Input/Output

Today
- Principles of I/O hardware & software
- I/O software layers
- Disks

Next
- Protection & Security
Operating Systems and I/O

- Two key operating system goals
  - Control I/O devices
  - Provide a simple, easy-to-use, interface to devices

- Problem – large variety
  - Data rates
  - Applications – what the device is used for
  - Complexity of control – a printer (simple) or a disk
  - Units of transfer – streams of bytes or larger blocks
  - Data representation – character codes, parity
  - Error condition – nature of errors, how they are reported, their consequences, …

- Makes a uniform & consistent approach difficult to get
Typical I/O Devices Data Rates

<table>
<thead>
<tr>
<th>Device</th>
<th>Data Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gigabit Ethernet</td>
<td>125 MB/sec</td>
</tr>
<tr>
<td>Graphic display</td>
<td>60 MB/sec</td>
</tr>
<tr>
<td>Hard disk</td>
<td>16.7 MB/sec</td>
</tr>
<tr>
<td>40x CDROM</td>
<td>6 MB/sec</td>
</tr>
<tr>
<td>Ethernet</td>
<td>1.25 MB/sec</td>
</tr>
<tr>
<td>Scanner</td>
<td>400 KB/sec</td>
</tr>
<tr>
<td>Laser printer</td>
<td>100 KB/sec</td>
</tr>
<tr>
<td>Modem</td>
<td>7 KB/sec</td>
</tr>
<tr>
<td>Mouse</td>
<td>100 B/sec</td>
</tr>
<tr>
<td>Keyboard</td>
<td>10 B/sec</td>
</tr>
</tbody>
</table>

Data rate
I/O Hardware - I/O devices

- I/O devices – roughly divided as
  - Block devices – stored info in fixed-size blocks; you can read/write each block independently (e.g. disk)
  - Character devices – I/O stream of characters (e.g. printers)
  - Of course, some devices don’t fit in here (e.g. clocks)

- I/O devices components
  - Device itself - mechanical component
  - Device controller - electronic component

- Controller
  - Maybe more than one device per controller
  - Converts serial bit stream to block of bytes
  - Performs error correction as necessary
  - Makes data available in main memory
I/O Controller & CPU Communication

- Device controllers have
  - A few registers for communication with CPU
    - Data-in, data-out, status, control, …
    - A data buffer that OS can read/write (e.g. video RAM)

- How does the CPU use that?
  - Separate I/O and memory space, each control register assigned an I/O port (a) – IBM 360
    
    ```
    IN REG, PORT
    ```
  - Memory-mapped I/O – first in PDP-11 (b)
  - Hybrid – Pentium (c) (graphic controller is a good example)

![Diagram](image)
Memory-mapped I/O

- **Pros:**
  - No special instructions needed
  - No special protection mechanism needed
  - Driver can be entirely written in C (how do you do IN or OUT in C?)

- **Cons:**
  - What do you do with caching? Disable it in a per-page basis
  - Only one AS, so all must check all mem. references “is it for me?”
    - Easy with single bus (a) but harder with dual-bus (b) arch
    - Possible solutions
      - Send all references to memory first
      - Snoop in the memory bus
      - Filter addresses in the PCI bridge (preloaded with range registers at boot time)

(a) [Diagram of architecture with three buses: CPU, Mem, I/O]

(b) [Diagram of architecture with high-bandwidth memory bus]
A more complex system

- Pentium system

- Level 1 in chip, level 2 off chip

- Industry Standard Architecture
  - Runs at 8.33MHz/2 bytes at a time: 16.7MB/sec

- Peripheral Component Interconnect
  - Runs at 66MHz/8 bytes at a time: 528MB/sec

- Pentium system
  - Running at 100MHz
  - Cache bus, Local bus, Memory bus
  - PCI bus
  - Available PCI slot
  - Available ISA slot
  - Modem, Sound card, Printer
Direct Memory Access (DMA)

- With or w/o memory-mapped I/O – CPU has to address the device controllers to exchange data
  - By itself, one byte at a time
  - Somebody else doing it instead – DMA

- Clearly OS can use it only if HW has DMA controller

- DMA operation
Some details on DMA

- One or more transfers at a time
  - Need multiple set of registers for the multiple channels
  - DMA has to schedule itself over devices served

- Buses and DMA can operate on one of two modes
  - Cycle stealing – as described
  - Burst mode (block) – DMA tells the device to take the bus for a while

- Two approaches to data transfer
  - Fly-by mode – just discussed, direct transfer to memory
  - Two steps – transfer via DMA; it requires extra bus cycle, but now you can do device-to-device transfers

- Physical (common) or virtual address for DMA transfer

- Why you may not want a DMA?

  *If the CPU is fast and there’s not much else to do anyway*
Interrupts revisited

- When I/O is done – interrupt by asserting a signal on a bus line
- Interrupt controller puts a # on address lines – index into interrupt vector (PC to interrupt service procedure)
- Interrupt service procedure ACK the controller
- Before serving interrupt, save context …
Interrupts revisited

Not that simple …

- Where do you save the state?
  - Internal registers? Hold your ACK (to protect you from another interrupt overwriting the internal registers)
  - In stack? You can get a page fault … pinned page?
  - In kernel stack? change to kernel mode ($$$ - change MMU context, invalid cache and TLB,…)

- Besides: pipelining, superscalar architectures, …

Ideally - a precise interrupt

- PC is saved in a known place
- All previous instructions have been fully executed
- All following ones have not
- The execution state of the instruction pointed by PC is known

The tradeoff – complex OS or really complex interrupt logic within the CPU (design complexity & chip area)
I/O software – goals & issues

- **Device independence**
  - Programs can access any I/O device w/o specifying it in advance

- **Uniform naming, closely related**
  - Name independent of device

- **Error handling**
  - As close to the hardware as possible (first the controller should try, then the device driver, …)

- **Buffering for better performance**
  - Check what to do with packets, for example
    - Decouple production/consumption

- **Deal with dedicated (tape drives) & shared devices (disks)**
  - Dedicated dev. bring their own problems – deadlock?
Ways I/O can be done (OS take)

- **Programmed I/O**
  - Simplest – CPU does all the work
  - CPU basically pools the device
  - … and it is tied up until I/O completes

- **Interrupt-driven I/O**
  - Instead of waiting for I/O, context switch to another process & use interrupts

- **Direct Memory Access**
  - Obvious disadvantage of interrupt-driven I/O?
    - An interrupt for every character
  - Solution: DMA - Basically programmed I/O done by somebody else
Three techniques for I/O

1. Programmed I/O
   - Issue read command to I/O module
   - Read status of I/O module
   - Read word from I/O module
   - Write word into memory
   - Check status
     - Ready
     - Error
   - Done?
     - Yes: Programmed I/O
     - No

2. Interrupt-driven I/O
   - Issue read command to I/O module
   - Read status of I/O module
   - Read word from I/O module
   - Write word into memory
   - Check status
     - Ready
     - Error
   - Done?
     - Yes: Interrupt-driven I/O
     - No

3. Direct Memory Access
   - Issue read command to I/O module
   - Read status of I/O module
   - Read word from I/O module
   - Write word into memory
   - Check status
     - Ready
     - Error
   - Interrupt
     - Do something else
     - DMA → CPU
   - Done?
     - Yes: Interrupt-driven I/O
     - No
I/O software layers

- I/O normally implemented in layers

<table>
<thead>
<tr>
<th>User-level I/O software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device-independent OS software</td>
</tr>
<tr>
<td>Device driver</td>
</tr>
<tr>
<td>Interrupt handlers</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>

- Interrupt handlers
  - Interrupts – an unpleasant fact of life – hide them!
  - Best way
    - Driver blocks (semaphores?) until I/O completes
    - Upon an interrupt, interrupt procedure handles it before unblocking driver
Layers - Device drivers

- Different device controllers – different registers, commands, etc → each I/O device needs a device driver

- Device driver – device specific code
  - Written by device manufacturer
  - Better if we have specs
  - Clearly, it needs to be reentrant
  - Must be included in the kernel (as it needs to access the device’s hardware) - How do you include it?
    - Is there another option?
  - Problem with plug & play
Layers - Device-independent SW

Some part of the I/O SW can be device independent

- Uniform interfacing with drivers
  - Fewer modifications to the OS with each new device
  - Easier naming (/dev/disk0) – major & minor device #s in UNIX (kept by the i-node of the device’s file)
  - Device driver writers know what’s expected of them

- Error reporting
  - Some errors are transient – keep them low
  - Actual I/O errors – reporting up when in doubt

- Allocating & releasing dedicated devices
- Providing a device-independent block size
- Buffering
Buffering

A process reading data from a modem

- **Unbuffered input**
  - Wake up the process when the buffer is full

- **Buffering in the kernel, copying to user space**
  - When kernel-buffer is full page with user buffer is brought in & copy done at once

- **Double buffering in the kernel**
  - What happen to characters arriving while the user page is brought in?

- **Careful – nothing is free, think of buffering and host-to-host communication**
Disk hardware

- Disk organization
  - Cylinders – made of vertical tracks
  - Tracks – divided into sectors
  - Sectors – minimum transfer unit

- Simplified model - careful with specs
  - Sectors per track are not always the same
  - Zoning – zone, a set of tracks with equal sec/track

- Hide this with a logical disk w/ constant sec/track
IBM 2314

- Announced April 1965
- Eight disk drives plus a spare one and a control unit
- Capacity: 8x29MB
- Average access time: 60msec
- Data rate: 312KB/sec
## Disk hardware nowadays

<table>
<thead>
<tr>
<th>Characteristics (All 512B/sector)</th>
<th>Seagate Cheetah 15-36LP</th>
<th>Seagate Barracuda 36ES</th>
<th>Toshiba HDD1242</th>
<th>IBM Microdrive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application</strong></td>
<td>High-performance server</td>
<td>Entry-level desktop</td>
<td>Portable</td>
<td>Handheld</td>
</tr>
<tr>
<td><strong>Capacity</strong></td>
<td>36.7GB</td>
<td>18.4GB</td>
<td>5GB</td>
<td>1GB</td>
</tr>
<tr>
<td><strong>Minimum seek time</strong></td>
<td>0.3ms</td>
<td>1.0ms</td>
<td>-</td>
<td>1.0ms</td>
</tr>
<tr>
<td><strong>Average seek time</strong></td>
<td>3.6ms</td>
<td>9.5ms</td>
<td>15ms</td>
<td>12ms</td>
</tr>
<tr>
<td><strong>Spindle speed</strong></td>
<td>15K rpm</td>
<td>7.2K rpm</td>
<td>4.2K rpm</td>
<td>3.6K rpm</td>
</tr>
<tr>
<td><strong>Average rotational delay</strong></td>
<td>2ms</td>
<td>4.17ms</td>
<td>7.14ms</td>
<td>8.33ms</td>
</tr>
<tr>
<td><strong>Max. transfer rate</strong></td>
<td>522-709MB/s</td>
<td>25MB/s</td>
<td>66MB/s</td>
<td>13.3MB/s</td>
</tr>
<tr>
<td><strong>Sector per track</strong></td>
<td>485</td>
<td>600</td>
<td>63</td>
<td>-</td>
</tr>
<tr>
<td><strong>Tracks per cylinder</strong></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Cylinders</strong></td>
<td>18,479</td>
<td>29,851</td>
<td>10,350</td>
<td>-</td>
</tr>
</tbody>
</table>
RAIDs

- Processing and I/O - parallelism
- Redundant Array of Inexpensive Disks & SLED  Single Large Expensive Disks

**RAID 0:** Striping w/o redundancy

**RAID 1:** Disk mirroring

**RAID 2:** Memory-style error-correcting-code organization; striping bits across disk & add ECC
RAIDs

RAID 3: Bit-interleaved parity organization. If one sector is damaged, you know which one; used the parity to figure out what the lost bit is.

RAID 4: Block-interleaved parity organization. Like 3 but at block level.

RAID 5: Block-interleaved distributed parity organization. Like 4 but distributed the parity block.
Disk formatting

- **Low-level formatting**
  - Sectors – [preamble, to recognize the start + data + ecc]
  - Spare sectors for replacements
  - ~20% capacity goes with it
    - Sectors and head skews to deal with moving head
    - Interleaving to deal with transfer time

- **Partitioning** – multiple logical disks – sector 0 holds master boot record (boot code + partition table)

- **High-level formatting**
  - Boot block, free storage admin, root dir, empty file system
Disk arm scheduling

- Time to read/write a disk block determined by
  - Seek time – dominates!
  - Rotational delay
  - Actual transfer time
- Sequence of request for blocks → scheduling
- Some algorithms
  - FCFS (FIFO) – not much you can do
  - Shortest Seek Time First (SSTF)
  - Elevator algorithm or SCAN
  - C-SCAN - circular scan
  - LOOK variation
- Reading beyond your needs
and the obvious problem is ...
SSTF

As SJF, possible starvation
Assuming a uniform distribution of requests, where’s the highest density when head is on the left?
C-SCAN

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

- Cool, but no need to be blind
Look for a request before moving in that direction
Next time

- Protecting access to your system and paying attention to the system’s environment
- Final review and a taste of systems research