, more doing
  - Learn OS concepts by coding them!
  - But this is a lot of work.
- Key Course Features
  - Workload rated in the 99% of CS/EE courses
  - -> legacy of CS240A/~200hrs+/8-week course
  - 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?



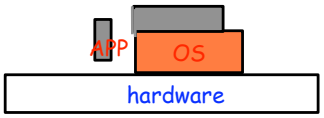
- software between applications and hardware:
  - abstracts hardware and makes useful and portable
  - makes finite into (near)infinite
  - provides protection

What's interesting here?/ Why study OS?

- OS = personality/mind/soul of computer system
  - Makes messy reality (hardware) pretty and friendly
  - OS is magic to most people. This course reveals its secrets!
- 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, relationships, …
- 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

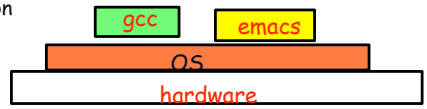
- Simple OS: One program, one user, one machine:
  - examples: early computers, early PCs, embedded controllers such as elevators, cars, NES/GameBoy, …



  - OS just a library of standard services. Examples: standard device drivers, interrupt handlers, I/O.
- Assumptions for a simple world: No bad people. No bad programs. A minimum of complex interactions
- Problems: poor hardware utilization; user must wait to run new program

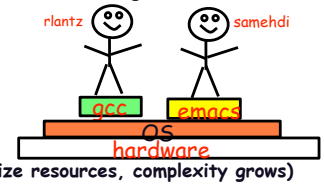
OS evolution: step 1 - Multitasking

- 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 - Multiuser

- Simple OS is expensive:
  - one user = one computer (compare to xenon, Google)
- (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 hogs?
  - 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 (king, deity)
  - 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.

Protection example, or: 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 foolish, 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?
  - Per user? Maybe not a great idea…
  - Usually treat all apps same, then monitor and re-apportion
- What's the best piece to take away?
  - OSes = last pure bastion of authoritarianism
  - Use system feedback rather than blind fairness
- How to handle gluttons?
  - Quotas (Stanford AFS), 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/broken resource hogs.

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
- Virtual Machines
- Review/Advanced Topics (if time)

The present

- Today: Read Silberschatz/Galvin
  - **Skim chapter 1,2 (history, background, hardware)**
- Next: processes & threads
  - **Implementation and scheduling**
  - **Synchronization, deadlocks, and communication**
  - **7th edition:**
    - **» read chapter 3, skip 3.4, 3.5, 3.6**
    - **» read chapter 4**
  - **6th edition:**
    - **read chapter 4, skip 4.5 & 4.6**
    - **read chapter 5**
- Pintos/Assignment 1 is coming Friday - try to compile/run pintos.
- Now: Mix!