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

Overview

> Basic System Structure
+ device types, rates
> Devices : H/W (and S/W) overview
> Controlling Devices : Problems + Solutions
> Case Study : Disks
overview, scheduling, trends

Themes

> Overlap + Parallelism
Buffering, CPU cost
> Abstractions
> Fairness v. 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 / multiplexor
⇒ Arbiter of system resources
Tension: fairness vs. performance of I/O requests

Protection: OS as secure VM
⇒ must share between legal users, disallow illegal uses

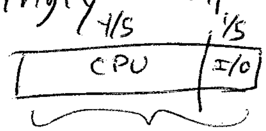
summary: who gets what when

Today's Focus: I/O

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

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

2) I/O performance: increasingly important
Amdahl's Law



Processors getting fast much faster!

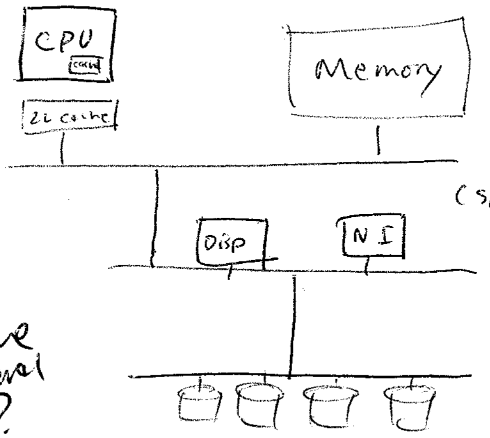
CPU gets 8x in 3 years
⇒ % of time in I/O?

⇒ what you do here matters

3) Storage + Networking:
storing your data
access to remote data
} what Internet is all about! (not processing)

→ I/O is where the action is!!

Basic System Structure



		Rate	Standard?
(DRAM)	Memory Bus	~ GB/s	(proprietary) specific
(sp.	I/O Bus	~ 100 MB/s	standard (PCI)
	Peripheral Bus	~ 10s of MB/s (5-80)	standard (SCSI)

Why have peripheral bus?

Observations:

- Closer to processor, lower the latency
- Faster the bus, shorter the bus (electrical properties)
  - ⇒ fewer devices / bus

Device Types

	I/O?	B/C?	S/C?	Rate?
Display	O	C	C (w/ human)	100s MB/s
Network Interface (NI)	I/O	C	C (w/ computer)	1/10s/100s
Disks, Tapes	I/O	B	S	10s MB/s
Keyboard, Mouse	I	C	C (w/ human)	bytes/sec

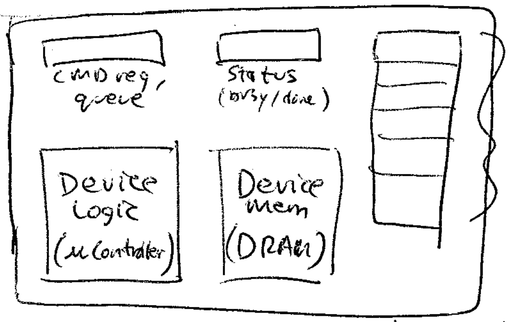
How to categorize?

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 (move disk arm to this location), correct minor errors (disk read bits)
  - => put some of this into H/W itself => device controller



Data registers

CPU, mem, etc =>

"Computer w/in the computer"!

=> expose cmds, H/W executes them (a little 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
if not, all done, do another I/O?

} exactly how we will discuss later

=> All done? no

~~Problem #1~~
~~Lots of devices out there~~
~~(all w/ different interfaces, registers, etc)~~

Some problems!

> Lots of different devices out there, how to simplify use?
> How to get processor to talk to devices?
> How to do so (the above) efficiently? (overlap)

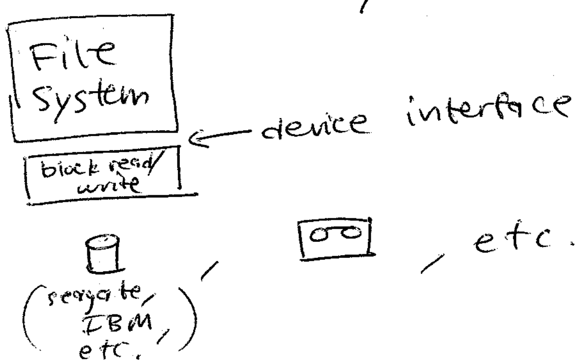
Problem: Lots of devices out there, how to simplify use?
not just for apps, but for rest of OS itself!
(characteristic of systems in general)

=> 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?



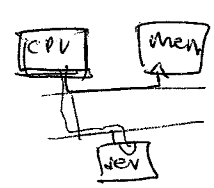
=> simplifies use, but at what cost

=> 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)

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

=> store reg_x, $[M_y]$



=> H/W S/W interface

Problem : How to interact efficiently?

Control : How to know when event is complete?

> Polling :

Handshake by setting, clearing flags

```
start-~~event~~action()
while (!done)
    check-status
```

{ Plus : simple
{ Minus : CPU cycles are wasted busy-waiting
(if not attentive, may lose data)

> Interrupts

Handle events asynchronously

1) (start-~~event~~action()
notify-me-when-done())

Device asserts interrupt request line
when device is ready

2) handler

CPU : jump to correct Int. 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 ctxt switch avoided)

Problem (cont.) : How to do so efficiently?

· Even w/ interrupts, CPU may have to move data to an fro

Data

⇒ Programmed I/O (PIO)

CPU moves every byte of data to/fro device

e.g. block read from disk, sitting in controller memory

CPU does device mem ⇒ main mem copy

{ Pro : simple
{ Con : CPU overhead!

⇒ Direct Memory Access (DMA)

Offload work to special-purpose transfer engine

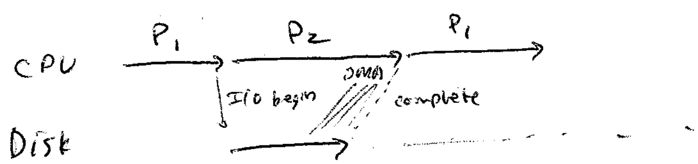
CPU sets up DMA

set up addresses for src, dest, xfer size

DMA controller handles ~~xfer~~

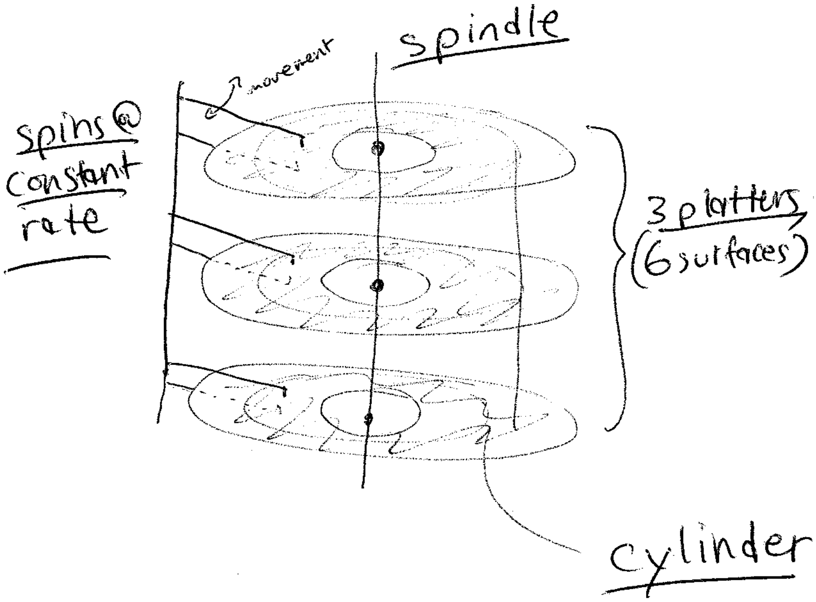
interrupts CPU when finished

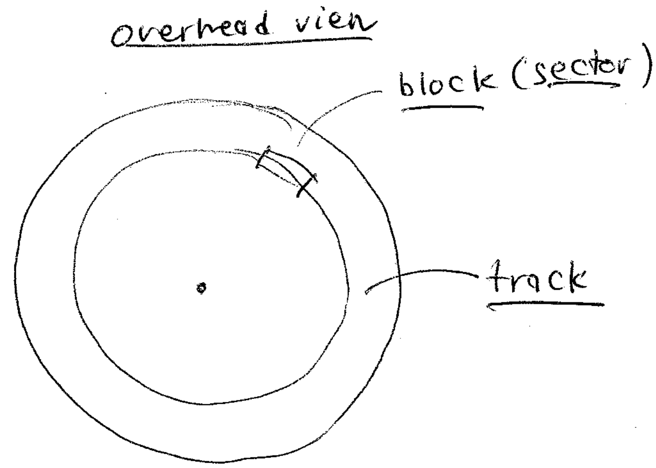
⇒ Now, true overlap is (almost) possible



Case Study: Disks (today and tomorrow)
(storage)

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

$$\Rightarrow T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

only real work

Typical "modern" disk

IBM 9ZX

- seek : 3 → 18 ms
- rot : 0 → 12 ms
- transfer : 12 → 20 MB/s

e.g. small large

why is outer track B/W higher than inner track?

Disk Scheduling, etc.

$T_{I/O} = T_S + T_R + \underbrace{T_{transfer}}_{\text{real work}}$

"How to make disks run fast"

- FS layout: > Long, sequential transfers
- layout + request scheduling: > Avoid seeks, rotations

} treat disk like a tape!

$T_{seek}, T_{rot} \Rightarrow 0$ OR $T_{transfer} \gg T_{seek}, T_{rot}$

Problem

queue of requests: read/write B_x
what order to process?

THEME: fairness/perf

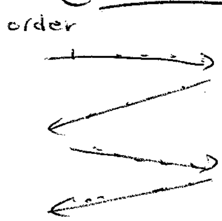
Most basic: FCFS (first come, first served)

service in order
problem? (pathological case?)

SSTF (shortest-seek-time first)

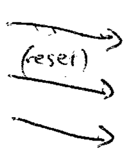
solves previous problem
but, adds a new one: ~~livelock~~ starvation

Scan (Elevator)   Look: variant (don't go to end)



starvation: not a problem
new problem:

C-SCAN



1) new request pile up
2) 2-scan delay for block (potentially)

oops! all seek centric
new research takes rotational delay into account too!