Today

- Operating System Evolution
- Unix Overview
  - Unix Structure
  - Shells and Utilities
  - System Calls and APIs

---

Operating System Evolution

- How did the OS evolve?
  - Dependent on hardware and technology
  - Dependent on applications and usage

Gen 1: Mono-programming (1945-55)

- Large computers (building-size)
- Mechanical relays, vacuum tubes
- Direct programming of hardware
- Basically no OSI
- Similar to a programmer writing raw binary code

---

CSCI 4061
Introduction to Operating Systems

Instructor: Abhishek Chandra
- Mainframes
- Programmers submit multiple jobs
  - Stack of punch cards
- Programs were batched and fed into the system
- Results were output after a few hours
- OS did some elementary job scheduling
- Similar to today's supercomputers

Gen 3: Multi-programming (1965-80)
- Multiple programs executing in parallel
- Allow more efficient use of CPU and I/O
- Time-sharing: Variant of multi-programming
  - Fast time-multiplexing between multiple jobs
  - Each user gets single-user view
- OS performs resource management and control
- MULTICS (MULTiplexed Information and Computing Service) and UNIX

Gen 4: Personal Computers (1980-)
- Started as single-user systems
- Evolved into multi-user, multi-programming systems
- Main focus: Ease of use
- Evolution of GUIs
- OS provides simple abstractions and resource management
- Examples: DOS, Windows, MacOS, Linux, FreeBSD

Gen 5: Distributed Systems (1985-)
- Multiple computers linked by a network/bus
- Various flavors:
  - Multiprocessor systems
  - Client-server systems
  - Peer-to-peer systems and Grids
  - Cloud computing
- Focus: Interaction of multiple independent entities
Gen 6: Mobile Computing (2000-)

- Handheld and wearable devices: smart phones, tablets, watches, ...
- Main constraints: power, size/weight
- Limited resources:
  - Slower CPU, less memory storage
- Many sensors:
  - Cameras, GPS, accelerometers, ...
- Focus: energy-efficiency, mobility, context
- Examples: Google Android, Apple iOS

Other Operating System Flavors

- Embedded OS
  - Limited CPU, memory, battery
  - E.g.: Home devices, environmental sensors, IoT
  - Special-purpose functionality
- Real-time OS
  - Time-based guarantees
  - E.g.: Space rockets, cars, production machinery

General Operating System Structure

<table>
<thead>
<tr>
<th>Applications</th>
<th>Operating System</th>
<th>Hardware</th>
</tr>
</thead>
</table>

- Hardware-OS interactions
- OS-Application interactions

Kernel

- Core of the Operating System
- Provides the main functionality:
  - Process and Thread Management
  - Memory Management
  - File System and I/O
  - Inter-Process Communication
Hardware-OS interactions

- We would mainly worry about hardware abstractions
  - E.g.: Processes, files, virtual memory, sockets
  - Control allocation and management of CPU, disks, devices, memory, network interfaces
  - Portability across variety of hardware
  - Asynchronous events and concurrency. E.g.: interrupts, I/O events

OS-Application interactions

- Shells and User interfaces
  - Allow users to interact with the OS
- Libraries
  - Allow programs to use common services
- System calls
  - Direct conduit into the OS
- Signals
  - OS interacting with user programs

Key Questions

- Q.1: How does the OS protect the hardware from (unruly or malicious) applications?
- Q.2: How do applications get access to desired resources (CPU, memory, disk, etc.)?

Kernel Mode

<table>
<thead>
<tr>
<th>User Mode</th>
<th>Kernel Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
<td></td>
</tr>
</tbody>
</table>

- OS runs in kernel mode: hardware-enabled
- Higher privileges than user mode
  - Access to hardware resources
  - Access to protected memory
  - Access to OS data structures
- Tighter control, security of system resources
**User Mode**

- Applications, utilities, shell run in user mode
- Restricted access to
  - System resources
  - Kernel data structures
- Protection boundaries

**Key Questions**

- Q.1: How does the OS protect the hardware from (unruly or malicious) applications?
- Q.2: How do applications get access to desired resources (CPU, memory, disk, etc.)?

**Example Scenario**

- Suppose an application needs to read data from a file
- Can the application directly read data from the disk?
- How does the program tell the OS to read the data for it?

**System Calls**

- Kernel API: well-defined, small set of operations
- Entry points into the kernel
- Provide restricted access to the kernel
System Call Operation

- User program executes a TRAP instruction
  - Switches to kernel mode
- Passes parameters, system call no.
- Kernel looks up system call table
- System call handler is invoked
- Results returned to user program

System Call Example: read

- E.g.: read 16 bytes from file F into buffer
- Parameters: (F, buffer, 16)
- User program passes:
  - Parameters, read syscall no. (3)
- User program executes TRAP
- Kernel looks up code for syscall no. 3
- Kernel executes code for read system call
  - Look up F (File system)
  - Transfer 16 bytes from disk to buffer (I/O)
- System call returns control to user space

System Call Implementation

- Requires performing TRAP instruction
- Requires passing parameters to kernel
  - In registers or in buffers
- Requires getting back control from kernel
- May be implemented in assembly language
- How to use in programs easily?

Libraries

- Pre-written and pre-tested functions
- Programmers can call library functions in their programs
- Two types:
  - Wrappers for system calls
  - User-level utility functions
System Call Wrapper Libraries
- Provide convenient function call to programmers
- Library call does the hard work:
  - Packaging the parameters
  - Executing the system call
  - Extracting the results
- E.g.: read library call
- Multiple library calls could map to same syscall
  - execve -> execl, execlp, execle, execv, execvp

User-level Utility Libraries
- Implement commonly-used functions
- Programmers don't have to reinvent the wheel
- Examples:
  - Random number generation (rand, srand, ...)
  - String operations (strcpy, strcmp, strlen, ...)

Advantages of Libraries
- Reduce load on programmers
- Standard functions
- Hide complexity
- Efficient implementation
- Available for high-level languages

Unix Overview
What is Unix?

- Highly Popular OS
- Programming Platform/Toolbox
- Many variants exist
  - Multiple implementations, similar functionality
- Principles:
  - Multiprogramming
  - Flexibility
  - Extensibility

Unix Evolution

- MULTICS (1965): MULTiplexed Information and Computing Service
- UNIX (1969): Developed at Bell Labs
  - Developed in C
  - Architecture-independent
- Several Evolution Paths:
  - AT&T System V (1983) -> SCO Unix
  - BSD (1980) -> FreeBSD, NetBSD
  - Linux (1991-) -> Android
  - Other variants: Solaris, HP/UX, IBM AIX, Apple OS/X

Common Features

- Similar tools and user interfaces
- Similar APIs
- POSIX Standards

Variations

- Shells, tools, and user interfaces
- APIs
- Underlying implementation
Unix: User’s View

Operating System Structure

Unix Structure

Shell

- Basic interface to the OS
- Allows users to interact with the OS
- Command interpreter
  - Parses and executes several commands
- Many Unix Shells
  - Bourne sh
  - csh, tcsh,
  - ksh,
  - bash
Shell as Command Interpreter

- Read-eval-print
  - Read user input
  - Evaluate the input and execute command(s)
  - Print output
- Example: `ls -l`
  - User types above command
  - Shell reads and parses the input
  - Shell invokes "ls" command with argument "-l"
  - Shell prints the output produced by the command

Shell as a Program

- Shell is just another user-level program
  - No special privileges
- Primary Objective: Run other useful programs
- Shell interacts with the OS using the API and through other programs
- Same functionality can be obtained using a C/Java/... program

Shell Commands

- Built-in commands
  - Implemented within the shell itself
  - E.g.: alias, echo, eval
- External commands
  - Other programs executed by the shell
  - Path needs to specified (or be in the environment)
  - File should be executable
  - E.g.: "ls" is the file "/bin/ls"

Environment variables

- Environment: A set of variables associated with the shell
- Useful for the shell to remember common settings
- Examples:
  - PATH: Set of directories to look for commands
  - HOME: Home directory of the user
  - PS1: Default prompt setting
  - SHELL: Path of the shell itself
  - x, foo: user-defined
Standard I/O
- Standard input (stdin): source of input data for a command or program
  - Default: Keyboard
- Standard output (stdout): destination of output from a command or program
  - Default: Display
- Standard error (stderr): destination of error messages
  - Default: Display

Shell Redirection
- Redirect the input or output of a command to a file
- Input redirection (<): Takes input from a file instead of keyboard
  - sort < file
- Output redirection (>): Send output to a file instead of display
  - ls -l > file
  - cat file1 >> file2

Shell Plumbing
- Pipes: prog1 | prog2
  - Allow multiple commands to be linked together
  - Connects output of prog1 to input of prog2
- Examples:
  - ls -l | more
  - cat foo | sort | head
- How will you do the above using only redirection?
- Most UNIX commands consume and produce plain text data

Tools and Utilities
- Unix is a toolbox
  - Several utilities come packaged/can be easily installed
- Shells and GUIs. E.g.: bash, X Windows
- Applications:
  - Editors. E.g.: vi, emacs
  - Compilers and debuggers. E.g.: gcc, gdb
  - Browsers. E.g.: Firefox
Unix File System

- Tree-based hierarchy
- File system has a root (/)
- Each user has a home directory ($HOME)
- There are several standard directories:
  - E.g.: /bin (binary files), /dev: device files, /lib: libraries, ...

Unix Directory Hierarchy

Unix Summary

- Applications, Utilities, and User Programs
  - Shell
  - Libraries
  - System Call Interface
  - Kernel
    - Processes, File System, Virtual Memory, Threads, Sockets
- Operating System
- Hardware
  - CPU, Memory, Disks, Devices
  - AUI (Application User Interface)