CSE 451: Operating Systems
Spring 2009

Course Introduction

Steve Gribble

Today's agenda

- Administrivia
  - course overview
    - course staff
    - general structure
    - your to-do list
- OS overview
  - functional
    - resource mgmt, major issues
  - historical
    - batch systems, multiprogramming, time shared OS's
    - PCs, networked computers

Course overview

- Everything you need to know is on the course web:

http://www.cs.washington.edu/education/courses/451/CurrentQtr

Overview

- course staff
  - Steve Gribble
  - Sean Anderson (TA)
  - Ryan McElroy (TA)
- general structure
  - read the text prior to class
  - class will supplement rather than regurgitate the text
  - sections will focus on the project
  - we really want to encourage *discussion*, both in class and in section

Your to do list

- please read the entire course web thoroughly, *today*
- please get yourself on the cse451 email list, *today*
  - and check your email *daily*
- homework 1 (reading + problems) is posted on the web; due Monday
- project 1 will be:
  - posted on the web Wednesday
  - discussed in section on Thursday
  - due a week from Friday

More about 451

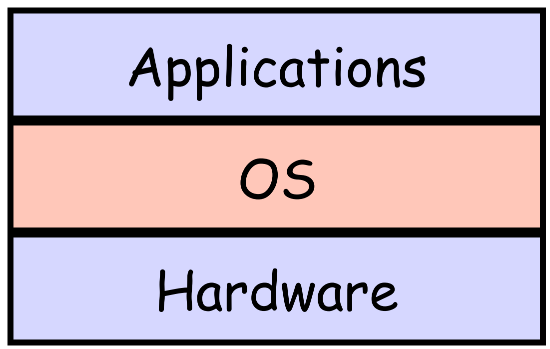
- This is really (at least!) two classes:
  - A classroom/textbook part (mainly run by me)
  - A project part (mainly run by the TAs)
- In a perfect world, we would do this as a two-quarter sequence
- Sometimes the projects and the lectures will mesh, sometimes they won't
- But in any case, you will have to keep up with both the classroom and the projects
- There will be a lot of work
- But you will learn a lot
- In the end, you'll understand much more deeply how computers work

Looking for volunteers

- I want to “try out” the MIT 6.828 project sequence
  - build a teeny OS starting from the HW
  - will likely be very intensive, might not work out
  - do this instead of current project sequence
- Looking for two project teams (two people per team)
  - be familiar with C programming, x86 assembly
    - or willing to learn *fast*
  - be comfortable with no safety net
- Email me if you’re interested

What is an Operating System?

- An operating system (OS) is:
  - a software layer to abstract away and manage details of hardware resources
  - a set of utilities to simplify application development



  - “all the code you didn’t write” in order to implement your application
- Key idea: *virtualization* of resources

The OS and hardware

- An OS mediates programs' access to hardware resources
  - Computation (CPU)
  - Volatile storage (memory) and persistent storage (disk, SSD, ..)
  - Network communications (TCP/IP stacks, ethernet cards, etc.)
  - Input/output devices (keyboard, mouse, display, sound card, ..)
- The OS abstracts hardware into logical resources and well-defined interfaces to those resources
  - processes (CPU, memory)
  - files (disk)
  - programs (sequences of instructions)
  - sockets (network)

Why bother with an OS?

- Application benefits
  - programming simplicity
    - see high-level abstractions (files) instead of low-level hardware details (device registers)
    - abstractions are reusable across many programs
  - portability (across machine configurations or architectures)
    - device independence: 3Com card or Intel card?
- User benefits
  - safety
    - program “sees” own virtual machine, thinks it owns computer
    - OS protects programs from each other (what if one crashes?)
    - OS fairly multiplexes resources across programs
  - efficiency (cost and speed)
    - share one computer across many users
    - concurrent execution of multiple programs

The major OS issues

- **structure**: how is the OS organized?
- **sharing**: how are resources shared across users?
- **naming**: how are resources named (by users or programs)?
- **security**: how is integrity of the OS and its resources ensured?
- **protection**: how is one user/program protected from another?
- **performance**: how do we make it all go fast?
- **reliability**: what happens if something goes wrong (either with hardware or with a program)?
- **extensibility**: can we add new features?
- **communication**: how do programs exchange information, including across a network?

More OS issues…

- **concurrency**: how are parallel activities (computation and I/O) created and controlled?
- **scale and growth**: what happens as demands or resources increase?
- **persistence**: how do you make data last longer than program executions?
- **distribution**: how do multiple computers interact with each other? how do we make distribution invisible?
- **accounting**: how do we keep track of resource usage, and perhaps charge for it?

There are a huge number of engineering tradeoffs

in dealing with these issues!

Hardware/Software Changes with Time

- 1960s: mainframe computers (IBM)
- 1970s: minicomputers (DEC)
- 1980s: microprocessors and workstations (SUN)
- 1990s: PCs (rise of Microsoft, Intel, then Dell)
- 1995-2005: Internet Services / Clusters (Amazon)
- 2006: General Cloud Computing (Google, Amazon)
- …..
- 2020: it's up to you!!

Is there anything new?

- New challenges constantly arise
  - embedded computing (e.g., iPod, GPS)
  - sensor networks (very low power, memory, etc.)
  - peer-to-peer systems (Kazaa, BitTorrent, etc.)
  - ad-hoc networking
  - global-scale server farms / cloud computing (e.g., Amazon EC2, Google)
  - software for utilizing huge clusters (e.g., MapReduce, Bigtable, GFS)
  - overlay networks (e.g., PlanetLab)
  - worms
  - finding bugs in system code (e.g., model checking)

OS history

- In the very beginning…
  - OS was just a library of code that you linked into your program; programs were loaded in their entirety into memory, and executed
  - interfaces were literally switches and blinking lights
- And then came **batch systems**
  - OS was stored in a portion of primary memory
  - OS loaded the next job into memory from the card reader
    - job gets executed
    - output is printed, including a dump of memory (why?)
    - repeat…
  - card readers and line printers were very slow
    - so CPU was idle much of the time (wastes $$)

Spooling

- Disks were much faster than card readers and printers
- Spool (Simultaneous Peripheral Operations On-Line)
  - while one job is executing, spool next job from card reader onto disk
    - slow card reader I/O is overlapped with CPU
  - can even spool multiple programs onto disk
    - OS must choose which to run next
    - job scheduling
  - but, CPU still idle when a program interacts with a peripheral during execution

Multiprogramming

- To increase system utilization, multiprogramming OSs were invented
  - keeps multiple runnable jobs loaded in memory at once
  - overlaps I/O of a job with computing of another
    - while one job waits for I/O completion, OS runs instructions from another job
  - to benefit, need asynchronous I/O devices
    - need some way to know when devices are done
      - interrupts
      - polling
  - goal: optimize system throughput
    - perhaps at the cost of response time…

Timesharing

- To support interactive use, create a timesharing OS:
  - multiple terminals into one machine
  - each user has illusion of entire machine to him/herself
  - optimize response time, perhaps at the cost of throughput
- Timeslicing
  - divide CPU equally among the users
  - if job is truly interactive (e.g. editor), then can jump between programs and users faster than users can generate load
  - permits users to interactively view, edit, debug running programs (why does this matter?)
- MIT Multics system (mid-1960's) was the first large timeshared system
  - nearly all OS concepts can be traced back to Multics

Timesharing

- In early 1980s, a *single* timeshared VAX/780 (like the one in the Allen Center atrium) ran computing for the *entire* CSE department.
- A typical VAX/780 was 1 MIPS (1 MHz) and had 16MB of RAM and 100MB of disk.
- An iPhone is 400 MIPS, has 128MB of RAM (way too little though) and 8GB of disk.

Parallel systems

- Some applications can be written as multiple parallel threads or processes
  - can speed up the execution by running multiple threads/ processes simultaneously on multiple CPUs [Burroughs D825, 1962]
  - need OS and language primitives for dividing program into multiple parallel activities
  - need OS primitives for fast communication among activities
    - degree of speedup dictated by communication/computation ratio
  - many flavors of parallel computers today
    - SMPs (symmetric multi-processors, multi-core)
    - SMT (simultaneous multithreading ["hyperthreading"]
    - MPPs (massively parallel processors)
    - NOWs (networks of workstations) [clusters]
    - computational grid (SETI @home)

Personal computing

- Primary goal was to enable new kinds of interactive applications
- Bit-mapped display [Xerox Alto,1973]
  - New graphic/visual apps
  - new input device (the mouse)
- Move computing near the display
  - why?
- Window systems
  - the display as a managed resource
- Local area networks [Ethernet]
  - why?
- Effect on OS?

Distributed OS

- distributed systems to facilitate use of geographically distributed resources
  - workstations on a LAN
  - servers across the Internet
  - 10,000 node cluster in a machine room
- supports communications between jobs
  - interprocess communication
    - message passing, shared memory
  - networking stacks
- sharing of distributed resources (hardware, software)
  - load balancing, authentication and access control, …
- speedup isn't always the issue
  - access to diversity of resources is goal
  - fault tolerance

Embedded, Mobile OSs

- Pervasive computing
  - cheap processors embedded everywhere
  - how many are on your body now? in your car?
  - cell phones, PDAs, games, iPod, network computers, …
- Typically very constrained hardware resources
  - slow processors
  - small amount of memory
  - no disk or tiny disk
  - typically only one dedicated application
  - limited power
- But technology changes fast
  - embedded CPUs are getting faster
  - storage is growing rapidly

CSE 451

- In this class we will learn:
  - what are the major components of most OS's?
  - how are the components structured?
  - what are the most important (common?) interfaces?
  - what policies are typically used in an OS?
  - what algorithms are used to implement policies?
- Philosophy
  - you may not ever build an OS
  - but as a computer scientist or computer engineer you need to understand the foundations
  - most importantly, operating systems exemplify the sorts of engineering design tradeoffs that you'll need to make throughout your careers – compromises among and within cost, performance, functionality, complexity, schedule …