OS Overview & Structure
CSE 410 Winter 2017

Instructor: Justin Hsia
Teaching Assistants: Kathryn Chan, Kevin Bi, Ryan Wong, Waylon Huang, Xinyu Sui

Internet of Things Teddy Bear Leaked 2 Million Parent and Kids Message Recordings

Spiral Toys left customer data of its CloudPets brand on a database that wasn't behind a firewall or password-protected.

As we've seen time and time again in the last couple of years, so-called "smart" devices connected to the internet—what is popularly known as the Internet of Things or IoT—are often left insecure or are easily hackable, and often leak sensitive data.

After this story was published, a security researcher revealed that the stuffed animals themselves could easily be hacked and turned into spy devices.

Quick Virtual Memory Review

- What do Page Tables map?
  
  VPN → PPN or disk address

- Where are Page Tables located?
  
  physical memory

- How many Page Tables are there?
  
  one per process

- Can your program tell if a page fault has occurred?
  
  No. The MMU throws the page fault exception; process just waits

- True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory
  
  True

- TLB stands for translation lookaside buffer and stores page table entries
Administrivia

- Lab 4 due Tuesday (3/7)
- Homework 5 due Thursday (3/9)

- Course evaluations now open
  - [https://uw.iastate.org/survey/172382](https://uw.iastate.org/survey/172382)

- **Final Exam:** Tue, Mar. 14 @ 2:30pm in MGH 241
  - Review Session: Sun, Mar. 12 @ 1:30pm in SAV 264
  - Cumulative (midterm clobber policy applies)
  - TWO double-sided handwritten 8.5×11” cheat sheets
    - Recommended that you reuse/remake your midterm cheat sheet
What is an Operating System?

Answers:

- “I don't know” – Ed Lazowska, longtime OS researcher
- Nobody knows
- The book has some ideas – see Section 1.7
- They’re programs – big, hairy programs
  - The Linux source has over 1.7 million lines of C code

Okay. Then what are some goals of an OS?
The Traditional Picture

- “The OS is everything you don’t need to write in order to run your application”
- This depiction invites you to think of the OS as a library
  - In some ways, it is:
    - All operations on I/O devices require OS calls (syscalls)
  - In other ways, it isn't:
    - You use the CPU/memory without OS calls
    - It intervenes without having been explicitly called
The OS and Hardware

- An OS *mediates* programs’ access to hardware resources (sharing and protection)
  - Computation (CPU)
  - Volatile storage (memory) and persistent storage (disk)
  - Network communications (TCP/IP stacks, Ethernet cards)
  - Input/output devices (keyboard, display, sound card)

- The OS *abstracts* hardware into *logical resources* and well-defined *interfaces* to those resources (*ease of use*)
  - Processes (CPU, memory)
  - Files (disk)
  - Programs (sequences of instructions)
  - Sockets (network)
Application Benefits of an OS

- Programming **simplicity**
  - Can deal with 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 of an OS

- **Safety**
  - Gives each process the illusion of its own virtual machine
  - *Protects* programs from each other
  - “Fairly” *multiplexes* resources across programs

- **Efficiency** (cost and speed)
  - *Share* one computer across many users
  - Allows for *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?
- **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 (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:** 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?
- **Accounting:** How do we keep track of resource usage, and perhaps charge for it?

These are recent issues: multicore, warehouse scale computers, cloud computing, cluster computing, distributed computing.

There are tradeoffs, not right and wrong!
Architectural features affecting OS’s

- These features were built primarily to support OS’s:
  - Timer (clock) operation
  - Synchronization instructions (*e.g.* atomic test-and-set)
  - Memory protection
  - I/O control operations
  - Interrupts and exceptions
  - Protected modes of execution (kernel vs. user)
  - Privileged instructions
  - System calls (and software interrupts)
  - Virtualization architectures
Privileged instructions

- **Privileged** instructions can only be executed by the OS
  - Directly access I/O devices (disks, network cards)
  - Manipulate memory state management
    - Page table pointers, TLB loads, etc.
  - Manipulate special ‘mode bits’
    - Interrupt priority level

- The architecture must support at least two modes of operation: **kernel** mode and **user** mode
  - Mode is set by status bit in a protected processor register
  - User programs must call an OS procedure to do something privileged \((\text{system calls})\)
System Calls

- **syscall instruction atomically:**
  - Saves the current PC
  - Sets the execution mode to privileged
  - Sets the PC (%rip) to a handler address

- It’s a lot like a **protected procedure call**
  - Caller puts arguments in a place callee expects
    - One of the arguments is the syscall number
  - Callee (OS) saves caller’s state (registers, other control state) so it can use the CPU
  - OS must **verify caller’s arguments**, then runs handler code
  - OS returns using a special instruction (**sysret**)
A Kernel Crossing Illustrated

Chrome: `read(int fileDescriptor, void *buffer, int numBytes)`

1. **User Mode**
   - Save user PC
   - `PC = trap handler address`
   - Enter kernel mode

2. **Kernel Mode**
   - **Trap Handler**
     - Save process state
     - Verify `syscall` number
     - Find `sys_read()` handler in vector table
   - **Sys_read() Kernel Routine**
     - Verify arguments
     - Initiate read
     - Choose next process to run
     - Setup return values
     - Restore app state

3. **System Exit**
   - `sysret` instruction
   - `PC = saved PC`
   - Enter user mode
OS Structure

- The OS sits between application programs (P for processes) and the hardware (D for devices)
  - It mediates access and abstracts away ugliness
  - Programs request services via *traps* or *exceptions*
  - Devices request attention via *interrupts*
Major OS Components

- Processes
- Memory
- Input/Output
- Secondary Storage (Disk)
- File Systems
- Protection
- Shells (command interpreter, or OS UI)
- GUI
- Networking
The Classic Diagram

Hardware (CPU, devices)

Hardware Abstraction Layer

Boot & Init

Interrupt Handlers

Process Manager

Memory Manager

File Systems

Application Interface (API)

Portable

Device Drivers

Network Support

Operating System

User Apps

Chrome

Powerpoint

Slack

Acrobat
But reality isn’t always that simple...
Process management

- An OS executes many kinds of processes:
  - Users’ programs
  - Batch jobs or scripts
  - System programs
    - Print spoolers, name servers, file servers, network daemons, ...

- Process includes the execution context
  - PC, registers, VM, OS resources (e.g. open files), etc...
  - The program itself (code and data)

  Process A: | code, stack | page tables |
              | PC, registers | resources |

- The OS’s **process module** manages these processes
States of a User Process

- **Running**
- **Ready**
- **Blocked**
- **Finished ("Zombie")**

- **terminate**
- **dispatch**
- **interrupt**
- **trap or exception**

- **difficult scheduling task**
Process Module Operations

- The OS provides the following kinds operations on processes (*i.e.* the process abstraction interface):
  - Create a process
  - Delete a process
  - Suspend a process
  - Resume a process
  - Clone a process
  - Inter-process communication
  - Inter-process synchronization
  - Create/delete a child process

*we've seen:*
- fork
- execve
- wait
- exit
Memory Management

- Main (primary) memory is directly accessed by CPU
  - Programs must be stored in memory to execute
  - But memory doesn’t survive power failures

- The OS must:
  - Allocate memory space for programs (explicitly & implicitly)
  - De-allocate space when needed by rest of system
  - Maintain physical to virtual memory mappings
  - Decide how much memory to allocate to each process
  - Decide when to remove a process from memory
Input/Output

- A big chunk of the OS kernel deals with I/O
- The OS provides a standard interface between programs and devices
  - e.g. file system (disk), sockets (network), frame buffer (video)
- Device drivers are the routines that interact with specific device types (encapsulate device-specific knowledge)
  - How to initialize a device, request I/O, handle interrupts or errors, etc.
  - Examples: SCSI device drivers, Ethernet card drivers, video card drivers, sound card drivers
Secondary Storage

- Secondary storage (disk, SSD) is persistent memory
  - Often magnetic media, survives power failures (hopefully)

- Routines that interact with disks are typically at a very low level in the OS
  - Used by many components (file system, VM, ...)
  - Handle scheduling of disk operations, head movement, error handling, and often management of space on disks

- Usually independent of file system
  - Although there may be cooperation
  - File system knowledge of device details can help optimize performance
    * e.g. place related files close together on disk
File Systems

- Secondary storage devices are crude and awkward

- A file system is a convenient abstraction
  - Defines logical objects like files and directories to hide details
  - Defines operations on objects like read and write

- A file is the basic unit of long-term storage
  - A file = a named collection of persistent information

- A directory is just a special kind of file
  - A directory = named file that contains names of other files and metadata about those files (e.g. file size)

- Note: Sequential byte stream is only one possibility!
File System Operations

- The file system interface defines standard operations:
  - File/directory creation, copy, and deletion
  - File/directory manipulation (e.g. read, write, extend, rename, protect, lock)

- File systems also provide higher level services:
  - Accounting and quotas
  - Backup (must be incremental and online!)
  - (Sometimes) indexing or search
  - (Sometimes) file versioning
Protection

- Protection mechanisms used throughout the OS help to detect and contain unintentional errors, as well as preventing malicious destruction.

- *All* resources need to be protected:
  - Memory
  - Processes
  - Files
  - Devices
  - CPU time
  - Etc.
Command Interpreter (shell)

- A particular program that handles the interpretation of users’ commands and helps to manage processes
  - User input may be from keyboard (command-line interface), from script files, or from the mouse (GUIs)
  - Allows users to launch and control new programs
- On some systems, command interpreter may be a standard part of the OS
  - Mostly for old/historical or tiny systems
- On others, it’s just non-privileged code that provides an interface to the user
  - *e.g.* `bash/csh/tcsh/zsh` on UNIX
OS Structure

- It’s not always clear how to stitch the OS modules together:
OS Structure

- An OS consists of all of these components, plus:
  - Many more!
  - System programs (privileged and non-privileged)
    - *e.g.* bootstrap code, the `init/systemd` program, ...

- Major issues:
  - What are all of the code modules, and where do they exist?
  - How do they cooperate?

- **Massive** software engineering and design problem
  - Must design a large, complex program that performs well, is reliable, is extensible, is backwards compatible, ...
Early OS Structure: Monolithic

- Traditionally, OS’s (like UNIX) were built as a monolithic entity:

```
+------------------+
| user programs    |
+------------------+
| OS              |
| everything       |
+------------------+
| hardware         |
+------------------+
```
Monolithic design

- **Major advantage:**
  - Cost of module interactions is low (procedure call)

- **Disadvantages:**
  - Hard to understand, modify, and maintain
  - Unreliable (no isolation between system modules)

- **What is the alternative?**
  - Find a way to organize the OS in order to simplify its design and implementation
Layering

- Implement OS as a set of layers
  - Layers can only invoke functions in lower layers
  - More organized and modular
  - Each layer can be tested and verified independently!

- Example layering approach:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 6:</td>
<td>Operator</td>
</tr>
<tr>
<td>Layer 5:</td>
<td>Job Managers</td>
</tr>
<tr>
<td>Layer 4:</td>
<td>I/O Management</td>
</tr>
<tr>
<td>Layer 3:</td>
<td>Console Manager</td>
</tr>
<tr>
<td>Layer 2:</td>
<td>Page Manager</td>
</tr>
<tr>
<td>Layer 1:</td>
<td>Kernel</td>
</tr>
<tr>
<td>Layer 0:</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

Execute users’ programs
Handle devices and provide buffering
Implements virtual consoles
Memory management
CPU scheduling and semaphores
Problems with Layering

- Imposes hierarchical structure
  - But real systems are more complex:
    - File system requires VM services (buffers)
    - VM would like to use files for its backing store
  - Strict layering isn’t flexible enough

- Poor performance
  - Each layer crossing has overhead associated with it

- Disjunction between model and reality
  - Systems modeled as layers, but not really built that way
Hardware Abstraction Layer

- An example of layering in modern operating systems
- **Goal:** Separate hardware-specific routines from the “core” OS
  - Provides portability
  - Improves readability

Diagram:

- **Core OS** (file system, scheduler, system calls)
- **Hardware Abstraction Layer** (device drivers, assembly routines)
Microkernels

- **Goal:** Minimize what goes in the kernel
  - Organize the rest of the OS as user-level processes

- **Advantages:**
  - Better reliability (isolation between components)
  - Ease of extension and customization

- **Disadvantages:**
  - Poor performance (user/kernel boundary crossings)
Microkernel Structure Illustrated

- User mode:
  - User processes:
    - chrome
    - powerpoint
    - spotify
    - apache
    - word
    - photoshop
  - System processes:
    - thread
    - file system
    - network
    - scheduling
    - paging
- Kernel mode:
  - Microkernel:
    - Communication
    - Processor control
    - Protection
    - Low-level VM
  - Hardware
Summary

- OS design has been an evolutionary process of trial and error – probably more error than success!
- Successful OS designs have run the spectrum from monolithic, to layered, to micro kernels, to virtual machine monitors
- The role and design of an OS are still evolving
- It is impossible to pick one “correct” way to structure an OS – it’s all about design trade-offs