CS162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2004

Lecture 1: Introduction

0.0 Administrivia:

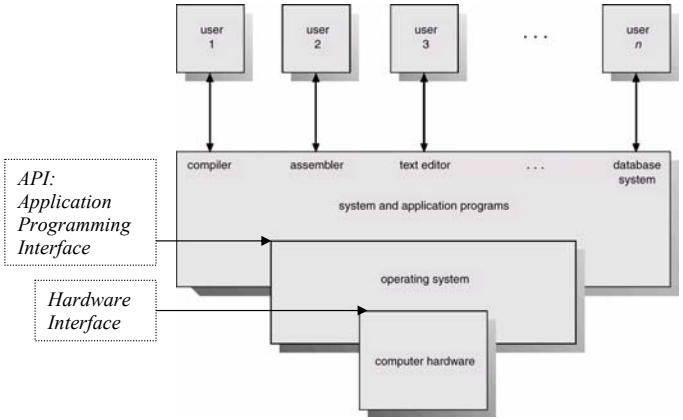
1.0 Main points:

What is an operating system --- and what isn't it?

Principles of operating system design

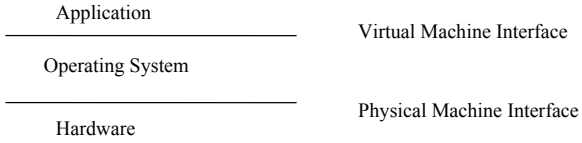
Why study operating systems?

1.1 What is an operating system?



(Figure is from Silberschatz and Galvin, Chapter 1)

Definition: *An operating system implements a virtual machine that is (hopefully) easier and safer to program and use 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), you 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. (examples, RISC architectures, VBasic libraries, etc.)

1.1.1 Virtual Machines

- Virtual machine model provides software emulation of an abstract machine
- Also used to allow programs for one hardware & OS platform to run on another one (e.g., Windows programs on a Macintosh), perhaps even running several VMs concurrently.
- Useful for OS research and development (much easier to debug)
- Protection and portability (e.g., Java VM)
- The project in this course is to build some of the portable components of an OS on top of Nachos, which provides a simulation environment. That is, it simulates the hardware and machine-dependent layer (interrupts, I/O, etc.) and the execution of user programs running on top of it. Note that Nachos runs on many different hardware/OS platforms.

1.1.2 Operating systems have two general functions:

Silberschatz and Galvin: "An OS is similar to a **government**".

This becomes political --- do you think a government does anything useful by itself?

1. **Coordinator & traffic cop:** allow multiple applications/users to work together in efficient and fair ways (examples: concurrency, memory protection, file systems, networking). This is **Resource Allocation** and **Control.** Goals are fair management of shared resources, protection, and efficiency.

2. **Standard services:** provide standard facilities that everyone needs (examples: standard libraries, windowing systems). View OS as a **facilitator**. Goal is to make application programming easier, faster, and less error-prone.

1.1.3 So, what is in an OS and what isn't?

Of course, there is no universal agreement on this but:

Most likely:

- Memory management
- I/O management
- CPU scheduling
- Communications?
- Multitasking/multiprogramming

What about?

- File System
- Multimedia support
- User Interface (window manager)
- Internet Browser? ☺

Bonus question: is this just of interest to academics? If not then why might it be important?

1.1.4 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!

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

Examples: very early computers, early PC's, embedded controllers (elevators, cars, Nintendos, ...), PDAs, intelligent light switches,…

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

1.1.6 More complex OS: what if you want to share the 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.

1.1.7 Example of OS coordination: protection

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

Goals of **protection**:

- Keep user programs from crashing OS
- Keep user programs from crashing each other

1.1.7.1 Hardware support for protection

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

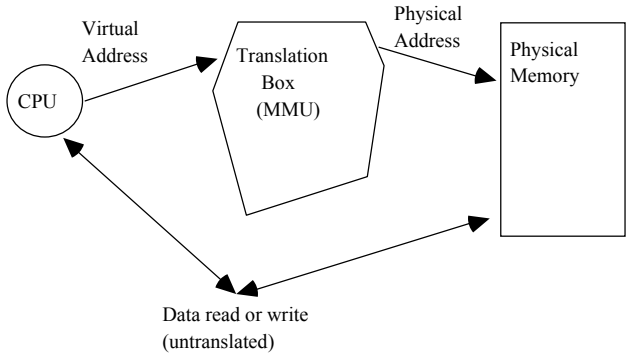
- Address translation
- Dual mode operation

1.1.7.2 Address translation

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

Achieve protection by restricting what a program can touch!

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

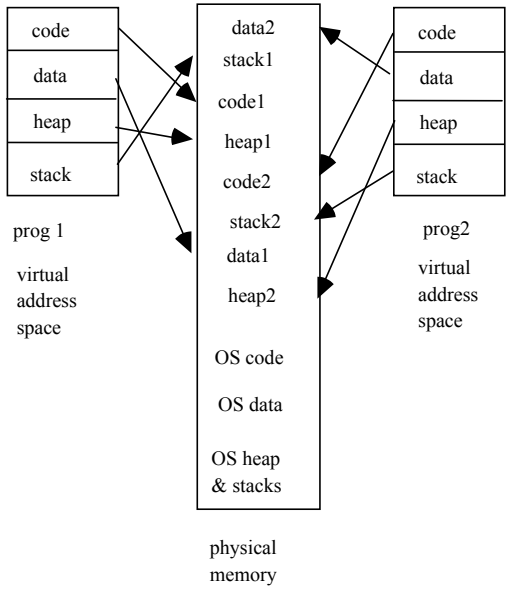


Address Translation in Modern Architectures

Two views of memory:

- View from the CPU – what program sees, virtual memory
- View from memory – physical memory

Translation box (also called a *memory management unit*) converts between the two views.



Example of Address Translation

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

Translation also helps with the issue of how to stuff multiple programs into memory.

Translation is implemented using 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 an application modify its own translation tables? If it could, then it could get access to all of physical memory. Has to be restricted somehow.

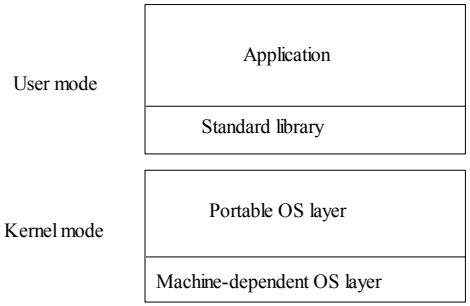
Dual-mode operation

- When in the OS, can do anything (called "kernel mode", "supervisor mode", or "protected mode")
- When in a user program, restricted to only touching that program's memory (user-mode)

Implemented by setting a hardware-provided bit. Restricted operations can only be performed when the "kernel-mode" bit is set. Only the operating system itself can set and clear this bit.

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

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.

1.2 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; but there is a constant tension between this and the desire to add more functionality and performance.

OS as history teacher – learn from past to predict the future in order to improve performance.

1.3 Why study operating systems?

You need to understand enough to make informed decisions about things like:

- Buying and using a personal computer:
  - Why do different PCs with the same CPU perform differently?
  - How do I choose between an AMD CPU and an Intel Itanium, Celeron, or Pentium 4 CPU?
  - Should I get Windows XP? Windows 2000? Linux? What's the difference?
  - Should I upgrade my hardware? Should I upgrade my OS?
  - What's going on with my PC, especially when I have to install something?
  - Should I use disk compression? Is there a cost to using it?
  - …
- Business (and personal) decisions about thin-clients versus PCs:
  - What are the issues involved?
  - What kinds of choices are being offered?
- Business decisions in general: how important are various capabilities (such as fault tolerance) and what should they cost?
- Security, viruses, and worms: what exposures do I have to worry about?
- Why is the Web so slow sometimes and is there anything I can do about it?

If you're going to be a software engineer then you'll need to understand various things about the environment offered by the OS you're running on:

- What **abstractions** does the OS provide? E.g., the OS may (or may not) provide illusions such as infinite CPU's, infinite memory, single worldwide computing, etc.
- What **system design trade-offs** have been made? E.g., what functionality has been put in hardware? What trade-offs have been made between simplicity and performance, putting functionality in hardware vs. software, etc?

Capstone: combines things from many other areas of computer science – languages, hardware, data structures, and algorithms. In general, systems smarts, complex software development (in groups), and *intuition* for general systems tradeoffs.