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
  - 1980 also 1 MIPS VAX-11/780 == $120,000
  - 2005: 3.5GHz Pentium 4 == $1,000

- Primary memory capacity
  - same story, same reason (Moore’s Law)
  - I remember pulling all kinds of strings to get a special deal: 512K of VAX-11/780 memory for $30,000
  - today:

- Disk capacity, 1975-1989
  - doubled every 3+ years
  - 25% improvement each year
  - factor of 10 every decade
  - Still exponential, but far less rapid than processor performance

- Disk capacity since 1990
  - doubling every 12 months
  - 100% improvement each year
  - factor of 1000 every decade
  - 10x as fast as processor performance!

- 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-)

- Only a few years ago, we purchased disks by the megabyte (and it hurt!)
- Today, 1 GB (a billion bytes) costs $1 $0.50 from Dell (except you have to buy in increments of 40 80 GB)
  - => 1 TB costs $50, 1 PB costs $1M $500K
  - In 2 years, 1 GB will cost $.10
  - => 1 TB for $100, 1 PB for $100K
• 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?

- Tape / Optical: 2,000 Years
- Disk: 2 Years
- Memory: 1.5 hr
- On Board Cache: 10 min
- On Chip Cache: 1 min
- Registers: 1 min
- System: 1 min
- My Head: 1 min
- This Room: 1 min
- This Building: 1 min
- Olympia: 1 min

Lower-level architecture affects the OS even more 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:
  - 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 band aid for non-runnable instructions!
  - Most current Intel-based PCs still lack support for 64-bit addressing (which has been available for a decade on other platforms: MIPS, Alpha, IBM, etc..)
    - Changing rapidly due to AMD’s new 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)
  - Protected instructions
  - System calls (and software interrupts)

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)
  - Manipulate memory state management
    - Page table pointers, TLB loads, etc.
  - Manipulate special ‘mode bits’
    - Interrupt priority level
  - Halts

OS protection:

- So how does the processor know if a protected 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 mode (OS == kernel)
- Protected instructions can only be executed in kernel mode
  - What happens if user mode executes a protected 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 I/O instructions?
- User programs must call an OS procedure:
  - OS defines a sequence of system calls
  - How does the user-mode to kernel-mode transition happen?
  - There must be a system call instruction, which:
    - Causes an exception (throws a software interrupt), which vectors to a kernel handler
    - Passes a parameter indicating which system call to invoke
    - Saves caller’s state (regs, mode bit) so they can be restored
  - OS must verify caller’s parameters (e.g., pointers)
  - Must be a way to return to user mode once done
A kernel crossing illustrated

Firefox: read()

trap to kernel mode; save app state

trap handler

find read() handler in vector table

read() kernel routine

restore app state, return to user mode, resume

System call issues

• What would happen if kernel didn’t save state?
• Why must the kernel verify arguments?
• How can you reference kernel objects as arguments or results to/from system calls?

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?

OS control flow

• after the OS has booted, all entry to the kernel happens as the result of an event
  – event immediately stops current execution
  – changes mode to kernel mode, event handler is called
• kernel defines handlers for each event type
  – specific types are defined by the architecture
    • e.g.: timer event, I/O interrupt, system call trap
  – when the processor receives an event of a given type, it transfers control to handler within the OS
    • handler saves program state (PC, regs, etc.)
    • handler functionality is invoked
    • handler restores program state, returns to program

Interrupts and exceptions

• Two main types of events: interrupts and exceptions
  – exceptions are caused by software executing instructions
    • e.g., the `int` instruction
    • e.g., a page fault, or an attempted write to a read-only page
  – an expected exception is a “trap”, unexpected is a “fault”
• Interrupts are caused by hardware devices
  • e.g., device finishes I/O
  • e.g., timer fires
I/O control

- Issues:
  - how does the kernel start an I/O?
    - special I/O instructions
  - memory-mapped I/O
  - how does the kernel notice an I/O has finished?
    - polling
    - interrupts
- Interrupts are 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

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 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
    - 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
- 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