Operating Systems History and Overview

Portions of this material courtesy Profs. Wong and Stark

So what is an OS?

One view of an OS

Another simple view of an OS

A less happy view of an OS

So which one is right?

- They all are
An OS serves three masters
1. Give users a desktop environment
2. Give applications a more usable abstraction of the hardware
3. Give hardware manufacturers an abstraction of the applications

Background (1)
- CPUs have 2 modes: user and supervisor
  - Sometimes more, but whatevs
- Supervisor mode:
  - Issue commands to hardware devices
  - Power off, Reboot, Suspend
  - Launch missiles, Do awesome stuff
- User mode:
  - Run other code, hardware tattles if you try anything reserved for the supervisor

OS architecture

Master #2: Applications
- Application Programming Interface (API)
  - Win32 (Windows)
  - POSIX (Unix/Linux)
  - Cocoa/Cocoa Touch (Mac OS/iOS)
- Application-facing functions provided by libraries
  - Injected by the OS into each application
Famous libraries, anyone?
- Windows: ntdll.dll, kernel32.dll, user32.dll, gdi32.dll
- Linux/Unix: libc.so, ld.so, libpthread.so, libm.so

Caveat 1
- Libraries include a lot of code for common functions
  - Why bother reimplementing sqrt?
- They also give high-level abstractions of hardware
  - Files, printer, dancing Homer Simpson USB doll
- How does this work?

System Call
- Special instruction to switch from user to supervisor mode
- Transfers CPU control to the kernel
  - One of a small-ish number of well-defined functions
- How many system calls does Windows or Linux have?
  - Windows ~1200
  - Linux ~350

Caveat 2
- Some libraries also call special apps provided by the OS, called a daemon (or service)
  - Communicate through kernel-provided API
- Example: Print spooler
  - App sends pdf to spooler
  - Spooler checks quotas, etc.
  - Turns pdf into printer-specific format
  - Sends reformatted document to device via OS kernel
Master 3: Hardware

- OS kernels are programmed at a higher low level of abstraction
  - Disk blocks vs. specific types of disks
- For most types of hardware, the kernel has a “lowest common denominator” interface
  - E.g., Disks, video cards, network cards, keyboard
  - Think Java abstract class
  - Sometimes called a hardware abstraction layer (HAL)
- Each specific device (Nvidia GeForce 600) needs to implement the abstract class
  - Each implementation is called a device driver

What about Master 1

- What is the desktop?
- Really just a special daemon that interacts closely with keyboard, mouse, and display drivers
  - Launches programs when you double click, etc.
  - Some program libraries call desktop daemon to render content, etc.

An OS serves three masters

1. Give users a desktop environment
   - Desktop, or window manager, or GUI
2. Give applications a more usable abstraction of the hardware
   - Libraries (+ system calls and daemons)
3. Give hardware manufacturers an abstraction of the applications
   - Device Driver API (or HAL)

Multiplexing Resources

- Many applications may need to share the hardware
- Different strategies based on the device:
  - Time sharing: CPUs, disk arm
    - Each app gets the resource for a while and passes it on
  - Space sharing: RAM, disk space
    - Each app gets part of the resource all the time
  - Exclusive use: mouse, keyboard, video card
    - One app has exclusive use for an indefinite period
So what is Linux?

• Really just an OS kernel
  – Including lots of device drivers
• Conflated with environment consisting of:
  – Linux kernel
  – Gnu libc
  – X window manager daemon
  – CUPS printer manager
  – Etc.

So what is Ubuntu?  Centos?

• A distribution: bundles all of that stuff together
  – Pick versions that are tested to work together
  – Usually also includes a software update system

OSX vs iOS?

• Same basic kernel (a few different compile options)
• Different window manager and libraries

What is Unix?

• A very old OS (1970s), innovative, still in use
• Innovations:
  – Kernel written in C (first one not in assembly)
  – Co-designed C language with Unix
  – Several nice API abstractions
  – Fork, pipes, everything a file
• Several implementations: *BSDs, Solaris, etc.
  – Linux is a Unix-like kernel

What is POSIX?

• A standard for Unix compatibility
• Even Windows is POSIX compliant!

History of Operating Systems

• Two ways to look at history:
  – Evolution of the Theory
  – Evolution of the Machine/Hardware
Evolution of OS Theory

1. Centralized operating system
   - Resource management and multiprogramming, Virtuality
2. Network operating system
   - Resource sharing to achieve Interoperability
3. Distributed operating system
   - Single computer view of a multiple computer system for Transparency
4. Cooperative autonomous system
   - Cooperative work with Autonomicity

Evolution of OS Machine/Hardware

1940’s – First Computers

- One user/programmer at a time (serial)
  - Program loaded manually using switches
  - Debug using the console lights
- ENIAC
  - 1st gen purpose machine
  - Calculations for Army
  - Each panel had specific function

Pros:
- Interactive – immediate response on lights
- Programmers were women

Cons:
- Lots of idle time
- Expensive computation
- Error-prone/tedious
- Each program needs all driver code

1950’s – Batch Processing

- Deck of cards to describe job
- Jobs submitted by multiple users are sequenced automatically by a resident monitor
- Resident monitor was a basic O/S
  - S/W controls sequence of events
  - Command processor
  - Protection from bugs (eventually)
  - Device drivers

Monitor’s Perspective

- Monitor controls the sequence of events
- Resident Monitor is software always in memory
- Monitor reads in job and gives control
- Job returns control to monitor
1950's – Batch Processing

Pros:
- CPU kept busy, less idle time
- Monitor could provide I/O services

Cons:
- No longer interactive – longer turnaround time
- Debugging more difficult
- CPU still idle for I/O-bound jobs
- Buggy jobs could require operator intervention

IBM 7090

Pros:
- CPU kept busy, less idle time
- Monitor could provide I/O services

Cons:
- No longer interactive – longer turnaround time
- Debugging more difficult
- CPU still idle for I/O-bound jobs
- Buggy jobs could require operator intervention

IBM 7090

Multiprogrammed Batch Systems

- CPU is often idle
  - Even with automatic job sequencing.
  - I/O devices are slow compared to processor

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read one record from file</td>
<td>15 µs</td>
</tr>
<tr>
<td>Execute DC instructions</td>
<td>3 µs</td>
</tr>
<tr>
<td>Write one record to file</td>
<td>15 µs</td>
</tr>
<tr>
<td>TOTAL</td>
<td>33 µs</td>
</tr>
</tbody>
</table>

CPU Utilization: \(\frac{33}{120} = 0.275 = 27.5\%\)

Figure 2.4 System Utilization Example

Uniprogramming

- Processor must wait for I/O instruction to complete before preceding

Multiprogramming

- When one job needs to wait for I/O, the processor can switch to the other job

1960's – Multiprogramming (time-sharing)

- CPU and I/O devices are multiplexed (shared) between a number of jobs
  - While one job is waiting for I/O another can use the CPU
  - SPOOLing: Simultaneous Peripheral Operation OnLine
    - 1st simplest multiprogramming system
- Monitor (resembles O/S)
  - Starts job, spools operations, I/O, switch jobs, protection between memory
1960’s – Multiprogramming (time-sharing)

Pros:
- Paging and swapping (RAM)
- Interactivity
- Output available at completion
- CPU kept busy, less idle time

Cons:
- H/W more complex
- O/S complexity?

1970’s - Minicomputers and Microprocessors
- Trend toward many small personal computers or workstations, rather than a single mainframe.
- Advancement of Integrated circuits
- Timesharing
  - Each user has a terminal and shares a single machine (Unix)

1980’s – Personal Computers & Networking
- Microcomputers = PC (size and $)
- MS-DOS, GUI, Apple, Windows
- Networking: Decentralization of computing required communication
  - Not cost-effective for every user to have printer, full copy of software, etc.
  - Rise of cheap, local area networks (Ethernet), and access to wide area networks (Arpanet).

OS issues:
- Communication protocols, client/server paradigm
- Data security, encryption, protection
- Reliability, consistency, availability of distributed data
- Heterogeneity
- Reducing Complexity
- Ex: Byte Ordering
- CPU/RAM/disk speed mismatch
- Send data to program vs. sending program to data
- QoS guarantees
- Security

1990’s – Global Computing
- Dawn of the Internet
  - Global computing system
- Powerful CPUs cheap! Multicore systems
- High speed links
- Standard protocols (HTTP, FTP, HTML, XML, etc)
- OS Issues:
  - Communication costs dominate
    - CPU/RAM/disk speed mismatch
    - Send data to program vs. sending program to data
  - QoS guarantees
  - Security

In the year 2000...
2000’s – Embedded and Ubiquitous Computing
• Mobile and wearable computers
• Networked household devices
• Absorption of telephony, entertainment functions into computing systems
• OS issues:
  – Security, privacy
  – Mobility, ad-hoc networks, power management
  – Reliability, service guarantees

Multi-core
• New hotness in CPU design. Not going away.
  – Why?
• Being able to program with threads and concurrent algorithms will be a crucial job skill going forward
  – Don’t leave SBU without mastering these skills
  – We will do some thread programming in Lab 3

Editorial
• Some textbooks imply modern OSes are microkernels
• This is false
  – Windows NT and OSX were designed as microkernels
  – Then reverted to essentially monolithic designs in practice
• Linux was never a microkernel
  – Google the famous Torvalds Tanenbaum debate
• Similarly, Distributed OSes are mostly abandoned

Object orientation
• Objects are a key feature of the Windows NT kernel design
  – IMO a good idea
• Linux actually has its own bizarre version of object orientation using C structs and function pointers
  – In Unix, everything is a file
  – How did they pull this off?
  – Poor-man’s object inheritance

Summary
• OS’s began with big expensive computers used interactively by one user at a time.
• Batch systems sequences jobs to keep computer busier. Interactivity sacrificed.
• Multiprogramming developed to make more efficient use of expensive hardware and restore interactivity.
• Cheap CPU/memory/storage make communication the dominant cost.
• Multiprogramming still central for handling concurrent interaction with environment.
Summary (2)

• Understand what an OS is
  – Three masters
  – Nomenclature
• Questions?