I/O

Hakim Weatherspoon
CS 3410, Spring 2013
Computer Science
Cornell University

See: P&H Chapter 6.5-6
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?

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
Attempt #3: I/O Controllers + Bridge

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
Attempt#3: I/O Controllers + 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>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
Attempt#4: 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

```c
// 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 ...
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

*Interact with cmd, status, and data device registers directly*

- `kbd status register`
- `kbd data register`

*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
## Device Drivers

### Programmed I/O

Polling examples,

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

But `mmap` I/O more efficient

### 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;
}
```

Polling examples,

But `mmap` I/O more efficient
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?

A: **Polling**: Periodically check I/O status register

- If device ready, do operation
- If device done, ...
- If error, take action

Pro? Con?

- Predictable timing & inexpensive
- But: wastes CPU cycles if nothing to do
- Efficient if there is always work to do (e.g. 10Gbps NIC)

Common in small, cheap, or real-time embedded systems

Sometimes for very active devices too...

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

    return inb(0x60);
}
```
Communication Method

Q: How does program learn device is ready/done?

A: **Interrupts**: Device sends interrupt to CPU
   - Cause register identifies the interrupting device
   - interrupt handler examines device, decides what to do

Priority interrupts
   - Urgent events can interrupt lower-priority interrupt handling
   - OS can disable defer interrupts

Pro? Con?
   - More efficient: only interrupt when device ready/done
   - Less efficient: more expensive since save CPU context
     - CPU context: PC, SP, registers, etc
   - Con: unpredictable b/c event arrival depends on other devices’ activity
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)) {}  // Very Expensive
for (i = 0..4k)
    buf[i] = disk->data;
```
I/O Data Transfer

Programmed I/O xfer: Device

\[
\text{for } (i = 1 \ldots n)
\]

• CPU issues read request
• Device puts data on bus & CPU reads into registers
• CPU writes data to memory
• \textit{Not} efficient

\textit{Everything} interrupts CPU
\textit{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

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 $\leftrightarrow$ Device

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 ($i = 1 \ldots n$)
  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;
device->mic_dma_count = dma_len;
device->cmd = DEV_MIC_INPUT | DEV_INTERRUPT_ENABLE | DEV_DMA_ENABLE;
```
Issue #1: DMA meets Virtual Memory

RAM: physical addresses
Programs: virtual addresses

Solution: DMA uses physical addresses
- OS uses physical address when setting up DMA
- OS allocates contiguous physical pages for DMA
- Or: OS splits xfer into page-sized chunks
  (many devices support DMA “chains” for this reason)
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

Solution 2: DMA uses virtual addresses

• OS sets up mappings on a mini-TLB
DMA Issues (2): Virtual Mem

Issue #2: DMA meets *Paged* Virtual Memory

DMA destination page may get swapped out

Solution: **Pin** the page before initiating DMA

Alternate solution: **Bounce Buffer**

- DMA to a pinned kernel page, then memcpy elsewhere
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: (software enforced coherence)

- OS flushes some/all cache before DMA begins
- Or: don't touch pages during DMA
- Or: mark pages as uncacheable in page table entries
  - (needed for Memory Mapped I/O too!)
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
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
No Lab section this week

Project3 extended to tonight Tuesday, April 23\textsuperscript{rd}

Project3 Cache Race Games night Friday, Apr 26\textsuperscript{th}, 5pm
- Come, eat, drink, have fun and be merry!
- Location: B17 Upson Hall

Prelim3: \textbf{Thursday}, April 25\textsuperscript{th} in evening
- Time: We will start at \textbf{7:30pm sharp}, so come early
- Two Locations: PHL101 and UPSB17
  - If NetID begins with ‘a’ to ‘j’, then go to PHL101 (Phillips Hall rm 101)
  - If NetID begins with ‘k’ to ‘z’, then go to UPSB17 (Upson Hall rm B17)
- Prelim Review: Today, Tue, at 6:00pm in Phillips Hall rm 101

Project4: Final project out next week
- Demos: May 14 and 15
- \textbf{Will not be able to use slip days}
Administrivia

Next two weeks

• Week 13 (Apr 22): Project3 Games Night and Prelim3
• Week 14 (Apr 29): Project4 handout

Final Project for class

• Week 15 (May 6): Project4 design doc due
• Week 16 (May 13): Project4 due by May 15th