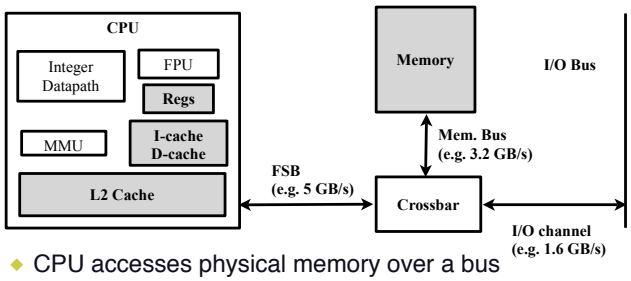
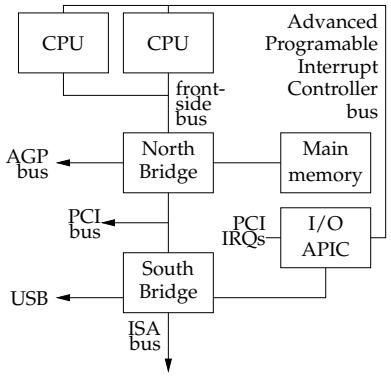
CS 140 – Summer 2008 – Handout #21: I/O and Device Drivers

Memory and I/O buses



- CPU accesses physical memory over a bus
- Devices access memory over I/O bus with DMA
  (DMA = Direct Memory Access)
- Devices can appear to be a region of memory

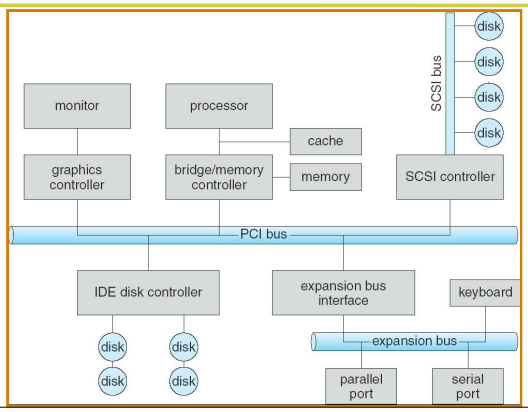
Realistic PC architecture



What is memory?

- SRAM – Static RAM
  - like two NOT gates circularly wired input-to-output
  - 4–6 transistors/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 sense amp. to decide if bit 1 or 0
  - Must re-write charge after (destructive) reading, periodically refresh
- VRAM – "Video RAM"
  - Dual ported, can write while someone else reads
  - uncommon in modern PCs (regular DRAM won)

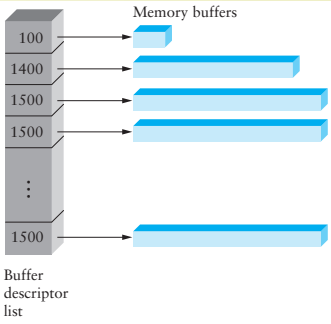
What is I/O bus (e.g. PCI)?



Communicating with a device

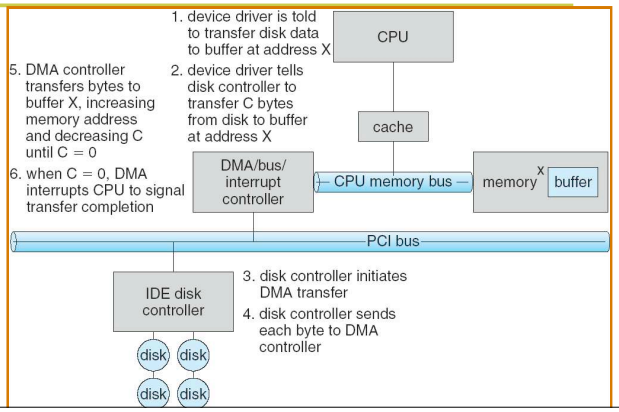
- Memory-mapped device registers
  - certain *physical* addresses correspond to device registers
  - Load/store gets status/sends instructions (not real memory!)
- Device memory
  - device may have memory/registers OS can write to directly!
- Special I/O instructions
  - Some CPUs have special I/O instructions (e.g. x86 in, out)
  - 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 need to "poke" card by writing to register
  - Overlaps computation with moving data over I/O bus

DMA buffers

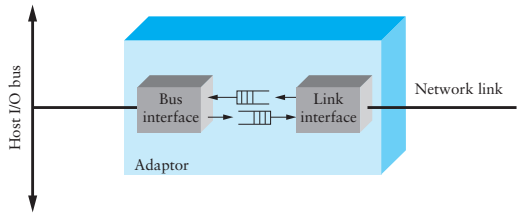


- List of buffer locations in main memory
- Card reads list, then accesses buffers w/DMA

Example 1: IDE disk with DMA



Example 2: Network Interface Card



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

Device Driver Architecture

- Device driver provides set of entry points for kernel
  - common interface -> simplicity, pluggability
- Unix: driver as a "file", `/dev/something`
  - `open(), close(), read(), write()`
  - everything else: `ioctl()`
  - block devices (e.g. disk), character devices (e.g. terminal)
  - powerful idea: namespace != implementation
    `/proc` filesystem, Portal/FUSE, Plan 9, etc.
- Question: How should driver synchronize with card?
  - e.g. need to know when transmit buffers free or packets arrive, or when disk request is complete?

Simplest approach: Polling

- Periodically query hardware for status
  - Sent a packet? Keep asking if buffer is free.
  - Waiting to receive? Keep asking if card has a packet.
  - Disk I/O? Keep checking disk ready bit.
- Disadvantages of polling
  - Takes up CPU time which could otherwise be used
  - Either busy-waiting (responsive but inefficient)
  - ... or periodic checks
    - schedule for time in the future
    - means you might just miss it
    -> longer latency for everything

Advanced? approach: Interrupt-driven drivers

- Instead, ask card to interrupt CPU on events
  - Interrupt handler runs at high priority
  - Asks card what happened (buffer free, new packet, etc.)
  - This is what most general-purpose OSes do
- Great for disks!
  - synchronous, predictable, rate limited...
- 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!
  - Worst case: 100% of time in interrupt handler – *receive livelock*
  - To avoid livelock/DoS: adaptive switching to/from polling