Lecture #1: Introduction, History, and Course Organization

****
Review -- 1 min
****

****
Outline - 1 min
****
Introduction **OS=coordination + abstraction**

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

Class Organization
History and Future
~~Dual mode operation~~
~~3 ways to invoke OS (if time)~~

****
Preview - 1 min
****
Today: Overview – basic principles

Then – how does OS manage basic system resources
-- next month CPU: concurrency (threads, synchronization, deadlock, scheduling)
– then management of memory
Then – IO: disk, networks, distributed systems, …

Next time: anatomy of an OS and a process

****

Lecture - 20 min

****

1. "Systems Principles"

Software systems -- here we mean low-level software on which programs you want to write rely

-- basic services that support programs you want to write
[systems v. applications]

Operating system, library, garbage collector, etc.

(Could add "compilers" in there, but you covered basics in Systems I, right?)

I'll often call this low-level software the "operating system" (but this is stretching the definition a bit to include things that are not normally thought of as the OS -- e.g., garbage collector -- but I regardless of traditional boundaries of OS, this seems fundamentally right. I could call it "runtime system" but that is clumsy...

2. Why Study Systems?

Systems software/runtime system/OS is everywhere: *coordination* and *abstraction* are key in any complex system. Plus many "applications" are really OS's or network OS's (e.g., netscape browser, JVM/Jini, Network Appliance, Akamai, Google, Amazon, Facebook, Tivoli, Yahoo, …)
--> not only will you use these abstractions when you write programs, but you may also implement some of them when you write big programs

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

System Design: tradeoffs between performance and simplicity, putting functionality in HW v. SW, etc.

How computers work: "look under the hood" of computer systems
-- to program computers well, need to understand how they work
-- CS major should know how computers work

Power tools: Key abstractions for solving common, difficult programming problems (threads, synchronization, transactions, 2PC, ...)
Project: This class should be particularly fun because it has an excellent project.

John Ousterhout – "When I read a paper, I can immediately tell whether the authors ever actually built the system and got it to work"
Proverb – "I hear and I forget, I see and I remember, I do and I understand"

3. Teach at 3 levels

1) How to approach problems
- fundamental issues -- *coordination, abstraction*
- design space
- case studies: historical and state of art techniques

Goal: When faced with similar (or very different) problem, you will be able to devise a good solution
Timescale: big long-term payoff

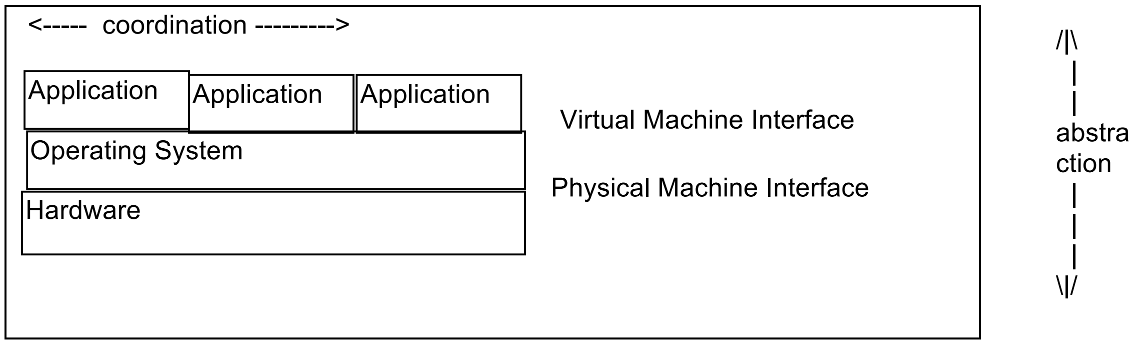
2) Specific techniques you should be able to apply
- Time-tested solutions to hard problems
- "Hacking" will not succeed
- e.g., concurrent programming, two-phase commit, transactions, …

Goal: be a good engineer
Timescale: immediately useful; still useful in 20 years

3) Details of modern OS
- e.g., FS, network stack, internal data structures, VM, … of XP, Solaris, Linux, …
- lots of material, changes relatively quickly
- not a priority of this class
  - but use "real" examples to help understand/motivate principles

4. What is an Operating System?

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



<----- coordination --------->
Application
Application
Application
Operating System
Hardware
Virtual Machine Interface
Physical Machine Interface
abstraction

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

For any OS area (file systems, virtual memory, 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, the OS, or hardware.

4.1 Operating systems have three general functions:

1. **Coordinator:** Allow multiple applications/users to work together in efficient and fair ways

3 aspects of coordination (*Draw Picture*)

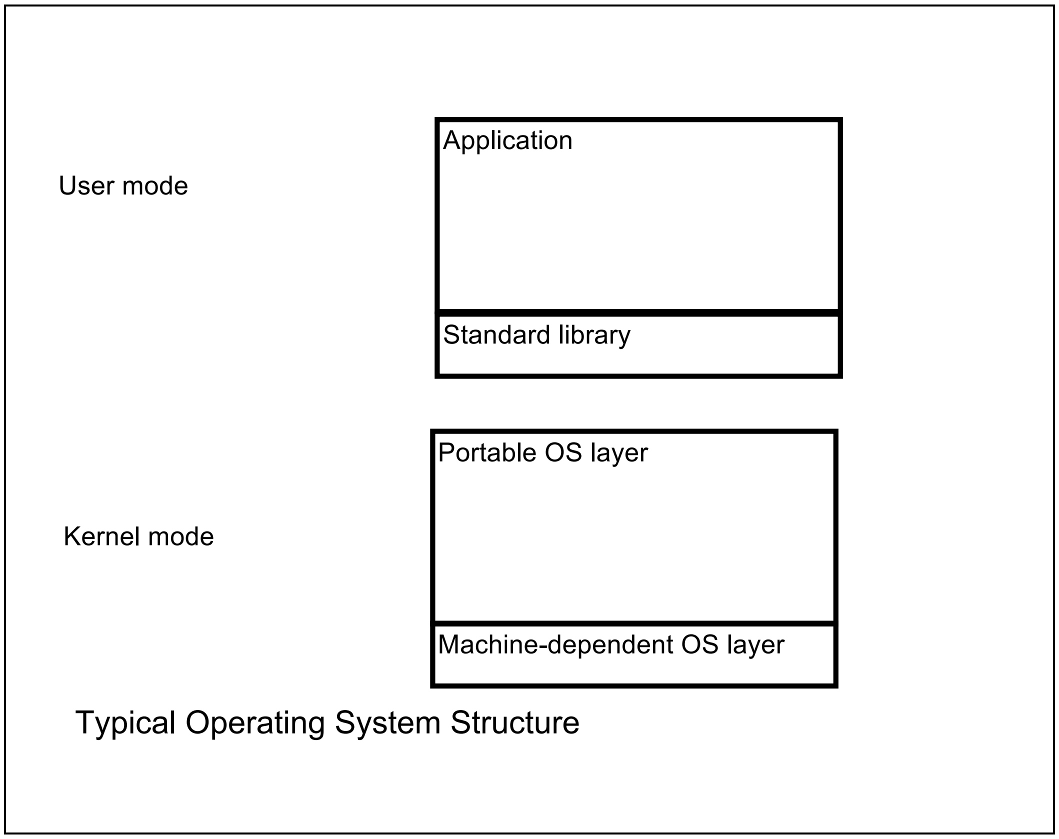
- Security – prevent jobs from interfering with one another
- Communication – let jobs talk to one another
- Resource Management – give jobs fair share of resources (memory, CPU, disk, …)

Dual Mode Operation
When in OS, program can do anything (kernel mode)
When in a user program, restricted to only touching that program's memory

Kernel mode:

- can issue physical addresses that are not translated by translation box (kernel can read/write process memory)
- can modify address translation tables

Structure of a Dual-mode operating system
emphasize -- not just kernel



Typical Operating System Structure

Also, want OS to be portable, so put a layer that abstracts out differences between different hardware architectures (e.g. Linux runs on x86 and ARM, Solaris runs on SPARC and x86, NT/win2k runs on x86 and (until recently) Alpha)

How do the security, communication, resource management facilities get reflected in dual-mode operation? (*draw in picture above)*

- Security: application can access its memory but not OS or other apps
- But OS can access any application's memory

- Communication: application can call OS (how does that work?) – then OS can read one application's memory and write another's → communication
- Resource management: OS can access hardware registers to, for instance, grow an application's memory allocation…

2. Abstraction -- raise level of programming
(a) hide details -- hundreds of different hardware devices --> a few common abstractions
e.g., hitachi disk, western digital disk, flash memory, CD --> block device --> FS

(b) better abstraction --
e.g., single-instruction, single-word atomic memory update --> critical section atomically updating multiple fields of a data structure

3. **Standard Services:** provide standard facilities that everyone needs ; facilitate sharing (e.g., window system, file cache, file system, network protocols, …)

Most basic OS: a way to start, stop, and clean up after a program
Modern OS -- much more **standard services**
-- threads, file system, transactions, network stack, windowing system, profiling, etc etc.
-- many/most of these you could do in an ad-hoc way as needed
-- having standard OS simplifies life
-- some of these require interoperability (e.g., FS, window system, ...)
-- need a standard version of service to use

"standard library" or "standard utilities" don't run in kernel, but can be regarded as part of OS

What if you didn't have an OS?

Source code → compiler → object code → hardware

How do you get object code onto the hardware?
How do you print an answer?
Before OS's: toggle in program in binary; read out answers from LED's!

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

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

Worry less about coordination, more about abstraction and standard services

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

History of OS – for each new platform, OS starts as abstraction + standard services (b/c only run one job at a time), evolves into "real" os that does coordination too
Mainframe, PC, palmtop?, cell phone?, …

4.3 More complex OS: what if we 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 (disks, printers, screen, keyboard), interrupts, etc

Of course, OS can still provide library of standard services

Discussion: What are key OS services? Do they do (1) coordination (resource mgmt, security, communication) or (2) standard services

- Example: file system – what aspects of file system are resource management? Security? Communication? Standard services?

4.4 Virtual machine monitor/hypervisor

"OS" (e.g., linux, windows, …) runs over hypervisor (e.g., vmware, Xen)

hypervisor provides protection among guest VMs
hypervisor provides few higher level services

guest OS provides higher level services to guest apps (and protection among guest apps)

4.5 Distributed systems

"OS" != everything that runs with "supervisor" bit set

NFS (network file service) is part of OS
Amazon S3 (simple storage service) and EC2 (elastic compute cloud) are OS services

Who knows/who cares if they run with supervisor bit set…

****

Admin - 3 min

****

(much more than 3 min today!)

Lectures

1 minute review
1 minute outline
40 minute lecture
3 minute admin
35 minute lecture
5 Q&A
20 "And now for something completely different"
1 minute summary (Don't get up!)

1 minute preview

Break up class (admin, break) b/c attention span
outline/summmary important to all presentations
"say what you're going to say, say it, say what you said"
preview – feedback from last class I taught

Philosophy

- Write on board instead of slides (or I'll go too fast)
    - feedback from class I taught – copies of my notes available on line (warning – not "polished")
- encourage discussion
- feedback – I'll hand out 3x5 cards in a month or so; feedback welcome any time

Discussion sections

W - focus on problem sets
F - focus on labs

Syllabus

see notes on syllabus

Book experiment "OSPP"
Excited to teach class
Using new (in progress) book
-- alpha testing; feedback please

-- most likely outcome -- substantially better class
- working through details; talking to 3 other faculty as interested in this stuff as I am; best ideas from all of us...
-- moderately likely outcome -- no significant effect for better or worse
-- non-negligible outcome -- substantially worse ("book is bad", "rough form of book is problem", "dahlin diving into too much detail", "dahlin too busy with book"
[[very likely outcome -- schedule will slip]]

OSPP + Bryant and O'Hallaron + outside readings

Project

- Overview and schedule
- Language C/C++ for the rest
- Project 1 begins now; due in 1 week;
-

Homework
Weekly problem sets
1-2 problems to be turned in each week
-- Wednesday deadline; will shift as needed
-- No late homeworks accepted
remaining problems "required but not turned in"
-- You need to do them to do well on exam!

Piazza

~~DEMO:~~
~~MDD: To set this up, get the bochs.img and fs.img files. Put on USB key~~
~~Boot machine dual-hard-drive using ubuntu live CD~~
~~Insert USB key~~

- ~~cd /media/usbdrive/bochdemo~~
- ~~sudo dd if=bochs.img of=/dev/hda~~
- ~~sudo dd if=fs.img of=/dev/hdb~~

~~NOTE: current demo code corrupts the file system, so if you get a kernel page fault, reload fs.img to /dev/hdb as above~~

Enrollment
Probably can't take more
Implications and adjustments
Bad news: load v. response time
Adjustment: Project design
Mal-adjustment: Homework
Adjustment: newsgroup
Adjustment: START EARLY, keep up with reading, HW, etc.

Adjustment: Exam design/grading

****
Lecture - 23 min
****

5. OS evaluation criteria

5.1 Reliability

reliability -- system does what it is supposed to do

OS breaks --> user stuck, may lose work
application breaks --> more limited effect (bc OS isolates failures)

Challenge -- hostile environment
v. debugging application.
Adversary v. random/incidental/accidental failures
--> test cases less effective

availability -- % time system is usable

disk crash --> data lost --> reliability failure
machine turned off/freezes --> can't read from disk --> availability failure

How would you create an operating system that is ultra-reliable and ultra-available?

5.2 Portability

portable -- does not change as the hardware changes

many dimensions
-- different IO devices (graphics cards, network interfaces, etc.)
    don't want different code for each
    --> OS defines common abstractions
-- different machine
    different architecture -- (x86 v. amd v. SPARC v. Atom v. ...)
    different generations of same architecture

(VAX v. Core 2 -- many abstractions from early 70's unix still there today)

porting applications expensive
--> OS presents common abstractions independent of HW
porting million-line OS expensive/difficult
--> OS built over HAL (hardware abstraction layer)

Within application, you know how you are going to use a module, and if you get some module's API wrong, you can change it.
For OS, we have to design abstractions for a wide range of applications. some of which don't even exist yet; and if we get it wrong, user may not be able to easily change it.
--> We think a lot about abstraction and interface design (and this training can help you even when you are facing the "simpler" problem of designing abstractions/APIs within applications)

What are the right, long lasting abstractions to present to applications?

5.3 Performance

several dimensions

overhead -- added resource cost of implementing abstraction
efficiency (inverse of overhead)

fairness among applications

response time (delay) -- time from start to end of task

throughput -- rate of task completion (efficiency of group of tasks, not just one)

NOTE: response time and throughput may not be directly related (pipelining)

predictability -- variation of response time (or throughput) which is better system that always has .5 second response time or a system that has .4 second *average* response time where most requests take .35 seconds but a few take 10 seconds? (A: almost always the former)

How do we build a system with minimal overhead, minimum response time, maximum throughput, high predictability? (v. "right long lasting abstractions" which may hide details (hurting predictability) or add overheads or interfere with pipelining or ...)

5.4 Trade-offs

portability v. performance (see above)

performance v. reliability
example: shave 1 instruction from assembly language path by assuming max size of OS kernel; years later, random crashes.

6. Operating Systems Principles

Throughout this course you will see four common themes recurring over and over:

- **OS as illusionist** – ***abstraction --*** make hardware limitations go away. OS provides illusion of dedicated machine with infinite memory and infinite processors
- **OS as government** – ***protection --*** protect users from each other and allocate resources fairly and efficiently
- **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

What is exciting about computer science v. other engineering disciplines – underlying technology changes rapidly → lets us do things that were unthinkable a few years ago (v. bridge building)

7. History of Operating Systems: Change!

Typical academic computer in 1981, 1996, 2005

	1981	1996	2005	factor
SPECint/ MIPS	1	300	3000	300, 3000
$/SPECint	$100K	$33	$.33	3000, 300K
DRAM capacity	128 KB	128MB	1024MB	1000, 10K
Disk Capacity	10MB	4 GB	400GB	400, 40K
Net BW	9600 b/s	100 Mbit/s	100Mbit/s	10K, 10K
#addr bits	16	64	64	4, 4
#users/machine	100	<1	<1	100, 100

Impact: Techniques have to vary over time, adapt to changing tradeoffs

7.1 History Phase 1: Hardware expensive, humans cheap

Computers cost millions of $ → optimize to make most efficient use of hardware

1) **User at console** – one user at a time; OS is a subroutine library (Literally a stack of cards you pulled off a shelf to, say, do a matrix multiply)

Problem – have to wait between jobs while user enters next job (innovations make job entry faster: binary switches → keyboard → card reader → tape reader)

2) **Batch monitor** – load program, run print
Advantage – can load next job immediately as previous one finishes

2 problems

- ♦ no protection – what if program has a bug and crashes the batch monitor → waste time rebooting
- ♦ computer idle during I/O

3) **Data channels, interrupts**: overlap 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 finishes (PICTURE)

4) **Memory protection + relocation**

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

Multiprogramming benefits

- Small jobs not delayed by large jobs
- 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
- OS360 released with 1000 bugs

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

7.2 History Phase 2: Hardware cheap, humans expensive

5) **interactive time sharing**

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

OS does timesharing to give illusion of each user has own computer

7.3 History Phase 3: Hardware very cheap, humans very expensive

6) **Personal computing**

Computers are cheap, so give everyone a computer.

Initially, OS became a subroutine library again (MSDos, MacOS)

Since then, adding back in memory protection, multiprogramming, etc. (when humans are expensive, don't waste their time by letting programs crash each other)

7.4 History phase 4: Distributed systems

Computers soo cheap – give people a bunch of them

I have a PC at home, 2 in my office, a portable, a palmtop and share some machines in a lab

→ how do I coordinate a bunch of machines?

Networks fast – allow machines to share resources and data easily

Networks cheap – allow geographically distributed machines to interact

Distributed systems abstractions mature --> can spread program over 1000s of machines (but we're back to sharing!)

Question: What does all this mean to OS?

****

Summary - 1 min

****

1) Key ideas: coordination (resource management, isolation), abstraction
2) Application of ideas changes
    a. Over time
    b. Across applications, services (not just kernels)
    c.

Point of describing change isn't "Look how stupid batch processing is" – it was right for 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 – 50M lines

100 - 1000 people-year

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