CSE 451: Operating Systems
Spring 2009

Architectural Support for Operating Systems

Steve Gribble
Administrivia

• Reminders:
  – sign up for class mailing list!
  – homework #1 is out, due on Monday
  – project #0 is out, due in 9 days (Apr 10)
    • project #0 should be done solo
  – other projects will be done in teams of 2
    • Start shopping for your project partner
  – no class on Friday

• Office hours are posted
  – Ryan will be holding his this week
  – Steve, Sean will start next week
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

• Current generation – everything is multicore:
  – UltraSPARC T2 (Sun): 8 cores, 64 threads
  – Intel “Nehalem”: 8 cores, 16 threads
• Primary memory capacity
  – same story, same reason (Moore’s Law)
  • 1978: 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 CPU performance

• Disk capacity since 1990
  – doubling every 12 months
  – 100% improvement each year
  – factor of 1000 every decade
  – 10x as fast as processor performance!
  – Today: 1TB disk for $150
Newly arrived, and coming soon…

• **Solid state storage (SSD)**
  – promises 10,000 - 100,000 random IOs per second
  – 700 MB/s transfer rates
  – still costly, but quickly riding Moore’s law
    • $5-10 per GB, compared to hard drives $0.10 per GB

• **Phase-change memory (PRAM)**
  – promises speed of DRAM, but non-volatile
  – still experimental, though early product shipping
• 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
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
  – 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)
  – 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?

<table>
<thead>
<tr>
<th>Prog A</th>
<th>base reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog B</td>
<td>limit reg</td>
</tr>
<tr>
<td>Prog C</td>
<td></td>
</tr>
</tbody>
</table>

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

• 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
  – how does the kernel exchange data with an I/O device?
    • programmed I/O (PIO)
    • direct memory access (DMA)
Asynchronous I/O

• Interrupts are basis for asynchronous I/O
  – device performs an operation asynchronous 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

• 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 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
    • doesn’t work so well on multi-processor machines
  – 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
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.