**Memory and I/O buses**

- CPU accesses physical memory over a bus
- Devices access memory over I/O bus with DMA
- Devices can appear to be a region of memory

---

**Realistic PC architecture**

**What is memory?**

- **SRAM – Static RAM**
  - Like two NOT gates circularly wired input-to-output
  - 4-6 transistors per bit, actively holds its value
  - Very fast, used to cache slower memory

- **DRAM – Dynamic RAM**
  - A capacitor + gate, holds charge to indicate bit value
  - 1 transistor per bit – extremely dense storage
  - Charge leaks—need slow comparator to decide if bit 1 or 0
  - Must re-write charge after reading, and periodically refresh

- **VRAM – “Video RAM”**
  - Dual ported, can write while someone else reads

---

**Communicating with a device**

- **Memory-mapped device registers**
  - Certain physical addresses correspond to device registers
  - Load/store gets status/sends instructions – not real memory

- **Device memory** – device may have memory OS can write to directly on other side of I/O bus

- **Special I/O instructions**
  - Some CPUs (e.g., x86) have special I/O instructions
  - Like load & store, but asserts special I/O pin on CPU
  - OS can allow user-mode access to I/O ports with finer granularity than page

- **DMA – place instructions to card in main memory**
  - Typically then need to “poke” card by writing to register
  - Overlaps unrelated computation with moving data over (typically slower than memory) I/O bus

---

**What is I/O bus? E.g., PCI**

**Example: parallel port (LPT1)**

- Simple hardware has three control registers:

  - **D7 D6 D5 D4 D3 D2 D1 D0**
    - **read/write data register (port 0x378)**
    - **BSY ACK PAP OFON ERR**
    - **read-only status register (port 0x379)**
    - **IRQ DSL INI ALF STR**
    - **read/write control register (port 0x37a)**

- Every bit except IRQ corresponds to a pin on 25-pin connector:

  - [Wikipedia][Messmer]
Writing bit to parallel port [osdev]

```c
void
sendbyte(uint8_t byte)
{
    /* Wait until BSY bit is 1. */
    while ((inb (0x379) & 0x80) == 0) delay ();

    /* Put the byte we wish to send on pins D7-0. */
    outb (0x378, byte);

    /* Pulse STR (strobe) line to inform the printer that a byte is available */
    uint8_t ctrlval = inb (0x37a);
    outb (0x37a, ctrlval | 0x01);
    delay ();
    outb (0x37a, ctrlval);
}
```

Memory-mapped IO

- **in/out instructions slow and clunky**
  - Instruction format restricts what registers you can use
  - Only allows $2^{16}$ different port numbers
  - Per-range access control turns out not to be useful (any port access allows you to disable all interrupts)
- **Devices can achieve same effect with physical addresses**, e.g.:
  ```c
  volatile int32_t *device_control = (int32_t *) 0xc00c0100;
  *device_control = 0x80;
  int32_t status = *device_control;
  ```
  - OS must map physical to virtual addresses, ensure non-cachable
- **Assign physical addresses at boot to avoid conflicts. PCI:**
  - Slow/clunky way to access configuration registers on device
  - Use that to assign ranges of physical addresses to device

DMA buffers

- **Idea:** only use CPU to transfer control requests, not data
- **Include list of buffer locations in main memory**
  - Device reads list and accesses buffers through DMA
  - Descriptions sometimes allow for scatter/gather I/O

Example: Network Interface Card

- **Link interface talks to wire/fiber/antenna**
  - Typically does framing, link-layer CRC
- **FIFOs on card provide small amount of buffering**
- **Bus interface logic uses DMA to move packets to and from buffers in main memory**

Example: IDE disk read w. DMA

- **Device driver provides several entry points to kernel**
  - Reset, ioctl, output, interrupt, read, write, strategy . . .
- **How should driver synchronize with card?**
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete
- **One approach:** **Polling**
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set
- **Disadvantages of polling?**
Driver architecture

- Device driver provides several entry points to kernel
  - Reset, ioctl, output, interrupt, read, write, strategy ...

- How should driver synchronize with card?
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete

- One approach: Polling
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set

- Disadvantages of polling?
  - Can’t use CPU for anything else while polling
  - Or schedule poll in future and do something else, but then high latency to receive packet or process disk block

Interrupt driven devices

- Instead, ask card to interrupt CPU on events
  - Interrupt handler runs at high priority
  - Asks card what happened (xmit buffer free, new packet)
  - This is what most general-purpose OSes do

- Bad under high network packet arrival rate
  - Packets can arrive faster than OS can process them
  - Interrupts are very expensive (context switch)
  - Interrupt handlers have high priority
  - In worst case, can spend 100% of time in interrupt handler and never make any progress – receive livelock
  - Best: Adaptive switching between interrupts and polling

- Very good for disk requests
- Rest of today: Disks (network devices in 3 lectures)

Anatomy of a disk [Ruemmler]

- Stack of magnetic platters
  - Rotate together on a central spindle @3,600-15,000 RPM
  - Drive speed drifts slowly over time
  - Can’t predict rotational position after 100-200 revolutions

- Disk arm assembly
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - Arms contain disk heads–one for each recording surface
  - Heads read and write data to platters

Disk
Storage on a magnetic platter

- Platters divided into concentric tracks
- A stack of tracks of fixed radius is a cylinder
- Heads record and sense data along cylinders
  - Significant fractions of encoded stream for error correction
- Generally only one head active at a time
  - Disks usually have one set of read-write circuitry
  - Must worry about cross-talk between channels
  - Hard to keep multiple heads exactly aligned

Disk positioning system

- Move head to specific track and keep it there
  - Resist physical shocks, imperfect tracks, etc.
- A seek consists of up to four phases:
  - speedup–accelerate arm to max speed or half way point
  - coast–at max speed (for long seeks)
  - slowdown–stops arm near destination
  - settle–adjusts head to actual desired track
- Very short seeks dominated by settle time (~1 ms)
- Short (200-400 cyl.) seeks dominated by speedup
  - Accelerations of 40g

Seek details

- Head switches comparable to short seeks
  - May also require head adjustment
  - Settles take longer for writes than for reads – Why?
- Disk keeps table of pivot motor power
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic “thermal recalibration”
  - But, e.g., ~500 ms recalibration every ~25 min bad for AV
- “Average seek time” quoted can be many things
  - Time to seek 1/3 disk, 1/3 time to seek whole disk

Sectors

- Disk interface presents linear array of sectors
  - Generally 512 bytes, written atomically (even if power failure)
- Disk maps logical sector #s to physical sectors
  - Zoning–puts more sectors on longer tracks
  - Track skewing–sector 0 pos. varies by track (why?)
  - sparing–flawed sectors remapped elsewhere
- OS doesn’t know logical to physical sector mapping
  - Larger logical sector # difference means larger seek
  - Highly non-linear relationship (and depends on zone)
  - OS has no info on rotational positions
  - Can empirically build table to estimate times
### Sectors
- **Disk interface presents linear array of sectors**
  - Generally 512 bytes, written atomically (even if power failure)
- **Disk maps logical sector #s to physical sectors**
  - *Zoning*—puts more sectors on longer tracks
  - *Track skewing*—sector 0 pos. varies by track (sequential access speed)
  - *Sparing*—flawed sectors remapped elsewhere
- **OS doesn’t know logical to physical sector mapping**
  - Larger logical sector # difference means larger seek
  - Highly non-linear relationship *(and depends on zone)*
  - OS has no info on rotational positions
  - Can empirically build table to estimate times

### Disk interface
- **Controls hardware, mediates access**
- Computer, disk often connected by bus (e.g., SCSI)
  - Multiple devices may contend for bus
- **Possible disk/interface features:**
  - **Disconnect from bus during requests**
  - **Command queuing:** Give disk multiple requests
    - Disk can schedule them using rotational information
  - **Disk cache used for read-ahead**
    - Otherwise, sequential reads would incur whole revolution
    - Cross track boundaries? Can’t stop a head-switch
- **Some disks support write caching**
  - But data not stable—not suitable for all requests

### SCSI overview [Schmidt]
- **SCSI domain consists of devices and an SDS**
  - Devices: host adapters & SCSI controllers
  - *Service Delivery Subsystem* connects devices—e.g., SCSI bus
- **SCSI-2 bus (SDS) connects up to 8 devices**
  - Controllers can have > 1 “logical units” (LUNs)
  - Typically, controller built into disk and 1 LUN/target, but “bridge controllers” can manage multiple physical devices
- **Each device can assume role of initiator or target**
  - Traditionally, host adapter was initiator, controller target
  - Now controllers act as initiators (e.g., copy command)
  - Typical domain has 1 initiator, ≥ 1 targets

### Executing SCSI commands
- **Each LUN maintains a queue of tasks**
  - Each task is dormant, blocked, enabled, or ended
  - Simple tasks are dormant until no ordered/head of queue
  - Ordered tasks dormant until no HoQ/more recent ordered
  - HoQ tasks begin in enabled state
- **Task management commands available to initiator**
  - Abort/terminate task, Reset target, etc.
- **Linked commands**
  - Initiator can link commands, so no intervening tasks
  - E.g., could use to implement atomic read-modify-write
  - Intermediate commands return status byte intermediate

### SCSI requests
- **A request is a command from initiator to target**
  - Once transmitted, target has control of bus
  - Target may disconnect from bus and later reconnect *(very important for multiple targets or even multitasking)*
- **Commands contain the following:**
  - **Task identifier**—initiator ID, target ID, LUN, tag
  - **Command descriptor block**—e.g., read 10 blocks at pos. N
  - Optional **task attribute**—simple, ordered, head of queue
  - Optional: output/input buffer, sense data
  - **Status byte**—good, check condition, intermediate, . . .

### SCSI exceptions and errors
- **After error stop executing most SCSI commands**
  - Target returns with check condition status
  - Initiator will eventually notice error
  - Must read specifics w. request sense
- **Prevents unwanted commands from executing**
  - E.g., initiator may not want to execute 2nd write if 1st fails
- **Simplifies device implementation**
  - Don’t need to remember more than one error condition
- **Same mechanism used to notify of media changes**
  - I.e., ejected tape, changed CD-ROM
Disk performance

- Placement & ordering of requests a huge issue
  - Sequential I/O much, much faster than random
  - Long seeks much slower than short ones
  - Power might fail any time, leaving inconsistent state
- Must be careful about order for crashes
  - More on this in next two lectures
- Try to achieve contiguous accesses where possible
  - E.g., make big chunks of individual files contiguous
- Try to order requests to minimize seek times
  - OS can only do this if it has a multiple requests to order
  - Requires disk I/O concurrency
  - High-performance apps try to maximize I/O concurrency
- Next: How to schedule concurrent requests

Scheduling: FCFS

- “First Come First Served”
  - Process disk requests in the order they are received
- Advantages
- Disadvantages

FCFS example

Shortest positioning time first (SPTF)

- Shortest positioning time first (SPTF)
  - Always pick request with shortest seek time
- Also called Shortest Seek Time First (SSTF)
- Advantages
- Disadvantages

Shortest positioning time first (SPTF)

- Shortest positioning time first (SPTF)
  - Always pick request with shortest seek time
- Also called Shortest Seek Time First (SSTF)
- Advantages
  - Exploits locality of disk requests
  - Higher throughput
- Disadvantages
  - Starvation
  - Don’t always know what request will be fastest
- Improvement?
**Shortest positioning time first (SPTF)**

- Shortest positioning time first (SPTF)
  - Always pick request with shortest seek time
- Also called Shortest Seek Time First (SSTF)

**Advantages**
- Exploits locality of disk requests
- Higher throughput

**Disadvantages**
- Starvation
- Don’t always know what request will be fastest

**Improvement: Aged SPTF**
- Give older requests higher priority
- Adjust “effective” seek time with weighting factor:
  \[ T_{\text{eff}} = T_{\text{pos}} - W \cdot T_{\text{wait}} \]

**“Elevator” scheduling (SCAN)**

- Sweep across disk, servicing all requests passed
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests

**Advantages**
- Takes advantage of locality
- Bounded waiting

**Disadvantages**
- Cylinders in the middle get better service
- Might miss locality SPTF could exploit

**CSCAN: Only sweep in one direction**

Very commonly used algorithm in Unix

Also called LOOK/CLOOK in textbook
- (Textbook uses [C]SCAN to mean scan entire disk uselessly)

**VSCAN(r)**

- Continuum between SPTF and SCAN
  - Like SPTF, but slightly changes “effective” positioning time
    - If request in same direction as previous seek: \( T_{\text{eff}} = T_{\text{pos}} \)
    - Otherwise: \( T_{\text{eff}} = T_{\text{pos}} + r \cdot T_{\text{max}} \)
  - when \( r = 0 \), get SPTF, when \( r = 1 \), get SCAN
  - E.g., \( r = 0.2 \) works well

**Advantages and disadvantages**
- Those of SPTF and SCAN, depending on how \( r \) is set

See [Worthington] for good description and evaluation of various disk scheduling algorithms
Flash memory

- Today, people increasingly using flash memory
- Completely solid state (no moving parts)
  - Remembers data by storing charge
  - Lower power consumption and heat
  - No mechanical seek times to worry about
- Limited # overwrites possible
  - Blocks wear out after 10,000 (MLC) – 100,000 (SLC) erases
  - Requires flash translation layer (FTL) to provide wear leveling, so repeated writes to logical block don’t wear out physical block
  - FTL can seriously impact performance
  - In particular, random writes very expensive [Birrell]
- Limited durability
  - Charge wears out over time
  - Turn off device for a year, you can easily lose data

Types of flash memory

- NAND flash (most prevalent for storage)
  - Higher density (most used for storage)
  - Faster erase and write
  - More errors internally, so need error correction
- NOR flash
  - Faster reads in smaller data units
  - Can execute code straight out of NOR flash
  - Significantly slower erases
- Single-level cell (SLC) vs. Multi-level cell (MLC)
  - MLC encodes multiple bits in voltage level
  - MLC slower to write than SLC

NAND Flash Overview

- Flash device has 2112-byte pages
  - 2048 bytes of data + 64 bytes metadata & ECC
- Blocks contain 64 (SLC) or 128 (MLC) pages
- Blocks divided into 2–4 planes
  - All planes contend for same package pins
  - But can access their blocks in parallel to overlap latencies
- Can read one page at a time
  - Takes 25 μs + time to get data off chip
- Must erase whole block before programming
  - Erase sets all bits to 1—very expensive (2 msec)
  - Programming pre-erased block requires moving data to internal buffer, then 200 (SLC)–800 (MLC) μs

Flash Characteristics [Caulfield’09]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>SLC</th>
<th>MLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density Per Die (GB)</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Page Size (Bytes)</td>
<td>2048+32</td>
<td>2048+64</td>
</tr>
<tr>
<td>Block Size (Pages)</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Read Latency (μs)</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write Latency (μs)</td>
<td>200</td>
<td>800</td>
</tr>
<tr>
<td>Erase Latency (μs)</td>
<td>2000</td>
<td>2000</td>
</tr>
</tbody>
</table>

40MHz, 16-bit bus

| Read b/w (MB/s) | 75.8 | 75.8 |
| Program b/w (MB/s) | 20.1 | 5.0 |

133MHz

| Read b/w (MB/s) | 126.4 | 126.4 |
| Program b/w (MB/s) | 20.1 | 5.0 |