Memory and I/O buses

- CPU accesses physical memory over a bus
- Devices access memory over I/O bus with DMA
- Devices can appear to be a region of memory
Realistic PC architecture

- CPU
- CPU
- Advanced Programable Interrupt Controller bus
- North Bridge
- AGP bus
- PCI bus
- South Bridge
- USB
- ISA bus
- Main memory
- I/O APIC
- PCI IRQs
- Front-side bus
What is memory?

- **SRAM – Static RAM**
  - Like two NOT gates circularly wired input-to-output
  - 4–6 transistors per 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 comparator to decide if bit 1 or 0
  - Must re-write charge after reading, and periodically refresh

- **VRAM – “Video RAM”**
  - Dual ported, can write while someone else reads
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 OS can write to directly on other side of I/O bus

• Special I/O instructions
  - Some CPUs (e.g., x86) have special I/O instructions
  - 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 then need to “poke” card by writing to register
  - Overlaps unrelated computation with moving data over (typically slower than memory) I/O bus
• Include list of buffer locations in main memory
• Card reads list then accesses buffers (w. DMA)
  - Allows for scatter/gather I/O
Example: Network Interface Card

- Link interface talks to wire/fiber/antenna
  - Typically does framing, link-layer CRC
- FIFOs on card provide small amount of buffering
- Bus interface logic uses DMA to move packets to and from buffers in main memory
Example: IDE disk with DMA

1. Device driver is told to transfer disk data to buffer at address X
2. Device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. Disk controller initiates DMA transfer
4. Disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. When C = 0, DMA interrupts CPU to signal transfer completion
Driver architecture

- Device driver provides several entry points to kernel
  - Reset, ioctl, output, interrupt, read, write, strategy …
- How should driver synchronize with card?
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete
- One approach: Polling
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set
- Disadvantages of polling
  - Can’t use CPU for anything else while polling
  - Or schedule poll in future and do something else, but then high latency to receive packet or process disk block
Interrupt driven devices

- Instead, ask card to interrupt CPU on events
  - Interrupt handler runs at high priority
  - Asks card what happened (xmit buffer free, new packet)
  - This is what most general-purpose OSes do

- **Bad under high network packet arrival rate**
  - Packets can arrive faster than OS can process them
  - Interrupts are very expensive (context switch)
  - Interrupts handlers have high priority
  - In worst case, can spend 100% of time in interrupt handler and never make any progress – *receive livelock*
  - Best: Adaptive switching between interrupts and polling

- **Very good for disk requests**

- Rest of today: Disks (network devices in 1.5 weeks)
Anatomy of a disk

- **Stack of magnetic platters**
  - Rotate together on a central spindle @3,600-15,000 RPM
  - Drive speed drifts slowly over time
  - Can’t predict rotational position after 100-200 revolutions

- **Disk arm assembly**
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - Arms contain disk heads—one for each recording surface
  - Heads read and write data to platters
Disk
Disk
Disk
Storage on a magnetic platter

- Platters divided into concentric tracks
- A stack of tracks of fixed radius is a cylinder
- Heads record and sense data along cylinders
  - Significant fractions of encoded stream for error correction
- Generally only one head active at a time
  - Disks usually have one set of read-write circuitry
  - Must worry about cross-talk between channels
  - Hard to keep multiple heads exactly aligned
Cylinders, tracks, & sectors

- track $t$
- sector $s$
- cylinder $c$
- platter
- arm assembly
- spindle
- read-write head
- arm
- rotation
Disk positioning system

- Move head to specific track and keep it there
  - Resist physical socks, imperfect tracks, etc.

- A *seek* consists of up to four phases:
  - *speedup*—accelerate arm to max speed or half way point
  - *coast*—at max speed (for long seeks)
  - *slowdown*—stops arm near destination
  - *settle*—adjusts head to actual desired track

- Very short seeks dominated by settle time (\(\sim 1\) ms)
- Short (200-400 cyl.) seeks dominated by speedup
  - Accelerations of 40g
Seek details

- **Head switches comparable to short seeks**
  - May also require head adjustment
  - Settles take longer for writes than reads

- **Disk keeps table of pivot motor power**
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic “thermal recalibration”
  - 500 ms recalibration every 25 min, bad for AV

- **“Average seek time” quoted can be many things**
  - Time to seek 1/3 disk, 1/3 time to seek whole disk,
Sectors

- Disk interface presents linear array of *sectors*
  - Generally 512 bytes, written atomically

- Disk maps logical sector #s to physical sectors
  - *Zoning*—puts more sectors on longer tracks
  - *Track skewing*—sector 0 pos. varies by track (why?)
  - *Sparing*—flawed sectors remapped elsewhere

- OS doesn’t know logical to physical sector mapping
  - Larger logical sector # difference means larger seek
  - Highly non-linear relationship (*and* depends on zone)
  - OS has no info on rotational positions
  - Can empirically build table to estimate times
Sectors

- Disk interface presents linear array of sectors
  - Generally 512 bytes, written atomically

- Disk maps logical sector #s to physical sectors
  - Zoning—puts more sectors on longer tracks
  - Track skewing—sector 0 pos. varies by track (sequential access speed)
  - Sparring—flawed sectors remapped elsewhere

- OS doesn’t know logical to physical sector mapping
  - Larger logical sector # difference means larger seek
  - Highly non-linear relationship (and depends on zone)
  - OS has no info on rotational positions
  - Can empirically build table to estimate times
Disk interface

- Controls hardware, mediates access
- Computer, disk often connected by bus (e.g., SCSI)
  - Multiple devices may content for bus

**Possible disk/interface features:**

- Disconnect from bus during requests
- Command queuing: Give disk multiple requests
  - Disk can schedule them using rotational information

- **Disk cache used for read-ahead**
  - Otherwise, sequential reads would incur whole revolution
  - Cross track boundaries? Can’t stop a head-switch

- **Some disks support write caching**
  - But data not stable—not suitable for all requests
SCSI overview

• **SCSI domain** consists of devices and an SDS
  - Devices: host adapters & SCSI controllers
  - *Service Delivery Subsystem* connects devices—e.g., SCSI bus

• **SCSI-2 bus (SDS) connects up to 8 devices**
  - Controllers can have > 1 “logical units” (LUNs)
  - Typically, controller built into disk and 1 LUN/target, but “bridge controllers” can manage multiple physical devices

• **Each device can assume role of initiator or target**
  - Traditionally, host adapter was initiator, controller target
  - Now controllers act as initiators (e.g., COPY command)
  - Typical domain has 1 initiator, ≥ 1 targets
SCSI requests

- A *request* is a command from initiator to target
  - Once transmitted, target has control of bus
  - Target may disconnect from bus and later reconnect
    (very important for multiple targets or even multitasking)

- Commands contain the following:
  - *Task identifier*—initiator ID, target ID, LUN, tag
  - *Command descriptor block*—e.g., read 10 blocks at pos. N
  - Optional *task attribute*—SIMPLE, ORDERD, HEAD OF QUEUE
  - Optional: output/input buffer, sense data
  - *Status byte*—GOOD, CHECK CONDITION, INTERMEDIATE, ...
Executing SCSI commdns

- Each LUN maintains a queue of tasks
  - Each task is DORMANT, BLOCKED, ENABLED, or ENDED
  - SIMPLE tasks are dormant until no ordered/head of queue
  - ORDERED tasks dormant until no HoQ/more recent ordered
  - HoQ tasks begin in enabled state

- Task management commands available to initiator
  - Abort/terminate task, Reset target, etc.

- Linked commands
  - Initiator can link commands, so no intervening tasks
  - E.g., could use to implement atomic read-modify-write
  - Intermediate commands return status byte INTERMEDIATE
SCSI exceptions and errors

- After error stop executing most SCSI commands
  - Target returns with CHECK CONDITION status
  - Initiator will eventually notice error
  - Must read specifics w. REQUEST SENSE

- Prevents unwanted commands from executing
  - E.g., initiator may not want to execute 2nd write if 1st fails

- Simplifies device implementation
  - Don’t need to remember more than one error condition

- Same mechanism used to notify of media changes
  - I.e., ejected tape, changed CD-ROM
Disk performance

- Placement & ordering of requests a huge issue
  - Sequential I/O much, much faster than random
  - Long seeks much slower than short ones
  - Power might fail any time, leaving inconsistent state
- Must be careful about order for crashes
  - More on this in next two lectures
- Try to achieve contiguous accesses where possible
  - E.g., make big chunks of individual files contiguous
- Try to order requests to minimize seek times
  - OS can only do this if it has a multiple requests to order
  - Requires disk I/O concurrency
  - High-performance apps try to maximize I/O concurrency
- Next: How to schedule concurrent requests
Scheduling: FCFS

• “First Come First Served”
  - Process disk requests in the order they are received
• Advantages

• Disadvantages
Scheduling: FCFS

• “First Come First Served”
  - Process disk requests in the order they are received

• Advantages
  - Easy to implement
  - Good fairness

• Disadvantages
  - Cannot exploit request locality
  - Increases average latency, decreasing throughput
Shortest positioning time first (SPTF)

- Shortest positioning time first (SPTF)
  - Always pick request with shortest seek time

- Advantages

- Disadvantages

- Improvement

- Also called Shortest Seek Time First (SSTF)
Shortest positioning time first (SPTF)

- Shortest positioning time first (SPTF)
  - Always pick request with shortest seek time

- Advantages
  - Exploits locality of disk requests
  - Higher throughput

- Disadvantages
  - Starvation
  - Don’t always know what request will be fastest

- Improvement: Aged SPTF
  - Give older requests higher priority
  - Adjust “effective” seek time with weighting factor:
    \[ T_{\text{eff}} = T_{\text{pos}} - W \cdot T_{\text{wait}} \]

- Also called Shortest Seek Time First (SSTF)
“Elevator” scheduling (SCAN)

- Sweep across disk, servicing all requests passed
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests

- Advantages

- Disadvantages

- CSCAN:

- Also called LOOK/CLOOK in textbook
  - (Textbook uses [C]SCAN to mean scan entire disk uselessly)
“Elevator” scheduling (SCAN)

• Sweep across disk, servicing all requests passed
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests

• Advantages
  - Takes advantage of locality
  - Bounded waiting

• Disadvantages
  - Cylinders in the middle get better service
  - Might miss locality SPTF could exploit

• CSCAN: Only sweep in one direction
  Very commonly used algorithm in Unix

• Also called LOOK/CLOOK in textbook
  - (Textbook uses [C]SCAN to mean scan entire disk uselessly)
VSCAN(r)

- **Continuum between SPTF and SCAN**
  - Like SPTF, but slightly uses “effective” positioning time
    If request in same direction as previous seek: $T_{\text{eff}} = T_{\text{pos}}$
    Otherwise: $T_{\text{eff}} = T_{\text{pos}} + r \cdot T_{\max}$
  - when $r = 0$, get SPTF, when $r = 1$, get SCAN
  - E.g., $r = 0.2$ works well

- **Advantages and disadvantages**
  - Those of SPTF and SCAN, depending on how $r$ is set