I/O

Prof. Kavita Bala and Prof. Hakim Weatherspoon
CS 3410, Spring 2014
Computer Science
Cornell University

See: *Online* P&H Chapter 6.9 (5th edition):
Also, *Online* P&H Chapter 6.5-6 (4th edition)
Administrivia

Project 3 submit “souped up” bot to CMS

Project 3 Cache Race Games night Monday, May 5th, 5pm
  • Come, eat, drink, have fun and be merry!
  • Location: B11 Kimball Hall

Prelim 2: *Today, Thursday*, May nd in evening
  • Time: We will start at **7:30pm sharp**, so come early
  • Two Locations: OLN155 and URSG01
    • If NetID *begins* with ‘a’ to ‘g’, then go to OLN155 (Olin Hall rm 155)
    • If NetID *begins* with ‘h’ to ‘z’, then go to URSG01 (Uris Hall rm G01)

Project 4:
  • Design Doc due May 7th, bring design doc to mtg May 5-7
  • Demos: May 13 and 14
  • *Will not be able to use slip days*
Administrivia

Next 2 weeks

• Prelim2 *Today*, Thu May 1\textsuperscript{st} : 7:30-9:30
  – Olin 155: Netid [a-g]*
  – Uris G01: Netid [h-z]*

• Proj3 tournament: Mon May 5 5pm-7pm (Pizza!)
  – Location: Kimball B11

• Proj4 design doc meetings May 5-7 (doc ready for mtg)

Final Project for class

• Proj4 due Wed May 14
• Proj4 demos: May 13 and 14
• Proj 4 release: in labs this week
• Remember: No slip days for PA4
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?
Big Picture: Input/Output (I/O)
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
## I/O Devices Enables Interacting with Environment

<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 ?!
Attempt#2: I/O Controllers

Decouple I/O devices from Interconnect
Enable smarter I/O interfaces
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>Devices 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
Attempts to I/O Controllers: Bridge

NUMA

Remove bridge as bottleneck with Point-to-point interconnects

E.g. Non-Uniform Memory Access (NUMA)
Takeaways

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:  PE  TO  AUXB  LOCK  AL2  SYSF  IBS  OBS

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

Interact with cmd, status, and data device registers directly

- Protection: only allowed in kernel mode

* Protection: only allowed in kernel mode

Kernel boundary crossing is expensive

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

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
Memory-Mapped I/O

Virtual Address Space
0x0000 0000
0xFFFF FFFF

Physical Address Space
0x0000 0000
0x00FF FFFF

I/O Controller
Display

I/O Controller
Disk

I/O Controller
Keyboard

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

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

Memory Mapped I/O

```c
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
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.
Next Goal

How does the processor know device is ready/done?
Communication Method

Q: How does program learn device is ready/done?
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
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
    disk->cmd = READ_4K_SECTOR;
    disk->data = 12;
    while (!(disk->status & 1)) {
        for (i = 0..4k)
            buf[i] = disk->data;
    }
```

- Very, Very, Expensive
I/O Data Transfer

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
• **Not** efficient

Everything interrupts CPU
Wastes CPU
I/O Data Transfer

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 \( i = 1 \ldots n \)

- 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 ↔ CPU ↔ RAM

for \(i = 1 \ldots n\)

- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory

**DMA xfer:** Device ↔ RAM

- CPU sets up DMA request
- for \(i = 1 \ldots n\)
  - 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 ...

```c
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
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
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