Input/Output

198:231 Introduction to Computer Organization
Lecture 15

Instructor:
Nicole Hynes
nicole.hynes@rutgers.edu
Organization of a Typical Computer System

We’ve discussed processor and memory

We’ll discuss I/O next
I/O Devices

Input
- Keyboard, computer mouse, graphic tablet, touch-screen, barcode reader, scanner, webcam, voice input/microphone

Output
- Monitor/graphics display, printer, projector, voice output/speakers

Input or Output
- Modem, network interface card (NIC)

Storage
- Magnetic disk, flash drive, optical disk, tape
## I/O Devices

### I/O Devices Categorized by Behavior, Partner and Data Rate

<table>
<thead>
<tr>
<th>Device</th>
<th>Behavior</th>
<th>Partner</th>
<th>Data Rate (bits/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>Human</td>
<td>3M</td>
</tr>
<tr>
<td>Graphics display</td>
<td>Output</td>
<td>Human</td>
<td>800M – 8G</td>
</tr>
<tr>
<td>Laser printer</td>
<td>Output</td>
<td>Human</td>
<td>3.2M</td>
</tr>
<tr>
<td>Sound output</td>
<td>Output</td>
<td>Human</td>
<td>8M</td>
</tr>
<tr>
<td>Modem</td>
<td>Input or output</td>
<td>Machine</td>
<td>16K – 64K</td>
</tr>
<tr>
<td>Network/LAN</td>
<td>Input or output</td>
<td>Machine</td>
<td>10M – 55M</td>
</tr>
<tr>
<td>Magnetic disk</td>
<td>Storage</td>
<td>Machine</td>
<td>240M – 2.5G</td>
</tr>
<tr>
<td>Magnetic tape</td>
<td>Storage</td>
<td>Machine</td>
<td>32M</td>
</tr>
<tr>
<td>Optical disk</td>
<td>Storage</td>
<td>Machine</td>
<td>80M</td>
</tr>
</tbody>
</table>
Typical PC Hierarchical Bus Architecture

- Intel Xeon 5300 processor
- Front Side Bus (1333 MHz, 10.5 GB/sec)
- FB DDR2 667 (5.3 GB/sec)
- Main memory DIMMs
- Memory controller hub (north bridge) 5000P
- PCIe x16 (or 2 PCIe x8) (4 GB/sec)
- ESI (2 GB/sec)
- PCIe x8 (2 GB/sec)
- Disk (Serial ATA 300 MB/sec)
- Disk (LPC 1 MB/sec)
- Keyboard, mouse, ...
- USB 2.0 (60 MB/sec)
- I/O controller hub (south bridge) Enterprise South Bridge 2
- PCIe x4 (1 GB/sec)
- PCI-X bus (1 GB/sec)
- Parallel ATA (100 MB/sec)
- CD/DVD

Adapted from Patterson’08
# I/O Buses

## Example Buses

<table>
<thead>
<tr>
<th>Name</th>
<th>Uses</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</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 v.31</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>
Device Manager

Application Process

File Manager

Device Manager

System Interface

Hardware Interface

Command  Status  Data

I/O Device Controller

Adapted from Nutt’04
Device Driver Interface

Device manager consists of a collection of device drivers:

- Provides abstract API to application programs.
- Device drivers implement API by performing low-level device-dependent operations on the I/O device controller.

```
... write(...);
...
```

Adapted from Nutt'04
I/O Device Controller

I/O Device Controller Registers

- **Command register**: holds the specific I/O operation to be performed (e.g., read, write)
- **Status register**: holds status information about attached device (e.g., whether device is busy or an error condition has occurred)
- **Data register(s)**: used to transfer data between the CPU and the attached device
Addressing Devices

Port I/O or Isolated I/O

- Device registers have port addresses separate from memory addresses.
- Special I/O machine instructions are used to select the proper I/O port and move the data into or out of a device register.
- Example: IA32 in and out instructions

```assembly
inb 0x02f8, %al  # read byte from device register at port address 0x02f8 into %al
outw %ax, 0x02fc  # write word in %ax to device register at port address 0x02fc
```

- May require dedicated hardware interface to devices.
- Virtualization can be difficult.
Addressing Devices

Implementing Port I/O

Adapted from Batten’12
Addressing Devices

Memory-Mapped I/O

- Device registers are mapped to memory addresses.
- Device registers may be accessed using normal load/store memory instructions.
- Example:

```c
movb 0x8004, %al  # move byte from device register at memory address 0x08004 into %al
movw %ax, 0x8008  # move word in %ax to device register at memory address 0x08008
```
- Re-uses standard memory hardware interface.
- Virtualization easily supported with normal virtual memory system.
Addressing Devices

Implementing Memory-Mapped I/O

- Address decoder determines which device communicates with processor
- Use address and rd/wr signals to generate write enable and read mux control signals

Adapted from Batten’12
Input/Output Techniques

1. Programmed I/O (a.k.a. polling I/O)
2. Interrupt-driven I/O
3. Direct memory access (DMA)
Programmed I/O

CPU does all the work.
- CPU writes next I/O operation on command register and waits for device to complete operation.
- Continuously checks status of device (busy, ready) – status polling.

Wastes CPU time.
- I/O devices much slower than CPU – wastes CPU cycles.
- May be acceptable in an embedded system with dedicated processor that does only one task and needs to complete task in real-time.

Example: read operation
- See flowchart.
Interrupt-Driven I/O

Instead of CPU polling device, device sends an interrupt signal to the CPU.

On receipt of interrupt signal CPU launches interrupt handler:

- Running program is “interrupted”.
- CPU context is saved; interrupt handler determines source of interrupt and completes I/O operation.
- CPU context is restored.
- Previously running program is resumed.

Example: read operation
- See flowchart.
Instruction Cycle with Interrupt Processing

CPU checks for interrupt signal at end of each instruction cycle

Instruction Cycle:
- Fetch
- Decode
- Execute
- Memory
- Write Back
- Interrupt?
  - Yes: Interrupt Handler
    - Save context
    - Get INTR ID
    - Invoke device handler to perform I/O operation
    - Restore context
  - No

Return from interrupt
I/O devices have (unique or shared) Interrupt Request Lines (IRQs)

- IRQs are mapped by special hardware to interrupt vectors, and passed to the CPU
- This hardware is called a Programmable Interrupt Controller (PIC)
- Early PC systems use two 8-input PICs (8259A)
Invoking Device Handler from IRQ

- IRQ from device results in PIC sending an interrupt signal to CPU.
- On acknowledgment from CPU, PIC sends interrupt vector corresponding to IRQ to CPU.
- CPU uses vector as index to the Interrupt Descriptor Table (IDT); IDT’s address in memory is stored in special register idtr.
- IDT entry corresponding to vector contains address of device handler corresponding to interrupt.
- Device handler is run to complete I/O operation.
Programmed I/O vs. Interrupt-Driven I/O

Each method incurs overhead when performing I/O

- Programmed I/O – polling overhead
  - CPU clock cycles consumed by repeatedly checking device status

- Interrupt-driven I/O – interrupt overhead
  - CPU clock cycles consumed by system call to interrupt handler, including saving/restoring CPU context

- Note: does not include clock cycles for performing actual data transfer

Case Study

- Assume the following:
  - 1 GHZ CPU (= $10^9$ clock cycles/sec)
  - Programmed I/O: polling overhead = 300 clock cycles/poll
  - Interrupt-driven I/O: interrupt overhead = 500 clock cycles/interrupt

- Study effect of I/O overhead for mouse and hard drive
Programmed I/O vs. Interrupt-Driven I/O

Example 1. Mouse

- Mouse changes position at a **maximum** rate of 60 times per second and an **average** rate of 45 times per second

**Programmed I/O**

- Must poll at the maximum rate or else miss some mouse movement → 60 polls/sec
- Percent of CPU consumed by polling overhead
  \[\frac{(60 \text{ polls/sec} \times 300 \text{ cycles/poll})}{10^9 \text{ cycles/sec}} = 0.0018\%\]

**Interrupt-driven I/O**

- Number of interrupts = number of times mouse actually moves
- On average, 45 interrupts/sec
- Percent of CPU consumed by interrupt overhead
  \[\frac{(45 \text{ interrupts/sec} \times 500 \text{ cycles/interrupt})}{10^9 \text{ cycles/sec}} = 0.00225\%\]

Performance comparison: close but programmed I/O wins
Programmed I/O vs. Interrupt-Driven I/O

Example 2. Hard drive

- Hard drive transfers data in 16-byte chunks
- Maximum data transfer rate is 20 MB/sec
- Assume on average that hard drive is transferring 10% of the time

Programmed I/O

- Number of 16-byte transfers/sec = \( \frac{20 \times 10^6}{16} = 1.25 \times 10^6 \) transfers/sec
- Must poll at the maximum rate → \( 1.25 \times 10^6 \) polls/sec
- Percent of CPU consumed by polling overhead
  \[ = \left( \frac{1.25 \times 10^6}{sec} \times 300 \text{ cycles/poll} \right) / (10^9 \text{ cycles/sec}) = 37.5\% \]

Interrupt-driven I/O

- Number of 16 byte transfers/sec = \( 1.25 \times 10^6 \) transfers/sec
- Since hard disk is transferring only 10% of time, average number of interrupts/sec
  \[ = \left( \frac{1.25 \times 10^6}{0.1} \right) = 1.25 \times 10^5 \text{ interrupts/sec} \]
- Percent of CPU consumed by interrupt overhead
  \[ = \left( \frac{1.25 \times 10^5 \text{ interrupts/sec} \times 500 \text{ cycles/interrupt}}{(10^9 \text{ cycles/sec})} \right) = 6.25\% \]

Performance comparison: interrupt-driven I/O wins hands down
Direct Memory Access

Interrupt-driven I/O removes wasteful status-polling in programmed I/O.

However, in both interrupt-driven and programmed I/O the CPU is still responsible for data transfer between device and memory.

- CPU is tied up while performing data transfer.
- Data transfer rate is limited – CPU can transfer only a few bytes at a time using standard move instructions.
- Woefully inefficient for transferring blocks of data to/from disk.

Solution: Direct Memory Access (DMA)

- Use special device, called a Direct Memory Access Controller (DMAC), to transfer data between I/O device and memory without CPU intervention.
- DMA is frequently part of the controller of block devices such as disks.
Direct Memory Access

Three Main Steps of DMA

1. CPU tells DMAC what to do (DMA initialization).
2. DMAC does the actual data transfer.
3. DMAC interrupts CPU upon completion of data transfer.

Taken from Stallings’10
Block Diagram of a DMAC

Address bus

Data bus

DMA select
Register select
Read
Write
Bus request
Bus granted
Interrupt

Data bus buffers

Control logic

Address register

Word-count register

Control register

DMA request

DMA acknowledge to I/O device

Taken from Mano’04
DMAC within a Computer System

Taken from Mano’04
DMA Initialization

CPU initializes the DMA by programming the DMAC registers:

- DMA address register is loaded with the starting address of the memory block to be read/written.
- DMA word (or byte) count register is loaded with the size of the memory block, in #words (or #bytes).
- Relevant bits of the DMA control register are set to specify the direction of data transfer (read or write).
DMA Transfer

1. When I/O device is ready, it sends a DMA request to the DMA controller (DMAC).

2. The DMAC activates the BR (Bus Request) line to inform the CPU to relinquish the bus. The CPU responds with its BG (Bus Grant) line, informing the DMAC that it is now the bus master.

3. The DMAC puts the current value of its address register on the address bus, activates the Read or Write control signal to memory, and sends a DMA acknowledge to the device.

4. When the device receives the DMA acknowledge, it puts a word on the data bus (for memory write) or receives a word from the data bus (for memory read).

5. For each word transferred, the DMAC increments its address register (to point to the next memory address) and decrements its word count register.

6. If the word count register is not yet zero, the DMAC waits for the next DMA request from the device, upon receipt of which it repeats step 3.

7. When the word count register becomes zero, the DMAC stops any further data transfer and deactivates its BR line. It also informs the CPU of the termination by means of an interrupt signal.
DMA Modes of Operation

Burst Mode

- An entire block of data is transferred in one contiguous sequence.
- Once the DMAC is granted access to the memory bus by the CPU, it transfers all bytes of data in the data block before releasing control of the bus back to the CPU.
- Also called "Block Transfer Mode".

Cycle Stealing Mode

- DMA controller takes over bus for a cycle; transfers one word of data.
- CPU pauses for one bus cycle.
- Not an interrupt – CPU does not switch context.
- Slows down CPU but does not idle it for long periods of time as in burst mode.
Direct Memory Access

Overhead Using DMA

- As in previous example, assume a 1 GHZ CPU and a hard drive with data transfer rate of 20 MB/sec.

- Assume DMA is used instead with following parameters:
  - Overhead due to DMA initialization = 1000 clock cycles
  - Overhead due to interrupt upon DMA completion = 500 clock cycles
  - Average transfer size from disk is 8KB

- What % of the CPU is consumed if the disk is actively transferring 100% of the time? Ignore any bus contention between CPU and DMAC.

- Answer:
  - No. of 8KB DMA transfers/sec = (20MB/sec)/8KB = 2500 DMA transfers/sec
  - Overhead per DMA transfer = 1000 + 500 = 1500 clock cycles
  - Percent of CPU consumed by DMA overhead = (2500 DMA transfers/sec) × (1500 clock cycles/DMA transfer) / (10^9 clock cycles/sec) = 0.375%
DMA and Virtual Memory

In a VM system, should DMA use virtual addresses or physical addresses?

- Either is possible, but give rise to different issues

Most DMA controllers use physical addresses

- Requires OS to convert the virtual address of the intended memory block into a physical address and write this physical address into the DMAC’s address register
- Must constrain DMA transfer to stay within a page or break up DMA transfer into a series of transfers, each confined within a single physical page

Some DMA controllers use virtual addresses

- Called virtual DMA
- DMAC must use MMU to have virtual-to-physical address translation done
DMA and Cache Coherency

DMA can lead to cache coherency problem:

- Example: See figure below; assume a write-back cache
  1. CPU writes the value Y into the cache without updating the main memory.
  2. DMAC transfers the old value X from main memory to an I/O device.
  \[\rightarrow\] I/O device gets old value X rather than new value Y.

![Diagram showing DMA and cache coherency](image)

- X: Old value
- Y: New value
DMA and Cache Coherency

Possible solutions to cache coherency problem

1. Route all I/O activity through cache
   - Ensures coherent copies across cache, main memory, and I/O device
   - However, performance hit could be very high
   - Also, may displace good data in cache needed by the running program

2. Operating system flushes cache
   - Invalidate affected cache entries on I/O-to-memory transfer
   - Force cache write-back on memory-to-I/O transfer
   - This is a software solution (OS) and hence incurs overhead due to system calls, but performance hit is not as high as solution 1.

3. Hardware support for cache coherency
   - Build cache flushing/invalidation into hardware
   - Already done in multi-processor systems (CC-NUMA)
   - Can also be used to solve DMA-induced cache coherency problem
Example Device Controllers

Intel 82C55A Programmable Peripheral Interface

- General-purpose I/O controller for use with Intel x86 processors.
- Three 8-bit ports (A, B, C); port C can be subdivided into two 4-bit ports.
- Each port can be programmed to either be input or output.
Example Device Controllers

Intel 82C55A Programmable Peripheral Interface

Printer interface

Keyboard and display interface
Example Device Controllers

Generic Disk Controller

Controller commands:
- Read sector
- Write sector
- Restore
- Seek
- Format
Buffering

Technique used to cope with the speed mismatch between the producer and a consumer of a data stream.
Buffering

- Buffering is the technique by which the device manager can keep slower I/O devices busy during times when a process is not requiring I/O operations.

- **Input buffering** is the technique of having the input device copy information into memory before the process requests it.

- **Output buffering** is the technique of saving information in memory and then writing it to the device while the process continues execution.
Hardware Buffering

Unbuffered

Process reads $b_{i-1}$
Controller brings in $b_i$

Process reads $b_i$
Controller brings in $b_{i+1}$
Double Buffering

Process

Controller

Device

Driver

Hardware
Circular Buffering

To data consumer

Buffer i

From data producer

Buffer j
Buffer Cache

- I/O operations with a block device such as a disk are often handled through a buffer cache that hold disk blocks that may be shared by multiple processes.

- The buffer cache is created in an area of main memory (RAM) accessible only to the OS.

- On a typical system approximately 85% of disk I/O can be avoided by using the buffer cache, though in general this depends on the job mix.
Using the Buffer Cache

Illustration

- Each block device has a request queue consisting of a list of pending read/write requests to blocks on the device

To service a request, OS first checks to see if requested disk block is in the buffer cache and if so, reads/writes the buffer cache copy.

If requested block is not in the buffer cache, then OS retrieves the block from disk and copies it into the buffer cache.

Uses write-back strategy: write back to disk only if “dirty” bit is set.
Using the Buffer Cache

Illustration

1. P1 issues read request for disk block 17

   ![Diagram](image1)

   Block 17 copied into cache

2. P2 issues read request for disk block 54

   ![Diagram](image2)

   Block 54 copied into cache
Using the Buffer Cache

Illustration

3. P3 issues read request for disk block 17

4. P3 issues write request for disk block 17

Block 17 in cache: read cache copy

Block 17 in cache: modify cache copy and mark as “dirty” (write to disk is delayed)
Using the Buffer Cache

Illustration

5. P4 issues read request for disk block 17

![Diagram showing P4 reading block 17 from the cache]

Block 17 in cache: read cache copy

6. Delayed-write buffers are written back to disk by executing sync( )

![Diagram showing sync() execution and block 17 written to disk]
Blocking, Non-Blocking, and Asynchronous I/O

Blocking I/O
- The process is put to sleep (i.e., suspended) when an I/O request is made, then awakened when the request completes

Non-blocking I/O
- The I/O request returns immediately, regardless of whether or not the requested I/O operation has completed
- The process gets a status about the I/O operation (completed or not), allowing it to re-try the I/O request, if it has not yet completed, at a later time
- The `select` system call is often used with non-blocking I/O

Asynchronous I/O
- Two-stage operation: a request for a `read` or `write` operation is first made, and returns immediately
- Later on, the process is notified that the operation is complete
Buffer Cache and Asynchronous I/O

In the kernel, the buffer cache provides asynchronous I/O interface to block devices

- A write to the buffer cache does not necessarily translate to an immediate write to the disk
- This is good because it reduces the amount of I/O needed to the device in case that same block gets modified again in the future
- There is a risk, however. If the system crashes or is shut off spontaneously before all modified blocks are written, that data is then lost

To minimize data loss, the operating system will periodically force a flush of data that's cached in the buffer cache

- On BSD systems, a user process, update, calls the sync system call every 30 seconds to flush data
- On Linux, the operation is done by a process called kupdated, the kernel update daemon
Device Types

Block Devices
- Data transfer in the form of blocks
- Examples: hard disk, CD-ROM drive, flash drive
- Usually support random access and generally use buffered input and output routines, typically via the buffer cache

Character Devices
- Data transfer can be of different types
- Anything that is not a block device or network device
  - Streams of characters: mouse, keyboard, serial modem, printer
  - Frame buffer: has its own buffer management policies and custom interfaces
  - Sound devices (DSPs), I2C controllers, etc.
- I/O does not go through the buffer cache

Network Devices
- Packet-oriented, rather than stream-oriented device
- The underlying network device may be either hardware (e.g., an ethernet controller) or software (e.g., a loopback driver)
- Not visible in the file system but is accessible through the socket interface
Device File Interface

Device manager interacts with file manager. Why?

Because devices are treated as files.

Adapted from Nutt’04
Device File Interface

In Linux/Unix systems, character and block devices are visible as files in the file system.

Device drivers implement operations similar to that for regular files – e.g., `read()`, `write()`, `seek()`.
In Linux, the user-level interface to a device is called a **device file**

- Device files are stored in the directory `/dev`

```
% ls -l /dev
```

<table>
<thead>
<tr>
<th>major #</th>
<th>minor #</th>
<th>Date</th>
<th>Time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>Sep 28</td>
<td>18:06</td>
<td>full</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>Mar 15</td>
<td></td>
<td>lp0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>Mar 15</td>
<td></td>
<td>lp1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Mar 15</td>
<td></td>
<td>lp2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Mar 15</td>
<td></td>
<td>fd0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Mar 15</td>
<td></td>
<td>fd1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Mar 15</td>
<td></td>
<td>hda</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>Mar 15</td>
<td></td>
<td>hdb</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>Mar 15</td>
<td></td>
<td>hdc</td>
</tr>
<tr>
<td>22</td>
<td>64</td>
<td>Mar 15</td>
<td></td>
<td>hdd</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>Mar 15</td>
<td></td>
<td>ttyS0</td>
</tr>
<tr>
<td>4</td>
<td>65</td>
<td>Mar 15</td>
<td></td>
<td>ttyS1</td>
</tr>
</tbody>
</table>

- **Major number** – identifies device class
- **Minor number** – identifies specific device within the class
Device File Interface

Applications use system calls to access devices
- `open()`, `read()`, `write()`, ...

These system calls have a counterpart in the device driver
- Not all functions may be supported by the driver – depends on the device

Example: Linux
To Learn More

We will discuss I/O, including device drivers, in more detail in the Operating Systems Course

For more information, I recommend the following:

- The Linux Documentation Project - http://www.tldp.org/guides.html
The End