CSE 451: Operating Systems Winter 2009

Introduction

Mark Zbikowski
Gary Kimura

Introduction

- Administration
  - Introductions
    - A bit about ourselves
    - Mark Zbikowski CSE 480
    - Gary Kimura CSE 476
  - Three sources of truth
    - Lectures
    - Reading
    - Projects/Source code
    - All are important
  - Lectures
    - Supplement rather than recapitulate text
    - Lots of historical/developmental info
    - Lots of “why was it done this way” info
    - ASK QUESTIONS!

Introduction

- More Administration
  - Homework
    - Keep up with the reading (Silberschatz, et al.). Far better for you to read the chapters BEFORE the class
    - Do/familiarize yourself with the problems at the end of each chapter
  - Quizzes
    - Regular quiz (one or two questions)
    - Last 10 to 15 minutes of class on Friday, returned to you on the following Wednesday.
    - Expect 9 quizzes throughout the quarter

Introduction

- More Administration
  - Projects based on Windows 2003 Server sources
    - 4 projects
      - Two individual projects and two group projects
      - You Will Write Code. You Will Read Lots of Code
      - You are either very familiar with C or will become so quickly
    - Online textbook, via class web page
    - Lab session to get students familiar with the development environment
    - Late policy

Introduction

- Last Administration
  - Final
    - Take home part (could be an essay)
    - Small in class portion
  - Grading
    - Goal is to determine what YOU have learned and can express
    - 30% quizzes (throw out the lowest quiz)
    - 35% projects
    - 30% Final
    - 5% incidentals
    - Scores available via Catalyst

Goals for this course

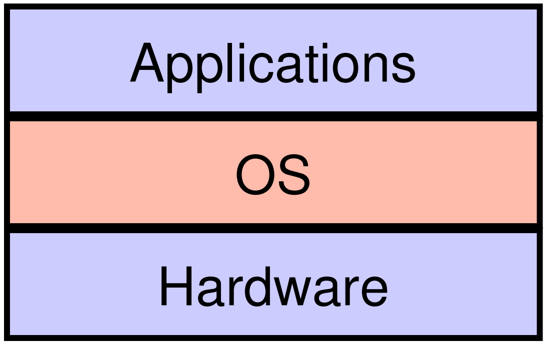
- Two views of an OS
  – The OS user's (i.e., application programmer's) view
  – The OS implementer's view
- In this class we will learn:
  – What are the major parts of an O.S.
  – How is the O.S. and each sub-part structured
  – What are the important common interfaces
  – What are the important policies
  – What algorithms are typically used
  – What engineering/practicality tradeoffs were used

Introduction to Operating Systems

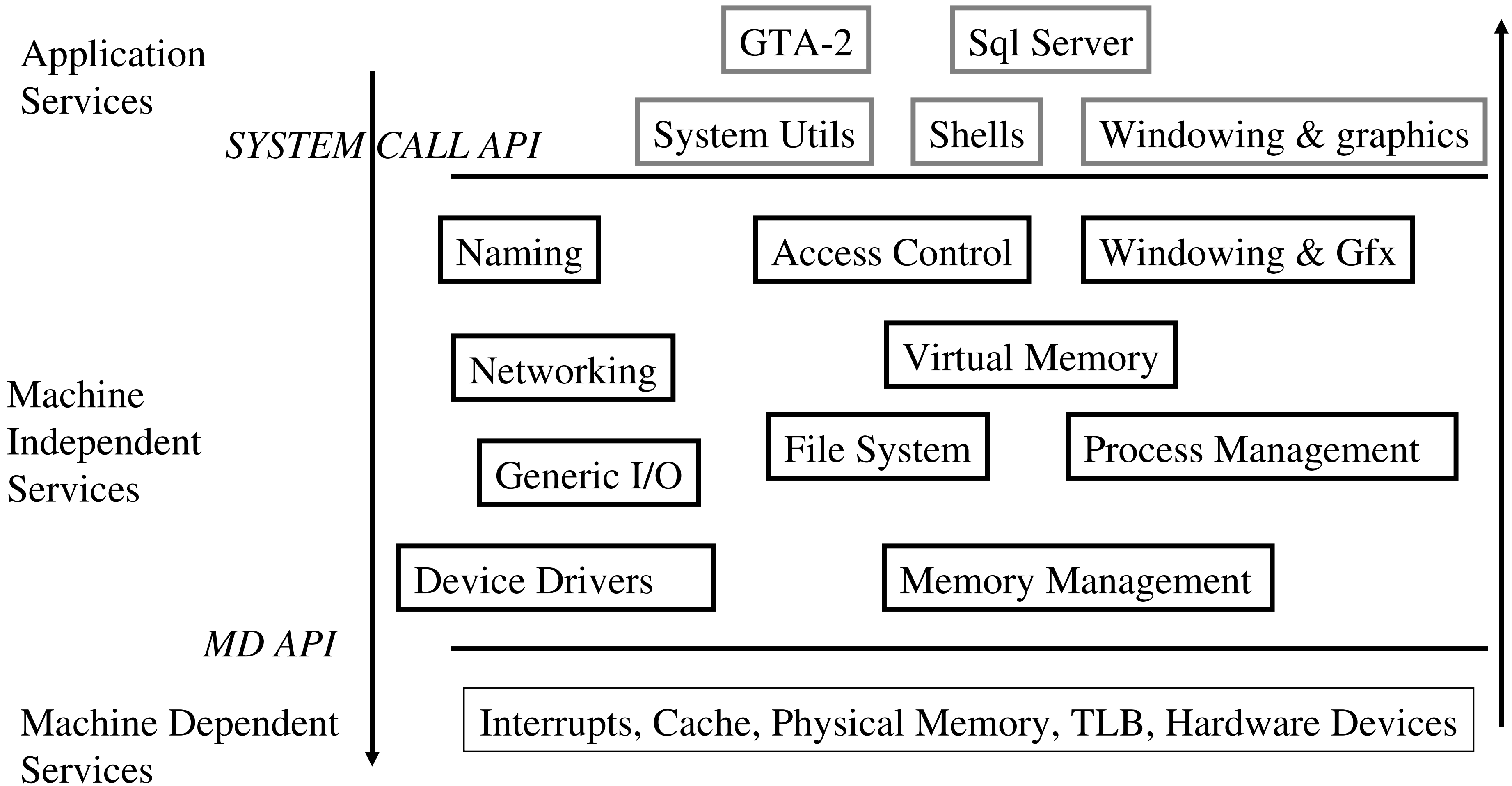
- What is it?
  - Textbook:
    - "… manages the computer hardware"
    - "… basis for application programs"
  - Once upon a time:
    - Programs were run one at a time, no multitasking
    - If you wanted to read data, you wrote the code to read from the punch card reader
    - If you wanted to output data, you wrote code to flash lights or to make the printer do things
    - If your application "crashed", YOU (or the operator) would push a button on the computer to get it to restart, and read the next program from the card reader
    - Was this an appropriate use of YOUR time?

What is an OS?

- How can we make this easier?
  - Let programs share the hardware (CPU, memory, devices, storage)
  - Supply software to *abstract* hardware (disk vs net or wireless mouse vs optical mouse vs wired mouse)
    - *Abstract* means to hide details, leaving only a common skeleton
  - *"All the code you didn't write"* in order to get your application to run. The little box, below, is simple, no?



What's in an OS?



Logical OS Structure

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
    - OS multiplexes resources across programs
  - efficiency (cost and speed)
    - share one computer across many users
    - concurrent execution of multiple programs

The major OS issues

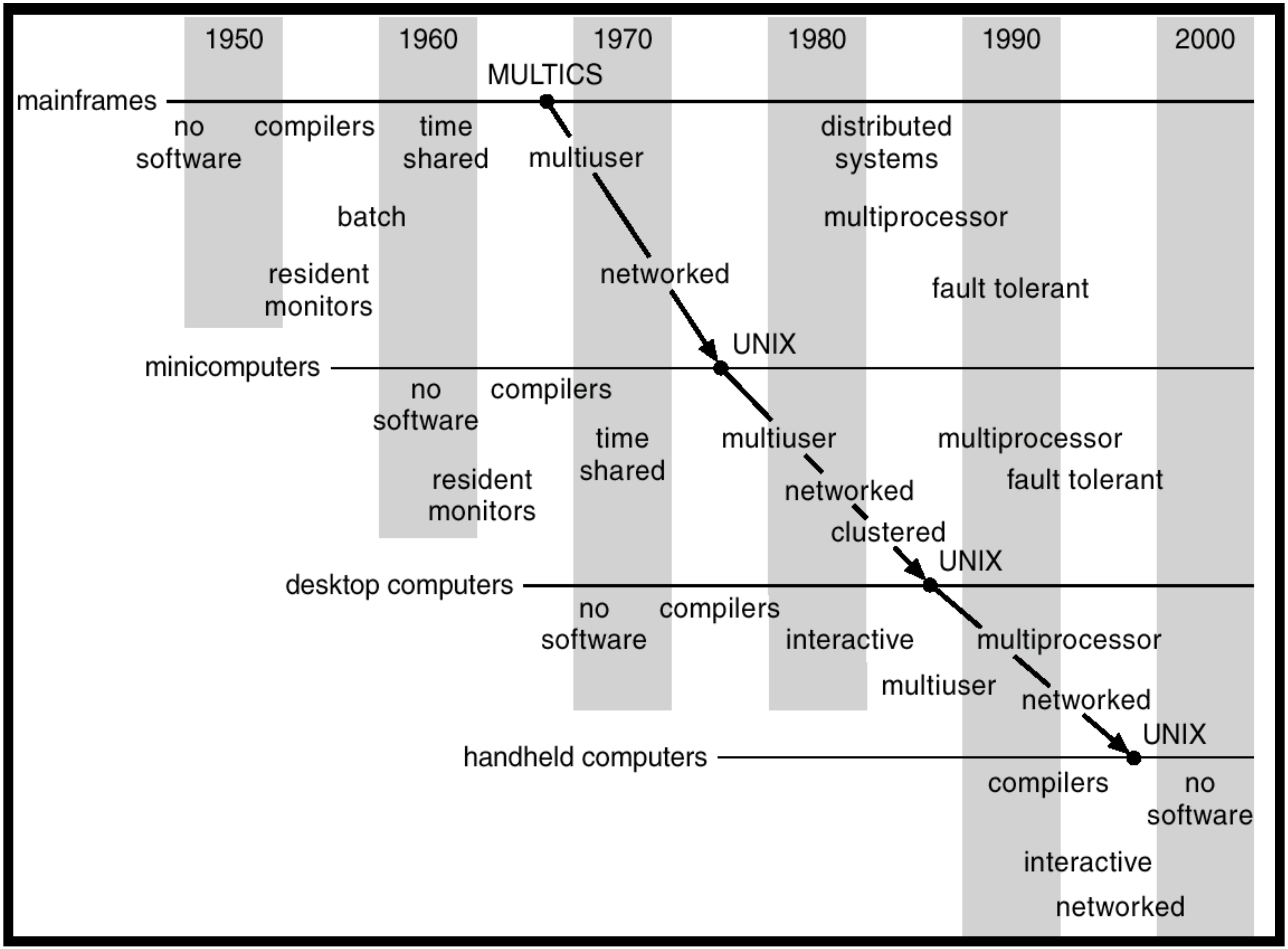
- structure: how is the OS organized? What are the resources a program can use?
- sharing: how are resources shared across users?
- naming: how are resources named (by users or programs)?
- security: how is the integrity of the OS and its resources ensured?
- protection: how is one user/program protected 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?

Major issues in OS (2)

- Concurrency: how are parallel activities created and controlled?
- Scale and growth: what happens as demands or resources increase?
- Persistence: how to make data last longer than programs
- Compatibility & Legacy Apps: can we ever do anything new?
- Distribution: Accessing the world of information
- Accounting: who pays the bills, and how do we control resource usage?

- These are engineering trade-offs
- Based on objectives and constraints

Progression of concepts and form factors



Has it all been discovered?

- New challenges constantly arise
  - embedded computing (e.g., iPod)
  - sensor networks (very low power, memory, etc.)
  - peer-to-peer systems
  - ad hoc networking
  - scalable server farm design and management (e.g., Google)
  - software for utilizing huge clusters (e.g., MapReduce, BigTable)
  - overlay networks (e.g., PlanetLab)
  - worm fingerprinting
  - finding bugs in system code (e.g., model checking)

Has it all been discovered?

- Old problems constantly re-define themselves
  - the evolution of PCs recapitulated the evolution of minicomputers, which had recapitulated the evolution of mainframes
  - but the ubiquity of PCs re-defined the issues in protection and security

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
- stuff you download and run knowingly (bugs, trojan horses)
- stuff you download and run obliviously (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
    - 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/drum
    - 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…

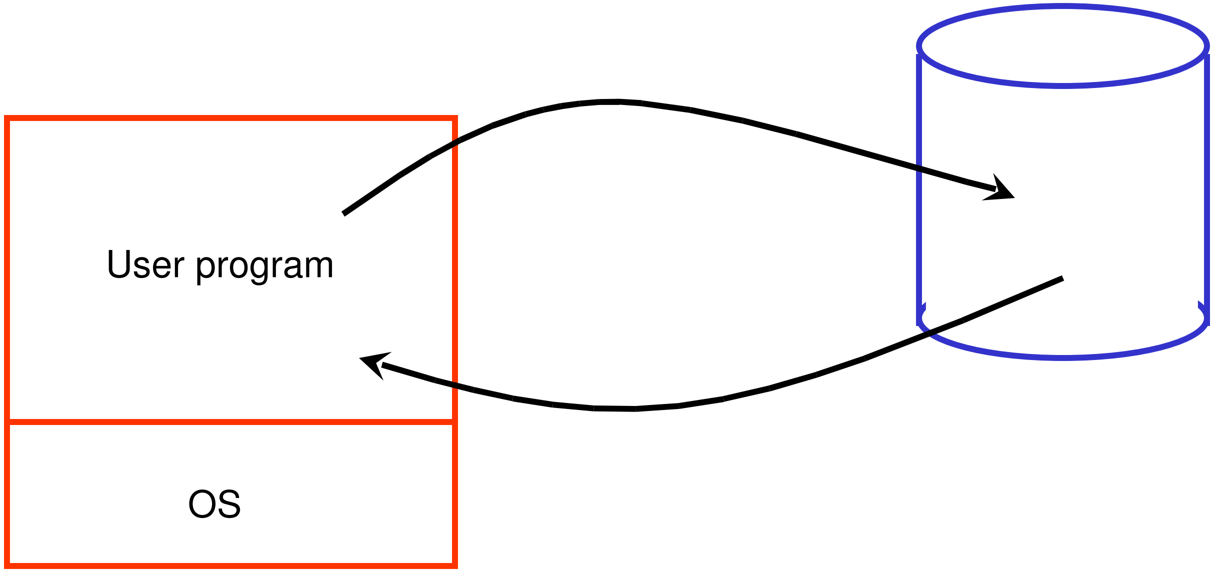
Timesharing

- 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?)

Timesharing

- MIT CTSS system (operational 1961) was among the first timesharing systems
  - only one user memory-resident at a time (32KB memory!)
- MIT Multics system (operational 1968) was the first large timeshared system
  - nearly all OS concepts can be traced back to Multics!
  - “second system syndrome”

- CTSS as an illustration of architectural and OS functionality requirements



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)
    - MPPs (massively parallel processors)
    - NOWs (networks of workstations)
    - computational grid (SETI @home)

Personal computing

- Primary goal was to enable new kinds of applications
- Bit mapped display [Xerox Alto,1973]
  – new classes of applications
  – 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
- Supports communications between programs
  - interprocess communication
    - message passing, shared memory
  - networking stacks
- Sharing of distributed resources (hardware, software)
  - load balancing, authentication and access control, …
- Speedup isn’t the issue
  - access to diversity of resources is goal

Client/server computing

- Mail server/service
- File server/service
- Print server/service
- Compute server/service
- Game server/service
- Music server/service
- Web server/service
- etc.

Peer-to-peer (p2p) systems

- Napster
- Gnutella
  - example technical challenge: self-organizing overlay network
  - technical advantage of Gnutella?
  - er … legal advantage of Gnutella?

Embedded/mobile/pervasive computing

- Pervasive computing
  - cheap processors embedded everywhere
  - how many are on your body now? in your car?
  - cell phones, PDAs, network computers, …
- Typically very constrained hardware resources
  - slow processors
  - very small amount of memory (e.g., 8 MB)
  - no disk
  - typically only one dedicated application
  - limited power
- But this is changing rapidly!

What is an OS?

- How were OS’s programmed?
  - Originally in assembly language
    - Maximal power, all features of the hardware exposed to developers
    - Minimal clarity, takes extreme effort
    - Minimal “portability”, OS is tightly tied to a single manufacturer’s architecture
    - GCOS (Honeywell/GE, ‘62), MVS and OS/360 (IBM, ‘64), TOPS-10 (Digital, ‘64)
  - Some special high-level languages
    - ESPOL, NEWP, DCALGOL (Burroughs, ‘61)
  - General high-level languages (with some assembly help)
    - PASCAL (UCSD p-system ’78, early Macintosh)
    - PL/1 (Multics, ’64)

What is an OS?

- What do we do today?
  - C
    - Adequate to hide most hardware issues
      - Precision, pointers
    - Procedural, reasonably type-safe, modular
    - Adequate for programmer to gauge efficiency
  - Plus some assembler
    - C does not reveal enough hardware
    - Assembler source files
    - In-line assembler in C files (only where it makes sense!)
  - Very little C+
    - Windows GUI completely in C++
    - Can hide inefficiencies!

CSE 451

- 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 …
- A good OS should be easily usable by everyone

Your next steps

- Familiarize yourself with course website
  - Read it often (daily)
- Get on cse451 mailing list. Read your email daily
- Read Chapters one and two by Wednesday
- Make sure you are familiar with C
  - Write and debug legible and correct code
  - Read, understand, and modify other's code