I/O Devices and Disks
I/O Devices

• Input and output (I/O) is critical part of computer systems

• Issues:
  • How should I/O be integrated into systems?
  • What are the general mechanisms?
  • How can we make I/O efficient?
Structure of input/output (I/O) device

CPU is attached to the main memory of the system via a memory bus.
Some devices are connected to the system via a general I/O bus.
I/O Architecture

• Buses
  • Data paths that provided to enable information between CPU(s), RAM, and I/O devices

• I/O bus
  • Data path that connects a CPU to an I/O device
  • I/O bus is connected to I/O device by three hardware components:
    • I/O ports
    • Interfaces
    • Device controllers
Canonical Device

- Canonical Devices has two important components
  - **Hardware interface** allows the system software to control its operation
  - **Internals** which is implementation specific
- Registers
  - **status register**
    - See the current status of the device
  - **command register**
    - Tell the device to perform a certain task
  - **data register**
    - Pass data to the device, or get data from the device
    - These registers are typically only a few bytes wide
- By reading and writing these registers, the OS can control device behavior
The Canonical Protocol

• Example of programmed I/O (PIO)
  • CPU is involved with the data movement

• Operating system waits until the device is ready by **repeatedly** reading the status register
  • Simple and correct
  • Wastes CPU time just waiting for the device
    • Switching to another ready process would utilize the CPU more

```c
While ( STATUS == BUSY)
    ; //wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    Doing so starts the device and executes the command
While ( STATUS == BUSY)
    ; //wait until device is done with your request
```

![Diagram of CPU utilization by polling](image)
Interrupts

- Put the I/O request process to sleep and context switch to another
- When the device is finished, wake the process waiting for the I/O by interrupt
  - Allows for the CPU and the disk to be properly utilized
Polling vs. interrupts

• Interrupts are not always the best solution
  • If a device performs very quickly interrupts slow down the system
    • Context switches are expensive

• If a device is fast, polling is preferred

• If the device is slow, interrupts are better
Data movement with PIO

- The CPU wastes time copying data from memory to the device
  - CPU copies one word at a time to the device
  - After data is copied to the device, then it can be written to disk

Diagram of CPU utilization:

- Task 1: Copy data from memory
- Task 2: Data copy one word at a time
DMA (Direct Memory Access)

- Copy data in memory by knowing “where the data lives in memory, how much data to copy”
- When completed, DMA raises an interrupt, I/O begins on Disk
  - Transfers between devices and main memory without much CPU intervention

### Diagram of CPU utilization by DMA

<table>
<thead>
<tr>
<th>CPU</th>
<th>DMA</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>C</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>2 2 2 2</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>2 2 2 2</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>2 2 2 2</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1: task 1, 2: task 2, C: copy data from memory
Device interaction

• Two main ways to communicate with devices

• I/O instructions: a way for the OS to send data to specific device registers
  • E.g., in and out instructions on x86
  • Typically, this a privileged instruction – why?

• Memory-mapped I/O
  • Device registers available as if they were memory locations
  • The OS loads (to read) or stores (to write) to the device instead of main memory
  • No new instructions, same as a memory read or write
The OS interface: the device driver

• How does the OS interact with different specific hardware interfaces?
  • E.g., we would like to build a single file system interface that works with:
    • SCSI disks
    • SATA disks
    • USB disks and so on

• **Abstraction** encapsulates any specifics of device interaction
  • At the lowest level, the OS must know how the hardware works
  • We call this software a device driver
  • Provides a higher-level interface to the rest of the system
Example Driver: File system Abstraction

• Application is unaware of the type of filesystem
• File system is unaware of which type of disk it is using
  • Issues block read/write requests to a generic block layer
• Many drivers expose a raw interface to allow for special applications
Problems With Device Driver Abstraction

• If there is a device with special capabilities, these capabilities will go unused in the generic interface layer

• Over 70% of Linux code is found in device drivers
  • Need a device driver for any piece of hardware you might plug into your system
  • Often not written by full-time kernel developers
    • Especially true for non commodity hardware
  • They are primary contributor to kernel crashes!
A Simple IDE Disk Driver: Registers

- **Control Register:**
  - Address 0x3F6: set to enable interrupts

- **Command Block Registers:**
  - Address 0x1F0: Data Port
  - Address 0x1F1: Error
  - Address 0x1F2: Sector Count
  - Address 0x1F3: LBA low byte
  - Address 0x1F4: LBA mid byte
  - Address 0x1F5: LBA hi byte
  - Address 0x1F6: 1B1D TOP4LBA: B=LBA, D=drive
  - Address 0x1F7 = Command/status

- **Status Register (Address 0x1F7):**

  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
  |-----------------------------|
  | BUSY | READY | FAULT | SEEK | DRQ | CORR | IDDEX | ERROR |

- **Error Register (Address 0x1F1):** (check when Status ERROR==1)

  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
  |-----------------------------|
  | BBK | UNC | MC | IDNF | MCR | ABRT | T0NF | AMNF |

BBK = Bad Block
UNC = Uncorrectable data error
MC = Media Changed
IDNF = ID mark Not Found
MCR = Media Change Requested
ABRT = Command aborted
T0NF = Track 0 Not Found
AMNF = Address Mark Not Found
Basic IDE Protocol

• Wait for drive to be ready
  • Read Status Register (0x1F7) until drive is not BUSY and READY

• Write parameters to command registers
  • Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number to command registers (0x1F2-0x1F6)

• Start the I/O
  • Write the READ/WRITE command to command register (0x1F7)

• Data transfer (for writes)
  • Wait until drive status is READY and DRQ (drive request for data)
  • Write data to data port

• Handle interrupts
  • In the simplest case, handle an interrupt for each sector transferred
  • DMA allows for batching and a final interrupt when the entire transfer is complete

• Error handling
  • After each operation, read the status register
  • If the ERROR bit is on, read the error register for details
I/O Summary

• For efficiency we use:
  • Interrupts: allow process to sleep while slow I/O takes place
  • DMA: Allow transfer between memory and a device with little CPU intervention

• Accessing Hardware
  • Explicit I/O instructions (\texttt{inb outb})
  • Memory mapped I/O (register access looks like a memory read or write)

• Drivers
  • Encapsulate low-level details of the hardware
  • Makes it easier to build the rest of the OS in a device-neutral fashion
Hard Disk Drives

• Hard disk drives have been the main form of persistent data storage in computer systems for decades
  • The drive consists of many sectors (e.g., 512-byte blocks)
    • Arranged in circular tracks around the disk
    • The only guarantee is that a single 512-byte write is atomic
  • Address Space
    • We can view a disk with n sectors as an array of sectors, 0 to n-1

• Multi-sector operations are possible
  • Many file systems will read or write 4KB at a time (common page size)
  • Torn write
    • If an untimely power loss or error occurs, only a portion of a larger write may complete

• Accessing blocks in a contiguous chunk is the fastest access mode
  • A sequential read or write
  • Much faster than a more random access pattern
Basic Geometry

• **Platter** (Aluminum coated with a thin magnetic layer)
  - A circular hard surface
  - Data is stored persistently by inducing magnetic changes to it
  - Each platter has 2 sides, each of which is called a **surface**

• **Spindle**
  - Spindle is connected to a motor that spins the platters around
  - The rate of rotations is measured in **RPM** (Rotations Per Minute)
    - Typical modern values: 7,200 RPM to 15,000 RPM
    - E.g., 10000 RPM: A single rotation takes about 6 ms

• **Track**
  - Concentric circles of sectors
  - Data is encoded on each surface in a track
  - A single surface contains many thousands and thousands of tracks
A Simple Disk Drive

- Disk head (one head per surface of the drive)
  - The process of reading and writing is accomplished by the disk head
  - Attached to a single disk arm, which moves across the surface
Example of a Disk
Single-track Latency: The Rotational Delay

- Rotational delay: Time for the desired sector to rotate
  - E.g., full rotational delay is $R$ and we start at sector 6
    - Read sector 0: Rotational delay = $\frac{R}{2}$ (average case)
    - Read sector 5: Rotational delay = $R$ (worst case)
Multiple Tracks: Seek Time

- E.g., move to sector 11
  - Seek: move the disk arm to the correct track
    - One of the costliest disk operations
  - Seek time: time to move head to the track contain the desired sector
Phases of Seek

- **Acceleration** ➔ **Coasting** ➔ **Deceleration** ➔ **Settling**
  - **Acceleration**: The disk arm gets moving
  - **Coasting**: The arm is moving at full speed
  - **Deceleration**: The arm slows down
  - **Settling**: The head is *carefully positioned* over the correct track
    - The settling time is often quite significant, e.g., 0.5 to 2ms
Transfer

• The final phase of I/O
  • Data is either read from or written to the surface

• Complete I/O time:
  • Seek
  • Waiting for the rotational delay
  • Transfer
Track Skew

- Make sure that sequential reads can be properly serviced even when crossing track boundaries

- Example:
  - Moving from 23 to 24

- Without track skew, the head would be moved to the next track, but the desired next block would have already rotated under the head
Cache (Track Buffer)

- Disk cache holds data read from or written to the disk
  - Allow the drive to quickly respond to requests
  - Small amount of memory (usually around 8 or 16 MB)

- Write back (Immediate reporting)
  - Acknowledge a write has completed when it has put the data in its memory
  - Faster but dangerous

- Write through
  - Acknowledge a write has completed only after the write has been written to disk
  - Slower but safer
I/O Time: Fun With Math

- I/O time ($T_{I/O}$):

\[
T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}
\]

- The rate of I/O ($R_{I/O}$):

\[
R_{I/O} = \frac{\text{Size}_{Transfer}}{T_{I/O}}
\]

<table>
<thead>
<tr>
<th></th>
<th>Cheetah 15K.5</th>
<th>Barracuda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>300 GB</td>
<td>1 TB</td>
</tr>
<tr>
<td>RPM</td>
<td>15,000</td>
<td>7,200</td>
</tr>
<tr>
<td>Average Seek</td>
<td>4 ms</td>
<td>9 ms</td>
</tr>
<tr>
<td>Max Transfer</td>
<td>125 MB/s</td>
<td>105 MB/s</td>
</tr>
<tr>
<td>Platters</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Cache</td>
<td>16 MB</td>
<td>16/32 MB</td>
</tr>
<tr>
<td>Connects Via</td>
<td>SCSI</td>
<td>SATA</td>
</tr>
</tbody>
</table>

Disk Drive Specs: SCSI Versus SATA
I/O Time Example

- **Random workload**: Issue 4KB read to random locations on the disk
- **Sequential workload**: Read 100MB consecutively from the disk

<table>
<thead>
<tr>
<th></th>
<th>Cheetah 15K.5</th>
<th>Barracuda</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{seek}}$</td>
<td>4 ms</td>
<td>9 ms</td>
</tr>
<tr>
<td>$T_{\text{rotation}}$</td>
<td>2 ms</td>
<td>4.2 ms</td>
</tr>
<tr>
<td>$T_{\text{transfer}}$</td>
<td>30 microsecs</td>
<td>38 microsecs</td>
</tr>
<tr>
<td>$T_{\text{I/O}}$</td>
<td>6 ms</td>
<td>13.2 ms</td>
</tr>
<tr>
<td>$R_{\text{I/O}}$</td>
<td>0.66 MB/s</td>
<td>0.31 MB/s</td>
</tr>
<tr>
<td><strong>Random</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{\text{transfer}}$</td>
<td>800 ms</td>
<td>950 ms</td>
</tr>
<tr>
<td>$T_{\text{I/O}}$</td>
<td>806 ms</td>
<td>963.2 ms</td>
</tr>
<tr>
<td>$R_{\text{I/O}}$</td>
<td>125 MB/s</td>
<td>105 MB/s</td>
</tr>
<tr>
<td><strong>Sequential</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Disk Drive Performance: SCSI Versus SATA**

There is a huge gap in drive performance between *random* and *sequential* workloads
Disk Scheduling

- **Disk Scheduler** decides which I/O request to schedule next
- **SSTF** (Shortest Seek Time First)
  - Order the queue of I/O request by track
  - Pick requests on the nearest track to complete first

SSTF: Scheduling Request 21 and 2
Issue the request to 21 → issue the request to 2
SSTF is not a panacea

- **Problem 1:** The drive geometry is not available to the host OS
  - Solution: OS can simply implement **Nearest-block-first (NBF)**

- **Problem 2:** Starvation
  - If there were a steady stream of request to the inner track, request to other tracks would then be ignored completely
Elevator (a.k.a. SCAN or C-SCAN)

- Move across the disk servicing requests in order across the tracks
  - **Sweep**: A single pass across the disk
    - If a request comes for a block on a track that has already been serviced on this sweep of the disk, it is queued until the next sweep

- **F-SCAN**
  - Freeze the queue to be serviced when it is doing a sweep
  - Avoid starvation of far-away requests by nearer by late coming requests

- **C-SCAN** (Circular SCAN)
  - Sweep from outer-to-inner, and then inner-to-outer, etc.
How to account for Disk rotation costs?

- If rotation is faster than seek: request 16 → request 8
- If seek is faster than rotation: request 8 → request 16

SSTF: Sometimes Not Good Enough

On modern drives, both seek and rotation are roughly equivalent: Thus, SPTF (Shortest Positioning Time First) is useful
Where is disk scheduling performed?

• Older systems:
  • OS did all the scheduling

• Newer systems:
  • Disks can handle multiple outstanding requests
  • Disks have sophisticated internal schedulers
    • Exact head position is available
    • Can implement SPTF accurately
  • OS issues a small number of disk requests (e.g., 16) and issues them all at once
    • Disk calculates the best possible SPTF order
Other scheduling issues

• I/O Merging
  • Reduce the number of request sent to the disk and lowers overhead
  • E.g., read blocks 33, then 8, then 34:
    • The scheduler merge the request for blocks 33 and 34 into a single two-block request

• How long to wait before issuing an I/O request?
  • Work-conserving – Issue I/O request right away
    • Disk will never be idle if there are requests to serve
  • Non-work-conserving – wait a little bit before issuing I/O request
    • A new and better request might arrive at the disk, increasing efficiency