Why are OS interesting?

OS are "magic"

- Most people don't understand them – including sysadmins and computer scientists!

OS are incredibly complex systems

- "Hello, World" – program really 1 million lines of code

Studying OS is learning how to deal with complexity

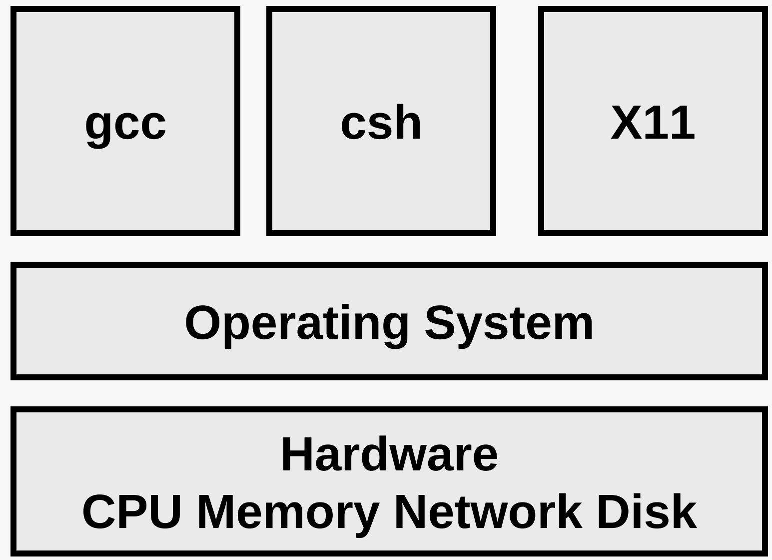
- Abstractions (+interfaces)
- Modularity (+structure)
- Iteration (+learning from experience)

What does an OS do?

Software layer that sits between applications and hardware

Performs services

- Abstracts hardware
- Provides protection
- Manages resources



OS vs Kernel

Can take a wider view or a narrower definition what an OS is

Wide view: Windows, Linux, Mac OSX are operating systems

- Includes system programs, system libraries, servers, shells, GUI etc.

Narrow definition:

- OS often equated with the *kernel*.
- The Linux kernel; the Windows executive – the special piece of software that runs with special privileges and actually controls the machine.

In this class, usually mean the narrow definition.

In real life, always take the wider view. (Why?)

Evolution of OS I

OSs as a library

- Abstracts away hardware, provide neat interfaces
  - Makes software portable; allows software evolution
- Single user, single program computers
  - No need for protection: no malicious users, no interactions between programs
- Disadvantages of uniprogramming model
  - Expensive
  - Poor utilization

Evolution of OS II

Invent multiprogramming

- First multi-programmed batch systems, then time-sharing systems

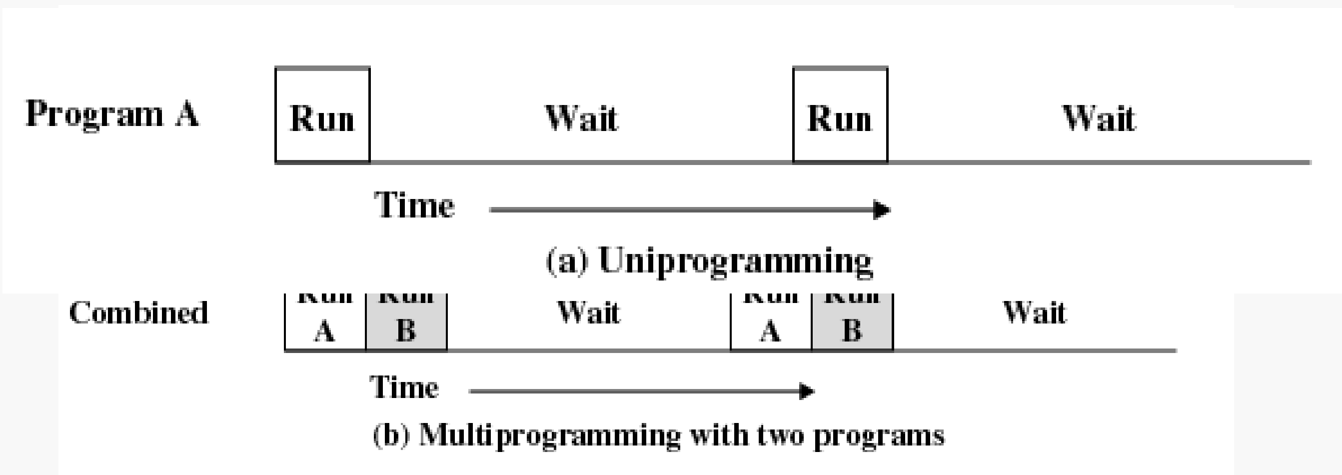
Idea:

- Load multiple programs in memory
- Do something else while one program is waiting, don’t sit idle (see next slide)

Complexity increases:

- What if programs interfere with each other (wild writes)
- What if programs don’t relinquish control (infinite loop)

Single Program vs Multiprogramming



Protection

Multiprogramming requires isolation

OS must protect/isolate applications from each other, and OS from applications

This requirement is absolute

- In Pintos also: if one application crashes, kernel should not! Bulletproof.

Three techniques

- Preemption
- Interposition
- Privilege

Protection #1: Preemption

Resource can be given to program and access can be revoked

- Example: CPU, Memory, Printer, “abstract” resources: files, sockets

CPU Preemption using *interrupts*

- Hardware timer interrupt invokes OS, OS checks if current program should be preempted, done every 1ms in Linux
- Solves infinite loop problem!

Q.: Does it work with all resources equally?

Protection #2: Interposition

OS hides the hardware

Application have to go through OS to access resources

OS can interpose checks:
- Validity (Address Translation)
- Permission (Security Policy)
- Resource Constraints (Quotas)

Protection #3: Privilege

Two fundamental modes:

- "kernel mode" – privileged
  - aka system, supervisor or monitor mode
  - Intel calls its PL0, Privilege Level 0 on x86
- "user mode" – non-privileged
  - PL3 on x86

Bit in CPU – controls operation of CPU

- Protection operations can only be performed in kernel mode. Example: hlt
- Carefully control transitions between user & kernel mode

```
int main()
{
        asm("hlt");
}
```

OS as a Resource Manager

OS provides illusions, examples:

- every program is run on its own CPU
- every program has all the memory of the machine (and more)
- every program has its own I/O terminal

"Stretches" resources

- Possible because resource usage is bursty, typically

Increases utilization

Resource Management

Multiplexing increases complexity

Car Analogy (by Rosenblum):

- Dedicated road per car would be incredibly inefficient, so cars share freeway. Must manage this.
- (abstraction) different lanes per direction
- (synchronization) traffic lights
- (increase capacity) build more roads

More utilization creates contention

- (decrease demand) slow down
- (backoff/retry) use highway during off-peak hours
- (refuse service, quotas) force people into public transportation
- (system collapse) traffic jams

Resource Management

OS must decide who gets to use what resource

Approach 1: have admin (boss) tell it

Approach 2: have user tell it

- What if user lies? What if user doesn't know?

Approach 3: figure it out through feedback

- Problem: how to tell power users from resource hogs?

Goals for Resource Management

Fairness

- Assign resources equitably

Differential Responsiveness

- Cater to individual applications' needs

Efficiency

- Maximize throughput, minimize response time, support as many apps as you can

These goals are often conflicting.

- All about trade-offs

Summary: Core OS Functions

Hardware abstraction through interfaces

Protection:

- Preemption
- Interposition
- Privilege (user/kernel mode)

Resource Management

- Virtualizing of resources
- Scheduling of resources

Evolution of OS III

Recent (last 15 years or so) trends

Multiprocessing

- SMP: symmetric multiprocessors
- OS now must manage multiple CPUs with equal access to shared memory

Network Operating Systems

- Most current OS are NOS.
- Users are using systems that span multiple machines; OS must provide services necessary to achieve that

Distributed Operating Systems

- Multiple machines appear to user as single image.
- Maybe future? Difficult to do.

OS and Performance

Time spent inside OS code is wasted, from user's point of view

- In particular, applications don't like it if OS does B in addition to A when they're asking for A, only
- Must minimize time spend in OS – how?

Provide minimal abstractions

Efficient data structures & algorithms

- Example: O(1) schedulers

Exploit application behavior

- Caching, Replacement, Prefetching

Common Performance Tricks

Caching

- Pareto-Principle: 80% of time spent in 20% of the code; 20% of memory accessed 80% of the time.
- Keep close what you predict you'll need
- Requires replacement policy to get rid of stuff you don't

Use information from past to predict future

- Decide what to evict from cache: monitor uses, use least-recently-used policies (or better)

Prefetch: Think ahead/speculate:

- Application asks for A now, will it ask for A+1 next?

Final thought: OS aren't perfect

Still way too easy to crash an OS

Example 1: "fork bomb"

- main() { for(;;) fork(); } stills brings down most Unixes

Example 2: livelock

- Can be result of denial-of-service attack
- OS spends 100% of time servicing (bogus) network requests
- What if your Internet-enabled thermostat spends so much time servicing ethernet/http requests that it has no cycles left to control the HVAC unit?

Example 3: buffer overflows

- Either inside OS, or in critical system components – read most recent Microsoft bulletin.