CS 162: Operating Systems and Systems Programming

Lecture 1: Course Intro

June 24, 2019
Instructor: Jack Kolb
https://cs162.eecs.berkeley.edu
Challenge: Huge Scale
Example: What's in a search query?

- Complex interaction of multiple components in multiple administrative domains
  - Systems, services, protocols, …
Challenge: Variety and Number of Devices

- Number crunching, Data Storage, Massive Services, Mining
- Productivity, Interactive
- Streaming from/to the physical world
**Challenge: Range of Timescales**

---

**Jeff Dean:**

"Numbers Everyone Should Know"

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>
Operating systems are crucial to addressing these challenges!

• Provide consistent abstractions to apps, even on different hardware
  • File systems
  • Processes, threads
  • VM, containers
  • Naming system

• Manage resources shared among multiple applications:
  • Memory, CPU, storage, …

• Achieve the above by implementing specific algorithms and techniques:
  • Scheduling
  • Concurrency
  • Transactions
  • Security
Why take CS 162?

• Some of you will build operating systems or components of them

• Many of you will use core concepts from operating systems in your work (hardware or software)
  • Caching, batching, indirection, synchronization, …

• All of you will write applications that use an operating system
  • The better you understand the OS's design and implementation, the more effectively you can make use of it
Aside: Systems Programming

• What makes something a system?
• Frame of mind: meticulous error handling, anticipating all possible failure cases, defending against malicious/careless users
• An important part of this class!
Today's Agenda

1. What is an operating system?
2. Logistics
3. Break
4. Unifying OS Themes & Ideas
5. The OS's most important job
What is an operating system?

- Layer of software that provides application software access to hardware resources
  - Convenient abstraction of complex hardware
  - Protected access to shared resources
  - Security
  - Basic Services – Communication, Storage, …
Operators Through History

Switchboard Operator

Computer Operators
What is an operating system?

Provides user programs with a more convenient *abstract machine* interface rather than the underlying *physical machine*
#include <stdio.h>
int main(void) {
    printf("Hello!\n")
}
Application View of the World

- Application's "machine" is the OS
- No hardware details (future-proof)
- More useful interfaces than raw hardware
Operating system translates from hardware interface to application interface.
Abstraction: Program → Process

```
#include <stdio.h>
int main(void) {
  printf("Hello!\n");
}
```
Multiple Processes: Context Switch

```c
#include <stdlib.h>
int main(void) {
    printf("Hello!\n")
}
```
Abstraction: Input / Output

```
#include <stdlib.h>
int main(void) {
    printf("Hello!\n")
}
```

Operating System

ISA/Hardware interface

CPU

Screen | Keyboard | Storage | Network

Input/Output

Memory
Security & Protection

```c
#include <stdio.h>
int main(void) {
    FILE* fh = open("/etc/shadow");
}
```

Permission Denied
Security & Protection

#include <stdlib.h>
int main(void) {
...
}

Segmentation Fault

Operating System

OS's interface

Threads  Address Spaces  Windows  Sockets
Processes  Files

ISA/Hardware interface

CPU

Screen  Keyboard  Storage  Network

Memory

Input/Output
Course Logistics
Course Staff

Instructor: Jack Kolb  jkolb@berkeley.edu
• OH: 1-2pm, Tuesdays, 3-5pm, Fridays; 411 Soda

Jonathan Murata  jmurata@berkeley.edu
• OH: 9-11am, Tuesdays, 341B Soda

Yiming Yang  yangyiming@berkeley.edu
• OH: 9-11am, Thursdays, 341B Soda

Jason Lee  lxj@berkeley.edu
• OH: 1-2pm, Tuesdays, 341B Soda
Infrastructure

• Course Website:  
  https://cs162.eecs.berkeley.edu

• Piazza for Q&A:  
  https://piazza.com/berkeley/summer2019/cs162

• No lecture recordings available during summer term

• Anonymous Feedback Form:  
  https://tinyurl.com/cs162anonfeedback
Textbook & Resources

• Course text: Operating Systems, Principles and Practice, 2nd ed., by Anderson and Dahlin

• Recommended Supplementary Material:
  • Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau, available for free online at: http://pages.cs.wisc.edu/~remzi/OSTEP
  • Linux Kernel Development, 3rd ed., by Robert Love

• Additional materials on course website "Readings" page
  • The Night Watch, relevant articles and papers
Syllabus Summary

1. **Core OS Concepts**: Processes, multiprogramming, protection

2. **Concurrency**: Threads, Scheduling, Synchronization

3. **Input/Output & Networking**: I/O Devices, Sockets, TCP, RPC

4. **File Systems**: Storage devices, Performance, Journaling, Transactions

5. **Address Space**: Virtual memory, Address translation, Paging

6. **Distributed Computing**: DHTs, Dist file systems, Consistency

7. **Reliability & Security**
Learning by Doing

• Three **group** projects using Pintos and C
  1. Threads and Scheduling
  2. User Programs
  3. File System

• Four **individual** homework assignments to practice systems programming
  1. HW0: Tools, autograding, C review, executables
  2. HW1: Simple Shell
  3. HW2: HTTP Server
  4. HW3: TBA
Start HW0 ASAP

• **Due Friday at 11:59PM**

• Gets you set up with the autograder, github, and a local VM

• Basic C programming task to serve as a refresher

• Entry survey so we can learn more about you
Group Projects

• Teams of 4 (Never 5, 3 would require serious justification)

• Find teams by **Friday** (or we will assign you to a team)

• Communication and cooperation will be essential
  • Design Documents
  • Slack/Messenger/whatever doesn’t replace face-to-face

• Everyone should do work and have clear responsibilities
  • You will evaluate your teammates at the end of each project
Preparing Yourself for this Class

• The projects will require you to be very comfortable with programming and debugging C
  • Pointers (including function pointers, void*)
  • Memory Management (malloc, free, stack vs heap)
  • Debugging with GDB

• You will be working on a larger, more sophisticated code base than anything you've likely seen in 61C
Preparing Yourself for this Class

• "Resources" page on course website
• C programming reference (still in beta):
  • https://cs162.eecs.berkeley.edu/ladder/
• Wednesday's section dedicated to programming and debugging review
• Review session this Friday 11 am-1 pm
  438 Soda Hall (Wozniak Lounge)
Project Grading

1. Initial Design Document & Design Review
2. Code Milestone
3. Final Version of Code
4. Peer Evaluations

Submission: git push to release branch
Course Grading

• **20%** 2-Hour Midterm Exam (7/18, 5-7pm, 155 Dwinelle)

• **25%** 3-Hour Final Exam (8/15, 5-8pm, Hearst Field Annex)

• **15%** Homework Assignments

• **35%** Pintos Projects

• **5%** Participation
Collaboration Policy

**OK:**
- Explaining a concept to someone in another group
- Discussing algorithms/testing strategies with other groups
- Helping debug someone else’s code (in another group)
- Searching online for generic algorithms (e.g., hash table)

**Not OK:**
- Sharing code or test cases with another group
- Copying OR reading another group’s code or test cases
- Copying OR reading online code or test cases from prior years
Late Policy

• Deadlines are at 11:59PM Pacific Time

• 1 slip day to use for individual homework assignments

• 2 slip days to use (as a group) for projects

• No credit for late work
Questions?
Break
What is an operating system?

• **Referee**
  - Resource sharing, protection, isolation

• **Illusionist**: clean, easy to use abstractions
  - Infinite memory, dedicated machine
  - Higher level objects: files, users, message
  - Masking limitations, virtualization

• **Glue**: Common Services
  - Storage
  - Window system
  - Networking
  - Authorization
Unifying Theme: Complexity

All definitions of operating systems are about hiding complexity
The Challenge of Complexity

• *Competing* applications with
  • Unexpected failures
  • Even *malicious* intentions…

• *Varying* hardware with
  • Evolving interfaces
  • Unexpected failures

Not feasible to test all possible configurations!
Complexity Example: Mars Pathfinder

- Lots of types of hardware: Radios, science experiments, batteries, solar panels, motors and robotic equipment
- Unique reliability requirements
  - No reset button, must reboot automatically when necessary
  - Must always be able to receive commands from Earth
- Remote debugging
  - Software will crash, how is that handled?
  - Need to debug from Earth
- Time-critical functions
  - Stop before hitting a Mars boulder
  - Earth communications
Complexity: Variety

- Different CPUs: x86, PowerPC, ARM, MIPS
- Different amounts of memory, disk space
- Different types of devices
  - Mice, keyboards, sensors, cameras
- Different networking equipment
  - Cable, DSL, Wireless, Ethernet
Questions

• Does the programmer need to write a single program that performs many independent activities?
• Does every program have to be altered for every piece of hardware?
• Does a faulty program crash everything?
• Does every program have access to all hardware?

Hopefully, no!
Unifying Theme: Helping the Programmer

• Write program once for lots of hardware

• Without reimplementing common functionality

• Run alongside other programs, possibly communicating and cooperating with them (safely)

• One program crashing doesn't crash everything
Abstract Virtual Machine Goals

• OS Goals:
  • Remove software/hardware quirks (*fight complexity*)
  • Optimize for convenience, utilization, reliability, … (*help the programmer*)

• For any OS area (e.g. file systems, virtual memory, networking, scheduling):
  • What hardware interface to handle? (physical reality)
  • What’s software interface to provide? (nicer abstraction)
Aside: Virtual Machines

• Software emulation of an abstract machine
  • Give programs illusion they own the machine
  • Make it look like hardware has features you want

• Two types of “Virtual Machines”
  • Process VM: supports the execution of a single program; this functionality typically provided by OS
  • System VM: supports the execution of an entire OS and its applications (e.g., VMWare Fusion, Virtual box, Parallels Desktop, Xen)
Process VMs

• Each process thinks...
  • It has all memory, CPU time, devices, …
  • All devices have the same convenient interface
  • Device interfaces more powerful than raw hardware
    • Bitmapped display ⇒ windowing system
    • Ethernet card ⇒ reliable, ordered, networking (TCP/IP)

• Fault Isolation
  • Processes unable to directly impact other processes
  • Bugs cannot crash whole machine

• Portability
  • Write your program for the OS rather than the raw hardware
System Virtual Machines: Layers of OSs

- Run entire operating systems
  - Good for OS development, portability, fault containment
- Interface closer to raw HW
  - To aid porting/running OSs built to use raw HW
  - Different kind of "application developer" to design for
What is an Operating System, Really?

• Always:
  • Memory Management
  • I/O Management
  • CPU Scheduling
  • Communications?
  • Multitasking/multiprogramming?

• Maybe:
  • File System?
  • Multimedia Support?
  • User Interface?
  • Web Browser?
What is an OS?

• **No universally accepted definition**

• “Everything a vendor ships when you order an operating system” is good approximation
  • But varies wildly!

• “The one program running at all times on the computer” is the **kernel**
  • Everything else is either a system program (ships with the operating system) or an application program
OS Goal: Protecting Processes & The Kernel

• Run multiple applications and:
  • Keep them from interfering with or crashing the operating system
  • Keep them from interfering with or crashing each other
How do we achieve this?

Two main hardware mechanisms:
1. Memory address translation
2. Dual mode operation

Use these to enact a simple policy: Programs are not allowed to read or write the memory of other programs or of the OS.
Address Translation

- **Abstraction**: Address Space
  - Group of memory addresses usable by something
  - Each process and the kernel have distinct address spaces
  - Illusion of owning all memory

- **Mechanism**: Address translation
  - Translate virtual addresses (used internally and emitted by CPU) to physical addresses
  - Usually performed in hardware by memory management unit (MMU)
Example of Address Translation

Translation Map 1

Translation Map 2

Physical Address Space
Address Translation Details

• Hopefully saw this in 61C?

• Does this get us our desired policy?
  • Can't read or write if no translation to the physical memory

• Problem: What if user code changes the page table pointer?
Dual Mode Operation

• Hardware provides at least two modes:
  1. Kernel Mode (or "supervisor" / "protected" mode)
  2. User Mode

• Certain operations are **prohibited** when running in user mode
  • Changing the page table pointer

• Carefully controlled transitions between user mode and kernel mode
  • System calls, interrupts, exceptions
### UNIX OS Structure

<table>
<thead>
<tr>
<th>User Mode</th>
<th>Kernel Mode</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications (the users)</td>
<td><strong>Standard Libs</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>shells and commands</td>
<td></td>
</tr>
<tr>
<td></td>
<td>compilers and interpreters</td>
<td></td>
</tr>
<tr>
<td></td>
<td>system libraries</td>
<td></td>
</tr>
<tr>
<td><strong>system-call interface to the kernel</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>signals terminal handling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>character I/O system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>terminal drivers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>file system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>swapping block I/O system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>disk and tape drivers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU scheduling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>page replacement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>demand paging</td>
<td></td>
</tr>
<tr>
<td></td>
<td>virtual memory</td>
<td></td>
</tr>
<tr>
<td><strong>kernel interface to the hardware</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>terminal controllers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>terminals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>device controllers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>disks and tapes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>memory controllers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>physical memory</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

Operating systems:

• Provide a **virtual machine abstraction** to handle diverse hardware

• **Coordinate resources** and protect users from each other

• **Simplify application development** by providing standard services

• Provide fault **containment**, fault **tolerance**, and fault recovery