CSE 451: Operating Systems
Winter 2004

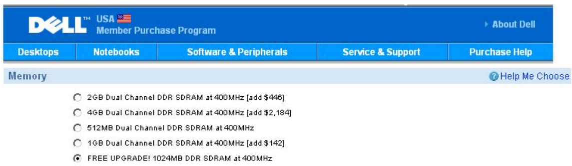
Module 2
Architectural Support for Operating Systems

Brian Bershad
bershad@cs.washington.edu
562 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

- Primary memory capacity
  - same story, same reason (Moore's Law)
    - 1982: 500KB VAX memory == $50K.
    - 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!
    - 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

- What are some of the implications of these hardware trends?
  - What was important yesterday may not be important today
    - We used to count instructions.
    - Then we counted memory references.
    - Today, we count critical security updates.
  - System interfaces matter more than the system itself
    - Linux vs Unix
  - Hardware advances may stall behind reluctant software
    - 64 bits "just around the corner"
  - Sometimes it may be better to just fake it
    - Virtual Machines

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
  - 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

Abstract Model of a Computer

- A computer system consists of two principle architectural elements
  - State (memory, registers)
    - Stores the values in use by a program
    - Example values:
      - The program text
      - The current PC
      - The value of register 7
      - The contents of the program's memory at location 104
    - For any system, see manual.
  - State changer (CPU)
    - Enables the values in use by a program to be changed
    - Most state changes occur as the result of instructions
      - Semantics defined precisely by the ISA
      - (again, see manual)
- The rate at which *state changes* is NOT generally defined by any interface
  - That is, programs generally have no guarantees
- And here in lies the leverage!

OS in a Nutshell

- The OS provides each program with the illusion of running on some abstract machine having
  - State that changes at some rate
  - Think what happens when you singlestep within the debugger.
    - There's a whole lot more than one instruction firing
- The hardware provides the OS with the services (instructions, registers, tables, indirections) it needs in order to allow the OS to convincingly implement the machine presented to the application
  - Essentially, program execution is cooperatively managed by the hardware and the OS.
    - The OS lets the hardware take over when things need to run fast.
    - The hardware gives over to the OS when things need to run correctly.
  - Architectural/OS evolution over the last 40 years essentially oriented towards understanding these transition points.

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)
    - why?
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
    - why?
  - manipulate special 'mode bits'
    - interrupt priority level
    - why?
  - halt instruction
    - why?

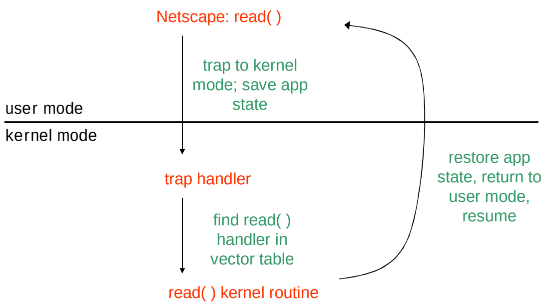
OS protection

- 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?

Crossing protection boundaries

- So how do user programs do something privileged?
  - e.g., 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)
  - Transfer to requested service
  - must be a way to return to user mode once done

A kernel crossing illustrated

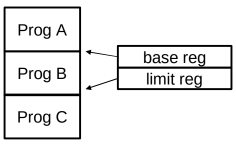


System call issues

- 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?



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

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 asynch 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
- Can be sugar-coated, but cannot be totally abstracted away
- Despite "easy to use" interfaces, remains a huge intellectual challenge
  - Unlike vulnerabilities due to buffer overruns, which are just sloppy programming

Example

```
int x = 0;

Count_up()
{
  x = x + 1;  /* load r0, x
                 r0 = r0 + 1
                 store r0, x */
}
```

```
int x = 0;

Count_up()
{
  x = x + 1;      /* load r0, x
}                    r0 = r0 + 1
                     store r0, x */
```