CSL373: Operating Systems

Instructor: Sorav Bansal

TA: Swati Sharma

Administrivia

- Class web page: http://cse.iitd.ernet.in/~sbansal/csl373
  - All assignments, reference material, lecture notes online
- Textbook: *Operating System Concepts, 8th edition*, by Silberschatz, Galvin and Gagne

Course Topics

- Threads & Processes
- Concurrency & Synchronization
- Scheduling
- Virtual Memory
- I/O
- Disks, File systems, Network file systems
- Protection & Security
- Virtual Machine Monitors

Course Goals

- Introduce you to operating system concepts
- Cover important systems concepts in general
  - Caching, concurrency, memory management, I/O, protection
- Teach you to deal with large software systems
  - Programming assignments bigger than any other course
  - Warning: This course will probably be your heaviest this semester
- Prepare you for advanced systems courses (advanced topics, recent developments, etc.)

Programming Assignments

- Implement parts of Pintos operating system
  - Built for x86 hardware, you will use hardware emulator
- Four implementation projects
  - Threads
  - Processes (Multiprogramming)
  - Virtual Memory
  - File System
- Implement projects in groups of up to 3 people

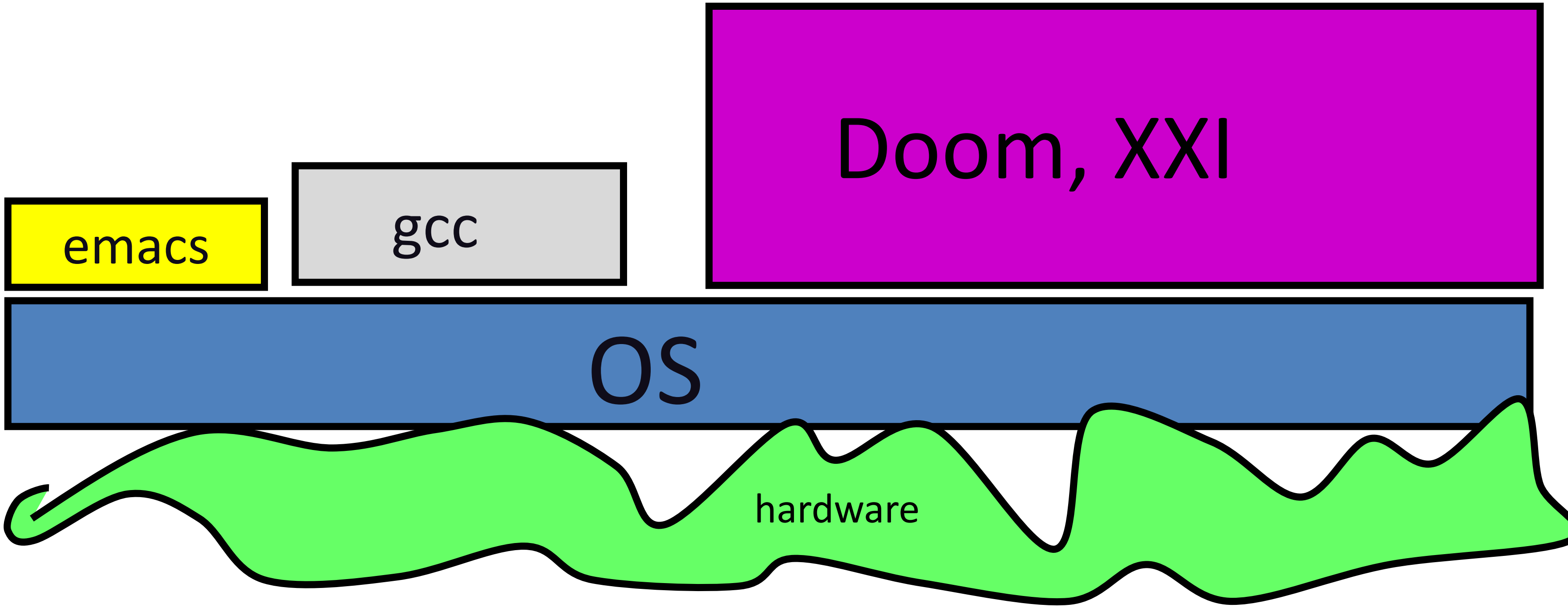
Grading

- 50% of final grade based on minors and major
- 50% of final grade based on projects
  - For each project, 50% of grade based on test cases
    - Please turn in working code, or no credit
  - Remaining 50% based on design, outlined in document
- Do not look at other people's solutions to projects
- Can read but don't copy other OSes (Linux, Open/FreeBSD, etc.)
- Cite any code that inspired your code

What is an operating system?

- OS = primal mud of a 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: 8 years, 1000s of people. Still doesn't work well.
  - Most interesting things are complex: 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.

What is an OS?



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

Abstraction

- What if? The entire software stack was one giant event-driven loop.
  - Possible, BUT clumsy
  - Certainly not practical for general purpose computers
- Better way:
  - Separate software into layers of abstraction
    - e.g., OS → JVM → Java bytecode
  - All programs are written (or compiled) to the abstraction provided by the OS
    - Abstractions may be different for different OS'es
      - e.g., Windows program will not run on Linux even though same underlying hardware

Why study operating systems?

- Operating systems are a maturing field
    - Most people use a handful of mature Oses
    - Hard to get people to switch operating systems
    - Hard to have impact with a new OS
- High-performance servers are an OS issue
    - Face many of the same issues as Oses
- Resource consumption is an OS issue
    - Battery life, radio spectrum, etc.
- Security is an OS issue
    - Hard to achieve security without a solid foundation
- Scalability is an OS issue
    - Large server farms need to solve hard OS problems
- New “smart” devices need new OSes

OS evolution: step 0

- Simple OS: One program, one user, one machine
  - Examples: early PCs, 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 number of complex interactions.
- Problem: poor utilization, expensive

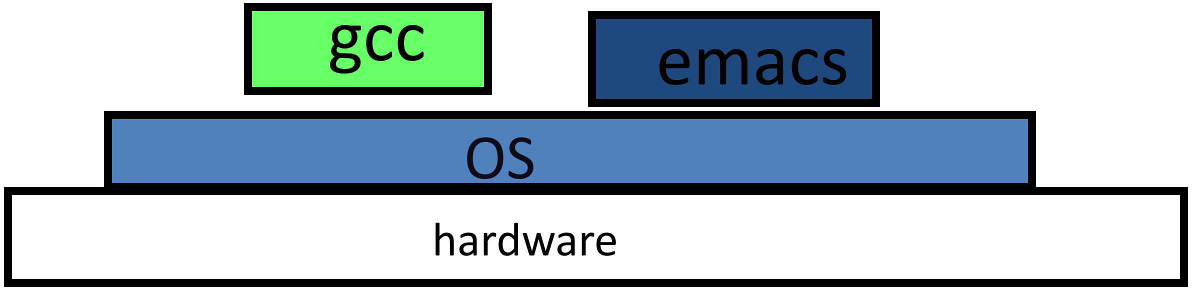
OS evolution: step 1

- 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

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

Protection at 50,000 feet

- Goal: isolate bad programs and people
  - main ideas: preemption + interposition + privileged ops
- Pre-emption:
  - Give application something, can always take it away
- Interposition:
  - OS between application and reality
  - 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 address
- "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, road, etc. per person, share. Same issues.

- BUT, more utilization = more complexity.
  - How to manage? (E.g., 1 road per car versus highway)
  - 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-apportion
- What’s the best piece to take away?
  - It is a dictatorship, with the OS having the final say
  - Use system feedback rather than blind fairness
- How to handle pigs?
  - Quotas (user accounts), ejection (swapping), buy more stuff (microsoft products), 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, classroom understanding, …

The present and the future

- 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
  - Read Ch 4, skip 4.6

- Future:
  - Memory management, virtual memory, file systems, networks