CSE 451: Operating Systems
Spring 2010

Module 2
Architectural Support for Operating Systems

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

Outline

- Part 1: Basics of Architecture / OS Interaction
  - Largely a review of CSE 378 material
- Part 2: The Hardware Architecture and Virtual Machines
  - Note: We'll need the basics of the Part 1 basics to understand Part 2...

Even coarse architectural trends impact tremendously the design of systems

- Processing power
  - doubling every 18 months
  - 60% improvement each year
  - factor of 100 every decade

  - 1980: 1 MHz Apple II+ == $2,000 ($5200)
    - 1980 also 1 MIPS VAX-11/780 == $120,000 ($312K)
  - 2011: 3.0GHz Quad Core == $530

© 2011 Gribble, Lazowska, Levy, Zahorjan

Power Consumption

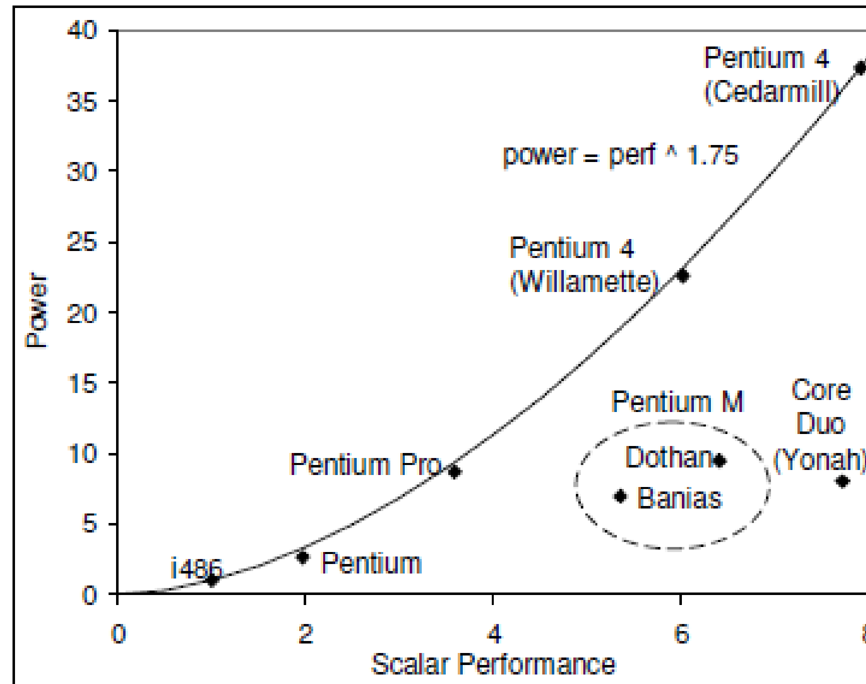
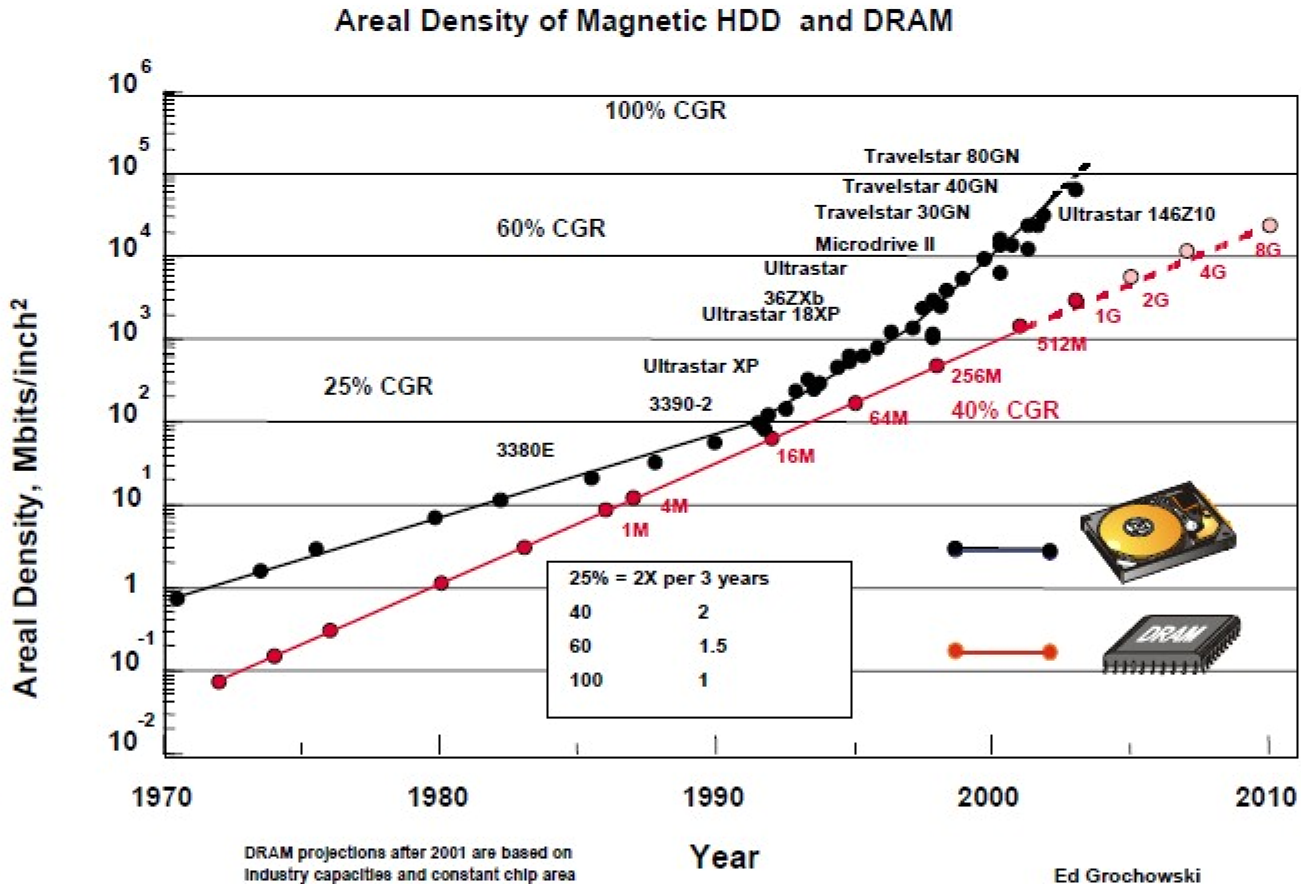


Figure 2: Normalized Power versus Normalized Scalar Performance for Multiple Generations of Intel Microprocessors

http://www.intel.com/pressroom/kits/core2duo/pdf/epi-trends-final2.pdf

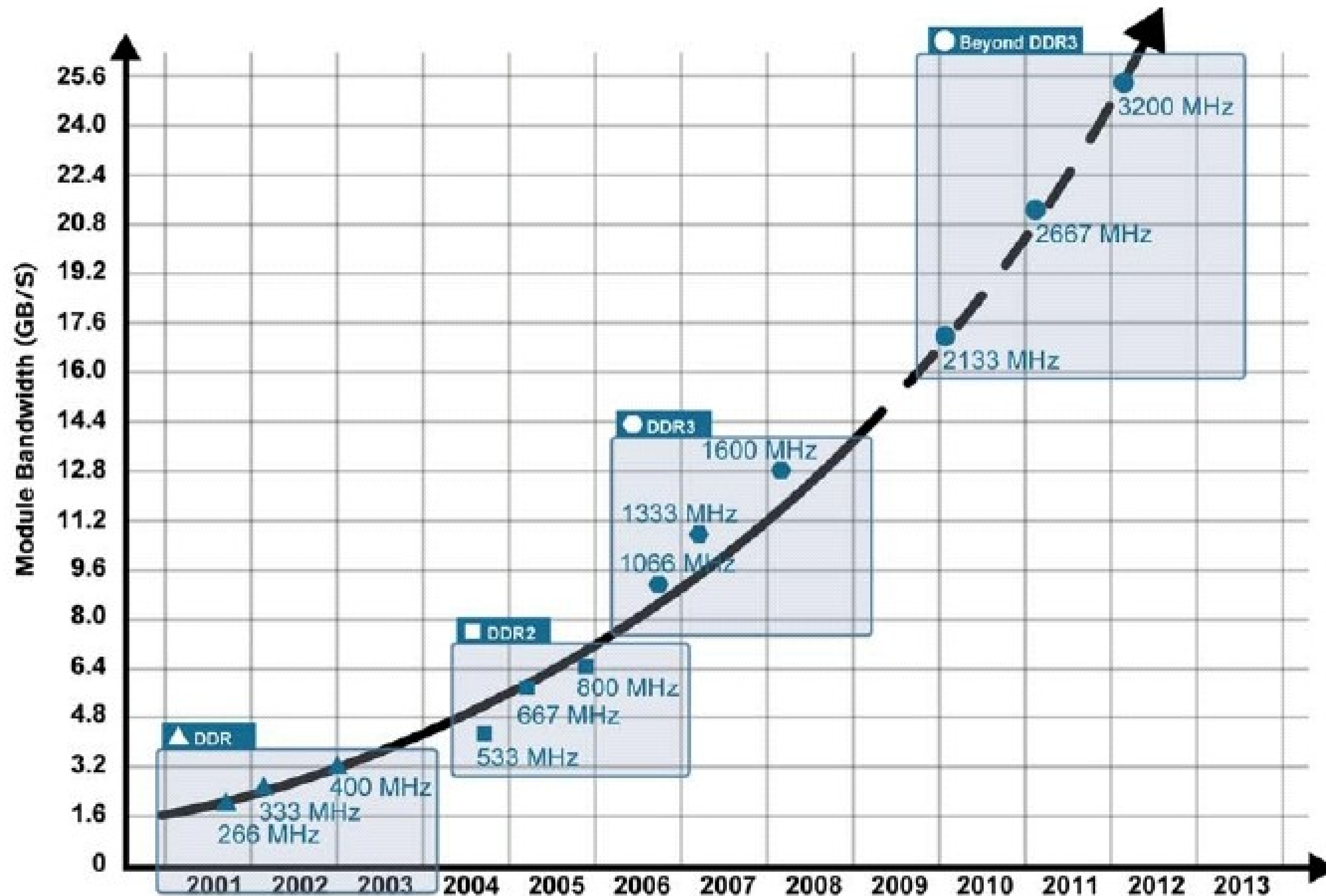
© 2011 Gribble, Lazowska, Levy, Zahorjan

Primary Memory / Disk Capacity



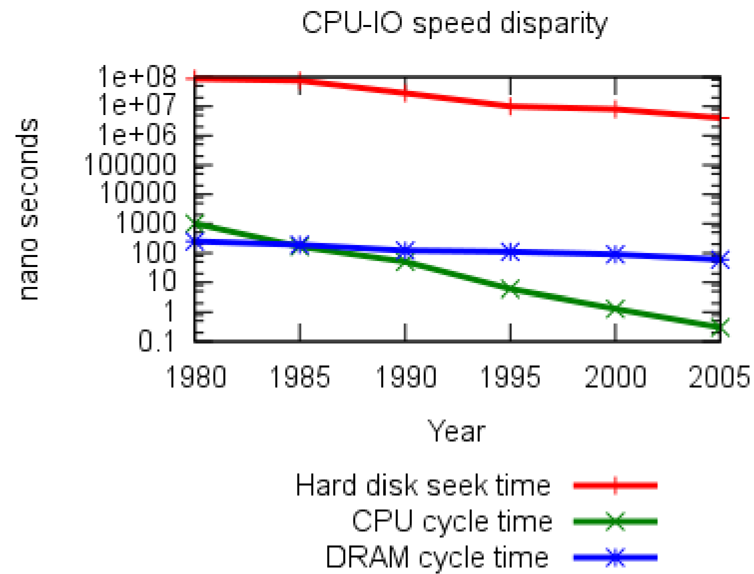
© 2011 Gribble, Lazowska, Levy, Zahorjan

Primary Memory Bandwidth



© 2011 Gribble, Lazowska, Levy, Zahorjan

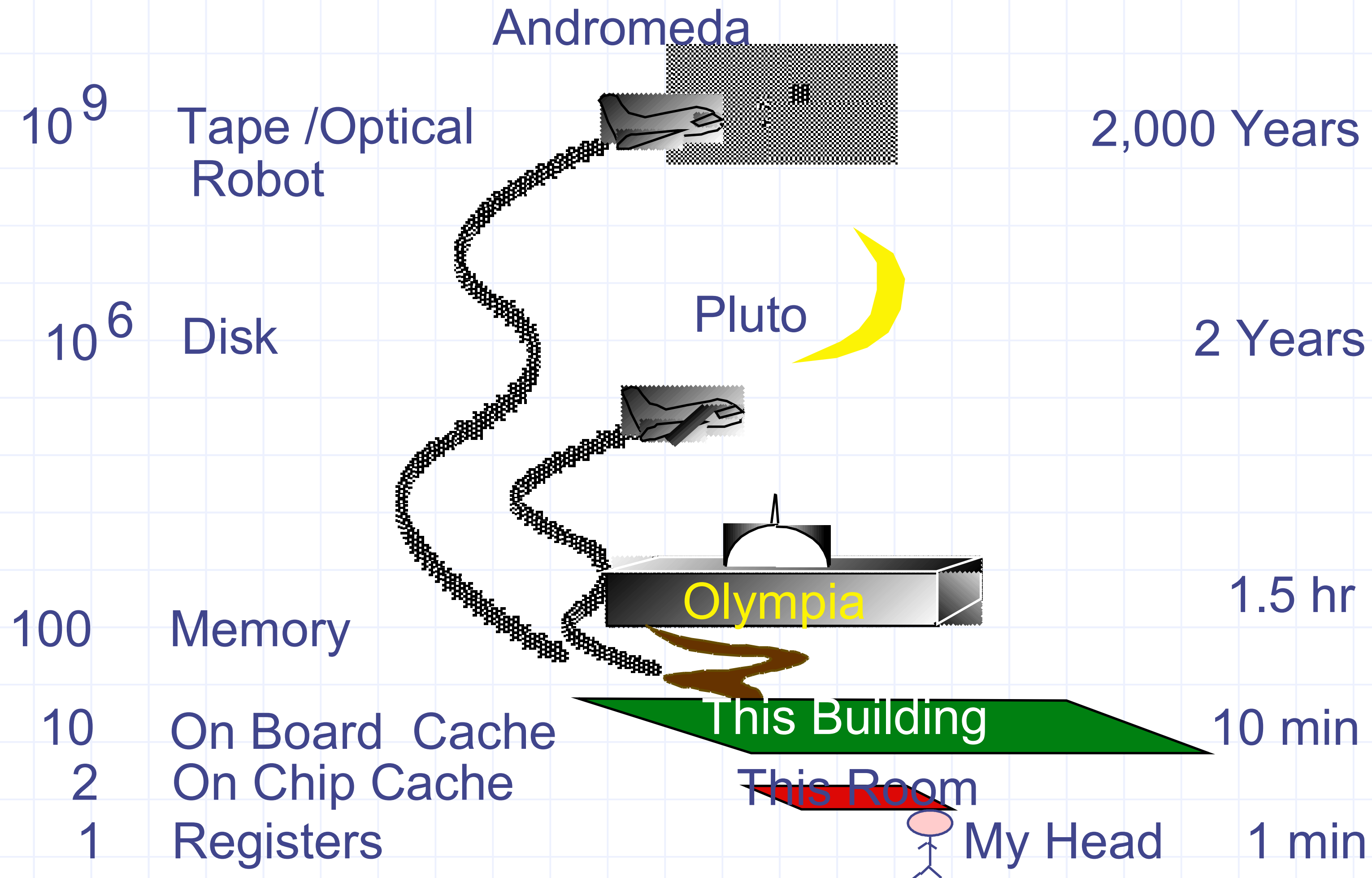
Relative Speeds



http://www.cs.cmu.edu/~amarp/cpu-io-gap.png

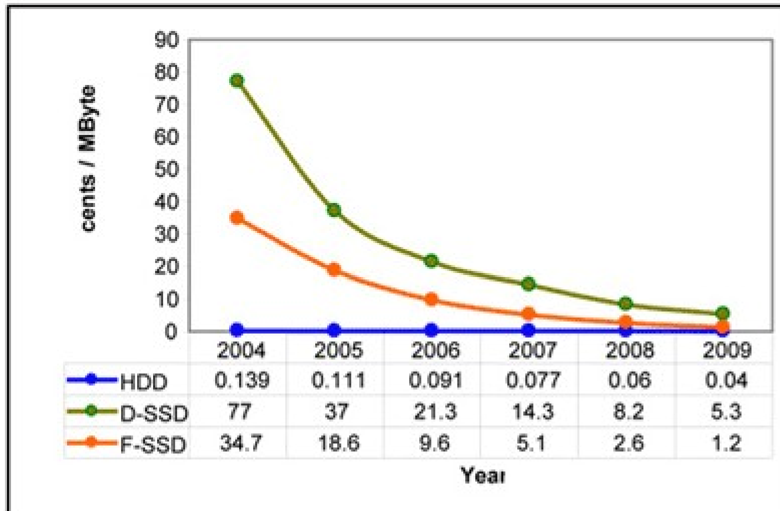
© 2011 Gribble, Lazowska, Levy, Zahorjan

Storage Latency: How Far Away is the Data?



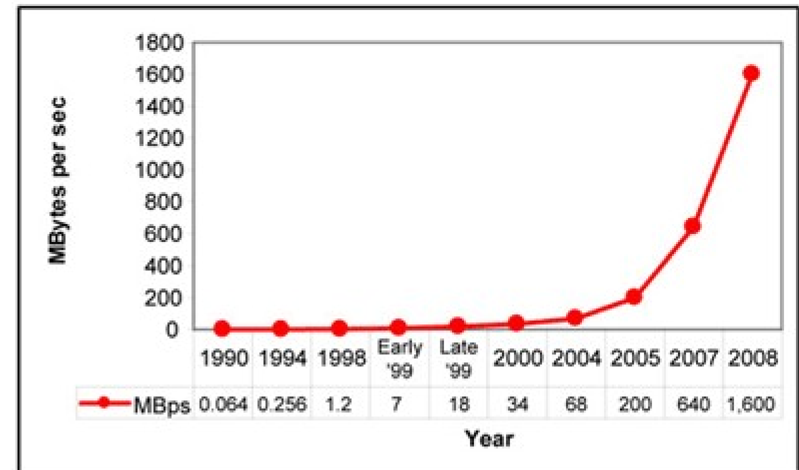
© 2004 Jim Gray, Microsoft Corporation

A Current Trend: Solid State Disks



	2004	2005	2006	2007	2008	2009
HDD	0.139	0.111	0.091	0.077	0.06	0.04
D-SSD	77	37	21.3	14.3	8.2	5.3
F-SSD	34.7	18.6	9.6	5.1	2.6	1.2

Figure B: HDD and SSD Storage Price Trend (2004-2009), cents / MByte
Source: Web-Feet Research



	1990	1994	1998	Early '99	Late '99	2000	2004	2005	2007	2008
MBps	0.064	0.256	1.2	7	18	34	68	200	640	1,600

Figure C: 3.5-inch Flash-SSD Sustained Random Read/Write Rates Trend

http://www.embeddedstar.com/articles/2005/2/article20050207-4.html

© 2011 Gribble, Lazowska, Levy, Zahorjan

Lower-level architecture affects the OS dramatically

- Operating system functionality is dictated, at least in part, by the underlying hardware architecture
  - includes instruction set (synchronization, I/O, …)
  - also hardware components like MMU or DMA controllers
- Architectural support can vastly simplify (or complicate!) OS tasks
  - e.g.: early PC operating systems (DOS, MacOS) lacked support for virtual memory, in part because at that time PCs lacked necessary hardware support

© 2011 Gribble, Lazowska, Levy, Zahorjan

Architectural features affecting OS's

- These features were built primarily to support OS's:
  - timer (clock) operation
  - synchronization instructions (e.g., atomic test-and-set)
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of execution (kernel vs. user)
  - protected instructions
  - system calls (and software interrupts)

- [2006] virtualization architectures
  - Intel: *http://www.intel.com/technology/itj/2006/v10i3/1-hardware/7-architecture-usage.htm*
  - AMD: *http://sites.amd.com/us/business/it-solutions/usage-models/virtualization/Pages/amd-v.aspx*

© 2011 Gribble, Lazowska, Levy, Zahorjan

Protected instructions

- some instructions are restricted to the OS
  - known as protected or privileged instructions

- e.g., only the OS can:
  - directly access I/O devices (disks, network cards)
    - why?
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
    - why?
  - manipulate special ‘mode bits’
    - interrupt priority level
    - why?
  - halt instruction
    - why?

© 2011 Gribble, Lazowska, Levy, Zahorjan

OS protection

- So how does the processor know if a protected instruction should be allowed?
  - the architecture must support at least two modes of operation: kernel mode and user mode
    - VAX, x86 support 4 protection modes
  - mode is set by status bit in a protected processor register
    - user programs execute in user mode
    - OS executes in kernel (privileged) mode (OS == kernel)
- Protected instructions can only be executed in kernel mode
  - what happens if user mode executes a protected instruction?

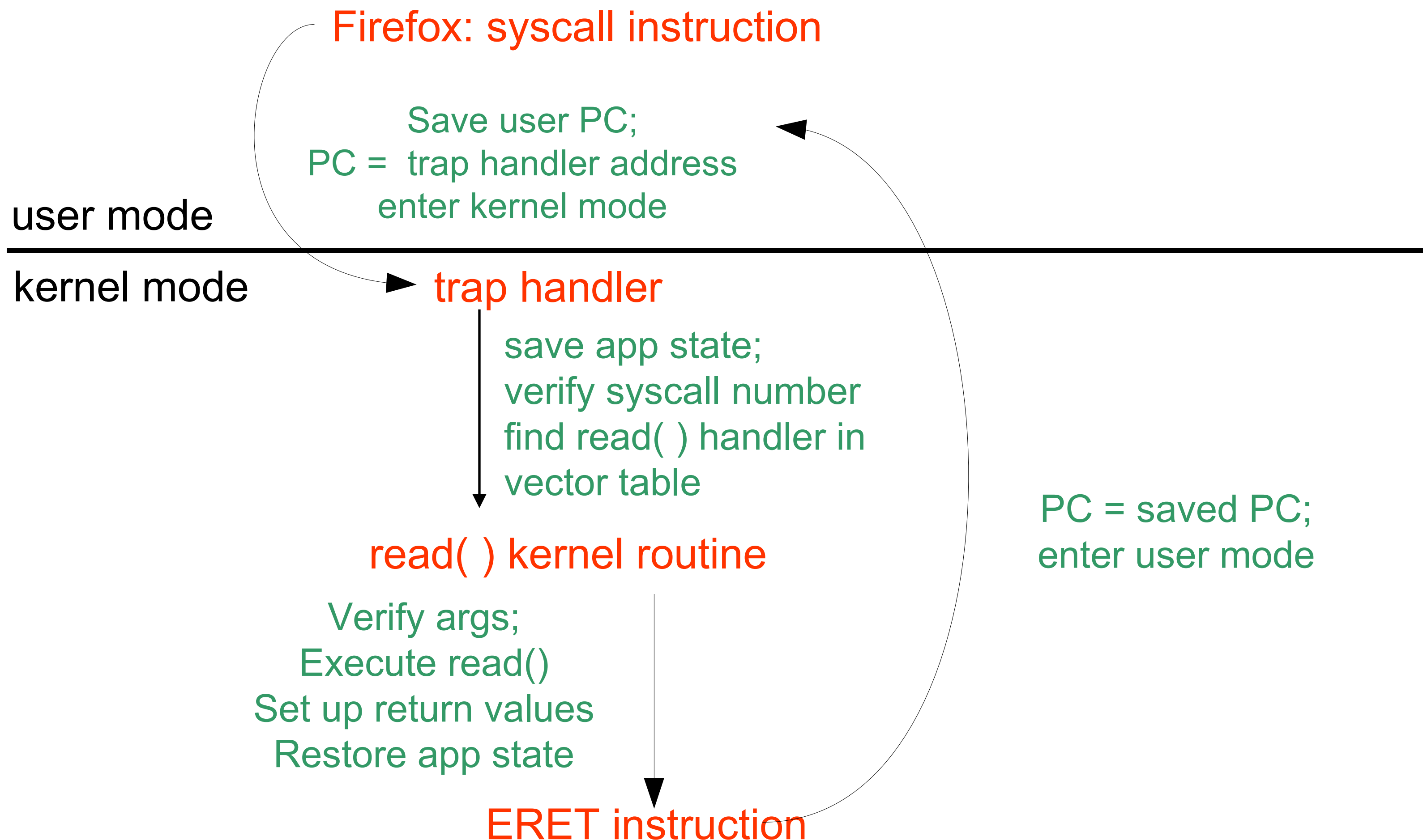
© 2011 Gribble, Lazowska, Levy, Zahorjan

Crossing protection boundaries

- User programs must call an OS procedure to do something privileged
  - OS defines a set of system calls
  - System calls act like protected procedure calls
- A syscall instruction atomically:
  - Saves the current PC
  - Sets the PC to a handler address
    - Writing the handler address is a privileged operation
  - Sets the execution mode to privileged
- With that, it's a lot like local procedure call (jal):
  - Caller puts arguments in a place callee expects (registers or stack)
    - One of the args is a syscall number, indicating which OS function to invoke
  - Callee (OS) saves caller's state (regs, other control info) so it can use CPU
  - OS function code runs
    - OS must verify caller's arguments (e.g., pointers)
  - OS returns using a special instruction (e.g., ERET – i.e., not JR)
    - Atomically sets PC to return address and sets execution mode to user

© 2011 Gribble, Lazowska, Levy, Zahorjan

A kernel crossing illustrated



© 2011 Gribble, Lazowska, Levy, Zahorjan

System call issues

- What would be wrong if syscall worked like subroutine call, with the caller specifying the next PC?
- What would happen if kernel didn’t save state?
- Why must the kernel verify arguments?
- How can you reference kernel objects as arguments to or results from system calls?

© 2011 Gribble, Lazowska, Levy, Zahorjan

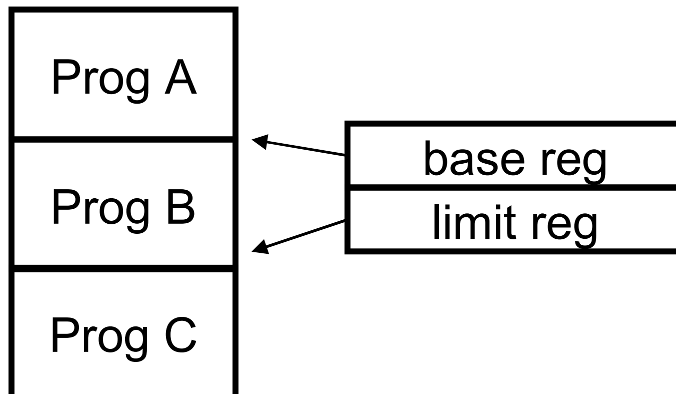
Exception Handling and Protection

- *All* entries to the OS occur via the mechanism just shown
  - Acquiring privilege mode execution and branching to the trap handler are inseparable
- Terminology:
  - Interrupt: asynchronous; caused by an external device
  - Exception: synchronous; unexpected problem with instruction
  - Trap: synchronous; intended transition to OS due to an instruction
- Privileged instructions and resources are the basis for most everything:
  - Memory protection
  - Protected I/O
  - Limiting user resource consumption

© 2011 Gribble, Lazowska, Levy, Zahorjan

Memory protection

- OS must protect user programs from each other
  - maliciousness, ineptitude
- OS must also protect itself from user programs
  - integrity and security
  - what about protecting user programs from OS?
- Simplest scheme: base and limit registers
  - are these protected?



base and limit registers are loaded by OS before starting program

© 2011 Gribble, Lazowska, Levy, Zahorjan

I/O control

- How does the kernel start an I/O?
    - special I/O instructions (privileged operation)
    - memory-mapped I/O (privileged memory)
- How does the OS know an I/O operation has finished?
    - Polling (sit in loop asking device if it's done)
    - Interrupt ("get back to me when you're finished")
- Interrupts are basis for asynchronous I/O
    - OS starts IO operation, then goes on doing whatever
    - device performs an operation asynchronously to CPU activity
    - device sends an interrupt signal on bus when done
    - Interrupt causes a kernel entry

© 2011 Gribble, Lazowska, Levy, Zahorjan

Protected CPU Use: Timers

- How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
  - use a hardware countdown timer
    - Generates an interrupt when it has counted down to 0
  - before it transfers to a user process, the OS sets the countdown timer
    - E.g., sets a number that will cause an interrupt in 10 msec.
  - now either the user process invokes the OS within the next 10 msec. or the timer goes off
    - Either way, the OS has a chance to choose a different process to “dispatch” next

© 2011 Gribble, Lazowska, Levy, Zahorjan

Part 2: Architecture and Virtual Machines

- Virtual machines date back to the 1960's
  - IBM has had a commercial product since then

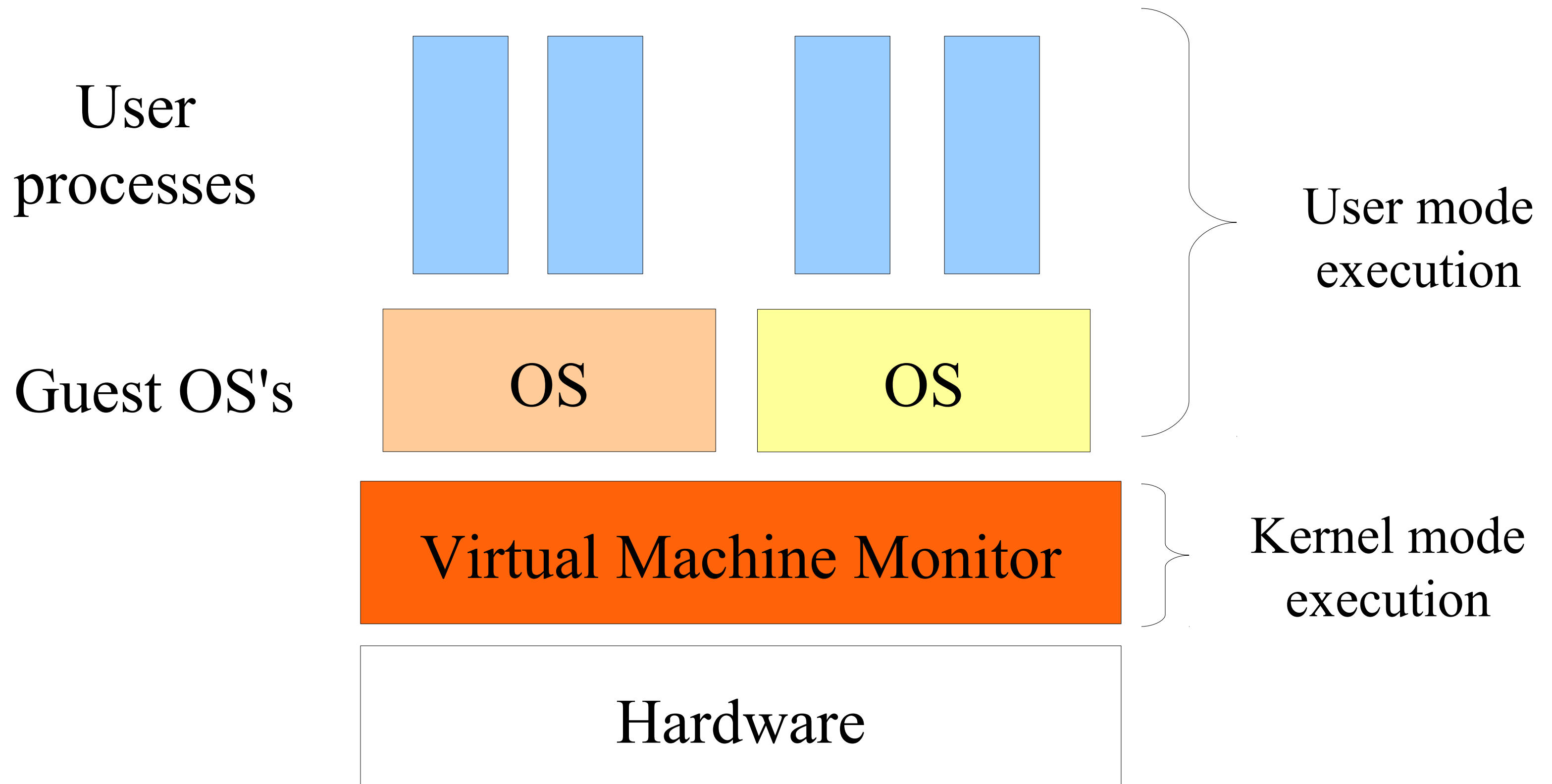
- A virtual machine is an efficient, software produced "hardware" execution environment
  - Efficient: virtual machine user processes run directly on the physical CPU
    - Eliminates Java or other software simulators
  - Hardware execution environment: any code that would run on an identically configured physical system will run on the virtual machine, including an OS

- Because each is like a separate hardware machine, they encapsulate complete namespaces for all resources
  - No sharing of names ⇒ Complete isolation

- They've undergone a huge resurgence the last decade

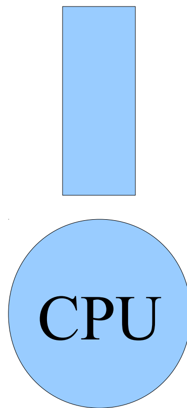
© 2011 Gribble, Lazowska, Levy, Zahorjan

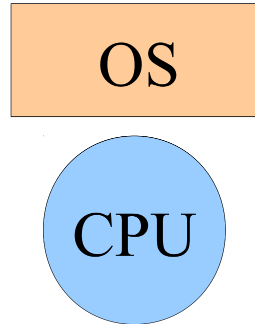
Virtual Machines: The Layered View

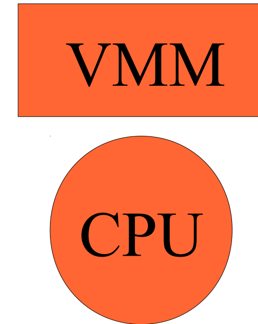


© 2011 Gribble, Lazowska, Levy, Zahorjan

Temporal Snapshots







The user process is executing in user mode, as expected

The guest OS is executing in user mode, which it isn't expecting

The VMM is executing in privileged mode, which it is expecting

© 2011 Gribble, Lazowska, Levy, Zahorjan

Basic Ideas

- When any kind of exception or interrupt occurs, we'll end up in the VMM, rather than the guest OS
  - VMM simulates state changes that would have been made by the hardware, then restarts VM at the guest OS handler address
    - E.g., stuffs the saved PC where the architecture says it should be
- When the guest OS tries to execute a privileged instruction:
  - Same thing
- What's so hard?
  - Not all instructions that require VMM intervention may be privileged
    - Depends on the architecture

© 2011 Gribble, Lazowska, Levy, Zahorjan

An Example Problem

- Suppose the ERET instruction (return to user process after handling exception) is not privileged
  - ERET sets the PC to the saved PC, and
  - Sets CPU privilege mode to user
  - There doesn't seem to be a reason to prevent user processes from doing that (even if there's no reason for them to want to)
- The problem for VMMs:
  - They have to track the virtual CPU protection level to know how to respond to a privileged instruction exception
    - When the virtual CPU state is privileged, the VMM should execute the effect of the privileged instruction
    - When it's user, it should execute the effect of a privileged instruction exception on the virtual machine
  - They therefore have to know when the guest OS executes an ERET
- Whether or not virtualization is possible depends on the hardware architecture

© 2011 Gribble, Lazowska, Levy, Zahorjan

"Formal Requirements for Virtualizable Third Generation Architectures"

- Popek and Goldberg, 1974
  - A local copy is available at http://www.cs.washington.edu/education/courses/cse451/11sp/protected/popekGoldberg.pdf
- A virtual machine is:
  - *Efficient*: only minor slowdowns for user processes
  - *Isolated*: the VMM controls physical resource allocation, not the guest OS
  - *A Duplicate*: programs get the same results as on physical machines, modulo resource availability and timing issues
    - Virtual machine configuration is smaller than the physical machine (less memory, less disk, …)
    - Virtual machine may have more or fewer devices
      - More disks, for example

© 2011 Gribble, Lazowska, Levy, Zahorjan

When Can An Architecture Be Virtualized?

- It helps to identify three classes of instructions
  - Privileged
    - Cause a trap when executed in user mode
  - Sensitive
    - Control Sensitive
      - Instruction changes machine mode (user/privileged) or control settings (e.g., memory mapping)
    - Behavior Sensitive
      - Instruction gets different results (including next PC value) depending on machine mode, or the real addresses of things
        - » S/360 Load Real Address (LRA) instruction
  - Innocuous
      - Everything else

© 2011 Gribble, Lazowska, Levy, Zahorjan

Formal Results

- *THEOREM 1. For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

- (Theorem 2: If you can virtualize, you can recursively virtualize.)

© 2011 Gribble, Lazowska, Levy, Zahorjan

x86 Architecture

- Doesn't meet the Popek & Goldberg virtualization criteria
- There are many reasons, but most of them stem from behavior sensitive instructions that don't fault in user mode
  - These are “safe” operations, when not implementing virtual machines, and so not privileged
  - E.g., reading control values (like page table pointers) is not a privileged operation
- The physical page table pointer registers must be set to what the VMM wants, not what the guest OS expects
  - If the guest OS reads them, there's no chance for the VMM to fix up the results of the read, and the guest OS gets confused
- So, how is the x86 being virtualized?

© 2011 Gribble, Lazowska, Levy, Zahorjan

Virtualizing the x86: Software Approach

- Basic idea (as always):
  - Execute innocuous instructions natively
  - Trap on sensitive instructions and simulate their effect
- How do we find the sensitive instructions?
  - Once the instructions are running natively, there's no trap when needed
  - Could simulate all instructions, but...
    - That isn't really a virtual machine, in the sense we mean here, because it's too slow
- Approach: find them *statically*, not *dynamically*
  - “Read the source,” not the stream of executing instructions
  - When you find a sensitive instruction...
    - What do we need to do here?
      - Remember, I want the innocuous instructions to execute natively, and the sensitive ones are hiding in their midst

© 2011 Gribble, Lazowska, Levy, Zahorjan

Binary Rewriting

- The VMM will have control whenever a code page is being loaded
  - Loading code involves setting page table permissions
  - That involves a privileged instruction
- When the page is loaded
  - Scan the instructions it contains, looking for the sensitive instructions
  - Replace them with something that will cause a fault
  - Remember the instruction that used to be there
  - Now allow native execution of the code page
- This is the original approach used to grapple with the x86 by the VMMs you've heard of (e.g., VMWare), and is still in use

© 2011 Gribble, Lazowska, Levy, Zahorjan

Virtualizing x86: The Hardware Approach

- Both Intel and AMD have developed virtualization extensions to the architecture (starting ~2006)
  - Intel: VT-x
  - AMD: AMD-V
- To get correctness, they introduce new machine modes and duplicate the problematic state
  - New modes: VMX root (actual root) and VMX non-root (you think you're root, but you're not)
  - One sees the actual control information; the other sees a decoy copy, whose contents are controlled by the VMM
    - Turns a behavior sensitive instruction into an innocuous one
  - Additionally, many previously non-faulting instructions cause a fault when in VMS non-root mode
- The extensions also provide some efficiency improvements
  - E.g., they know that guest OS and VMM page tables need to be composed

© 2011 Gribble, Lazowska, Levy, Zahorjan

(Aside) An Alternative SW Approach: Paravirtualization

- Don't try to create a duplicate of hardware; create something useful and similar
  - Define result of problem sensitive instructions as "undefined"
  - Introduce new architectural features that improve virtual machine performance
    - New instructions: idle-with-timeout instruction (rather than idle loop)
    - Slightly higher level of abstraction for hardware device interfaces than hw provides
- Because it's not a duplicate, some modification of the guest OS may be required
- While you're at it, address some performance issues that trouble even virtualizable architectures
  - Example: Both the VMM and the guest OS are managing paging
    - Duplication of effort
    - Guest OS doesn't actually know what the state of memory is
    - Resolving a fault requires composing the guest OS and VMM page tables
  - Solution: remove paging from guest OS
- Qualitatively, paravirtualization is to virtual machines what hints are to caches
  - We change the problem definition slightly, and end up with much higher performance implementations

© 2011 Gribble, Lazowska, Levy, Zahorjan

OS/Architecture Summary

- Some architectural features are required to run an OS in a reasonable way and user processes at full machine speed
  - (At least) two modes of operation (privileged, user)
  - Privileged instructions
  - Memory protection
  - Interrupts / exceptions / timers
- We'll be seeing more of virtual machines for the foreseeable future
  - Their efficiency, and in fact the ability to create them at all, is strongly affected by the architecture
  - Architectural features are evolving explicitly to provide better support for them

© 2011 Gribble, Lazowska, Levy, Zahorjan