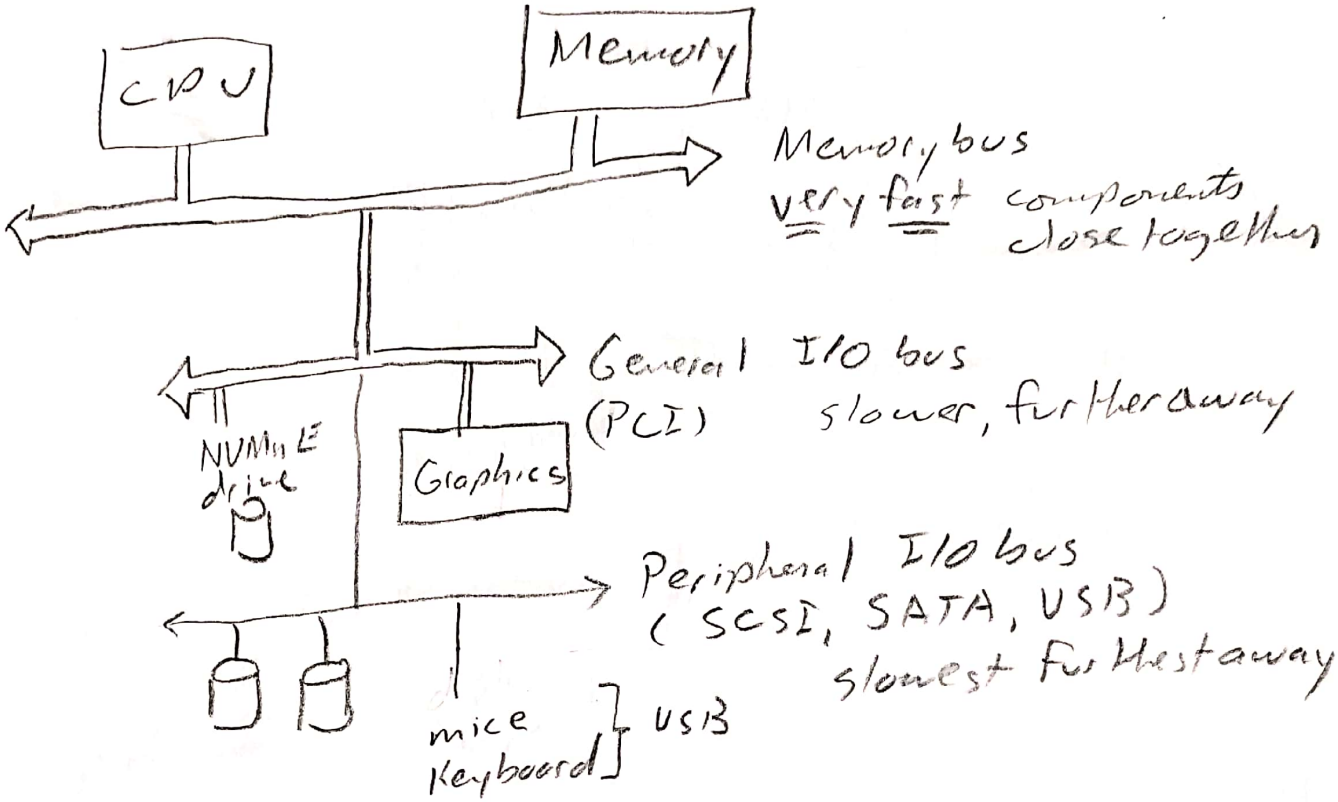
I/O Devices (chapter 36)

How to integrate I/O into systems?



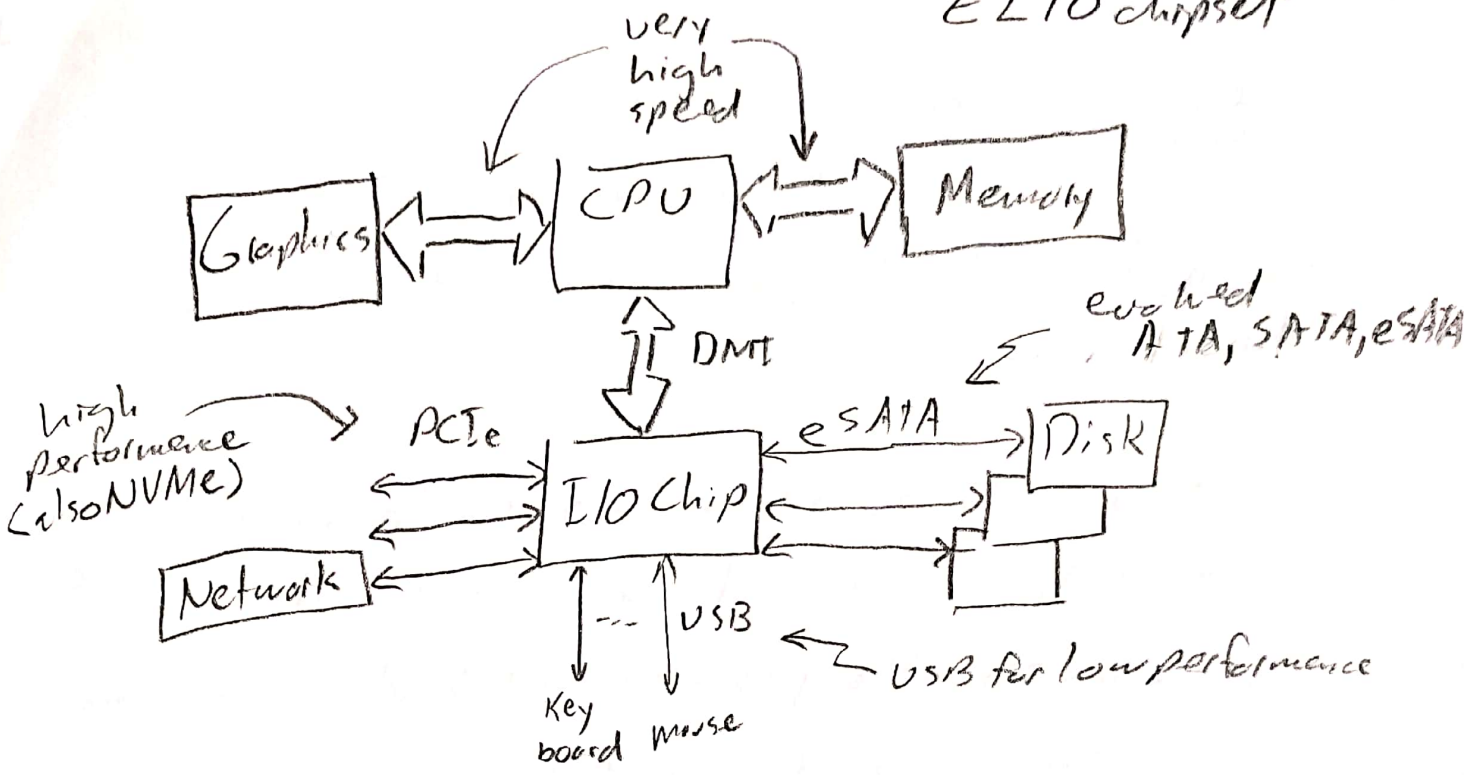
Why do we need this?

physics: the faster a bus is the shorter it must be.
(why memory is always close to CPU)

costs: higher clocks, harder to work around, cross coupling & EMI effects go up as switching speeds increase. Harder to engineer.

High performance, less of, closer
low performance, more of, further away

Z270 chipset



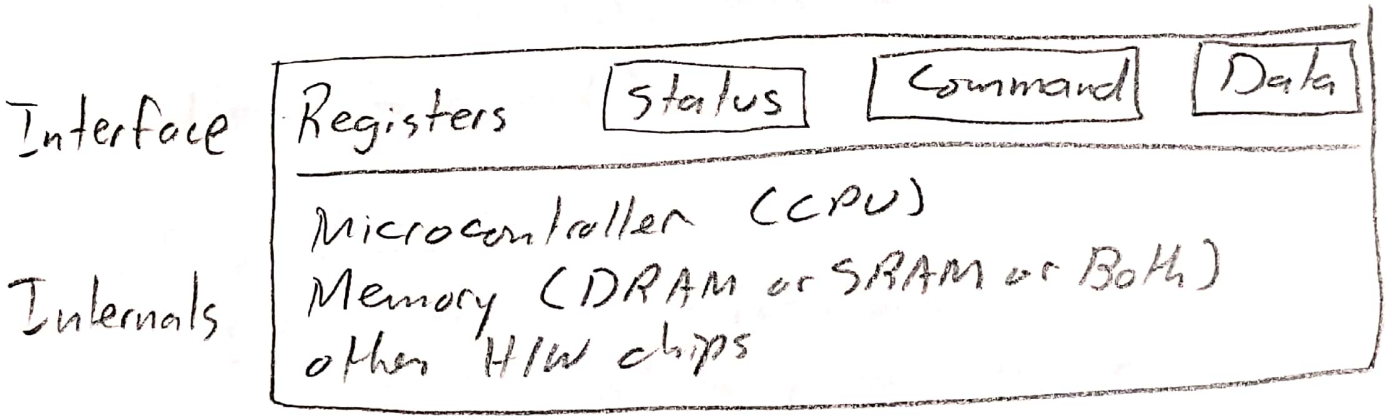
Approximation of Intel Z270 chipset.

- High performance up top
- CPU connects to I/O chip via Intel proprietary DMI (Direct Media Interface) bus.
- Note single chip handles all the bus protocols

generic Device

① - has hardware interface - allows the system software to control it.
- consists of interface + protocol for typical interaction

② internal structure - control of the device,
simple - 1 or a few HW chips to implement functionality
complex - a simple CPU, memory other device chips
ex. Modern RAID controllers have thousands of lines of firmware.



status - what's going on
command - tell it what to do
Data - data that goes in & out

OS controls device by reading/writing

```
while (status == busy)        <- polling!
    ; // wait until not
write Data to Data reg
write a command to Command reg.
    (starts the device & executes the command)
while (status == busy)        <- polling
    ; // wait until device done w/ request.
```

Probs?

Polling - could waste lots of CPU time waiting on slow device.

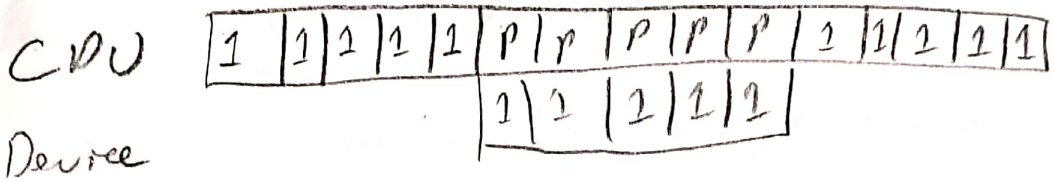
How to avoid?

Solve - Interrupts

OS issues request for work
puts the calling process to sleep
switch to other process
when device finishes work
device raises H/W interrupt
CPU jumps to OS at ISR {Interrupt Service Routine}
ISR finishes request {maybe reads data & maybe error code}
and wakes the process waiting for I/O

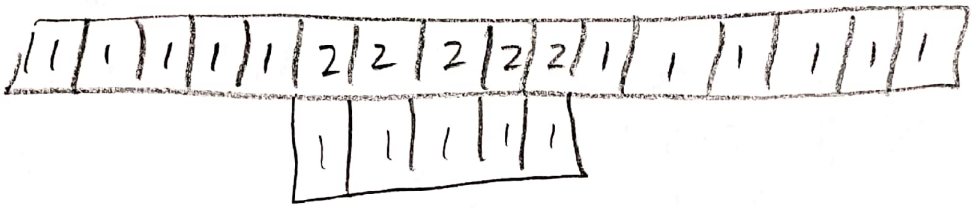
overlap

p=polling



above process 1 runs, issues request to read from disk, process 1 polls waiting for data

Interrupts



OS runs 2 while waiting for disk
to service 1's request
when disk read is done, device interrupts CPU, 1 reawakened by OS, runs again

---

Interrupts are not always the best solution!

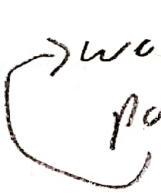
if device performs it's task quickly
then 1st poll finds it done.
Better to poll (few clock cycles) verses
context switches + ISR (100,000 cycles)

So interrupts make sense for slow devices

If you don't know speed maybe use hybrid, poll for a bit then switch to interrupts

Livelock interrupt problem

- arises in networks - huge stream of incoming packets, each generating interrupt
- possible for system to only process interrupts (livelock)
- maybe better to...



} know there going to be a job(s) don't bother w/ context switch overhead.

- or coalesce

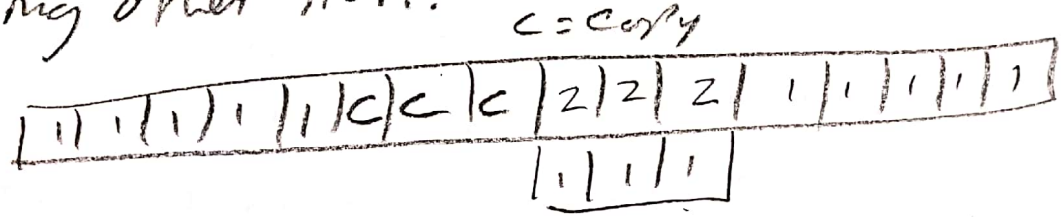
if device needs to gen an interrupt wait a bit & see if more tasks complete then generate one interrupt for all. (multiple → single) lowers int processing overhead.

but wait to long increase latency (time to service interrupt)

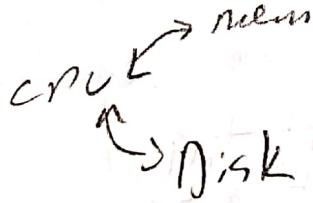
Better Data movement

DMA (Direct Memory Access)

- so far CPU directly involved with all transfer programmed IO (PIO)
- to transfer large chunk of data to device CPU involved with each Xfer,
- waste a lot of CPU time when it could be doing other stuff.



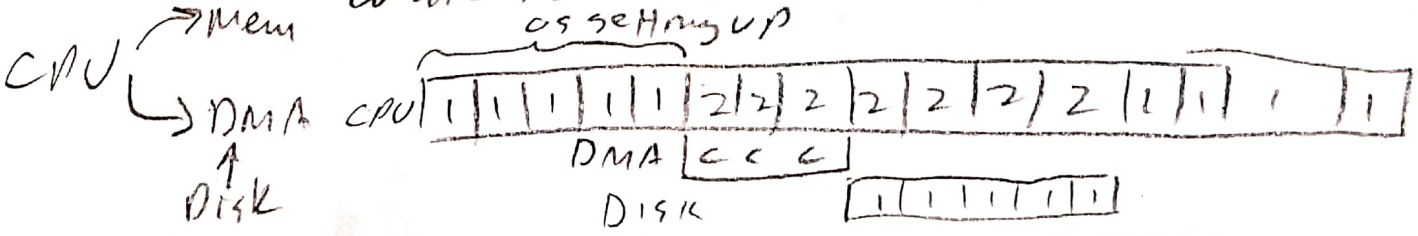
CPU copies from mem to disk



initiates copy from mem to device
device writes to disk

Fix

OS programs DMA engine tells it
where data is in memory
how much to copy
which device to send to



copying handled by DMA controller.

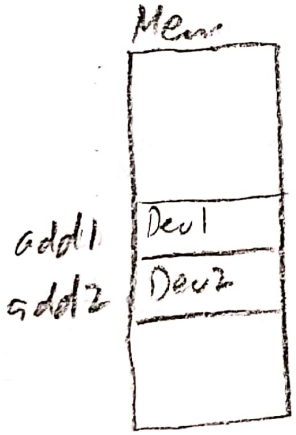
How does OS communicate with Device?

2 ways

① Explicit I/O instructions

② Mem Mapped I/O

- How makes device registers available as if they were mem locations.



- to access register, OS issues a load (read) or store (write) command at the device address, How routes request to device not mem.

Finally

- How to fit devices, (each having unique characteristics)
- ex filesystem - would like it to work on SCSI, IDE, USB/Keychain disks. like filesystem to be oblivious to details of particular filesystem

Abstraction -

OS

↓

general API (open, read, write close etc...)

↑ ↑

device driver, implements all for a particular bit of hardware

OS sees only generic API, just reads & writes to it.

Vendors supply device driver for their H/W

Consequence -- lots of devices, lots of drivers

- sometimes written by amateurs, run as kernel can cause system crash
- all must be available on system even though you only use a few
  - so most OS consists of mostly unused device drivers. (Linux 70% of OS code is device drivers)