The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.
Announcements

Project5 Cache Race Games night Monday, May 7th, 5pm
- Come, eat, drink, have fun and be merry!
- Location: B11 Kimball Hall

Prelim2: **Thursday**, May 3rd in evening
- Time and Location: **7:30pm sharp** in **Statler Auditorium**
- Old prelims are online in CMS

Project6: Malloc
- Design Doc due May 9th, bring design doc to mtg May 7-9
- Project due Tuesday, May 15th at 4:30pm
- **Will not be able to use slip days**

Lab Sections are Optional this week
- Ask Prelim2 or Project6 questions
Goals for Today

Computer System Organization

How does a processor interact with its environment?
  • I/O Overview

How to talk to device?
  • Programmed I/O or Memory-Mapped I/O

How to get events?
  • Polling or Interrupts

How to transfer lots of data?
  • Direct Memory Access (DMA)
Next Goal

How does a processor interact with its environment?
How does a processor interact with its environment?
Big Picture: Input/Output (I/O)

How does a processor interact with its environment?

Computer System Organization =
Memory +
Datapath +
Control +
Input +
Output
<table>
<thead>
<tr>
<th>Device</th>
<th>Behavior</th>
<th>Partner</th>
<th>Data Rate (b/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>Input</td>
<td>Human</td>
<td>100</td>
</tr>
<tr>
<td>Mouse</td>
<td>Input</td>
<td>Human</td>
<td>3.8k</td>
</tr>
<tr>
<td>Sound Input</td>
<td>Input</td>
<td>Machine</td>
<td>3M</td>
</tr>
<tr>
<td>Voice Output</td>
<td>Output</td>
<td>Human</td>
<td>264k</td>
</tr>
<tr>
<td>Sound Output</td>
<td>Output</td>
<td>Human</td>
<td>8M</td>
</tr>
<tr>
<td>Laser Printer</td>
<td>Output</td>
<td>Human</td>
<td>3.2M</td>
</tr>
<tr>
<td>Graphics Display</td>
<td>Output</td>
<td>Human</td>
<td>800M – 8G</td>
</tr>
<tr>
<td>Network/LAN</td>
<td>Input/Output</td>
<td>Machine</td>
<td>100M – 10G</td>
</tr>
<tr>
<td>Network/Wireless LAN</td>
<td>Input/Output</td>
<td>Machine</td>
<td>11 – 54M</td>
</tr>
<tr>
<td>Optical Disk</td>
<td>Storage</td>
<td>Machine</td>
<td>5 – 120M</td>
</tr>
<tr>
<td>Flash memory</td>
<td>Storage</td>
<td>Machine</td>
<td>32 – 200M</td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>Storage</td>
<td>Machine</td>
<td>800M – 3G</td>
</tr>
</tbody>
</table>
Attempt#1: All devices on one interconnect

Replace *all* devices as the interconnect changes
e.g. keyboard speed == main memory speed ?!
Decouple I/O devices from Interconnect

Enable smarter I/O interfaces

Unified Memory and I/O Interconnect

- Memory Controller
- I/O Controller
- I/O Controller
- I/O Controller
- I/O Controller

- Memory
- Display
- Disk
- Keyboard
- Network
Separate high-performance processor, memory, display interconnect from lower-performance interconnect.
Bus Parameters

**Width** = number of wires

**Transfer size** = data words per bus transaction

**Synchronous** (with a bus clock)

or **asynchronous** (no bus clock / “self clocking”)
Bus Types

Processor – Memory ("Front Side Bus". Also QPI)

- Short, fast, & wide
- Mostly fixed topology, designed as a “chipset”
  - CPU + Caches + Interconnect + Memory Controller

I/O and Peripheral busses (PCI, SCSI, USB, LPC, ...)

- Longer, slower, & narrower
- Flexible topology, multiple/varied connections
- Interoperability standards for devices
- Connect to processor-memory bus through a bridge
Separate high-performance processor, memory, display interconnect from lower-performance interconnect
## Example Interconnects

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Devics per channel</th>
<th>Channel Width</th>
<th>Data Rate (B/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firewire 800</td>
<td>External</td>
<td>63</td>
<td>4</td>
<td>100M</td>
</tr>
<tr>
<td>USB 2.0</td>
<td>External</td>
<td>127</td>
<td>2</td>
<td>60M</td>
</tr>
<tr>
<td>USB 3.0</td>
<td>External</td>
<td>127</td>
<td>2</td>
<td>625M</td>
</tr>
<tr>
<td>Parallel ATA</td>
<td>Internal</td>
<td>1</td>
<td>16</td>
<td>133M</td>
</tr>
<tr>
<td>Serial ATA (SATA)</td>
<td>Internal</td>
<td>1</td>
<td>4</td>
<td>300M</td>
</tr>
<tr>
<td>PCI 66MHz</td>
<td>Internal</td>
<td>1</td>
<td>32-64</td>
<td>533M</td>
</tr>
<tr>
<td>PCI Express v2.x</td>
<td>Internal</td>
<td>1</td>
<td>2-64</td>
<td>16G/dir</td>
</tr>
<tr>
<td>Hypertransport v2.x</td>
<td>Internal</td>
<td>1</td>
<td>2-64</td>
<td>25G/dir</td>
</tr>
<tr>
<td>QuickPath (QPI)</td>
<td>Internal</td>
<td>1</td>
<td>40</td>
<td>12G/dir</td>
</tr>
</tbody>
</table>
Interconnecting Components

Interconnects are (were?) busses

• parallel set of wires for data and control

• shared channel
  – multiple senders/receivers
  – everyone can see all bus transactions

• bus protocol: rules for using the bus wires

Alternative (and increasingly common):

• dedicated point-to-point channels

e.g. Intel Xeon

e.g. Intel Nehalem
Remove bridge as bottleneck with Point-to-point interconnects

E.g. Non-Uniform Memory Access (NUMA)
Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.
Next Goal

How does the processor interact with I/O devices?
I/O Device Driver Software Interface

Set of methods to write/read data to/from device and control device
Example: Linux Character Devices

// Open a toy " echo " character device
int fd = open("/dev/echo", O_RDWR);

// Write to the device
char write_buf[] = "Hello World!";
write(fd, write_buf, sizeof(write_buf));

// Read from the device
char read_buf[32];
read(fd, read_buf, sizeof(read_buf));

// Close the device
close(fd);

// Verify the result
assert(strcmp(write_buf, read_buf)==0);
I/O Device API

Typical I/O Device API

- a set of read-only or read/write registers

Command registers

- writing causes device to do something

Status registers

- reading indicates what device is doing, error codes, ...

Data registers

- Write: transfer data to a device
- Read: transfer data from a device

Every device uses this API
I/O Device API

Simple (old) example: AT Keyboard Device

8-bit Status:

<table>
<thead>
<tr>
<th>PE</th>
<th>TO</th>
<th>AUXB</th>
<th>LOCK</th>
<th>AL2</th>
<th>SYSF</th>
<th>IBS</th>
<th>OBS</th>
</tr>
</thead>
</table>

8-bit Command:

0xAA = “self test”
0xAE = “enable kbd”
0xED = “set LEDs”
...

8-bit Data:

scancode (when reading)
LED state (when writing) or ...
Communication Interface

Q: How does program OS code talk to device?

A: special instructions to talk over special busses

Programmed I/O

- `inb $a, 0x64`
- `outb $a, 0x60`

- Specifies: device, data, direction
- Protection: only allowed in kernel mode

*x86: $a implicit; also inw, outw, inh, outh, ...*

Interact with cmd, status, and data device registers directly

- kbd status register
- kbd data register

Kernel boundary crossing is expensive
Q: How does program-OS code talk to device?
A: Map registers into virtual address space

Memory-mapped I/O ← Faster. Less boundary crossing

- Accesses to certain addresses redirected to I/O devices
- Data goes over the memory bus
- Protection: via bits in pagetable entries
- OS+MMU+devices configure mappings
Device Drivers

Programmed I/O
Polling examples,
But mmap I/O more efficient

char read_kbd()
{
do {
    sleep();
    status = inb(0x64);
} while(!(status & 1));
return inb(0x60);
}

Memory Mapped I/O

struct kbd {
    char status, pad[3];
    char data, pad[3];
};

kbd *k = mmap(...);

char read_kbd()
{
do {
    sleep();
    status = k->status;
} while(!(status & 1));
return k->data;
}
Comparing Programmed I/O vs Memory Mapped I/O

Programmed I/O

• Requires special instructions
• Can require dedicated hardware interface to devices
• Protection enforced via kernel mode access to instructions
• Virtualization can be difficult

Memory-Mapped I/O

• Re-uses standard load/store instructions
• Re-uses standard memory hardware interface
• Protection enforced with normal memory protection scheme
• Virtualization enabled with normal memory virtualization scheme
Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.
Next Goal

How does the processor know device is ready/done?
Q: How does program learn device is ready/done?
Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient.
Next Goal

How do we transfer a *lot* of data *efficiently*?
I/O Data Transfer

How to talk to device?
  • Programmed I/O or Memory-Mapped I/O

How to get events?
  • Polling or Interrupts

How to transfer lots of data?

```c
#include <stdio.h>

void transfer_data(int n)
{
    int i;
    i = 0;
    while (!(disk->status & 1))
    {
        for (i = 0; i < n; i++)
        {
            buf[i] = disk->data;
        }
    }
}
```

Very, Very, Expensive
Programmed I/O xfer: Device

for (i = 1 .. n)
  • CPU issues read request
  • Device puts data on bus & CPU reads into registers
  • CPU writes data to memory
  • **Not** efficient

Read from Disk
Write to Memory
*Everything* interrupts CPU
*Wastes* CPU
Q: How to transfer lots of data efficiently?
A: Have device access memory directly

Direct memory access (DMA)

• 1) OS provides starting address, length
• 2) controller (or device) transfers data autonomously
• 3) Interrupt on completion / error
DMA: Direct Memory Access

Programmed I/O xfer: Device ↔ CPU ↔ RAM

for (i = 1 .. n)

- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory
DMA: Direct Memory Access

Programmed I/O xfer: Device $\leftrightarrow$ CPU $\leftrightarrow$ RAM
for (i = 1 .. n)
- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory

DMA xfer: Device $\leftrightarrow$ RAM
- CPU sets up DMA request
- for (i = 1 ... n) Device puts data on bus & RAM accepts it
- Device interrupts CPU after done
DMA Example

DMA example: reading from audio (mic) input
  • DMA engine on audio device... or I/O controller ... or ...

```c
int dma_size = 4*PAGE_SIZE;
int *buf = alloc_dma(dma_size);
...
dev->mic_dma_baseaddr = (int)buf;
dev->mic_dma_count = dma_len;
dev->cmd = DEV_MIC_INPUT | DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```
DMA Issues (1): Addressing

Issue #1: DMA meets Virtual Memory

RAM: physical addresses

Programs: virtual addresses
DMA Example

DMA example: reading from audio (mic) input
  • DMA engine on audio device... or I/O controller ... or

    int dma_size = 4*PAGE_SIZE;
    void *buf = alloc_dma(dma_size);
    ...
    dev->mic_dma_baseaddr = virt_to_phys(buf);
    dev->mic_dma_count = dma_len;
    dev->cmd = DEV_MIC_INPUT | DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
DMA Issues (1): Addressing

Issue #1: DMA meets Virtual Memory

RAM: physical addresses
Programs: virtual addresses
DMA Issues (2): Virtual Mem

Issue #2: DMA meets *Paged* Virtual Memory

DMA destination page may get swapped out.
Issue #4: DMA meets Caching

DMA-related data could be cached in L1/L2

- DMA to Mem: cache is now stale
- DMA from Mem: dev gets stale data
DMA Issues (4): Caches

Issue #4: DMA meets Caching

DMA-related data could be cached in L1/L2

- DMA to Mem: cache is now stale
- DMA from Mem: dev gets stale data

Solution 2: (hardware coherence aka snooping)

- cache listens on bus, and conspires with RAM
- DMA to Mem: invalidate/update data seen on bus
- DMA from mem: cache services request if possible, otherwise RAM services
Takeaways

Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies.

Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores.

Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient.

Modern systems combine memory-mapped I/O, interrupt-based I/O, and direct-memory access to create sophisticated I/O device subsystems.
I/O Summary

How to talk to device?
Programmed I/O or Memory-Mapped I/O

How to get events?
Polling or Interrupts

How to transfer lots of data?
DMA