I/O Handling

Tyler Bletsch
Duke University

Slides are adapted from Brian Rogers (Duke)
• Typical application flow consists of alternating phases
  – Compute
  – I/O operation
  – Often I/O is the primary component with very short compute bursts

• Recall that OS manages resources
  – Also includes I/O resources
  – Initiates and controls I/O operations
  – Controls I/O devices and device drivers

• I/O systems allow process to interact w/ physical devices
  – Both within the computer: Disks, printer, keyboard, mouse
  – And outside the computer: Network operations
Processor Interface to IO Devices

- Processor Chip has IO Pins
  - E.g. for connection to buses
    - Memory bus
    - PCIe bus
  - Other dedicated IO to chip
    - E.g. for power

To main memory (e.g. DRAM)
IO System Connections

- Monitor
- Graphics controller
- Disk controller
- Disk
- Processor
- Memory
- USB controller
- Universal Serial Bus
- Mouse
- Keyboard
- USB PCIe Card
IO System

• Devices connect via a port or a bus
  – A bus is a set of wires with a well defined protocol

• Controller operates a port, bus or device
  – Wide ranging complexities
    • Disk controllers can be very complex
  – Sometimes even a dedicated embedded processor is used
    • Runs the controller software

• Two sides of the communication
  – Processor:
    • On-chip hardware (e.g. PCIe controller) interfaces to the bus protocol
    • Or bridge / IO controller on separate chip in older systems
  – IO devices:
    • Via the controller mentioned above
• Processor interacts with controller for a target device
  – Processor can send commands / data (or receive)

• Controller contains registers for commands / data
  – Two ways for processor to communicate with these registers
    • Dedicated I/O instructions that transfer bits to I/O port address
    • Memory mapped I/O: controller regs are mapped to mem address
      – Standard load/store instructions can write to registers
      – E.g. graphics controller has large mem mapped space for pixel data
  – Control register bit patterns indicate different commands to device

• Usually at least 4 register
  – Data-in (to the processor) and Data-out (from the processor)
  – Status: state of the device (device busy, data ready, error, etc.)
  – Control Register: written by device to initiate command or change device settings
Processor – Device Interaction

- Handshake protocol
  1. Host reads a busy bit in the status register until device free
  2. Host sets write bit in command register & writes data into data-out
  3. Host sets the command ready bit in the command register
  4. Controller detects command ready bit set & sets busy bit
  5. Controller reads command register; sees command; does I/O w/ device
  6. Controller clears command ready bit; clear error & busy bits in status reg

- How to handle step 1
  - Polling (busy-waiting) executing a small code loop
    - Load – branch if bit not set
    - Performance-inefficient if device is frequently busy
  - Interrupt mechanism to notify the CPU
    - Recall our previous lecture
More on Interrupts & I/O

• Steps for reading from disk
  – Initiate I/O read operations for disk drive
    • Bring data into kernel buffer in memory
  – Copy data from kernel space buffer into user space buffer

• Initiating I/O read ops from disk is high priority
  – Want to efficiently utilize disk

• Use pair of interrupt handlers
  – High priority handler handshakes w/ disk controller
    • Keeps I/O requests moving to disk
    • Raises low-priority interrupt when disk operations are complete
  – Low priority handler services interrupt
    • Moves data from kernel buffer to user space
    • Calls scheduler to move process to ready queue

• Threaded kernel architecture is a good fit
We’ve talked about a tight control loop (handshake) so far
  – Processor monitors status bits (or interrupts)
  – Move data in bytes or words at a time via data-in / data-out regs
    •  **Programmed I/O (PIO)**

Some devices want to perform large data transfers
  – E.g. disk, network

**Direct Memory Access (DMA):**
Typically done w/ dedicated HW engine or logic
  – Processor writes DMA commands to a memory buffer
    • Pointer to src and dest addresses, # of bytes to transfer
  – Processor writes address of DMA command block to DMA engine
  – DMA engine operates on memory & handshakes with device
DMA Operation

• DMA-request & DMA-acknowledge to device controller
  ▪ Device asserts DMA-request when data is available to transfer
  ▪ DMA controller obtains bus control
    • Puts appropriate request address on the bus
    • Asserts DMA-acknowledge wire
  ▪ Device controller puts data on the bus

• DMA controller generates CPU interrupt when transfer is complete
Application Interface to I/O System

• Many different devices
  ▪ All with different functionality, register control definitions, etc.
  ▪ How can OS talk to new devices without modification?
  ▪ How can OS provide consistent API to applications for I/O?

• Solution to all computer science problems
  ▪ Either add a level of indirection (abstraction)...or cache it!

• Abstract away IO device details
  ▪ Identify sets of similar devices; provide standard interface to each
  ▪ Add a new layer of software to implement each interface
    • Device Drivers
    • Type of kernel module (OS extensions that can be loaded / unloaded)
Device Drivers

• Purpose: hide device-specific controller details from I/O subsystem as much as possible
  ▪ OS is easier to develop & maintain
  ▪ Device manufacturers can conform to common interfaces
    • Can attach new I/O devices to existing machines

• Device driver software is typically OS-specific
  ▪ Different interface standards across OSes

• Several different device categories (each w/ interface)
  ▪ Based on different device characteristics
    • Block I/O, Character-stream I/O, Memory-mapped file, Network sockets
  ▪ OS also has low-level system calls (ioctl on Linux)
    • Look at man page
Block-Device Interface

• API for accessing block-oriented devices
  – read, write, seek (if random access device)
• Applications normally access via file system interface
• Low-level device operation & policies are hidden by API
• Examples: Hard drive, optical disc drive
Character-Stream Interface

• Keyboard, mice, for example

• API:
  – get(), put() a character at a time

• Often libraries are implemented on top of this interface
  – E.g. buffer and read a line at a time
  – Useful for devices that produce input data unpredictably

• Examples: Serial port, modem
Memory-mapped File Interface

• Layer on top of block-device interface
• Provides access to storage as bytes in memory
  – System call sets up this memory mapping
  – We’ve seen an example of this for memory-mapped disk files
• Processor can read & write bytes in memory
• Data transfers only performed as needed between memory & device
• Example: Video card (frame buffer)
Network Device Interface

• UNIX network sockets for example

• Applications can
  - Create socket
  - Connect a local socket to a remote address
    • Address = host IP address and port number
    • This will plug the socket into an application on the remote machine
  - Use select() to monitor activity on any of a number of sockets

• Example: Ethernet or WiFi NIC
Blocking vs. Nonblocking (vs. Async)

• Blocking
  – Process is suspended on issuing a blocking IO system call
  – Moved from ready queue to wait queue
  – Moved back to ready queue after IO completes

• Nonblocking
  – Process does not wait for IO call completion
    • Any data that is ready is returned
  – E.g. user Interface receives keyboard & mouse input

• Asynchronous
  – IO call returns immediately & IO operation is initiated
  – Process is notified of IO completion via later interrupt
  – E.g. select() w/ wait time of 0
    • Followed by read() if any source has data ready
OS Kernel I/O Subsystem

• Provides many services for I/O
  – Scheduling
  – Buffering
  – Caching
  – Spooling
  – Device Reservation
  – Error Handling
  – Protection of I/O
I/O Scheduling

• Scheduling = Ordering application requests to IO devices
  – OS does not necessarily have to send them in order received

• Can impact many aspects of the system
  – Performance
    • Average wait time by applications for I/O requests
    • IO device utilization (how often are they busy performing useful work)
  – Fairness
    • Do applications get uniform access to I/O devices?
    • Should some users / applications be prioritized?

• Implementation
  – OS implements a wait queue for requests to each device
  – Reorders queue to schedule requests to optimize metrics
• Traditional hard disk has two access time components
  – Seek time: disk arm moves heads to cylinder containing sector
  – Rotational latency: disk rotates to desired sector
  – Bandwidth is also important (# bytes per unit time)

• Somewhat analogous to CPU scheduling we discussed
  – FCFS: first-come, first-served
    • Fair, but generally not fast or high bandwidth
  – SSTF: shortest seek time first
    • Equivalent to SJF (see pros & cons from CPU scheduling)
  – SCAN: move disk arm from one end to the other, back & forth
    • Service requests as disk arm reaches their cylinder
    • “Elevator” algorithm
  – C-SCAN: move disk arm in a cyclical round trip (servicing forward, skipping back)
    • Improves wait time relative to SCAN
I/O Buffering

• Memory region to store in-flight data
  ▪ E.g. between two devices or a device and application

• Reasons for buffering
  ▪ Speed mismatch between source and destination device
    • E.g. data received over slow network going to fast disk
      ▪ Want to write big blocks of data to disk at a time, not small pieces
    • Double buffering
      ▪ Alternate which buffer is being filled from source and which is written to destination
      ▪ Removes need for timing requirements between producer / consumer
  ▪ Efficiently handle device data with different transfer sizes
I/O Caching

• Similar concept to other types of caching you’ve learned
  ▪ CPU caching (L1, L2, L3 caches for main memory)
  ▪ Disk caching using main memory

• Use memory to cache data regions for IO transfers
  ▪ Similar to buffering, but for a different purpose

• E.g. for disk IO, cache buffers in main memory
  ▪ Improves efficiency for shared files that are read/written often
  ▪ Improve latency for reads; Reduce disk bandwidth for writes
    • Reads serviced by memory instead of slow disk
    • Writes can be “gathered” and a single bulk disk write done later
I/O Spooling

• Spool: type of buffer to hold data for device that cannot accept interleaved data streams
  ▪ Printers!

• Kernel stores each applications print I/O data
  ▪ Spooled to a separate disk file

• Later, the kernel queues a spool file to the printer
  ▪ Often managed by a running daemon process
  ▪ Allows applications to view pending jobs, cancel jobs, etc.

• Device Reservation:
  ▪ For similar purposes as spooling
  ▪ Kernel facility for allocating an idle device & deallocating later
I/O Error Handling & Protection

- I/O system calls return information about status
  - `errno` variable in UNIX
  - Indicate general nature of failure
  - Failures can happen due to transient problems
    - OS can compensate by re-trying failed operations

- Protection mechanisms for I/O by kernel
  - All I/O instructions are privileged
    - cannot be executed directly by user process
  - User process must execute system call
  - System call can check for valid request & data