CSE 451: Operating Systems
Winter 2004

Module 2
Architectural Support for Operating Systems

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

Even coarse architectural trends tremendously impact the design of systems

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

- Primary memory capacity
  - same story, same reason (Moore’s Law)
    - I remember pulling all kinds of strings to get a special deal: $12K of VAX-11/780 memory for $30,000
    - today:

- 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-)

- 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!

- Only a few years ago, we purchased disks by the megabyte (and it hurt!)
- Today, 1 GB (a billion bytes) costs $1 from Dell
  (except you have to buy in increments of 20 GB)
  - => 1 TB costs $1K, 1 PB costs $1M
- In 3 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!
  – What else?

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
  – 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-
      reordable instructions!
  – 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…)
    • this will change mostly 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
- Example: 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?

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
    - Why more than 2?
      - 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 the kernel mode
  - What happens if user mode executes a protected instruction?

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
    - Why more than 2?
      - 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 the kernel mode
  - What happens if user mode executes a protected instruction?

Crossing protection boundaries

- So how do user programs do something privileged?
  - Example: how can you write to a disk if you can’t do 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

- Netscape: read()
  - Trap to kernel mode; save app state
  - Find read() handler in vector table
  - 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?
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

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 x86 'int' instruction
    – e.g., a page fault, 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 async 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 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