I/O and Devices: H/W S/W Interaction

Overview

- Basic System Structure
- Devices: H/W and S/W overview
- Controlling Devices: Problems + Solutions
- Case Study: Disks

Themes

- Overlap + Parallelism
- Buffers, CPU cost
- Abstractions
- Fairness vs. Performance

Review: Why should OS manage I/O devices?

(What are the roles of OS?)

Abstractions: OS as virtual machine
  => consistent interface to many different devices
    => 2 levels: internal to rest of OS and to user via FS
  Tension: too general = poor performance
        lack of feature exploitation

Resource Manager: OS as scheduler/multiplexer
  => arbiter of system resources
  Tension: fairness vs. performance of I/O requests

Protection: OS as secure vm
  => must same between legal users, disallow illegal uses

Summary: Who gets what when

Today's Focus: I/O

- why care about I/O?
  (after all, processors are the coolest part of computer systems)

1) Without I/O programs would produce same result each time
   O: no point in running code

2) I/O performance is increasingly important

   Amdahl's Law
   = what you do here matters

   => Processors getting fast much faster

3) Storage + Networking:
   Storing your data
   Access to remote data
   "Internet is all about!
   (not processing)

I/O is where the action is!"
Basic System Structure

Why have a peripheral bus?
- Closer to processor, lower the latency
- Faster the bus, shorter the bus (electrical properties)
  ⇒ fewer devices / bus

Device Types

<table>
<thead>
<tr>
<th>Display</th>
<th>I/O?</th>
<th>B/c?</th>
<th>S/c?</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>C</td>
<td>C</td>
<td>100s MB/s</td>
</tr>
<tr>
<td>Network Interface (NI)</td>
<td>I/O</td>
<td>C</td>
<td>C</td>
<td>1/10s-100s</td>
</tr>
<tr>
<td>Disks / Tapes</td>
<td>I/O</td>
<td>S</td>
<td>C</td>
<td>10s MB/s</td>
</tr>
<tr>
<td>Keyboard, Mouse</td>
<td>I</td>
<td>C</td>
<td>C</td>
<td>bytes/sec</td>
</tr>
</tbody>
</table>

How to categorize?
- Input / Output / Both
- Block / Character
- Storage / Communication
- Rate of operation
Devices: The H/W side

"care + feeding"

Device may require lots of attention during operation

Observe low-level progress, provide detailed cmds, correct minor errors (e.g., disk read/writes)

⇒ put some of this into H/W itself ⇒ device controller

CPU, mem, etc. ⇒ "computer w/ in
the computer"

⇒ expose cmds, H/W executes them (e.g., in the program)

Basic operation

⇒ check status, wait until free
⇒ put data in data register(s)
⇒ put cmd in cmd reg
  (device executes cmd)
⇒ check status
  if busy, keep checking
⇒ all done? yes

No = Problems:

⇒ lots of devices out there, how to simplify use?
⇒ how to get processor to talk to devices?
⇒ how to do so efficiently? (overlap)
Problem: lots of devices out there, how to simplify use? not just for apps, but for rest of OS itself! (characteristic of systems in general)

Internal Abstraction Layer
Uniform interface to similar devices (e.g. disk => block_read block_write)

S/W takes abstract requests => specific H/W register manipulations

What's this S/W called? Device driver

How to build file system then?

File System
device interface

Block read/write

(sergate, IBM, etc.)

=> simplifies use, but at what cost

Implementation Notes
> Old days: new driver => recompile, reboot entire OS
now: loadable on the fly "plug & play"

> Linux: over half of source code is in drivers

Problem: how to comm. w/ devices?

Special I/O instructions only valid in kernel mode was popular in mainframes (no longer) => store regx, [m]

Memory-mapped I/O read + write special memory addresses protect by placing in KVM or PM
simple, general, widely used

=> H/W S/W interface
Problem: How to interact efficiently?

Control: How to know when event is complete?

- **Polling**
  - Handshake by setting, clearing flags
    - Plus: simple
    - Minus: CPU cycles are wasted, busy-waiting
      - (if not attentive, may lose data)

- **Interrupts**
  - Handle events asynchronously
  - Device asserts interrupt request line when device is ready
  - CPU: jump to correct interrupt service routine (ISR)
    - Interrupt vector: table of handler addresses
      - Index by interrupt #
    - Plus: frees CPU from checking
    - Minus: could be costly - context switch into handler routine

=> When to use polling, when interrupts?

  - => Depends on speed of device
    - If slow, interrupts (allows OS to switch to other job)
  => OVERLAP
    - If quick, polling (cost of context switch avoided)
Problem (cont.): How to do so efficiently?

Data

- Programmed I/O (PIO)
  CPU moves every byte of data to/from device
  e.g., block read from disk, sitting in controller memory
  CPU does device \( \rightarrow \) mem \( \rightarrow \) copy

  - Pro: simple
  - Con: CPU overhead

- Direct Memory Access (DMA)
  Offload work to special-purpose transfer engine
  CPU sets up DMA
  set of addresses for src, dest, xfer size
  DMA controller handles transfer interrupts CPU when finished

Now, true overlap is possible (Async)

\[
\begin{array}{c}
\text{CPU} \xrightarrow{P_1} \text{PIO} \xrightarrow{P_2} \text{Mem} \xrightarrow{P_3} \text{Disk} \xrightarrow{\text{IO busy}} \text{complete}
\end{array}
\]
Case Study: Disks (today and tomorrow)

H/W structure:
Important to understand so we can design better file system!
(general rule: understand H/W to enable you to build S/W)

To read/write block:

- Seek: Position arm/head over correct cylinder (accelerate, coast, decel., settle)
- Rotational Delay: Wait for right block/sector to rotate underneath
- Transfer: When right block is there, transfer to device memory

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

Typical "modern" disk
IBM 92X

\[
\begin{align*}
\text{seek} & : 3 \to 18 \text{ ms} \\
\text{rot} & : 0 \to 12 \text{ ms} \\
\text{transfer} & : 12 \to 20 \text{ MB/s}
\end{align*}
\]

\( \text{e.g. small} \quad \text{large} \)

Why is outer track B/W higher than inner track?
Disk Scheduling, etc.

T_{io} = T_s + T_r + T_{transfer}

> real work

"How to make disks run fast"

> Long, sequential transfers

> Avoid seeks, rotations

T_{seek}, T_{rot} \rightarrow 0 \quad or \quad T_{transfer} \gg T_{seek}, T_{rot}

Problem

Queue of requests: read/write B_x

What order to process?

Theme: fairness/perf

Most basic: FCFS (first come, first served)

Service in order? (Pathological case?)

SSTF (shortest-seek-the-first)

Solves previous problem

But, adds a new one: starvation

Scan (Elevator)

Look: variant (don't go to end)

Starvation: not a problem

New problem:

C-Scan

1) New request pile up

2) Z-scan delay for block (potentially)

Oops! All seek-centric

New research takes rotational delay into account too!