I/O Devices & SSD
System Architecture

- **Hierarchical approach**
- **Memory bus**
  - CPU and memory
  - Fastest
- **I/O bus**
  - e.g., PCI
  - Graphics and higher-performance I/O devices
- **Peripheral bus**
  - SCSI, SATA, or USB
  - Connect many slowest devices
Device Structure

- **Hardware Interface**
  - Allows the system software to control its operation
  - status register: can be read to see the current status of the device
  - command register: can be written to tell the device to perform a certain task
  - data register: transfer data to/from the device

- **Internal Structure**
  - Implementation specific
  - Simple devices
    - have one or a few hardware chips to implement their functionality;
  - Complex devices
    - include a simple CPU running firmware, some general purpose memory, and other device-specific chips
Protocol

• By reading and writing internal registers of a device, the operating system can control device behavior.

```c
While (STATUS == BUSY)
    ; // wait until device is not busy -> polling
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
```

• Inefficiencies and Inconveniences
  – Polling
    • Repeatedly reading the status register
  – Programmed I/O (PIO)
    • The main CPU is involved with the data movement
Lowering CPU Overhead With Interrupts

- **Interrupt**
  - OS can issue a request, put the calling process to sleep, and context switch to another task.
  - When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a pre-determined interrupt service routine (ISR)
  - The handler in operating system code that will finish the request
    - e.g., reading data and perhaps an error code from the device
    - Wake the process waiting for the I/O
    - Allow for overlap of computation and I/O
- Interrupt is not *always* the best solution
  - If a device is fast, it may be best to poll.
  - Otherwise, use interrupt
Interrupt Optimization

- **Unknown device: hybrid approach**
  - polls for a little while and then, if the device is not yet finished, uses interrupts. (two-phased approach)

- **Network systems**
  - When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to **livelock**
    - only processing interrupts and never allowing a user-level process to run
  - Occasionally use polling to better control what is happening in the system (e.g., slashdot effect)

- **Interrupt Coalescing**
  - Multiple interrupts are coalesced into a single interrupt delivery
    - lower the overhead of interrupt processing
  - Long waiting
    - Many interrupts can be coalesced
    - Increase the latency of a request
More Efficient Data Movement With DMA

- OS programs the DMA engine
  - Source/destination address in memory or device
  - Data amount
- Other processes can be serviced
- When the DMA is complete, the DMA controller raises an interrupt, and the OS thus knows the transfer is complete.

![Programmed IO Diagram]

Dongkun Shin, SKKU
Methods Of Device Interaction

• **I/O instructions**
  – E.g., in and out on x86
  – To send data to a device, register $\rightarrow$ **port**.
  – Usually *privileged* instructions

• **Memory-mapped I/O**
  – Device registers are mapped to memory address space
  – load (to read) or store (to write)
Device Driver

- How can we keep most of the OS device-neutral, thus hiding the details of device interactions from major OS subsystems?
- Solution: **Abstraction**
  - The device driver in the OS know in detail how a device works
  - Any specifics of device interaction are encapsulated within D/D

File System Stack:

- Application
- POSIX API [open, read, write, close, etc.]
- File System
- Generic Block Interface [block read/write]
- Generic Block Layer
- Specific Block Interface [protocol-specific read/write]
- Device Driver [SCSI, ATA, etc.]

Issues block requests to the generic block layer
Routes block requests to the appropriate D/D
handles the details of issuing the specific request
Device Driver

- **Drawbacks of encapsulation**
  - Even if a device has many special capabilities, it has to present a generic interface to the rest of the kernel, those special capabilities will go unused.
  - e.g., SCSI devices which have very rich error reporting

- **Over 70% of OS code is found in device drivers**
  - for any given installation, most of that code may not be active
  - have many more bugs and thus are a primary contributor to kernel crashes
Disk System
Disk System

- **Data access in disk system**
  - Seek time
  - Rotational delay (latency time)
  - Data transmission time

1. Head movement from current position to desired cylinder:
   - **Seek time** (0-10s ms)

2. Disk rotation until the desired sector arrives under the head:
   - **Rotational latency** (0-10s ms)

3. Disk rotation until sector has passed under the head:
   - **Data transfer time** (< 1 ms)
Disk System

• **Average time to access some target sector**
  - \( T_{access} = T_{avg\text{-seek}} + T_{avg\text{-rotation}} + T_{avg\text{-transfer}} \)
• **Seek time (\( T_{avg\text{-seek}} \))**
  - Time to position heads over cylinder containing target sector
  - Typical \( T_{avg\text{-seek}} \) is 3~9ms
• **Rotational latency (\( T_{avg\text{-rotation}} \))**
  - Time waiting for first bit of target sector to pass under head
  - \( T_{avg\text{-rotation}} = \frac{1}{2} \times \frac{1}{\text{RPMs}} \times \frac{60s}{1\text{min}} \)
  - Typical \( T_{avg\text{-rotation}} = 1\text{~}4\text{ms} \)
• **Transfer time (\( T_{avg\text{-transfer}} \))**
  - Time to read the bits in the target sector
  - \( T_{avg\text{-transfer}} = \frac{1}{\text{RPM}} \times \frac{1}{\text{(avg. # sectors/track)}} \times \frac{60s}{1\text{min}} \)
Flash-based SSD

- **NAND flash memory**
  - Page: read/write unit, 4KB~16KB
  - Block: erase unit, *erase-before-write*, 128 KB or 256 KB
  - Limited lifetime: wear out
  - SLC, MLC, TLC
Flash Operations

- **Read (a page)**
  - Random access

- **Erase (a block)**
  - set each bit to the value 1

- **Program (a page)**
  - change some of the 1’s within a page to 0’s

<table>
<thead>
<tr>
<th>Device</th>
<th>Read ($\mu$s)</th>
<th>Program ($\mu$s)</th>
<th>Erase ($\mu$s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLC</td>
<td>25</td>
<td>200-300</td>
<td>1500-2000</td>
</tr>
<tr>
<td>MLC</td>
<td>50</td>
<td>600-900</td>
<td>~3000</td>
</tr>
<tr>
<td>TLC</td>
<td>~75</td>
<td>~900-1350</td>
<td>~4500</td>
</tr>
</tbody>
</table>

Erase Block

Program Page 0

Previous contents of Page 1, 2, 3 are all gone!
They must be moved before the block erase operation.
Flash Performance And Reliability

- **Wear out**
  - Erased/program slowly accrues a little bit of extra charge.
  - As that extra charge builds up, the block becomes unusable.
  - MLC block: 10,000 P/E (Program/Erase) cycle lifetime;
  - SLC block: 100,000 P/E cycles.

- **Disturbance**
  - When accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages
  - read disturbs or program disturbs

Dongkun Shin, SKKU
From Raw Flash to Flash-Based SSDs

- Flash-based SSD provides standard block interface atop the raw flash flash chips inside it.
  - Sector (512 KB) read/write
Samsung Galaxy Note8

- Samsung K3UH6H60AM-NGCJ 6 GB LPDDR4X SDRAM layered over a Qualcomm Snapdragon 835
- Samsung KLUCG4J1ED-B0C1 64 GB UFS flash storage
- Qualcomm WCD9341 Aqstic audio codec
- Skyworks 78160-11 power amplification module
- Avago AFEM-9066 power amplification module
- Wacom W9018 touch control IC
Solid-State Disk

- **SATA SSD**
- **NVMe SSD**
- **M.2 NVMe**
- **MacBook Air**

**Appearance of Serial ATA Connectors**
(Drawing courtesy of Molex)

- **Device plug connector**
- **Serial ATA signal connector (pin S1)**
- **Serial ATA power connector (pin P1)**
- **Host receptacle connector**

---

Dongkun Shin, SKKU
FTL

- FTL takes read and write requests on logical blocks and turns them into low-level read, erase, and program commands on the underlying physical blocks and physical pages.

- Performance issues
  - Utilizing multiple flash chips in parallel
  - Reduce write amplification
    - total write issued to flash chips/total write issued by host

- Reliability issues
  - Wear leveling
    - spread writes across the blocks of the flash as evenly as possible
    - ensure that all of the blocks of the device wear out at roughly the same time
  - Program disturbance
    - sequential-programming
      - program pages within an erased block in order
      - low page → high page
FTL: Direct Mapped

- A write to logical page N
  - Read in the entire block that page N is contained within
  - Erase the block
  - Program the old pages as well as the new one.
  - Read-modify-write approach

- Poor performance & high write amplification
- The physical blocks containing popular data will quickly wear out.
A Log-Structured FTL

- Upon a write to logical block N
  - The device appends the write to the next free spot in the currently-being-written-to block
  - no overwrite, out-of-place update
  - To allow for subsequent reads of block N, the device keeps a **mapping table**, which stores the physical address of each logical block.

- Problems: GC & mapping table

Where is the mapping table? What happens if the device loses power?
Garbage Collection

- Dead-block reclamation to provide free space for new writes
  - Find a block that contains one or more garbage pages
  - Read in the live (non-garbage) pages from that block, write out those live pages to the log
  - (Finally) reclaim the entire block for use in writing
- Must know
  - Whether each page is live or dead ➔ Mapping table or bitmap
  - The number of live pages in each block for victim selection
- Overprovision can delay GC, and push to the background
Mapping Table Size

- Page-level mapping
  - With a large 1-TB SSD, a single 4-byte entry per 4-KB page results in 1 GB of memory needed the device

<table>
<thead>
<tr>
<th>LPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

LPN: logical page number
PPN: physical page number

“write(9, A)”

```
Block 0
 Block 1
 Block 2
 Block 3
```

```
PPN 0
PPN 1
PPN 2
PPN 3
 PPN 15
```

data area
OOB

flash memory
Mapping Table Size

- Block-level mapping

LBN: logical block number
PBN: physical block number

“write(9, A)”

LBN: 9/4 = 2
offset: 1

PBN: 1
offset: 1

mapping table

<table>
<thead>
<tr>
<th>LBN</th>
<th>PBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

PPN 0
PPN 1
PPN 2

... A

data area
OOB

Block 0
Block 1
Block 2
Block 3

flash memory

Dongkun Shin, SKKU
Update at Block Mapping

1. Initial block address mapping

2. Update data (write new data) in an empty block

3. Block address re-mapping

4. Erase the original block (can be re-used now)
Hybrid Mapping

- To enable flexible writing but also reduce mapping costs
- FTL keeps a few log blocks and directs all writes to them
  - Page-level mapping (log table)
  - Small number of log blocks $\Rightarrow$ small-sized log table
- Normal data blocks
  - Block-level mapping (data table) $\Rightarrow$ small-sized data table
- For read request
  - first consult the log table; if not found, consult the data table
- To keep the number of log blocks small, log blocks must be merged with data blocks if no free space is available in log blocks.
Hybrid Mapping (BAST)

Hybrid Mapping – Merge Operation

(a) switch merge
(b) partial merge
(c) full merge in BAST
(d) full merge in FAST
SSD Performance

- Random I/O
  - Large difference between SSD and HDD
- Sequential I/O
  - Much less of a difference between SSD and HDD
  - HDD still a good choice
- SSD random read performance is not as good as SSD random write performance
  - Write buffer
- Performance difference between sequential and random in SSD
  - many of the file system techniques for HDDs are still applicable to SSDs

<table>
<thead>
<tr>
<th>Device</th>
<th>Random Reads (MB/s)</th>
<th>Random Writes (MB/s)</th>
<th>Sequential Reads (MB/s)</th>
<th>Sequential Writes (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung 840 Pro SSD</td>
<td>103</td>
<td>287</td>
<td>421</td>
<td>384</td>
</tr>
<tr>
<td>Seagate 600 SSD</td>
<td>84</td>
<td>252</td>
<td>424</td>
<td>374</td>
</tr>
<tr>
<td>Intel SSD 335 SSD</td>
<td>39</td>
<td>222</td>
<td>344</td>
<td>354</td>
</tr>
<tr>
<td>Seagate Savvio 15K.3 HDD</td>
<td>2</td>
<td>2</td>
<td>223</td>
<td>223</td>
</tr>
</tbody>
</table>