Operating Systems and Systems Programming

Professor: Tom Anderson

Lecture 1: Introduction

1.0 Main points:

Why study operating systems?
What is an operating system?
Principles of operating system design
History of operating systems

1.1 Why study operating systems?

Abstraction: OS is a wizard, providing illusion of infinite CPUs, infinite memory, single worldwide computing, etc.

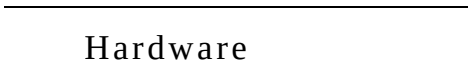
System Design: tradeoffs between performance and simplicity, putting functionality in hardware vs. software, etc.

How computers work: "look under the hood" of computer systems

Capstone: combines things from many other areas of computer science -- languages, hardware, data structures, algorithms

1.2 What is an operating system?

Definition: An operating system implements a virtual machine that is (hopefully) easier to program than the raw hardware:



In some sense, OS is just a software engineering problem: how do you convert what the hardware gives you into something that the application programmers want?

For any OS area (file systems, virtual memory, networking, CPU scheduling), begin by asking two questions:

what's the hardware interface? (the physical reality)
what's the application interface? (the nicer abstraction)

Of course, should also ask why the interfaces look the way they do, and whether it might be better to push more responsibilities into applications or into hardware, or vice versa.

1.2.1 Operating systems have two general functions:

1. **Coordinator:** allow multiple applications/users to work together in efficient and fair ways (examples: concurrency, memory protection, file systems, networking)

2. **Standard services:** provide standard facilities that everyone needs (examples: standard libraries, windowing systems)

1.2.2 What if you didn't have an operating system?

source code -> compiler -> object code -> hardware

How do you get object code onto the hardware? How do you print out the answer? Before OS's, used to have to toggle in program in binary, and then read out answers from LED's!

1.2.3 Simple OS: What if only one application at a time?

Examples: very early computers, early PC's, embedded controllers (elevators, cars, Nintendos, ...)

Then OS is just a library of standard services. Examples: standard device drivers, interrupt handlers, math libraries, etc.

1.2.3 More complex OS: what if share machine among multiple applications?

Then OS must manage interactions between different applications and different users, for all hardware resources: CPU, physical memory, I/O devices like disks and printers, interrupts, etc.

Of course, the OS can still provide libraries of standard services.

1.2.4 Example of OS coordination: protection

Problem: How do different applications run on the same machine at the same time, without stomping on each other?

Goal of **protection**:

Keep user programs from crashing OS

Keep user programs from crashing each other

1.2.4.1 Hardware support for protection

Hardware provides two things to help isolate a program's effects to within just that program:

address translation

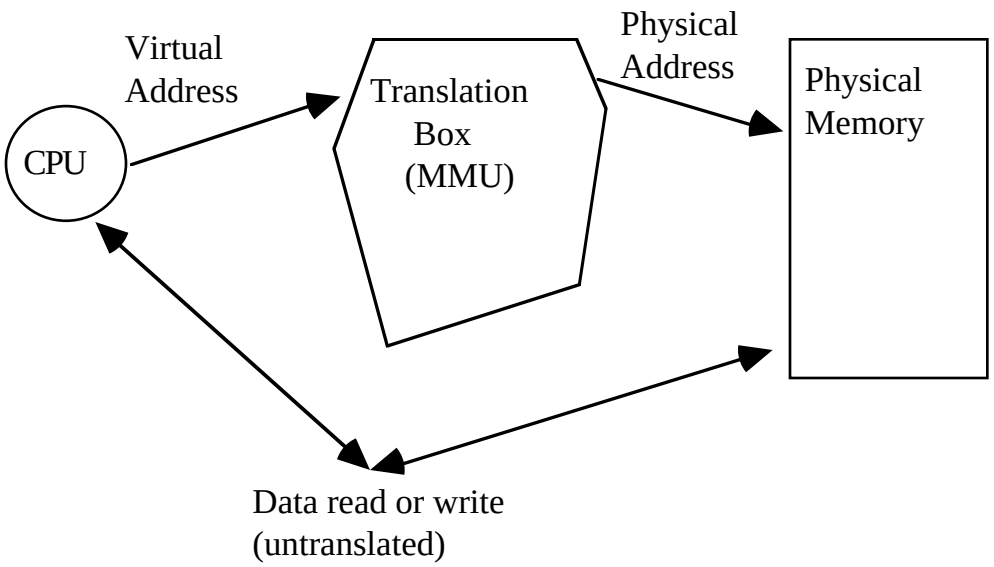
dual mode operation

1.2.4.2 Address translation

Address space: literally, all the addresses a program can touch. All the state that a program can affect or be affected by.

Restrict what a program can do by restricting what it can touch!

Hardware translates every memory reference from virtual addresses to phyiscal addresses; software sets up and manages the mapping in the translation box.

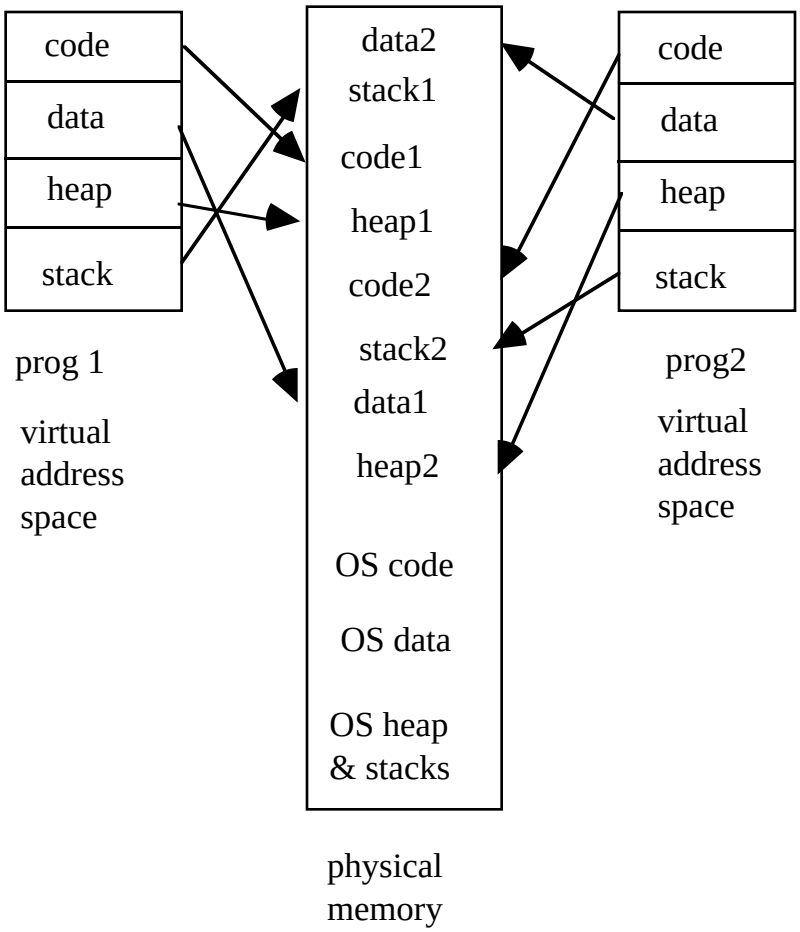


Address Translation in Modern Architectures

Two views of memory:

view from the CPU -- what program sees, virtual memory
view from memory -- physical memory

Translation box converts between the two views.



Example of Address Translation

Translation helps implement protection because no way for program to even talk about other program's addresses; no way for them to touch operating system code or data.

Translation implemented by some form of table lookup (we'll discuss various options for implementing the translation box later). Separate table for each user address space.

1.2.4.3 Dual mode operation

Can application modify its own translation tables? If it could, could get access to all of physical memory. Has to be restricted somehow.

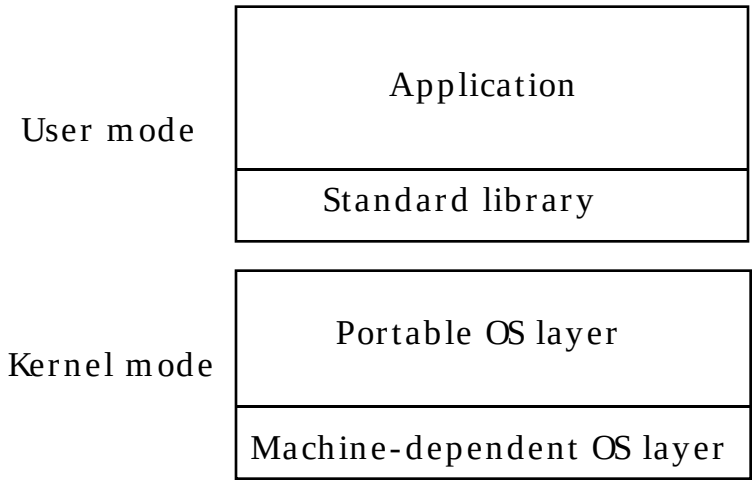
Dual-mode operation

when in the OS, can do anything (kernel-mode)

when in a user program, restricted to only touching that program's memory (user-mode)

Hardware requires CPU to be in **kernel-mode** to modify address translation tables.

Want to isolate each address space so its behavior can't do any harm, except to itself.



Typical UNIX Operating System Structure

Remember: don’t need boundary between kernel and application if system is dedicated to a single application.

Also, want OS to be portable, so put in a layer that abstracts out differences between different hardware architectures.

Project in this course, Nachos, is to build the portable OS kernel. We've built a simulation environment to surround the portable OS -- to simulate the hardware and machine-dependent layer (interrupts, etc.), and the execution of user programs running on top.

But to the code you write, simulator is exactly the same as if you were running on real hardware, and in fact, someone in West Virginia has ported Nachos to run native on an x86.

So why use a simulator instead of real hardware? To make debugging easier! In fact, most commercial OS's are now run first on a simulator, before being put on the real hardware.

1.3 Operating Systems Principles

Throughout the course, you'll see four common themes recurring over and over:

OS as illusionist -- make hardware limitations go away. OS provides illusion of dedicated machine with infinite memory and infinite processors.

OS as government -- protect users from each other and allocate resources efficiently and fairly.

OS as complex system -- keeping things simple is key to getting it to work.

OS as history teacher -- learn from past to predict the future.

Meta-principle: OS design tradeoffs change as technology changes.

1.4 History of Operating Systems: Change!

Typical academic computer in 1981 and 1996

	1981	1996	factor
SPECint (MIPS)	1	400	400
$/SPECint	$100K	$50	2000
DRAM capacity	128KB	64MB	500
disk capacity	10 MB	4 GB	400
net bandwidth	9600 b/s	155 Mb/s	15000
# address bits	16	64	4
# users/mach.	10s	<= 1	0.1

What does this mean?

Techniques have to vary over time, adapt to changing tradeoffs.

1.4.1 History Phase 1: hardware expensive, humans cheap

When computers cost millions of $'s, optimize for more efficient use of the hardware!

1. **User at console:** one user at a time. OS as subroutine library.

2. **Batch monitor:** load program, run, print.

No protection: what if batch program had a bug and wrote over batch monitor?

3. **Data channels, interrupts:** overlap of I/O and computation.

DMA -- direct memory access for I/O devices. OS requests I/O, goes back to computing, gets interrupt when I/O device has finished.

4. Memory protection + relocation.

Multiprogramming: several programs run at the same time; users share the system.

Multiprogramming benefits:

1. Small jobs not delayed by large jobs

2. More overlap between I/O and CPU

Multiprogramming requires memory protection to keep bugs in one program from crashing the system or corrupting other programs.

Bad news: OS must manage all these interactions between programs. Each step seems logical, but at some point, fall off cliff -- just gets too complicated.

Multics: announced in 1963 -> ran in 1969

OS 360 released with 1000 bugs.

UNIX based on Multics, but simplified so they could get it to work!

1.4.2 History, Phase 2: hardware cheap, humans expensive

5. Interactive timesharing: Use cheap terminals to let multiple users interact with the system at the same time. Sacrifice CPU time to get better response time for users.

Problem: thrashing -- performance falls off a cliff as you add users.

1.4.3 History, Phase 3: hardware very cheap, humans very expensive

6. Personal computing - Computers are cheap, so give everyone a computer. Initially, OS became subroutine library again, but since then, have added back in memory protection, multiprogramming, etc.

1.4.3 History, Phase 4: Distributed systems

Networking: allow different machines to share resources easily.

1.5 Summary

Point of change isn't: look how stupid batch processing is.
It was right for the tradeoffs of the time -- but not anymore.

Point is: have to change with changing technology.

Situation today is much like it was in the late 60's: OS's today are enormous, complex things:

small OS -- 100K lines
big OS -- 10M lines

100-1000 people-years

NT under development for the last 7 years, still doesn't work very well.

Key aspect of this course -- understand OS's, so we can simplify them!