CS 537
Lecture 2
Computer Architecture and Operating Systems

Michael Swift

Administrivia

- First reading assignment is up on web

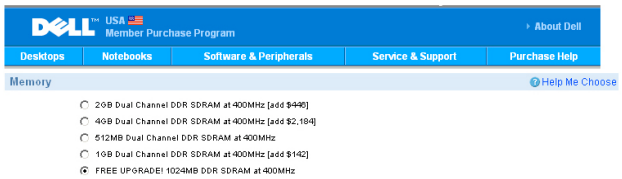
What you should learn

- How do architectural trends impact operating systems?
- How does architecture support OS functionality?

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)
    - 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 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?

Archive The New York Times

HOME | HELP | SEARCH | Go to Advanced Search/Archive | Past 30 Days | GO TO MEMBER CENTER | LOG OUT | Welcome, lazowska

This page is print-ready, and this article will remain available for 90 days. Instructions for Saving | About this Service | Purchase History

May 26, 2003, Monday

BUSINESS/FINANCIAL DESK

TECHNOLOGY; From PlayStation to Supercomputer for $50,000

By JOHN MARKOFF (NYT) 913 words

As perhaps the clearest evidence yet of the computing power of sophisticated but inexpensive video-game consoles, the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign has assembled a supercomputer from an army of Sony PlayStation 2's.

The resulting system, with components purchased at retail prices, cost a little more than $50,000. The center's researchers believe the system may be capable of a half trillion operations a second, well within the definition of supercomputer, although it may not rank among the world's 500 fastest supercomputers.

Perhaps the most striking aspect of the project, which uses the open source Linux operating system, is that the only hardware engineering involved was placing 70 of the individual game machines in a rack and plugging them together with a high-speed Hewlett-Packard network switch. The center's scientists bought 100 machines, but are holding 30 in reserve, possibly for high-resolution display application.

"It took a lot of time because you have to cut all of these things out of the plastic packaging," said Craig Steffen, a senior research scientist at the center, who is one of four scientists working part time on the project.

The scientists are taking advantage of a standard component of the Sony video-game console that was originally intended to move and transform pixels rapidly on a television screen to produce lifelike graphics. The chip is not the PlayStation 2's MIPS microprocessor, but rather a graphics co-processor known as the Emotion Engine. That custom designed silicon chip is capable of producing up to 6.5 billion mathematical operations a second.

How do arch. trends impact OS design?

- Human:computer ratio
  - Batch - time sharing - personal computers - embedded / pervasive computing
  - Single job - time shared - internetworked
- Programmer:processor cost ratio
  - assembly to C to Java to Perl languages
  - command line to GUI to pen / voice interfaces
- Networking
  - Isolation to dialup to LAN to WAN
    - OS must devote more effort to communications
  - Disconnected to wired to wireless
    - OS must manage connectivity more
  - Isolated to shared to attacked
    - OS must provide more security / protection

More trends

- Disk size: data size
  - Deleting is not as important
  - Extra space is available for metadata
  - Finding data is as important as storing it
- Disk speed: memory speed
  - Important apps don't page

Processor Trends

- CPU performance improved 52% per year from 1986-2002
- From 2002-2006, performance improved less than 20% per year
- Modern trend: multi-core, multi-threading
  - Pentium 4: hyperthreading
  - Core II Duo: 2 cores
  - Sun Niagara II: 8 cores, 8 threads per core
- Single thread performance has stopped growing
- All future performance gains from compilers, OS, multithreading

Low-level architecture support for OS

- 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
  - e.g.: virtual machines arrived on PCs 25 years after they arrived on mainframes because X86 processors lacked support for virtualization

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?

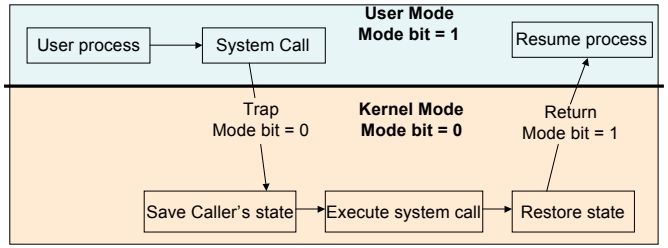
OS protection

- 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)
  - must be a way to return to user mode once done

A kernel crossing illustrated



System call details

- How does the kernel know which system call?
  - In a register
- Where are the parameters?
  - in a register
  - on the stack
  - in a memory block

```
<open>:         push   %ebx
<open+1>:       mov    0x10(%esp),%edx
<open+5>:       mov    0xc(%esp),%ecx
<open+9>:       mov    0x8(%esp),%ebx
<open+13>:      mov    $0x5,%eax
<open+18>:      int    $0x80
<open+20>:      pop    %ebx
<open+21>:      cmp    $0xfffff001,%eax
<open+26>:      jae    0x2a189d <open+29>
<open+28>:      ret
```

```
# system call handler stub
ENTRY(system_call)
    pushl %eax              # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
    syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,EAX(%esp)     # store the return value
```

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?

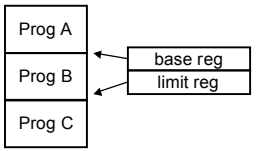
What functions are system calls?

- Process control
  - Create process, allocate memory
- File management
  - Create, read, delete file
- Device management
  - Open device, read/write device, mount device
- Information maintenance
  - Get time, get system data/parameters
- Communications
  - Create/delete channel, send/receive message

- Programmers generally do **not** use system calls directly
  - They use runtime libraies (e.g. Java, C)
  - Why?

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
- 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