CSE 451: Operating Systems
Winter 2013

Introduction to Operating Systems

Gary Kimura
Introduction to Operating Systems

• What is it?
  • “… manages the computer hardware”
  • “… basis for application programs”
  • “Software to manage a computer’s resources for its users and applications”

• 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 \textit{abstract} hardware (disk vs net or wireless mouse vs optical mouse vs wired mouse)
    • \textit{Abstract} means to hide details, leaving only a common skeleton
  – “\textit{All the code you didn’t write}” in order to get your application to run. The little box, below, is simple, no?

\begin{center}
\begin{tikzpicture}
\draw[step=1cm,black,thin] (0,0) grid (3,3);
\draw[red,thick] (0.5,0) rectangle (2.5,1.5);
\draw[blue,thick] (0.5,0) rectangle (2.5,3);
\node at (1.5,1) {Applications};
\node at (1.5,0.5) {OS};
\node at (1.5,0) {Hardware};
\end{tikzpicture}
\end{center}
What's in an OS?

Logical OS Structure

Application Services
- System Utilities
- Shells
- Windowing & Gfx
- GTA-2
- Sql Server

Machine Independent Services
- Application Services
  - System Call API
    - Naming
    - Access Control
    - Networking
    - Virtual Memory
    - Generic I/O
    - File System
    - Process Management
    - Memory Management
- Device Drivers

Machine Dependent Services
- MD API
  - Interrupts, Cache, Physical Memory, TLB, Hardware Devices
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?
  - 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, not right and wrong
- Based on objectives and constraints
Progression of concepts and form factors

- **Mainframes**
  - 1950: Mainframes
  - No software
  - Compilers
  - Resident monitors
  - Batch

- **Multics**
  - 1970: Multiuser
  - Networked
  - Unix

- **Minicomputers**
  - 1960: No software
  - Compilers
  - Resident monitors
  - Time shared

- **Desktop computers**
  - 1980: Interactive
  - Multiuser
  - Networked

- **Handheld computers**
  - 1990: Multiprocessor
  - Fault tolerant
  - Networked

- **2000**
  - Distributed systems
  - Multiprocessor
  - Fault tolerant

© Silberschatz, Galvin and Gagne
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

• Before the beginning
  – Computers were rare, huge, power-sucking and hugely expensive
  – More expensive than *people*. This leads to a huge effort to make the most out of the hardware

• 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 (sometimes 10’s of minutes just to read in a program)
    • so CPU was idle much of the time (wastes an expensive resource)
Spooling

- Disks were much faster than card readers and printers (once they were invented)
- 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 (wastes an expensive resource)
  - 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 computation 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. That’s ok until people start getting more expensive than computers…
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
    (person-time more expensive than computer time!)

• 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”
Parallel systems

• Some applications can be written as multiple activities
  – 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 (Amdahl’s Law)
  – many flavors of parallel computers today
    • SMPs (symmetric multi-processors, multi-core)
    • MPPs (massively parallel processors)
    • NOWs (networks of workstations)
    • computational grid (SETI @home, FoldIt!)
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
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++, next to zero Java
    • Windows GUI completely in C++
    • Can hide inefficiencies!