OS-Related Hardware & Software
Lecture 2 Overview

OS-Related Hardware & Software
- complications in real systems
- brief introduction to memory protection, relocation, virtual memory and I/O
Key Points From Class 1

Why do we need an interrupt mechanism?
Why do we need a timer device?
Why do we need privileged instructions?
Why are system calls different to procedure calls?
How are system calls different to interrupts?
Why is memory protection necessary?
Real World Complexity

Our simple hardware model is valid, but real systems have more complexity

- Pipelined CPUs
- Superscalar CPUs
- Multi-level memory hierarchies
- Virtual memory addressing
- Complexity of devices and buses
Pipelined CPUs

Execution of current instruction is performed in parallel with decode of next instruction and fetch of the one after that.
Superscalar CPUs

Fetch unit → Decode unit → Holding buffer → Execute unit

Fetch unit → Decode unit → Holding buffer → Execute unit

Fetch unit → Decode unit → Holding buffer → Execute unit
What does this mean for the OS?

Pipelined CPUs
- more complexity in capturing the state of a running application
- more expensive to suspend and resume applications

Superscalar CPUs
- even more complexity in capturing state of a running application
- even more expensive to suspend and resume applications
- but we have support from hardware (precise interrupts)

More details, but its fundamentally the same task

The BLITZ CPU is not pipelined or superscalar and it has precise interrupts
The Memory Hierarchy

2GHz processor has 0.5 ns clock cycle
Data/instruction cache access time is \(~0.5\)ns
   This is where the CPU looks first!
   Cache is fast but small and can’t hold everything

Main memory access time is \(~100\) ns
   It’s slow but it can hold everything (8 GB+)

Secondary storage access time is \(~10\) ms
   Tbytes in size, but 20 million times slower access time!
## Typical Latency Numbers

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 ns</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000 ns</td>
</tr>
<tr>
<td>Send 1K bytes over 1 Gbps network</td>
<td>10,000 ns</td>
</tr>
<tr>
<td>Read 4K randomly from SSD*</td>
<td>150,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000 ns</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from SSD*</td>
<td>1,000,000 ns</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000 ns</td>
</tr>
<tr>
<td>Send packet USA-&gt;Europe-&gt;USA</td>
<td>150,000,000 ns</td>
</tr>
</tbody>
</table>
Managing the Memory Hierarchy

Data is copied from lower levels to higher levels
- it is temporarily resident in higher levels

Space is allocated in multi-byte units at each level
- cache lines (64 bytes on x86)
- memory pages (often 4KB or 8KB)
- disk blocks/sectors

Data access attempts cause cache lines to be loaded automatically
- Placement and replacement policy is fixed by hardware
Managing the Memory Hierarchy

Operating Systems have some control
- Lay out data carefully in memory
- Be careful about the order in which they access data items
- Use instructions to “flush” the cache when necessary

Movement of data between lower levels is under direct control of the OS
- Virtual memory page faults
- File system calls
Other Memory-Related Issues

How do you protect one application’s area of memory from that of another application?

How do you *relocate* an application in memory?

- *How does the programmer know where the program will ultimately reside in memory?*
Memory Protection & Relocation

Memory protection – the basic idea

Virtual vs physical addresses

- Address range in each application starts at 0
- Applications use virtual addresses, but hardware and OS translate them automatically into physical addresses

- A simple way to do this via a base register:
  - Get the CPU to interpret address indirectly via a base register
  - Base register holds starting, or base address
  - Add base value to address to get a real address before main memory is accessed
  - Relocation is simple: change the base register value
Paged virtual memory

The base register idea doesn’t work well in practice

Paged virtual memory is a similar concept, but ...

- Supports non-contiguous allocation of memory
- Allows allocated memory to grow and shrink dynamically
- Requires hardware support for page-based address translation
  * Sometimes referred to as a memory management unit (MMU) or a translation lookaside buffer (TLB)

Much more on this later ...
Device Input/Output
Simple Model of I/O Devices
More Realistic
Device Terminology

Device (mechanical hardware)
Device controller (electrical hardware)
Device driver (software)
Device Controllers

The Device vs. its Controller

Some duties of a device controller:
- Interface between CPU and the Device
- Start/Stop device activity
- Convert serial bit stream to a block of bytes
- Deal with error detection/correction
- Move data to/from main memory

Some controllers may handle several (similar) devices
Software’s View of Devices

Hardware supports I/O ports or memory mapped I/O for accessing device controller registers and buffers.
I/O Ports

Each port has a separate number.

CPU has special I/O instructions

\[
\text{in} \quad r4,3 \\
\text{out} \quad 3,r4
\]

Port numbers form an “address space”... separate from main memory

Contrast with

\[
\text{load} \quad r4,3 \\
\text{store} \quad 3,r4
\]
Memory-Mapped I/O

One address space for
main memory
I/O devices

CPU has no special instructions

load    r4,addr
store   addr,r4

I/O devices are “mapped” into
very high addresses
## Wide Range of Device Speeds

<table>
<thead>
<tr>
<th>Device</th>
<th>Data rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>10 bytes/sec</td>
</tr>
<tr>
<td>Mouse</td>
<td>100 bytes/sec</td>
</tr>
<tr>
<td>56K modem</td>
<td>7 KB/sec</td>
</tr>
<tr>
<td>Telephone channel</td>
<td>8 KB/sec</td>
</tr>
<tr>
<td>Dual ISDN lines</td>
<td>16 KB/sec</td>
</tr>
<tr>
<td>Laser printer</td>
<td>100 KB/sec</td>
</tr>
<tr>
<td>Scanner</td>
<td>400 KB/sec</td>
</tr>
<tr>
<td>Classic Ethernet</td>
<td>1.25 MB/sec</td>
</tr>
<tr>
<td>USB (Universal Serial Bus)</td>
<td>1.5 MB/sec</td>
</tr>
<tr>
<td>Digital camcorder</td>
<td>4 MB/sec</td>
</tr>
<tr>
<td>IDE disk</td>
<td>5 MB/sec</td>
</tr>
<tr>
<td>40x CD-ROM</td>
<td>6 MB/sec</td>
</tr>
<tr>
<td>Fast Ethernet</td>
<td>12.5 MB/sec</td>
</tr>
<tr>
<td>ISA bus</td>
<td>16.7 MB/sec</td>
</tr>
<tr>
<td>EIDE (ATA-2) disk</td>
<td>16.7 MB/sec</td>
</tr>
<tr>
<td>FireWire (IEEE 1394)</td>
<td>50 MB/sec</td>
</tr>
<tr>
<td>XGA Monitor</td>
<td>60 MB/sec</td>
</tr>
<tr>
<td>SONET OC-12 network</td>
<td>78 MB/sec</td>
</tr>
<tr>
<td>SCSI Ultra 2 disk</td>
<td>80 MB/sec</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>125 MB/sec</td>
</tr>
<tr>
<td>Ultrium tape</td>
<td>320 MB/sec</td>
</tr>
<tr>
<td>PCI bus</td>
<td>528 MB/sec</td>
</tr>
<tr>
<td>Sun Gigaplane XB backplane</td>
<td>20 GB/sec</td>
</tr>
</tbody>
</table>
Hardware Performance Challenges

How to prevent slow devices from slowing down CPU’s access to memory due to bus contention
- Only one thing can use the bus at a time
- Moving a single character from the keyboard can prevent millions of memory accesses by the CPU

The challenge: how to perform I/O without interfering with memory performance
A hardware solution: Dual Bus Architecture

CPU
Memory
I/O

Bus

CPU reads and writes of memory go over this high-bandwidth bus

CPU
Memory
I/O

This memory port is to allow I/O devices access to memory
Pentium Bus Architecture
Software Performance Challenges

For slow devices: how to prevent CPU throughput from being limited by I/O device speed
  - Why would slow devices affect the CPU?
  - CPU must wait for devices to be ready for next command

For fast devices: how to prevent I/O throughput from being limited by CPU speed
  - When devices can move data faster than the CPU

We must also achieve good utilization of CPU and I/O devices and meet the real-time requirements of devices
Programmed I/O

Steps in printing a string
Programmed I/O

Example: Writing a string to a serial output or printing a string

```
CopyFromUser(virtAddr, kernelBuffer, byteCount)
for i = 0 to byteCount-1
    while *serialStatusReg != READY
        endwhile
    *serialDataReg = kernelBuffer[i]
endFor
return
```

Called “Busy Waiting” or “Polling”
Problem: CPU is continually busy working on I/O!
    If I/O device is slow, CPU is idle most of the time
Interrupt-Driven I/O

Getting the I/O started:

```
CopyFromUser(virtAddr, kernelBuffer, byteCount)
EnableInterrupts()
while *serialStatusReg != READY
endWhile
*serialDataReg = kernelBuffer[0]
Sleep()
```

The Interrupt Handler:

```
if i == byteCount
    Wake up the requesting process
else
    *serialDataReg = kernelBuffer[i]
i = i + 1
endIf
Return from interrupt
```
Hardware Support For Interrupts

Connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires.
Interrupt Driven I/O Problem

Problem:
- CPU is still involved in every data transfer
- Interrupt handling overhead is high
- Overhead cost is not amortized over much data
- Overhead is too high for fast devices
  - Gbps networks
  - Disk drives
Direct Memory Access (DMA)

Data transferred from device straight to/from memory

CPU not involved

*The DMA controller:*
- Does the work of moving the data
- CPU sets up the DMA controller ("programs it")
- CPU continues
- The DMA controller moves the bytes
Sending Data Using DMA

Getting the I/O started:

CopyFromUser(virtAddr, kernelBuffer, byteCount)
Set up DMA controller
Sleep()

The Interrupt Handler:

Acknowledge interrupt from DMA controller
Wake up the requesting process
Return from interrupt
Direct Memory Access (DMA)
Direct Memory Access (DMA)

Cycle Stealing
- DMA Controller acquires control of bus
- Transfers a single byte (or word)
- Releases the bus
- The CPU is slowed down due to bus contention

Burst Mode
- DMA Controller acquires control of bus
- Transfers all the data
- Releases the bus
- The CPU operation is temporarily suspended
Program-Device Interaction

Devices vs device controllers vs device drivers
- Device drivers are part of the OS (ie. software)
- Programs call the OS which calls the device driver
- Device drivers interact with device controllers using special IO instructions or by reading/writing controller registers that appear as memory locations
- Device controllers are hardware that communicate with device drivers via interrupts
Device to Program Interaction

Diagram showing the interaction between CPU, interrupt controller, disk controller, and disk drive.
Types of Interrupt

**Timer interrupts**
- Allows OS to regain control of the CPU
- One way to keep track of time

**I/O interrupts**
- Keyboard, mouse, disks, network, etc...

**Program generated (traps & faults)**
- Address translation faults (page fault, TLB miss)
- Programming errors: seg. faults, divide by zero, etc.
- System calls like read(), write(), gettimeofday()
## Interrupts, Traps, Faults and Exceptions

<table>
<thead>
<tr>
<th></th>
<th>Synchronous?</th>
<th>Intentional?</th>
<th>Error?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interrupt</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Trap</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Fault</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Exception</strong></td>
<td>Yes</td>
<td>Maybe</td>
<td>Sometimes</td>
</tr>
</tbody>
</table>
System calls

*System calls* are the mechanism by which programs invoke the OS

Implemented via a TRAP instruction

Example UNIX system calls:

- open(), read(), write(), close()
- kill(), signal()
- fork(), wait(), exec(), getpid()
- link(), unlink(), mount(), chdir()
- setuid(), getuid(), chown()
System calls

If all system calls cause a trap, how can the OS tell which one the application wants it to execute?

How can we pass parameters to the call and receive results?
System Call Implementation

User-level code

Process usercode
{ ... 
  read (file, buffer, n);
} ...

Procedure read(file, buff, n)
{ ... 
  read_syscall(file, buff, n)
} ...

_library code_

_read_syscall:
  LOAD r1, @SP+2
  LOAD r2, @SP+4
  LOAD r3, @SP+6
  TRAP
Communication Protocol Could use Stack or Dedicated Registers
Before Next Class

Do the reading for next week’s class
Finish project 1 – Introduction to BLITZ