CSE 451: Operating Systems
Spring 2010

Module 1
Course Introduction

John Zahorjan
zahorjan@cs.washington.edu
534 Allen Center

Today's agenda

- Administrivia
  - course overview
    - course staff
    - general structure
    - the text
    - policies
    - your to-do list
- OS overview
  - Trying to make sense of the topic

© 2010 Gribble, Lazowska, Levy, Zahorjan

Course overview

- Everything you need to know will be on the course web page:

  http://www.cs.washington.edu/451/

© 2010 Gribble, Lazowska, Levy, Zahorjan

More Stuff You Can Read On The Course Web

- Course staff
  - John Zahorjan
  - Sal Guarnieri

- General Course Structure
  - Read the text prior to class
  - Class doesn't aim to repeat the text
  - Homework exercises to motivate reading by non-saints
  - Sections are likely to focus on projects
  - You're paying for interaction
    - Plus the degree...

© 2010 Gribble, Lazowska, Levy, Zahorjan

- the text
  - Silberschatz, Galvin & Gagne, *Operating System Concepts*, eigth **edition**
    - if using an earlier edition, watch chapter numbering, exercise numbering
- other resources
  - many online; some of them are required reading; some of them are prohibited reading
- policies
  - Collaboration vs. cheating
  - Homework exercises: late policy
  - Projects: late policy

© 2010 Gribble, Lazowska, Levy, Zahorjan

- your to-do list …
  - please read the entire course web thoroughly, *today*
  - keep up with the reading
  - homework 1 (problems) is posted on the web **now**
    - due at **the start of class** next Monday

© 2010 Gribble, Lazowska, Levy, Zahorjan

What is an Operating System?

- Answers:
  - I don't know.
  - Nobody knows.
  - (The book knows. Read Chapter 1.)

Okay. What Are Its Goals?

- Answers:
  - Well, they're programs. They *can* do anything a program can do.
  - Did I mention they're programs? Big programs?
    - The Linux source you'll be compiling has over 1.7M lines of C code.

© 2010 Gribble, Lazowska, Levy, Zahorjan

Getting a Grip

- Operating systems are the result of a 60 year long evolutionary process.
  - They were born out of need

- We'll follow a bit of their evolution

- That should help make clear what some of their functions are, and why

© 2010 Gribble, Lazowska, Levy, Zahorjan

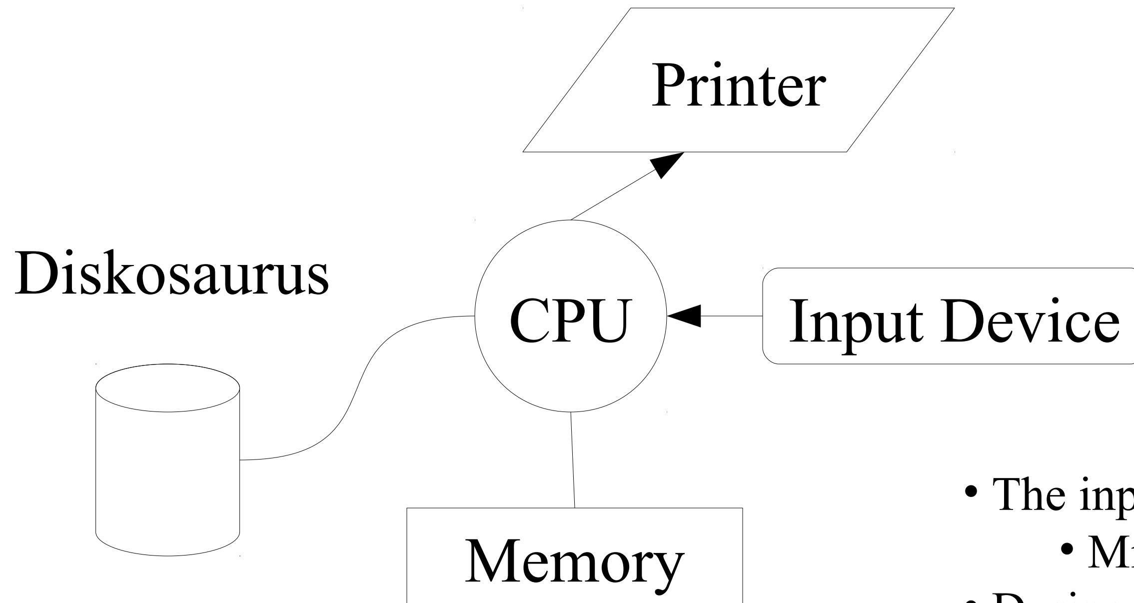
In the Beginning…

- 1943:
  - T.J. Watson (created IBM):
    "I think there is a world market for maybe five computers."

- Fast forward: 1950
  - There are maybe 20 computers in the world
  - Why do we care?
    - They were unbelievably expensive
    - Imagine this: machine time is more valuable than person time!
    - Ergo: efficient use of the hardware is paramount
  - Operating systems are born
    - They carry with them the vestiges of these ancient forces

© 2010 Gribble, Lazowska, Levy, Zahorjan

The Primordial Computer



- The input device is very slow
  - Minutes to read a job
- During those minutes, the mainframe is idle!
- Idea: Let's have a “resident monitor” load the next job into memory while the current job is running

© 2010 Gribble, Lazowska, Levy, Zahorjan

The Resident Monitor Needs Protection

- This is a good plan, but what happens if the job in execution:
  - Goes into an infinite loop?
  - Has a bug and corrupts the resident monitor?

- We need:
  - Interrupt/exception mechanism
    - Regain use of CPU, no matter what
  - Memory protection
    - User program can't overwrite monitor code
  - "user mode vs. supervisor mode" ("user/privileged", "user/kernel", "user/root", ...)

© 2010 Gribble, Lazowska, Levy, Zahorjan

Hey, That Worked!

- Overlap of job input with job processing resulted in higher CPU utilization (a good thing)

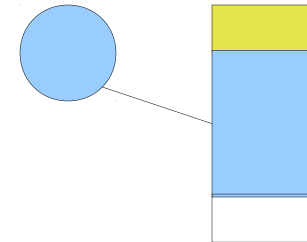
- The new bottleneck: the diskosaurus
  - Disks were (are) slow
  - The CPU was spending a lot of time waiting around for data from the disk
  - What to do

- **Course theme**:
  - There are a handful of good/great ideas
  - (Re)Use them!

© 2010 Gribble, Lazowska, Levy, Zahorjan

I/O Overlap: Parallelism

- Add hardware so that disk operate without tying up the CPU
  - Disk controller

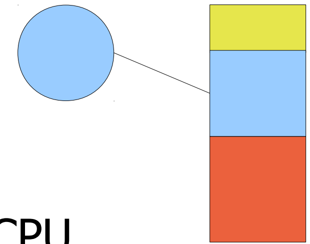


- Hotshot programmers could now write code that:
  - Starts an I/O
  - Goes off and does some computing
  - Checks if the I/O is done at some later time
  - I'm going to refer to this kind of overlap, whose goal is to improve the performance of a single "job," as **parallelism**
- Upside
  - it helps increase CPU utilization
- Downsides
  - it's hard to get right
  - the benefits are job specific: is there enough available parallelism?

© 2010 Gribble, Lazowska, Levy, Zahorjan

I/O Overlap: Concurrency

- Run more than one job at a time
- When one starts an I/O, switch CPU to run a different one



- Upsides:
  - If you have enough jobs in memory, there's always some CPU work to do
- Downsides:
  - Memory allocation issues
  - Protection of one job from another (memory, disk, CPU)
  - CPU allocation issues
  - (Disk I/O allocation issues)

© 2010 Gribble, Lazowska, Levy, Zahorjan

Concurrency

- The official name for loading more than one job in memory and switching the CPU among them is *multiprogramming*
  - All modern systems, even on fairly rudimentary devices, are multiprogrammed
  - Why?
- I'm going to refer to overlapped execution that simplifies programming effort as **concurrency**
  - Concurrent executions involve parallelism
  - They *can* have beneficial performance impacts for individual applications
  - Most often, though, the biggest win is that the computation is more easily built / managed / understood
- How is multiprogramming concurrency, by that definition?

© 2010 Gribble, Lazowska, Levy, Zahorjan

(An Aside)

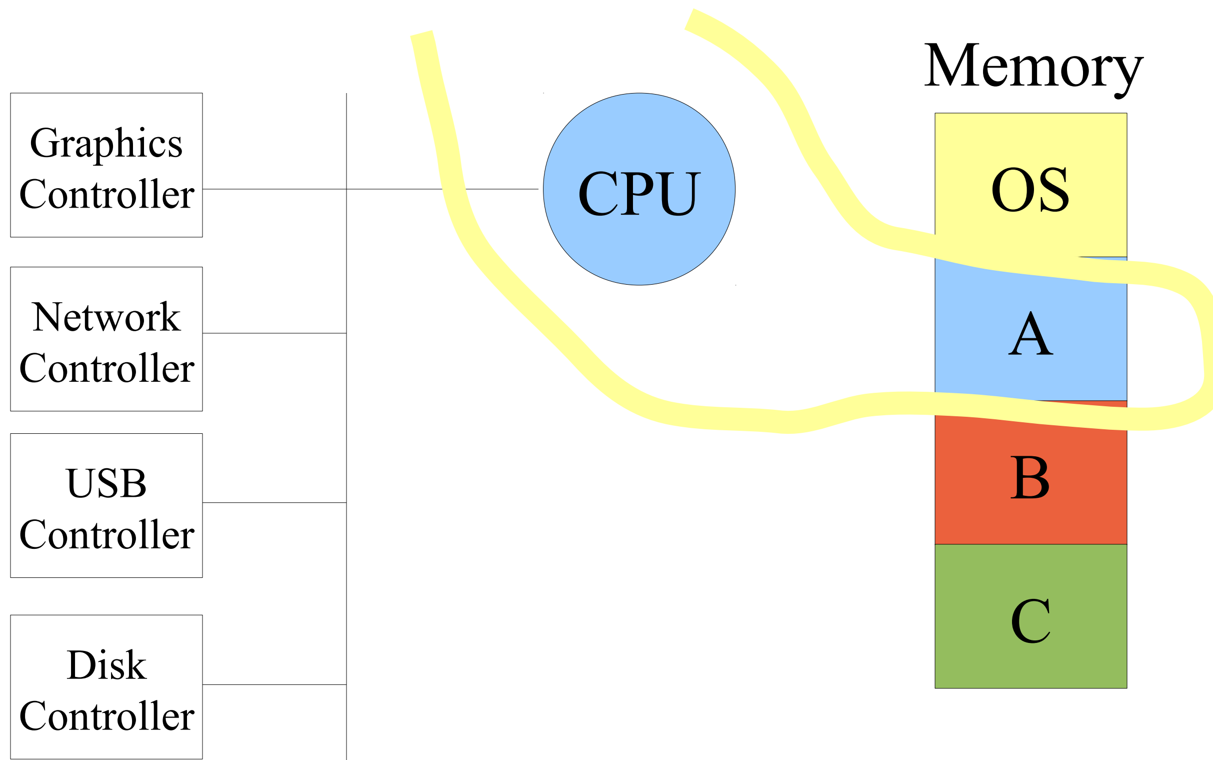
- CPU architects tell us that individual cores aren't going to be getting faster (very fast), but that they can double the number of cores on the old 18 month cycle (or so)

- The burden is on the programmer to use an ever increasing number of cores
  - You can use parallelism
  - You can use concurrency

- A lot of this course is about concurrency
  - It used to be a bit esoteric
  - It now seems likely to be one of the most important things you'll learn (in our courses)

© 2010 Gribble, Lazowska, Levy, Zahorjan

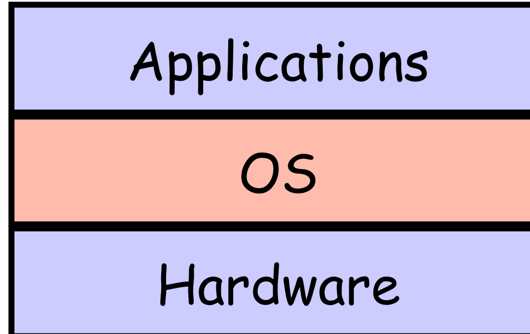
Where Were We?



Protection Requirements ⇒
Programs execute directly on the CPU, but cannot touch anything other than their own memory without OS help

© 2010 Gribble, Lazowska, Levy, Zahorjan

The More Customary Drawing



- This depiction invites you to think of the OS as a library
  - It isn't:
    - you use the CPU/memory without OS calls
    - it intervenes without having been explicitly called
  - It is:
    - all operations on I/O devices require OS calls (*syscalls)*
- So long as it is a library as far as I/O devices go, it might as well be a useful one
  - Presents nicer abstractions to program to than the raw hardware

© 2010 Gribble, Lazowska, Levy, Zahorjan

Device Abstractions

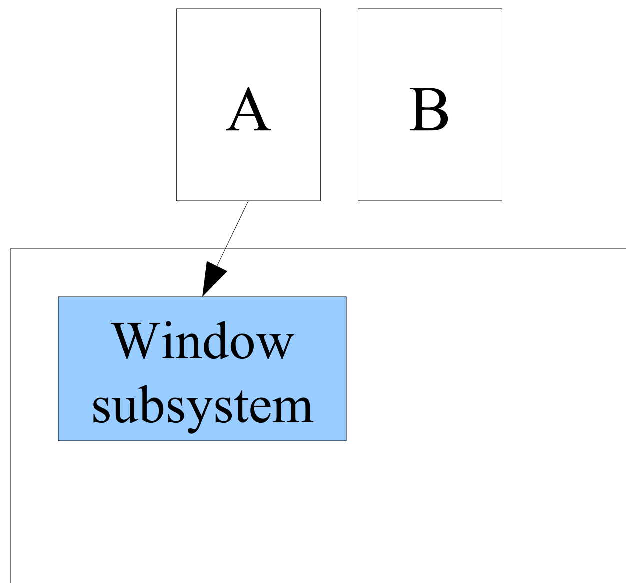
- Examples:
  - Raw disk storage $\Rightarrow$
  - Keyboard/mouse $\Rightarrow$
  - Graphics card $\Rightarrow$
  - Network interface card $\Rightarrow$

  - CPU $\Rightarrow$ process (/ thread)
  - Memory $\Rightarrow$ virtual address space
- Besides protection, allocation, and performance, another role of the OS is programming convenience

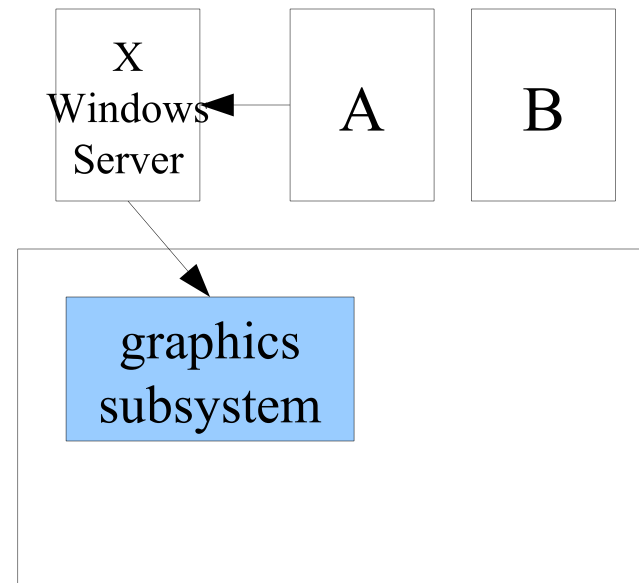
© 2010 Gribble, Lazowska, Levy, Zahorjan

(Back To) What Is An Operating System?

User processes



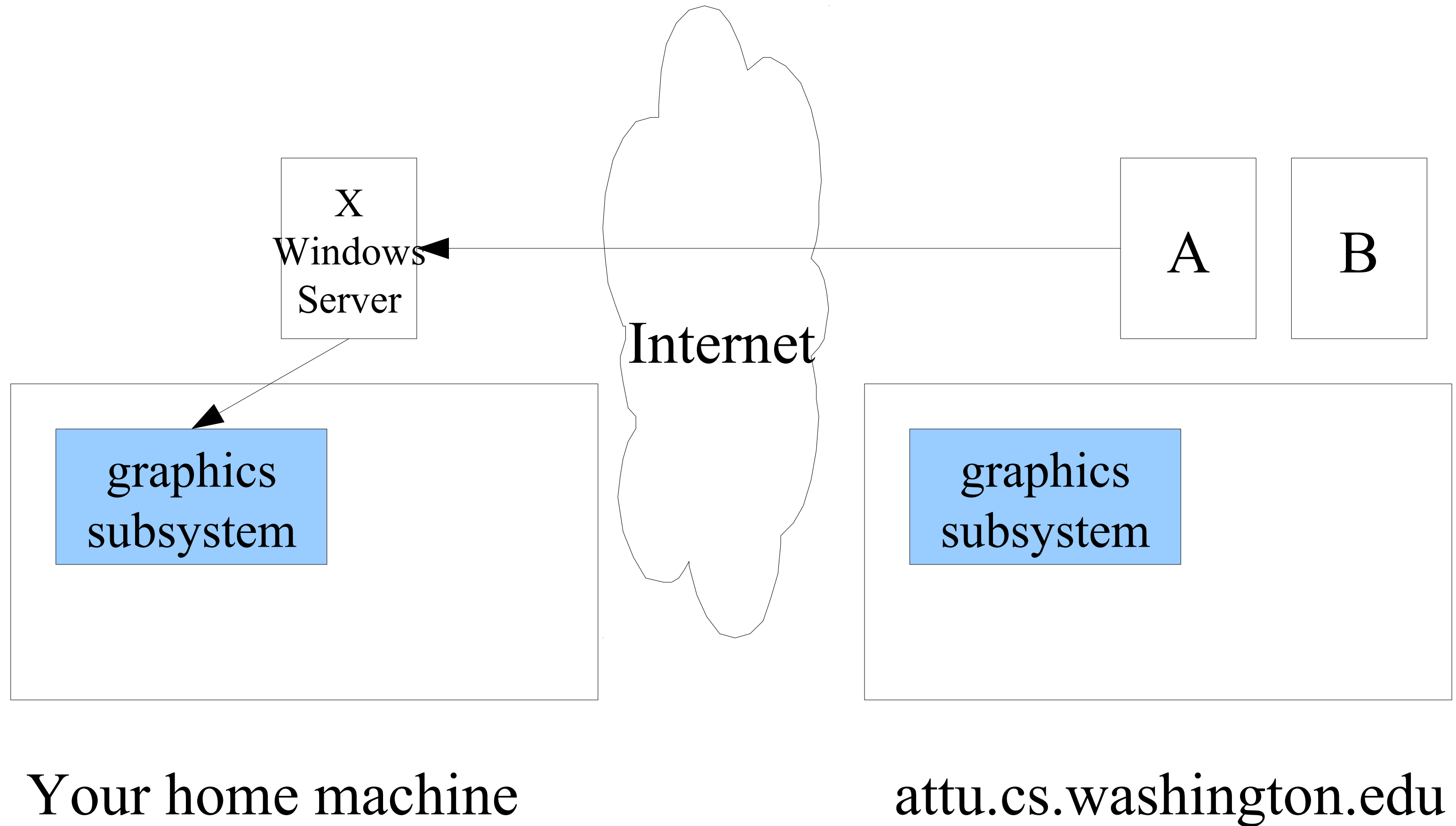
Windows (OS)



Unix

© 2010 Gribble, Lazowska, Levy, Zahorjan

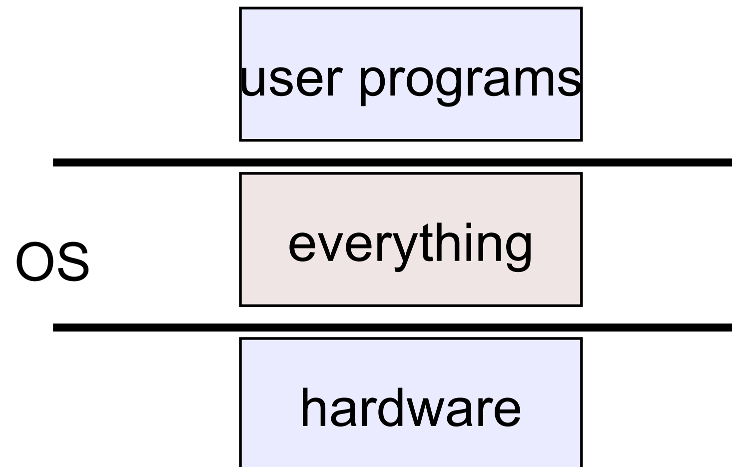
Impact of That Decision



© 2010 Gribble, Lazowska, Levy, Zahorjan

Hey, That Worked! (OS Structure)

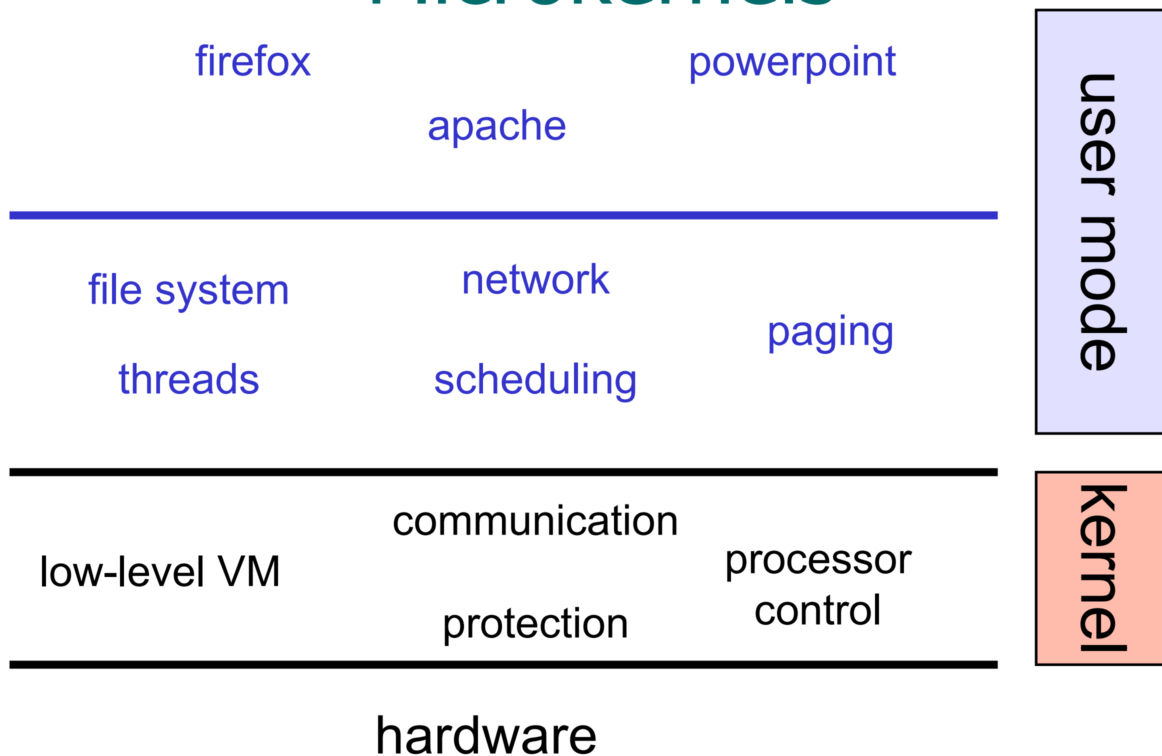
- OS's evolved as *monolithic* implementations



- Pros:
  - Fast
- Cons:
  - Complicated
  - Inflexible

© 2010 Gribble, Lazowska, Levy, Zahorjan

Microkernels

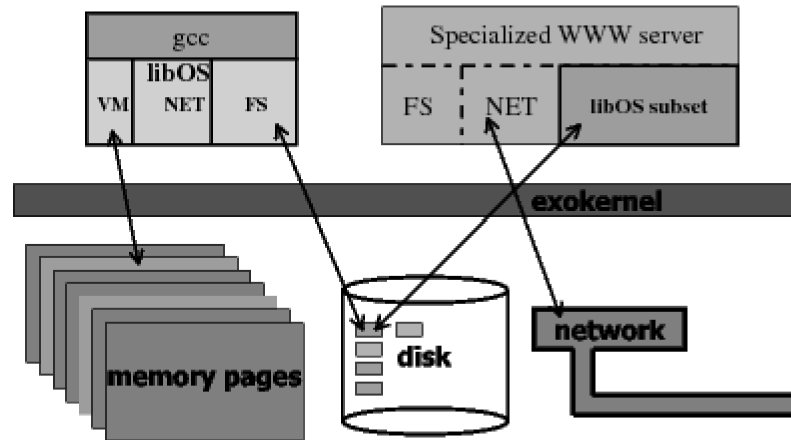


- Pros:
  - Flexible
  - Debuggable
- Cons:
  - Slow
  - Complicated for applications

© 2010 Gribble, Lazowska, Levy, Zahorjan

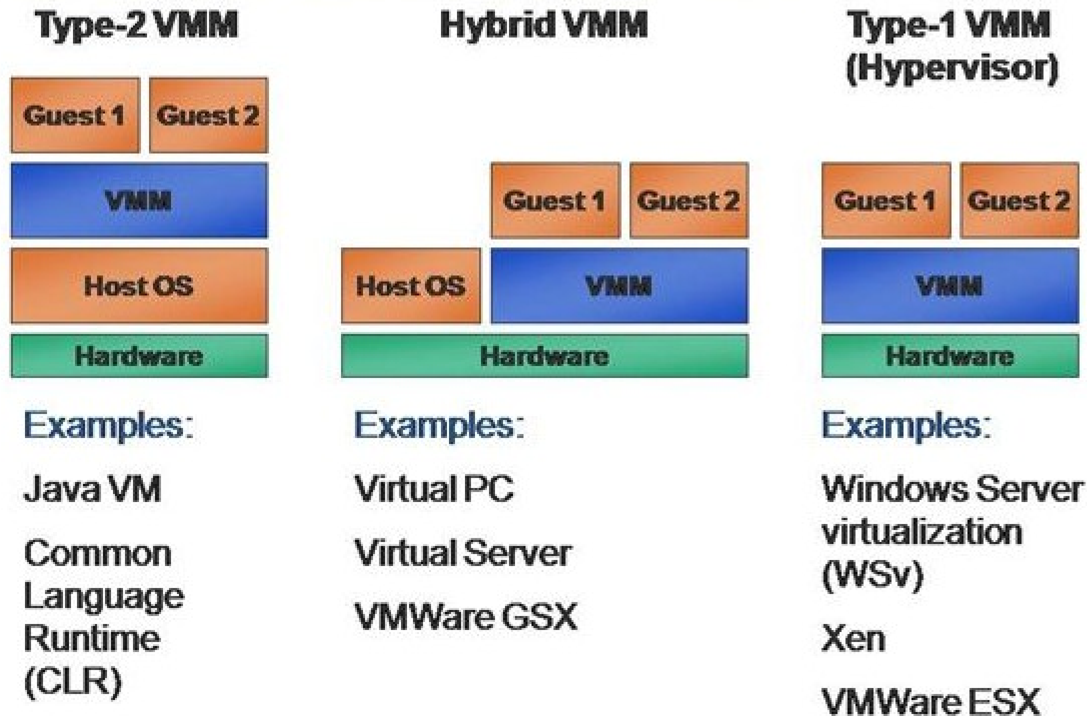
Exokernel ("No Kernel")

- Export hardware to user level (in a protected way)



- Pros:
  - Flexible
  - Arguably more efficient (than microkernel)
- Cons
  - Approximately 1.5B existing applications

© 2010 Gribble, Lazowska, Levy, Zahorjan

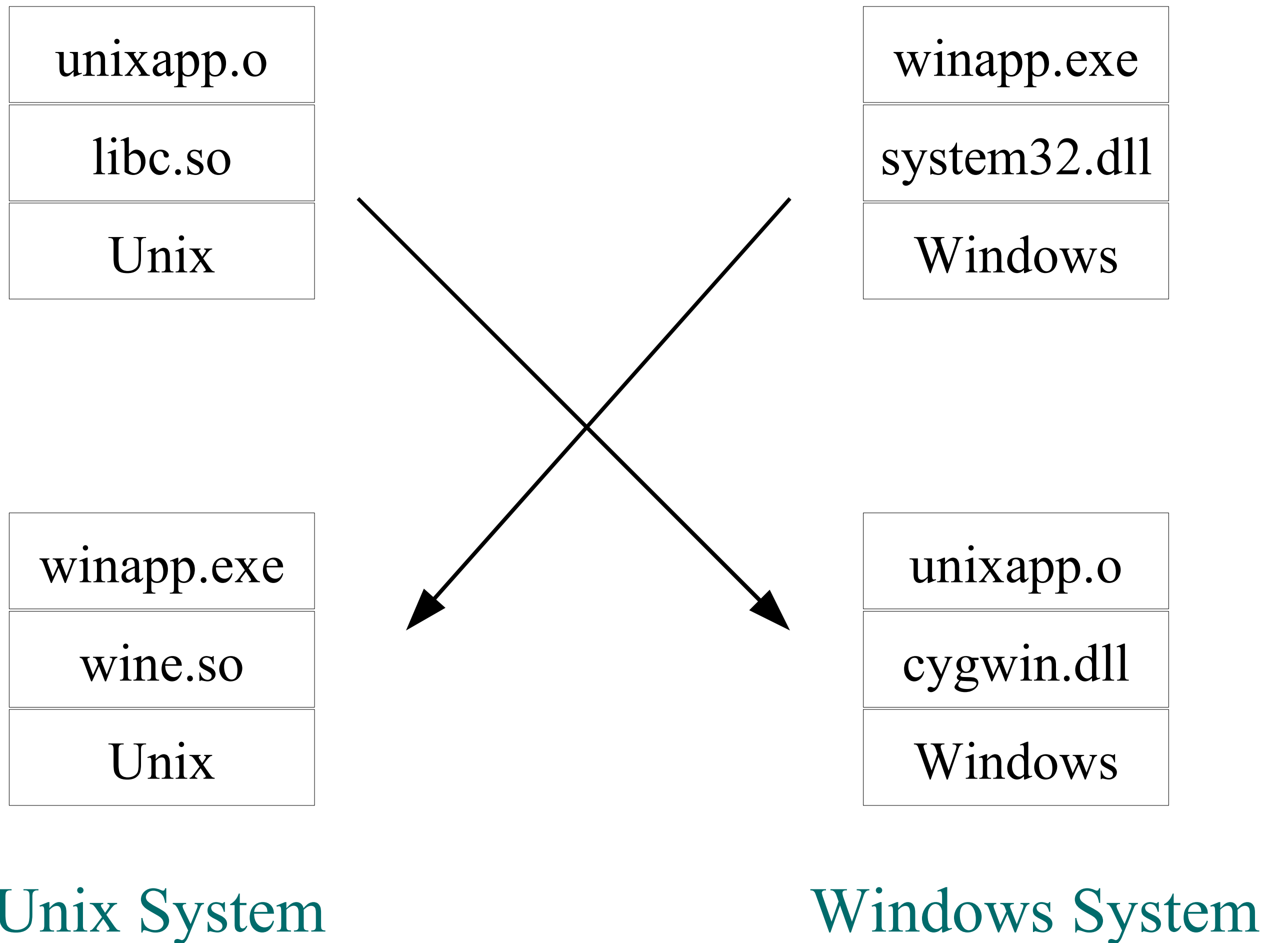


http://port25.technet.com/photos/images/images/4155/640x480.aspx

- Transparently implement "hardware" in software
- Voilà, you can boot a "guest OS"

© 2010 Gribble, Lazowska, Levy, Zahorjan

(Another aside) Cross-system Application Portability



© 2010 Gribble, Lazowska, Levy, Zahorjan

Core OS Functions

- Programming convenience
  - OS provides abstractions / implements objects
- Concurrency
  - More than one computation is going on at a time
- Protection
  - Which then requires providing ways around protection
- Allocation
  - Hardware is shared; no way around that
- Performance / Efficiency
  - Achieving user specified objectives

© 2010 Gribble, Lazowska, Levy, Zahorjan

What Now?

- Decisions about how to provide these things are intertwined
  - E.g., the level of abstraction will have a strong impact on the cost of providing the abstraction
- This leads to an exponential number of choices
  - That can tend to turn learning the material into memorization
- We're now going to try to look at some key concepts that underlie the choices
  - Pro: Simple, and with sweeping applicability
    - Even beyond CSE 451 / operating systems
  - Con: We'll be ignoring various warts, and sometimes the devil is in the warts

© 2010 Gribble, Lazowska, Levy, Zahorjan

1. Programming Convenience / OS Abstractions

- The OS provides a set of abstractions
  - Process / thread
  - File
  - Disk device
  - Address space
  - …
- It provides functions to manipulate them (create, delete, alter,...)
- This requires a name space
- It also results in metadata

© 2010 Gribble, Lazowska, Levy, Zahorjan

Namespaces

- Examples:
  - Files:
    - /home/zahorjan/.bash_profile
    - 0 (stdin file handle number)
  - Processes: 17543 (process id (pid))

- Self-test:

  class foo { …}

  - What is the namespace for instances of foo in a Java program?

- Namespaces have a scope
  - Often we say "global" vs. "local," but the meaning isn't precise
    - Memory address 0x00753A2C is local to the process
    - /bin/cat is global (to the system)
    - www.cs.washington.edu is global (to the world)

© 2010 Gribble, Lazowska, Levy, Zahorjan

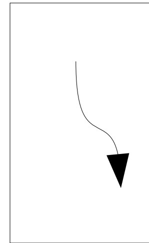
Metadata

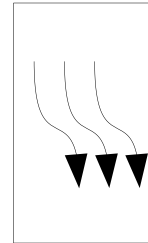
- It's easy to think of the object as being (solely) it's primary concept
  - E.g., a file is its contents; a process is an execution (thread and memory contents)
- The OS, and many applications, require additional information to sensibly manipulate these objects
  - Files: owner, creation time, modification time, size, permissions, …
  - Process/thread: state (running, runnable, blocked, terminated); creator (parent process); …
- This is metadata

© 2010 Gribble, Lazowska, Levy, Zahorjan

2. Concurrency

- First, some terminology:





- Key to dealing with concurrency is understanding temporal relationships

© 2010 Gribble, Lazowska, Levy, Zahorjan

Temporal Relationships

- Binary relationships on operations
  - before, after, simultaneous with, …

```
A:    x = 0;
B:    y = x + 1;
C:    z = y > 0 ? y : 0;
```

  - A < B (A happens before B)
  - C > A (C happens after A)
  - B ~ B (B is simultaneous with B)
  - A < B or B < A or A ~ B == true
- *Distinct operations performed by a single thread are totally ordered in <*
  - A < B or B < A or A == B is true
  - Note: we don't have any idea of how much "real time" passes between statement executions, and we don't care
- This is our intuitive notion of time
  - Your life is single threaded...

© 2010 Gribble, Lazowska, Levy, Zahorjan

Concurrent Executions

- Two threads

```
A:   x = 0;                          D:  s = 0;
B:   y = x + 1;                      E:  t = func(s, t);
C:   z = y > 0 ? y : 0;              F:  z = 5;
```

- A < B < C and D < E < F == true
- C < F? F < C?
  - No, and no
  - Therefore, C ~ F (A<B or B<A or A~B == true, always)
- Simultaneous becomes "not definitely before and not definitely after"
  - In a practical sense, this is "sometimes before, sometimes after"
  - What's the value of z in the statement that follows C?
    - Sometimes 1, sometimes 5

© 2010 Gribble, Lazowska, Levy, Zahorjan

Simplifying Concurrency: Atomicity

- Atomic operation: set of operations appear to outsiders to happen at once
  - Either all have executed, or none have

```
A:    x = 0;
B:    y = x + 1;
C:    z = y > 0 ? y : 0;
```

$\Rightarrow$ A' (x=0; y=1; z=1;)

- Implementation requires cooperation of all threads
  - Example:
    - I'm going to update the course web only between 2:00AM and 3:00AM
    - You're going to look at the course web only between 3:00AM and 2:00AM
  - Note: This is a lousy example...
- Atomicity lets us think of many operations as a single one

© 2010 Gribble, Lazowska, Levy, Zahorjan

Simplifying Concurrency: Enforcing Ordering

- Enforcing an ordering is an example of synchronization
- For operations A and B executed by distinct threads, enforcing A<B involves some kind of communication
  - T_A advertises that it has executed A
  - T_B waits to execute B until it sees the advertisement
- Example:
  - I'll phone when I get to your apartment building door
  - You'll come unlock the door when you get my call
- What happens if you try to synchronize without explicit communication?
- Synchronization often leads to waiting
  - Why not the scheme “When I get to the door, you be there”?
  - (“When I need the next YouTube video frame to display, you have it already fetched”)

© 2010 Gribble, Lazowska, Levy, Zahorjan

Mutual Exclusion

- Mutual exclusion is a peculiar sort of ordering: Not A ~ B
  - In other words, either A < B or B < A, I don't care
- In real life, mutual exclusion often involves acquiring some object:
  - Examples:
    - Restroom key
    - The waiter
    - The next space in a revolving door
- The same is true on computers
- Example: reservationless Flex-car program
  - ME criterion: only one driver per car at a time, please
  - Protocol: when you see a free car, take it
    - This mostly works, but not always. Why?
    - Mostly isn't good enough

© 2010 Gribble, Lazowska, Levy, Zahorjan

Concurrency Summary

- The hard part of concurrency is thinking about time
- The hard part of thinking about time is the unintuitive meaning of "simultaneous with"
- Operations performed by a single thread are easy to reason about
  - They're totally ordered
- Reasoning about concurrent executions usually involves synchronization to impose ordering constraints
- Synchronization can induce overheads
  - The time to communicate
  - Waiting

© 2010 Gribble, Lazowska, Levy, Zahorjan

(Aside) Concurrency in the Real World

- Banks "clear" transactions nightly
- For two transactions, A and B, made between nightly runs, not A<B and not B < A
  - They're "simultaneous"

> Of the various industry tactics, the resequencing of transactions to maximize overdrafts is perhaps the most obscure. In a hypothetical case, a card-user with $100 in available funds might buy a $75 sweater, a $2 cup of coffee, a $4 hamburger and $30 worth of groceries — going over the limit only on the final purchase. But banks often tally each day's transactions by order of the purchase amount — largest to smallest — not by chronology. In this example, the consumer would exceed the limit after just the two largest purchases ($75 + $30 = $105), and thus be hit with overdraft fees on the two smaller purchases as well. The result? The bank gets three overdraft fees ($81) instead of just one ($27).
>
> Feddis, said that tactic is precisely what customers want, arguing that the the most vital purchases tend to be the most expensive. "People want their important expenses paid," she said.

http://washingtonindependent.com/38975/house-dems-eye-overdraft-reform (4/16/2009)

© 2010 Gribble, Lazowska, Levy, Zahorjan

3. Protection

- Authorization
  - Should $F_P(r)$ be allowed?
    - F is some operation; e.g., read, write, create, delete, ...
    - r is some resource; e.g., a file, a process, some memory, …
    - P is a principal, the agent making the request

- Authentication
  - Establishing yourself as principal P
  - Always involves P having something non-P's don't
    - A password
    - Photo ID
    - A particular retina
    - A decryption key

- Delegation
  - If P can execute F(r), can P authorize Q to execute F(r)?

- Revocation
  - If can execute F(r) now, will P always be able to execute F(r)?

© 2010 Gribble, Lazowska, Levy, Zahorjan

Basic Enforcement Techniques

- Interposition
  - Some trusted code is guaranteed to run each time an operation requiring authorization is requested
- Naming
  - No authorization check is required (at the time the operation is requested) because its impossible to name a resource you don't have authorization to use
- (Hybrid: Virtual Memory and access rights)

© 2010 Gribble, Lazowska, Levy, Zahorjan

Interposition: Protected Operations: $F_P(r)$

- No one can execute F, except for the OS
  - Therefore, must “call” the OS to do F (Privileged instructions)
  - When OS gets request $F_P(r)$, check authorization

- An optimization: caching
  - Require an authorization call to look up what the principal can do with the resource
    - E.g., a file open
  - Save result in OS memory
  - On $F_P(r)$, look up cached result
    - If no cached result, protection violation
    - If cached result says 'no,' protection violation

- A further optimization:
  - Hand back a fast-lookup key in response to original authorization call
    - A file handle, h: an integer index into the open file table
    - Read/write invocation is actually $F_P(h)$

© 2010 Gribble, Lazowska, Levy, Zahorjan

Interposition: Protected Resources: $F_P(r)$

- Does P have authorization to access r?
- Example: files
  - A "typical system" associates an owner and group with a file
  - It also records rwx permissions for the ower, group, and "other" with each file
  - When P tries to access file r, the OS
    - Checks to see if P is the owner, then...
    - Checks to see if P is in r's group, then...
    - Checks to see if F is allowed for "others"
- Note: there are other ways to do this
  - E.g., associate a list of authorized reader's with the file (NTFS)
  - Associate a list of files P is allowed to read with P (capabilities)
  - Idea is the same, though...

© 2010 Gribble, Lazowska, Levy, Zahorjan

Protection Via Namespace Manipulation

- We've just seen an example – file open:
  - Global name r (e.g., /bin/cat) is translated to a local name h (e.g., 4)
  - All requests that actually manipulate the file must use names in the local namespace
  - The OS controls the entries in the local namespace
    - You can ask to read file handle 20, but it can't possibly succeed unless '20' has been inserted into the namespace
- Another example: virtual memory
  - You can try to read/write any memory location, but the address you give is a local name (a virtual address)
  - The OS controls insertions into the local namespace (the page tables)
- Theme
  - Create a namespace in which only accessible objects are named
  - Have some trusted entity (e.g., the OS) control insertion into the namespace
  - Require a local name for naming an object

© 2010 Gribble, Lazowska, Levy, Zahorjan

Local Namespaces and Protection

- Local namespaces provide very strong security
  - They're the fundamental reason virtual machines are secure: ALL names are local (to that virtual machine)
- Local namespaces aren't handy for sharing
- Imagine a world in which file names were local
  - Your /bin/cat and my /bin/cat wouldn't mean the same thing
  - What's wrong with that?
- It's common for their to be a way around the protection
  - E.g., memory segments
    - A nameable chunk of real memory
    - Two or more processes can ask the OS to insert that memory into their local namespaces (virtual address spaces)

© 2010 Gribble, Lazowska, Levy, Zahorjan

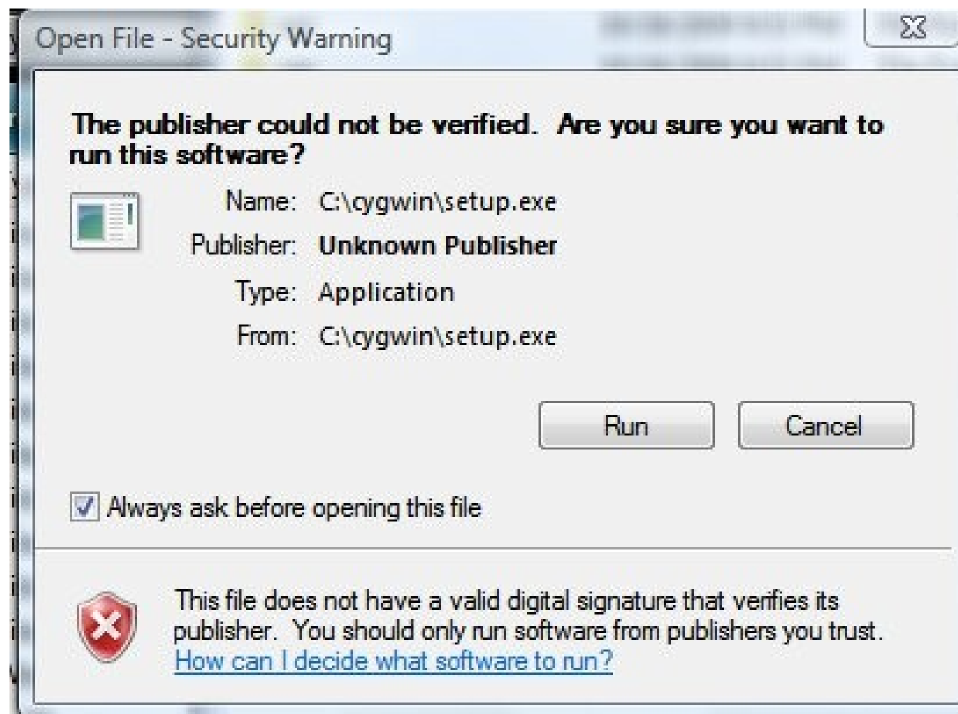
Protection Via Global Names

- Another approach is to combine authentication and authorization via the ability to name an object
  - If you can name it, you can use it
- For this to be viable, it must not be possible to simply manufacture a name
  - You get a name only because someone who had it gave it to you
  - This is an instance of delegation
- Example:
  - Names are 256-bit strings
- Benefits:
  - Scalable: No per-client state maintained by OS; in fact, no per-client identities
- Issues:
  - As described can't control delegation
  - Can't implement revocation

© 2010 Gribble, Lazowska, Levy, Zahorjan

Security Policy

- Policy: who should be allowed to do what?
- Often specifying policy is the problem:



© 2010 Gribble, Lazowska, Levy, Zahorjan

4. Allocation

- The OS allocates resources
  - Use of CPU, disk, other devices
  - Memory space, disk space
- There are questions of mechanism
  - How are the resources organized
- There are questions of policy
- The policy-mechanism distinction is worth remembering

© 2010 Gribble, Lazowska, Levy, Zahorjan

Example Policies

- Policies reflect goals
- Examples (using CPU):
  - Maximize CPU utilization
    - Minimize OS overhead by running (basically) non-preemptively
  - Provide fairness
    - Allocate CPU to each process an equal fraction of time
  - Provide unfairness:
    - Undergrad processes have priority over faculty processes
  - Meet real-time constraints
    - Allocate in a way that all processes meet their next deadline

© 2010 Gribble, Lazowska, Levy, Zahorjan

Policy/Mechanism Dichotomy

- Cleanly separating policy from mechanism allows pushing policy to the application
  - We've seen mechanisms to do this: micro-kernels, exokernel
- In theory, applications can make custom policy decisions that work better for them than a system-wide policy implemented in the OS

© 2010 Gribble, Lazowska, Levy, Zahorjan

5. Performance

- Much of what the OS does is overhead
  - E.g., all the work required to implement allocation policy
- You'd of course like these overheads to be as small as possible
  - Use efficient algorithms / data structures
  - Design decisions and overhead are intimately linked
- We'll look here at a general model of cost, and general approaches to addressing them

© 2010 Gribble, Lazowska, Levy, Zahorjan

The Cost Model

- Costs can be measured in many ways
  - User centric: the elapsed time from starting an operation until it completes
  - System centric: the total amount of resource consumed to complete the operation
- Our model:

  Cost =fixed overhead + #units * time/unit

- Examples:
  - System call: time to enter/exit OS + time to run syscall procedure
  - Disk: fixed disk overhead (seek/latency) + transfer size/bandwidth

© 2010 Gribble, Lazowska, Levy, Zahorjan

Reducing Costs

- If the fixed cost is small, life is easy:
  - The total cost to complete a fixed number of units of work is independent of the size of each operation
  - Best strategy: "on demand"
    - Do exactly the work requested, when it's requested

- When the fixed cost is appreciable, life is hard
  - Want to do big chunks of work, to amortize fixed costs
  - An efficient chunk size may be larger than what is currently requested
    - Speculate (pre-fetch)
    - Support more complicated operations
      - Web servers led to a system call that would stream an entire file out to the network, instead of having to do a sequence of file_read / network_write operations

© 2010 Gribble, Lazowska, Levy, Zahorjan

Reducing Costs: Caching

- Technically, caching means remembering the result of an operation in a way that makes re-accessing it cheap, in case it's needed again
  - We often associate it with the memory hierarchy
- We might think of it as any way of saving state to reduce the cost of subsequent operations
  - Example: file handle namespace
  - Other examples:
    - memoization in programming languages
    - dynamic programming
    - just-in-time compilation

© 2010 Gribble, Lazowska, Levy, Zahorjan

Caching Issues

- Caching changes the question answered from “What is the value now of...” to “What was the value back then?”

- This always raises the issue of coherence:
  - How do I make sure the cached value is current?
  - Sub-issues: invalidation, write policy, and all the other topics from CSE 378

- Software caching is the source of many bugs...

© 2010 Gribble, Lazowska, Levy, Zahorjan

Relaxing Coherence Constraints

- One way to get the benefits of caching without the complexities and costs of coherence is to forget about coherence
  - We'll call these cached values hints
    - The stored value may or may not be "right"
- Example: file open
  - Permissions stored in the open file table are those that were in force when the file was opened
  - User's right to read the file may have been revoked since then
  - They get to read the file anyway
- Whether hints are good enough is application dependent...

© 2010 Gribble, Lazowska, Levy, Zahorjan

A Sense of Absolute Costs

Obtained using lmbench

Proc	AMD Athlon 64 X2 (2.8GHz, 0.358 nsec. Clock)				
OS	Linux 2.6.31.4				
32-bit int add	2.3	nsec		int parallelism	1.26
32-bit int div	47.6	nsec			
float add	2.5	nsec		float parallelism	2.7
float div	18.1	nsec			
null syscall	220.2	nsec			
stat	1,311.1	nsec			
file open/close	2,727.6	nsec.			
sig hdlr install	326.5	nsec.			
sig hdlr ovrhd	1,602.5	nsec			
protection fault	303.1	nsec			
page fault	1,556.7	nsec.			
ctx switch	2,290.0	nsec.	2 processes writing 0 data bytes		
	5,270.0	nsec.	2 processes writing 64KB data		
	22,470.0	nsec.	16 processes writing 64KB data		
fork	331,500.0	nsec			
fork + exec	342,400.0	nsec.			
fork + sh cmd	1,960,700.0	nsec.			
disk seek	6,000,000.0	nsec.	highly variable		
disk xfer rate	50,000.0	nsec./4KB	highly variable		

© 2010 Gribble, Lazowska, Levy, Zahorjan

Recap: What is an Operating System?

- We're still not sure
- An operating system (OS) is:
  - a software layer to abstract away and manage details of hardware resources
  - a set of utilities to simplify application development
  - "all the code you didn't write" in order to implement your application
  - the code that runs in privileged mode
  - The code that enforces allocation *policy*

© 2010 Gribble, Lazowska, Levy, Zahorjan

The major OS issues

- **structure**: how is the OS organized?
- **sharing**: how are resources shared across users?
- **naming**: how are resources named (by users or programs)?
- **security**: how is the integrity of the OS and its resources ensured?
- **protection**: how is one user/program protected from another?
- **performance**: how do we make it all go fast?
- **reliability**: what happens if something goes wrong (either with hardware or with a program)?
- **extensibility**: can we add new features?
- **flexibility**: are we in the way of new apps?
- **communication**: how do programs exchange information, including across a network?

© 2010 Gribble, Lazowska, Levy, Zahorjan

More OS issues…

- **concurrency**: how are parallel activities (computation and I/O) created and controlled?
- **scale**: what happens as demands or resources increase?
- **persistence**: how do you make data last longer than program executions?
- **distribution**: how do multiple computers interact with each other?
- **accounting**: how do we keep track of resource usage, and perhaps charge for it?

There are tradeoffs, not right and wrong.

© 2010 Gribble, Lazowska, Levy, Zahorjan

CSE 451

- In this class we will learn:
  - what are the major components of most OS's?
  - how are the components structured?
  - what are the most important (common?) interfaces?
  - what policies are typically used in an OS?
  - what algorithms are used to implement policies?

- Philosophy
  - you may not ever build an OS
  - but as a computer scientist or computer engineer you need to understand the foundations
  - most importantly, operating systems exemplify the sorts of engineering design tradeoffs that you'll need to make throughout your careers – compromises among and within cost, performance, functionality, complexity, schedule …

© 2010 Gribble, Lazowska, Levy, Zahorjan