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 will be on the course web page:

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

- But to tide you over for the next hour …
  - course staff
    - Hank Levy
    - Roxana Geambasu (grad TA)
    - Nick Hunt (ugrad TA)
    - Kristin Lee (ugrad 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
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
- The world isn’t perfect ☺
- 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

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, etc.)
  - 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)
- 2000: 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)

Protection and security as an example

- none
- OS from my program
- your program from my program
- my program from my program
- access by intruding individuals
- access by intruding programs
- denial of service
- distributed denial of service
- spoofing
- spam
- worms
- viruses
- cross-site scripting attacks (in the browser)
- stuff you download and run knowingly (bugs, trojan horses)
- stuff you download and run unknowingly (cookies, spyware)

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
      - buffering, double-buffering
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 OS

- 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 …