CSE 451: Operating Systems
Spring 2013

Module 2
Architectural Support for Operating Systems

Ed Lazowska
lazowska@cs.washington.edu
570 Allen Center

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 (~$5,000 today)
    - 1980 also 1 MIPS VAX-11/780 = $120,000 (~$300,000 today)
  - 2006: 3.0GHz Pentium D = $800
  - 2010: 3.0GHz Dual Core = $500
  - 2013: 2.7GHz Quad Core = $369

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 (~$5,000 today)
    - 1980 also 1 MIPS VAX-11/780 = $120,000 (~$300,000 today)
  - 2006: 3.0GHz Pentium D = $800
  - 2010: 3.0GHz Dual Core = $500
  - 2013: 2.7GHz Quad Core = $369

Power Consumption

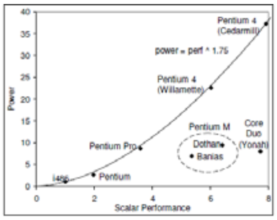
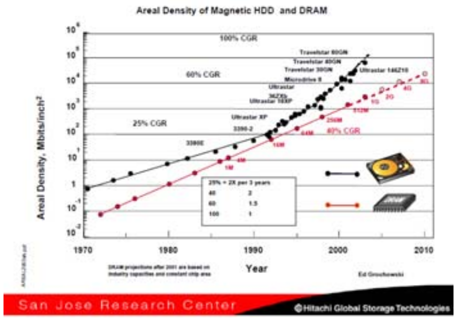


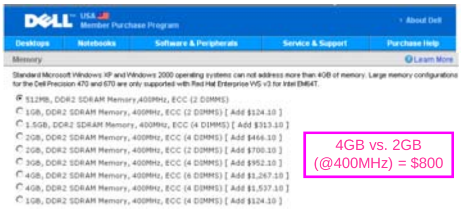
Figure 2: Normalized Power versus Normalized Scalar Performance for Multiple Generations of Intel Microprocessors

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

Primary Memory / Disk Capacity



- Primary memory cost
  - 1972: 1MB = $1,000,000
  - 1982: I remember pulling all kinds of strings to get a special deal: 512K of VAX-11/780 memory for $30,000
  - 2005:

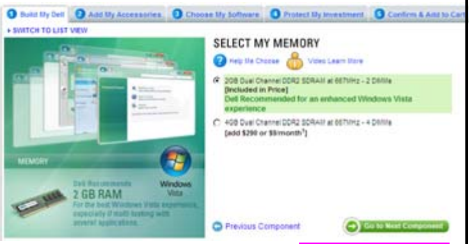


Standard Microsoft Windows XP and Windows 2000 operating systems can not address more than 4GB of memory. Large memory configurations for the Dell Precision 470 and 670 are only supported with Red Hat Enterprise WS v3 for Intel EM64T.

- ● 512MB, DDR2 SDRAM Memory,400MHz, ECC (2 DIMMS)
- ○ 1GB, DDR2 SDRAM Memory, 400MHz, ECC (2 DIMMS) [Add $124.10]
- ○ 1.5GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add $313.10]
- ○ 2GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add $466.10]
- ○ 2GB, DDR2 SDRAM Memory, 400MHz, ECC (2 DIMMS) [Add $700.10]
- ○ 3GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add $952.10]
- ○ 4GB, DDR2 SDRAM Memory, 400MHz, ECC (6 DIMMS) [Add $1,267.10]
- ○ 4GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add $1,537.10]
- ○ 1GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add $124.10]

4GB vs. 2GB (@400MHz) = $800

- 2007:



SELECT MY MEMORY

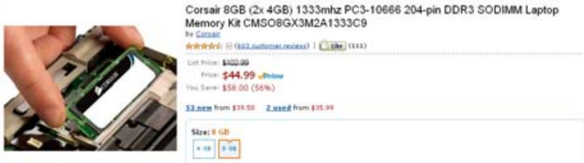
● 2GB Dual Channel DDR2 SDRAM at 667MHz - 2 DIMMs
[Included in Price]
Dell Recommended for an enhanced Windows Vista experience

○ 4GB Dual Channel DDR2 SDRAM at 667MHz - 4 DIMMs
[add $290 or $9/month¹]

Dell Recommends
2 GB RAM
For the best Windows Vista experience, especially if multi tasking with several applications.

4GB vs. 2GB (@667MHz) = $290

- Today:



Corsair 8GB (2x 4GB) 1333mhz PC3-10666 204-pin DDR3 SODIMM Laptop Memory Kit CMSO8GX3M2A1333C9
by Corsair
(403 customer reviews) | Like (111)

List Price: ~~$102.99~~
Price: $44.99 Prime
You Save: $58.00 (56%)

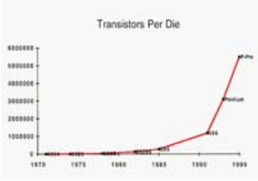
53 new from $39.50   2 used from $35.99

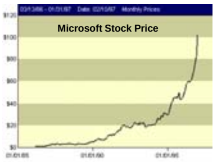
Size: 8 GB
4 GB | 8 GB

8GB (@1333MHz) = $44.99

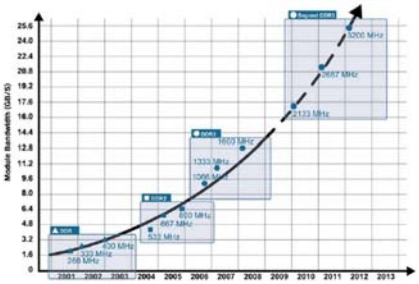
- Disk cost:
  - Only a few years ago, we purchased disks by the megabyte (and it hurt!)
  - Today, 1 GB (a billion bytes) costs ~~$1~~ ~~$0.50~~ $0.05 from Dell (except you have to buy in increments of ~~40~~ ~~80~~ 250 GB)
    - => 1 TB costs ~~$1K~~ ~~$500~~ $20, 1 PB costs ~~$1M~~ ~~$500K~~ $20K

- Aside: Where does it all go?
  - Facetiously: “What Gordon giveth, Bill taketh away”
  - Realistically: our expectations for what the system will do increase relentlessly
    - e.g., GUI
  - “Software is like a gas – it expands to fill the available space” – Nathan Myhrvold (1960-)



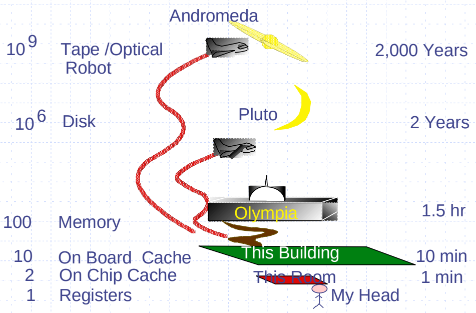


Primary Memory Bandwidth

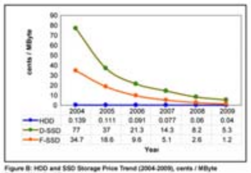


- Optical bandwidth today
  - *Doubling every 9 months*
  - 150% improvement each year
  - Factor of 10,000 every decade
  - 10x as fast as disk capacity!
  - 100x as fast as processor performance!!
- What are some of the implications of these trends?
  - Just one example: We have always designed systems so that they “spend” processing power in order to save “scarce” storage and bandwidth!

Storage Latency: How Far Away is the Data?

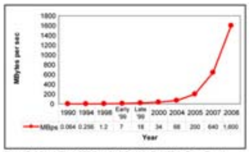


A Current Trend: Solid State Disks



	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	16	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

Lower-level architecture affects the OS even more dramatically

- The operating system supports sharing and protection
  - multiple applications can run concurrently, sharing resources
  - a buggy or malicious application can't nail other applications or the system
- There are many approaches to achieving this
- The architecture determines which approaches are viable (reasonably efficient, or even possible)
  - 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
    - Apollo workstation used two CPUs as a bandaid for non-restartable instructions!
  - Until very recently, Intel-based PCs still lacked support for 64-bit addressing (which has been available for a decade on other platforms: MIPS, Alpha, IBM, etc…)
    - Changed driven by AMD's 64-bit architecture

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)
  - privileged instructions
  - system calls (and software interrupts)
  - 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

Privileged instructions

- some instructions are restricted to the OS
  - known as 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?

OS protection

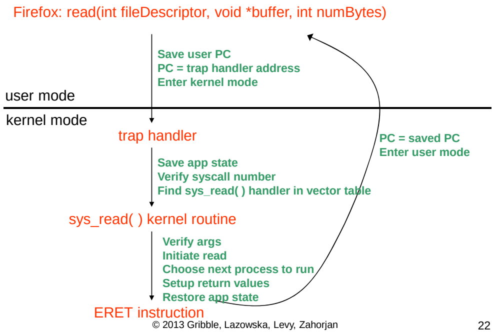
- So how does the processor know if a privileged instruction should be executed?
  - – the architecture must support at least two modes of operation: kernel mode and user mode
    - VAX, x86 support 4 protection modes
  - – mode is set by status bit in a protected processor register
    - user programs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

Crossing protection boundaries

- So how do user programs do something privileged?
  - – e.g., how can you write to a disk if you can't execute an I/O instructions?
- User programs must call an OS procedure – that is, get the OS to do it for them
  - – OS defines a set of system calls
  - – User-mode program executes system call instruction
- Syscall instruction
  - – Like a protected procedure call

- The syscall instruction atomically:
  - – Saves the current PC
  - – Sets the execution mode to privileged
  - – Sets the PC to a handler address
- With that, it's a lot like a local procedure call
  - – 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 (registers, other control state) so it can use the CPU
  - – OS function code runs
    - OS must verify caller's arguments (e.g., pointers)
  - – OS returns using a special instruction
    - Automatically sets PC to return address and sets execution mode to user

A kernel crossing illustrated



Firefox: read(int fileDescriptor, void *buffer, int numBytes)

Save user PC
PC = trap handler address
Enter kernel mode

user mode

kernel mode

trap handler

Save app state
Verify syscall number
Find sys_read( ) handler in vector table

sys_read( ) kernel routine

Verify args
Initiate read
Choose next process to run
Setup return values
Restore app state

ERET instruction

PC = saved PC
Enter user mode

System call issues

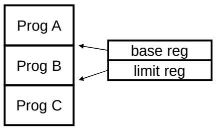
- What would be wrong if a syscall worked like a regular subroutine call, with the caller specifying the next PC?
- What would happen if kernel didn't save state?
- Why must the kernel verify arguments?
- How can you reference kernel objects as arguments to or results from system calls?

Exception Handling and Protection

- *All* entries to the OS occur via the mechanism just shown
  - – Acquiring privileged mode 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, …

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

More sophisticated memory protection

- coming later in the course
- paging, segmentation, virtual memory
  - page tables, page table pointers
  - translation lookaside buffers (TLBs)
  - page fault handling

I/O control

- Issues:
  - how does the OS start an I/O?
    - special I/O instructions
    - memory-mapped I/O
  - how does the OS notice an I/O has finished?
    - polling
    - Interrupts
  - how does the OS exchange data with an I/O device?
    - Programmed I/O (PIO)
    - Direct Memory Access (DMA)

Asynchronous I/O

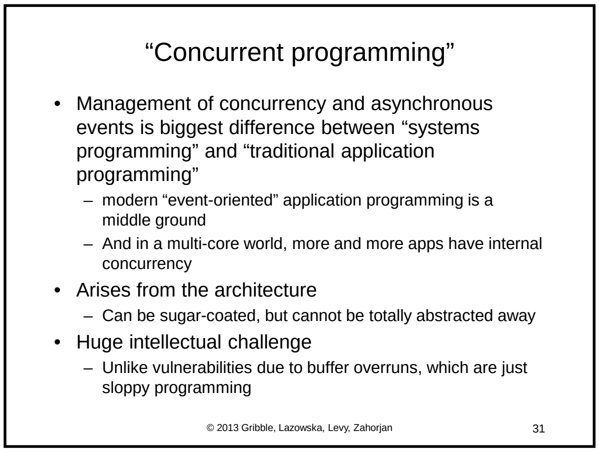
- Interrupts are the basis for asynchronous I/O
  - device performs an operation asynchronously to CPU
  - device sends an interrupt signal on bus when done
  - in memory, a vector table contains list of addresses of kernel routines to handle various interrupt types
    - who populates the vector table, and when?
  - CPU switches to address indicated by vector index specified by interrupt signal
- What's the advantage of asynchronous I/O?

Timers

- How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
  - use a hardware timer that generates a periodic interrupt
  - before it transfers to a user program, the OS loads the timer with a time to interrupt
    - "quantum" – how big should it be set?
  - when timer fires, an interrupt transfers control back to OS
    - at which point OS must decide which program to schedule next
    - very interesting policy question: we'll dedicate a class to it
- Should access to the timer be privileged?
  - for reading or for writing?

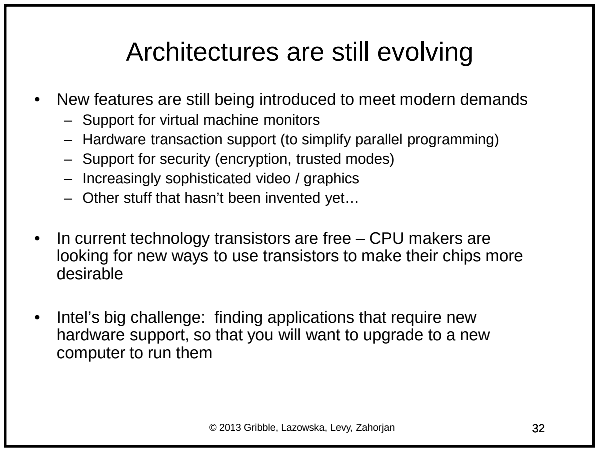
Synchronization

- Interrupts cause a wrinkle:
  - may occur any time, causing code to execute that interferes with code that was interrupted
  - OS must be able to synchronize concurrent processes
- Synchronization:
  - guarantee that short instruction sequences (e.g., read-modify-write) execute atomically
  - one method: turn off interrupts before the sequence, execute it, then re-enable interrupts
    - architecture must support disabling interrupts
      - Privileged???
  - another method: have special complex atomic instructions
    - read-modify-write
    - test-and-set
    - load-linked store-conditional



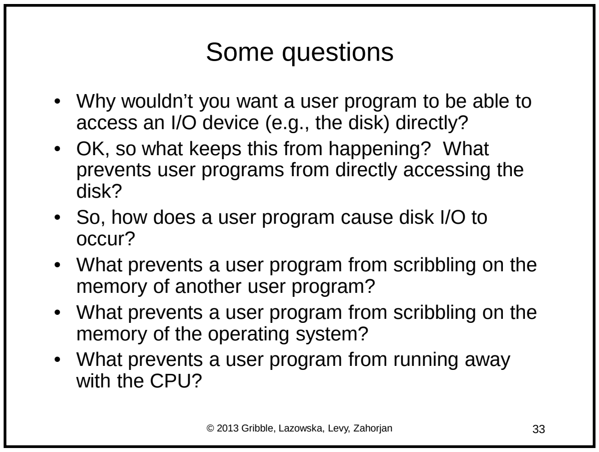
“Concurrent programming”

- Management of concurrency and asynchronous events is biggest difference between “systems programming” and “traditional application programming”
  - modern “event-oriented” application programming is a middle ground
  - And in a multi-core world, more and more apps have internal concurrency
- Arises from the architecture
  - Can be sugar-coated, but cannot be totally abstracted away
- Huge intellectual challenge
  - Unlike vulnerabilities due to buffer overruns, which are just sloppy programming



Architectures are still evolving

- New features are still being introduced to meet modern demands
  - Support for virtual machine monitors
  - Hardware transaction support (to simplify parallel programming)
  - Support for security (encryption, trusted modes)
  - Increasingly sophisticated video / graphics
  - Other stuff that hasn’t been invented yet…
- In current technology transistors are free – CPU makers are looking for new ways to use transistors to make their chips more desirable
- Intel’s big challenge: finding applications that require new hardware support, so that you will want to upgrade to a new computer to run them



Some questions

- Why wouldn’t you want a user program to be able to access an I/O device (e.g., the disk) directly?
- OK, so what keeps this from happening? What prevents user programs from directly accessing the disk?
- So, how does a user program cause disk I/O to occur?
- What prevents a user program from scribbling on the memory of another user program?
- What prevents a user program from scribbling on the memory of the operating system?
- What prevents a user program from running away with the CPU?