CSE451 Operating Systems
Winter 2009

Module 2
Architectural Support

Mark Zbikowski
Gary Kimura
Today

• Quick look at hardware trends
• What special hardware support is there for an OS?
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
  - 2008: Intel Quad-Core 2.66GHz == $900
    er... make that $700
• Primary memory capacity
  – same story, same reason (Moore’s Law)
    • 1972: 1MB = $1,000,000
    • 1982: 4 MB for DECSYSTEM 20 - $60,000
    • 2009: 2 GB for Dell Inspiron - $75
• 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!

• Disk Performance has NOT kept up
  – Speed in 1983: 500KB/sec
  – Speed in 2009: ~40MB/sec (less than 90x!)
• Optical network 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?

- **Register**: 1
- **On Chip Cache**: 2
- **On Board Cache**: 10
- **Memory**: 100
- **Disk**: $10^6$
- **Tape/Optical Robot**: $10^9$
- **Andromeda**: 2,000 Years
- **Pluto**: 2 Years
- **Olympia**: 1.5 hr
- **This Building**: 10 min
- **This Room**: 1 min
- **My Head**: 1 min

© 2004 Jim Gray, Microsoft Corporation
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…)
    • changing rapidly due to 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
  – disabling hardware interrupts
  – protected modes of execution (kernel vs. user)
  – protected and privileged instructions
  – system calls (and software interrupts)
Privileged 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 (registers, 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

App: ReadFile( Handle, Buffer, Count, &BytesRead, Overlapped )

User mode

kernel mode

trap to kernel mode; save app state

trap handler

find read( ) handler in vector table

NtReadFile( ) 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?

```
Prog A
Prog B
Prog C
```

```
<table>
<thead>
<tr>
<th>base reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>limit reg</td>
</tr>
</tbody>
</table>
```

base and limit registers are loaded by OS before starting program
More sophisticated memory protection

- coming later in the course
- virtual memory
  - paging, segmentation
  - 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
Exceptions

- Hardware must detect special conditions: page fault, write to a read-only page, overflow, trace trap, odd address trap, privileged instruction trap, syscall...
- Must transfer control to handler within the OS
- Hardware must save state on fault (PC, etc) so that the faulting process can be restarted afterwards
- Modern operating systems use VM traps for many functions: debugging, distributed VM, garbage collection, copy-on-write...
- Exceptions are a performance optimization, i.e., conditions could be detected by inserting extra instructions in the code (at high cost)
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 specified by interrupt signal
device interrupts

CPU stops current operation, switches to
kernel mode, and saves current PC and
other state on kernel stack

CPU fetches proper vector from
vector table and branches to that
address (to routine to handle
interrupt)

interrupt routine examines device database
and performs action required by interrupt

handler completes operation, restores saved
(interrupted state) and returns to user mode
(or calls scheduler to switch to another
program)
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
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?
Next Time

• We now know what the hardware gives us to use, so

• How do we conceptually organize an OS to put it all together?