This Unit: I/O

- I/O system structure
  - Devices, controllers, and buses
- Device characteristics
  - Disks
- Bus characteristics
- I/O control
  - Polling and interrupts
  - DMA
I/O Control and Interfaces

• Now that we know how I/O devices and buses work...
• How does I/O actually happen?
  • How does CPU give commands to I/O devices?
  • How do I/O devices execute data transfers?
  • How does CPU know when I/O devices are done?
Sending Commands to I/O Devices

- Remember: only OS can do this! Two options ...
- **I/O instructions**
  - OS only? Instructions must be privileged (only OS can execute)
  - E.g., IA-32
- **Memory-mapped I/O**
  - Portion of *physical* address space reserved for I/O
  - OS maps physical addresses to I/O device control registers
  - Stores.loads to these addresses are commands to I/O devices
    - Main memory ignores them, I/O devices recognize and respond
    - Address specifies both I/O device and command
  - These address are not cached – why?
  - OS only? I/O physical addresses only mapped in OS address space
  - E.g., almost every architecture other than IA-32 (see pattern??)
Querying I/O Device Status

- Now that we’ve sent command to I/O device ...
- How do we query I/O device status?
  - So that we know if data we asked for is ready?
  - So that we know if device is ready to receive next command?

- **Polling**: Ready now? How about now? How about now???
  - Processor queries I/O device status register (e.g., with MM load)
    - Loops until it gets status it wants (ready for next command)
    - Or tries again a little later
  - Simple
    - Waste of processor’s time
  - Processor much faster than I/O device
Polling Overhead: Example #1

- Parameters
  - 500 MHz CPU
  - Polling event takes 400 cycles

- Overhead for polling a mouse 30 times per second?
  - Cycles per second for polling = (30 poll/s)\*(400 cycles/poll)
  - \( \rightarrow \) 12000 cycles/second for polling
  - \((12000 \text{ cycles/second})/(500 \text{ M cycles/second}) = 0.002\% \text{ overhead}\)
  + Not bad
Polling Overhead: Example #2

- Same parameters
  - 500 MHz CPU, polling event takes 400 cycles

- Overhead for polling a 4 MB/s disk with 16 B interface?
  - Must poll often enough not to miss data from disk
  - Polling rate = (4MB/s)/(16 B/poll) >> mouse polling rate
  - Cycles per second for polling = [(4MB/s)/(16 B/poll)] * (400 cyc/poll)
  - → 100 M cycles/second for polling
  - (100 M cycles/second)/(500 M cycles/second) = 20% overhead
  - Bad
  - This is the overhead of polling, not actual data transfer
    - Really bad if disk is not being used (pure overhead!)
Interrupt-Driven I/O

- **Interrupts**: alternative to polling
  - I/O device generates interrupt when status changes, data ready
  - OS handles interrupts just like exceptions (e.g., page faults)
    - Identity of interrupting I/O device recorded in ECR
      - ECR: exception cause register

- I/O interrupts are **asynchronous**
  - Not associated with any one instruction
  - Don’t need to be handled immediately

- I/O interrupts are **prioritized**
  - Synchronous interrupts (e.g., page faults) have highest priority
  - High-bandwidth I/O devices have higher priority than low-bandwidth ones
Interrupt Overhead

- **Parameters**
  - 500 MHz CPU
  - Polling event takes 400 cycles
  - Interrupt handler takes 400 cycles
  - Data transfer takes 100 cycles
  - 4 MB/s, 16 B interface disk, transfers data only 5% of time

- **Percent of time processor spends transferring data**
  - $0.05 \times \frac{(4 \text{ MB/s})}{(16 \text{ B/xfer})} \times \left[\frac{(100 \text{ c/xfer})}{(500 \text{M c/s})}\right] = 0.25\%$

- **Overhead for polling?**
  - $(4 \text{ MB/s})/(16 \text{ B/poll}) \times \left[\frac{(400 \text{ c/poll})}{(500 \text{M c/s})}\right] = 20\%$

- **Overhead for interrupts?**
  - $+ 0.05 \times \frac{(4 \text{ MB/s})}{(16 \text{ B/int})} \times \left[\frac{(400 \text{ c/int})}{(500 \text{M c/s})}\right] = 1\%$

Note: when disk is transferring data, the interrupt rate is same as polling rate
Direct Memory Access (DMA)

- Interrupts remove overhead of polling...
- But still requires OS to transfer data one word at a time
  - OK for low bandwidth I/O devices: mice, microphones, etc.
  - Bad for high bandwidth I/O devices: disks, monitors, etc.

- Direct Memory Access (DMA)
  - Transfer data between I/O and memory without processor control
  - Transfers entire blocks (e.g., pages, video frames) at a time
    - Can use bus “burst” transfer mode if available
  - Only interrupts processor when done (or if error occurs)
DMA Controllers

- To do DMA, I/O device attached to **DMA controller**
  - Multiple devices can be connected to one DMA controller
  - Controller itself seen as a memory mapped I/O device
    - Processor initializes start memory address, transfer size, etc.
  - DMA controller takes care of bus arbitration and transfer details
    - So that’s why buses support arbitration and multiple masters!
I/O Processors

- A DMA controller is a very simple component
  - May be as simple as a FSM with some local memory
- Some I/O requires complicated sequences of transfers
  - **I/O processor**: heavier DMA controller that executes instructions
    - Can be programmed to do complex transfers
    - E.g., programmable network card
DMA Overhead

- Parameters
  - 500 MHz CPU
  - Interrupt handler takes 400 cycles
  - Data transfer takes 100 cycles
  - 4 MB/s, 16 B interface, disk transfers data 50% of time
  - DMA setup takes 1600 cycles, transfer 1 16KB page at a time

- Processor overhead for interrupt-driven I/O?
  - $0.5 \times \frac{4\text{ M B/s}}{16\text{ B/transfer}} \times \frac{(500 \text{ c/transfer})}{(500\text{ M c/s})} = 12.5\%$

- Processor overhead with DMA?
  - Processor only gets involved once per page, not once per 16 B
  + $0.5 \times \frac{4\text{ M B/s}}{16\text{ K B/page}} \times \frac{(2000 \text{ c/page})}{(500\text{ M c/s})} = 0.05\%$
DMA and Memory Hierarchy

• DMA is good, but is not without challenges

• Without DMA: processor initiates all data transfers
  • All transfers go through address translation
    + Transfers can be of any size and cross virtual page boundaries
  • All values seen by cache hierarchy
    + Caches never contain stale data

• With DMA: DMA controllers initiate data transfers
  • Do they use virtual or physical addresses?
  • What if they write data to a cached memory location?
DMA and Address Translation

- Which addresses does processor specify to DMA controller?

- **Virtual DMA**
  - Can specify large cross-page transfers
  - DMA controller has to do address translation internally
    - DMA contains small translation buffer (TB)
    - OS initializes buffer contents when it requests an I/O transfer

- **Physical DMA**
  - DMA controller is simple
    - Can only do short page-size transfers
      - OS breaks large transfers into page-size chunks
DMA and Caching

- Caches are good
  - Reduce CPU’s observed instruction and data access latency
  + But also, reduce CPU’s use of memory...
  + ...leaving majority of memory/bus bandwidth for DMA I/O

- But they also introduce a coherence problem for DMA
  - Input problem
    - DMA write into memory version of cached location
    - Cached version now stale
  - Output problem: write-back caches only
    - DMA read from memory version of “dirty” cached location
    - Output stale value
Solutions to Coherence Problem

- Route all DMA I/O accesses to cache
  - Solves problem
  - Expensive: CPU must contend for access to caches with DMA

- Disallow caching of I/O data
  - Also works
  - Expensive in a different way: CPU access to those regions slow

- Selective flushing/invalidations of cached data
  - Flush all dirty blocks in “I/O region”
  - Invalidate blocks in “I/O region” as DMA writes those addresses
  - The high performance solution
    - **Hardware cache coherence** mechanisms for doing this
      - Expensive in yet a third way: must implement this mechanism
H/W Cache Coherence (more later on this)

- D$ and L2 “snoop” bus traffic
  - Observe transactions
  - Check if written addresses are resident
  - **Self-invalidate** those blocks
    - Doesn’t require access to data part
    - Does require access to tag part
      - May need 2nd copy of tags for this
      - That’s OK, tags smaller than data

- Bus addresses are physical
  - L2 is easy (physical index/tag)
  - D$ is harder (**virtual index**/physical tag)
Designing an I/O System for Bandwidth

- **Approach**
  - Find bandwidths of individual components
  - Configure components you can change...
  - To match bandwidth of bottleneck component you can’t

- **Example (from P&H textbook, 3rd edition)**
  - **Parameters**
    - 300 MIPS CPU, 100 MB/s backplane bus
    - 50K OS insns + 100K user insns per I/O operation
    - SCSI-2 controllers (20 MB/s): each accommodates up to 7 disks
    - 5 MB/s disks with $t_{\text{seek}} + t_{\text{rotation}} = 10$ ms, 64 KB reads
  - **Determine**
    - What is the maximum sustainable I/O rate?
    - How many SCSI-2 controllers and disks does it require?
Designing an I/O System for Bandwidth

• First: determine I/O rates of components we can’t change
  • CPU: \( \frac{300 \text{M insn/s}}{150 \text{K Insns/IO}} = 2000 \text{ IO/s} \)
  • Backplane: \( \frac{100 \text{M B/s}}{64 \text{K B/IO}} = 1562 \text{ IO/s} \)
  • Peak I/O rate determined by bus: \( \textbf{1562 \text{ IO/s}} \)

• Second: configure remaining components to match rate
  • Disk: \( \frac{1}{10 \text{ ms/IO} + \frac{64 \text{K B/IO}}{5 \text{M B/s}}} = 43.9 \text{ IO/s} \)
  • How many disks?
    • \( \frac{1562 \text{ IO/s}}{43.9 \text{ IO/s}} = \textbf{36 disks} \)
  • How many controllers?
    • \( \frac{43.9 \text{ IO/s} \times 64 \text{K B/IO}}{2.74 \text{M B/s per disk}} = 2.74 \text{M B/s per disk} \)
    • \( \frac{20 \text{M B/s}}{2.74 \text{M B/s}} = 7.2 \text{ disks per SCSI controller} \)
    • \( \frac{36 \text{ disks}}{7 \text{ disks/SCSI-2}} = \textbf{6 SCSI-2 controllers} \)

• Caveat: real I/O systems modeled with simulation
Designing an I/O System for Latency

- Previous system designed for bandwidth
- Some systems have latency requirements as well
  - E.g., database system may require maximum or average latency

- Latencies are actually harder to deal with than bandwidths
  - **Unloaded system**: few concurrent IO transactions
    - Latency is easy to calculate
  - **Loaded system**: many concurrent IO transactions
    - Contention can lead to queuing
    - Latencies can rise dramatically
    - Queuing theory can help if transactions obey fixed distribution
    - Otherwise simulation is needed
Summary

• Role of the OS

• Device characteristics
  • Data bandwidth
  • Disks
    • Structure and latency: seek, rotation, transfer, controller delays

• Bus characteristics
  • Processor-memory, I/O, and backplane buses
  • Width, multiplexing, clocking, switching, arbitration

• I/O control
  • I/O instructions vs. memory mapped I/O
  • Polling vs. interrupts
  • Processor controlled data transfer vs. DMA
    • Interaction of DMA with memory system