I/O

Anne Bracy
CS 3410
Computer Science
Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.
How does a processor interact with its environment?

Computer System = Memory + Datapath + Control + Input + Output
# I/O Devices Enables Interacting w/ 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>
Round 1: All devices on one interconnect

Replace all devices as the interconnect changes

E.g. keyboard speed == main memory speed ?!
Round 2: I/O Controllers

Decouple I/O devices from Interconnect
Enable smarter I/O interfaces
Round 3: I/O Controllers + Bridge

Separate high-performance processor, memory, display interconnect from lower-performance interconnect

- **Core0**
  - Cache
  - Memory Controller
  - Memory
  - Display

- **Core1**
  - Cache
  - Memory Controller
  - Memory
  - Display

**High Performance Interconnect**

**Lower Performance Legacy Interconnect**

**I/O Controllers**

- Disk
- Keyboard
- Network
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
# 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>
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 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
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

Physical Address Space

agreed-upon locations for communication

I/O Controller
I/O Controller
I/O Controller
I/O Controller

Display
Disk
Keyboard
Network

Less-favored alternative = Programmed I/O:
- Syscall instructions that communicate with I/O
- Communicate via special device registers
Programmed I/O

Both 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;
}
```
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?

disk->cmd = READ_4K_SECTOR;
disk->data = 12;
while (!(disk->status & 1) { }
for (i = 0..4k)
  buf[i] = disk->data;

Very, Very, Expensive
1. Programmed I/O: 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

2. Direct Memory Access (DMA): 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

*Which one is the winner? Which one is the loser?*
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;
```
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;
```
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
Polling vs. Interrupts

How does program learn device is ready/done?

1. **Polling**: Periodically check I/O status register
   - Common in small, cheap, or real-time embedded systems
   - Predictable timing, inexpensive
   - Wastes CPU cycles

2. **Interrupts**: Device sends interrupt to CPU
   - Cause register identifies the interrupting device
   - Interrupt handler examines device, decides what to do
   - Only interrupt when device ready/done
   - Forced to save CPU context (PC, SP, registers, etc.)
   - Unpredictable, event arrival depends on other devices’ activity

Which one is the winner? Which one is the loser?
I/O 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.