Chapter 12: I/O Systems
Chapter 12: I/O Systems

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- STREAMS
- Performance
Objectives

- Explore the structure of an operating system’s I/O subsystem
- Discuss the principles and complexities of I/O hardware
- Explain the performance aspects of I/O hardware and software
Overview

- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- Ports, busses, device controllers connect to various devices
- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem
I/O Hardware

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface

- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device
  - **Bus - daisy chain** or shared direct access
    - **PCI** bus common in PCs and servers, PCI Express (**PCIe**)
    - **expansion bus** connects relatively slow devices
    - **Serial-attached SCSI (SAS)** common disk interface
  - **Controller (host adapter)** – electronics that operate port, bus, device
    - Sometimes integrated
    - Sometimes separate circuit board (host adapter)
    - Contains processor, microcode, private memory, bus controller, etc
      - Some talk to per-device controller with bus controller, microcode, memory, etc
A Typical PC Bus Structure

- Monitor
- Processor
- Graphics controller
- Bridge/memory controller
- Cache
- Memory
- SAS controller
  - Disk
  - Disk
  - Disk
  - Disk
- Expansion bus interface
- Keyboard
- Expansion bus
  - USB port
  - USB port
I/O Hardware (Cont.)

- **Fibre channel (FC)** is complex controller, usually separate circuit board (**host-bus adapter, HBA**) plugging into bus
- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped I/O**
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)
### Device I/O Port Locations on PCs (partial)

<table>
<thead>
<tr>
<th>I/O address range (hexadecimal)</th>
<th>device</th>
</tr>
</thead>
<tbody>
<tr>
<td>000–00F</td>
<td>DMA controller</td>
</tr>
<tr>
<td>020–021</td>
<td>interrupt controller</td>
</tr>
<tr>
<td>040–043</td>
<td>timer</td>
</tr>
<tr>
<td>200–20F</td>
<td>game controller</td>
</tr>
<tr>
<td>2F8–2FF</td>
<td>serial port (secondary)</td>
</tr>
<tr>
<td>320–32F</td>
<td>hard-disk controller</td>
</tr>
<tr>
<td>378–37F</td>
<td>parallel port</td>
</tr>
<tr>
<td>3D0–3DF</td>
<td>graphics controller</td>
</tr>
<tr>
<td>3F0–3F7</td>
<td>diskette-drive controller</td>
</tr>
<tr>
<td>3F8–3FF</td>
<td>serial port (primary)</td>
</tr>
</tbody>
</table>
Polling

- For each byte of I/O
  1. Read busy bit from status register until 0
  2. Host sets read or write bit and if write copies data into data-out register
  3. Host sets command-ready bit
  4. Controller sets busy bit, executes transfer
  5. Controller clears busy bit, error bit, command-ready bit when transfer done

- Step 1 is **busy-wait** cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - But if miss a cycle data overwritten / lost
Interrupts

- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?
- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable**
  - Interrupt chaining if more than one device at same interrupt number
Interrupt-Driven I/O Cycle

1. CPU

2. Device driver initiates I/O

3. CPU executing checks for interrupts between instructions

4. CPU receiving interrupt, transfers control to interrupt handler

5. CPU initiates I/O

6. Interrupt handler processes data, returns from interrupt

7. Input ready, output complete, or error generates interrupt signal

8. CPU resumes processing of interrupted task
Interrupts (Cont.)

- Interrupt mechanism also used for exceptions
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via \textit{trap} to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast
Latency

- Stressing interrupt management because even single-user systems manage hundreds or interrupts per second and servers hundreds of thousands.
- For example, a quiet macOS desktop generated 23,000 interrupts over 10 seconds.
<table>
<thead>
<tr>
<th>vector number</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>divide error</td>
</tr>
<tr>
<td>1</td>
<td>debug exception</td>
</tr>
<tr>
<td>2</td>
<td>null interrupt</td>
</tr>
<tr>
<td>3</td>
<td>breakpoint</td>
</tr>
<tr>
<td>4</td>
<td>INTO-detected overflow</td>
</tr>
<tr>
<td>5</td>
<td>bound range exception</td>
</tr>
<tr>
<td>6</td>
<td>invalid opcode</td>
</tr>
<tr>
<td>7</td>
<td>device not available</td>
</tr>
<tr>
<td>8</td>
<td>double fault</td>
</tr>
<tr>
<td>9</td>
<td>coprocessor segment overrun (reserved)</td>
</tr>
<tr>
<td>10</td>
<td>invalid task state segment</td>
</tr>
<tr>
<td>11</td>
<td>segment not present</td>
</tr>
<tr>
<td>12</td>
<td>stack fault</td>
</tr>
<tr>
<td>13</td>
<td>general protection</td>
</tr>
<tr>
<td>14</td>
<td>page fault</td>
</tr>
<tr>
<td>15</td>
<td>(Intel reserved, do not use)</td>
</tr>
<tr>
<td>16</td>
<td>floating-point error</td>
</tr>
<tr>
<td>17</td>
<td>alignment check</td>
</tr>
<tr>
<td>18</td>
<td>machine check</td>
</tr>
<tr>
<td>19–31</td>
<td>(Intel reserved, do not use)</td>
</tr>
<tr>
<td>32–255</td>
<td>maskable interrupts</td>
</tr>
</tbody>
</table>
Direct Memory Access

- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - **Cycle stealing** from CPU but still much more efficient
  - When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient - **DVMA**
Six Step Process to Perform DMA Transfer

1. device driver is told to transfer drive 2 data to buffer at address “x”

2. device driver tells drive controller to transfer “c” bytes to buffer at address “x”

3. drive controller initiates DMA transfer

4. DMA controller transfers bytes to buffer “x”, increasing memory address and decreasing “c” until c = 0

5. when c = 0, DMA interrupts CPU to signal transfer completion
Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks

Devices vary in many dimensions

- **Character-stream** or **block**
- **Sequential** or **random-access**
- **Synchronous** or **asynchronous** (or both)
- **Sharable** or **dedicated**
- **Speed of operation**
- **read-write**, **read only**, or **write only**
## A Kernel I/O Structure

<table>
<thead>
<tr>
<th>Hardware</th>
<th>SAS device controller</th>
<th>keyboard device controller</th>
<th>mouse device controller</th>
<th>PCIe bus device controller</th>
<th>802.11 device controller</th>
<th>USB device controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS devices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>keyboard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mouse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCIe bus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>802.11 devices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB devices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Characteristics of I/O Devices

<table>
<thead>
<tr>
<th>aspect</th>
<th>variation</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-transfer mode</td>
<td>character block</td>
<td>terminal disk</td>
</tr>
<tr>
<td>access method</td>
<td>sequential</td>
<td>modem</td>
</tr>
<tr>
<td></td>
<td>random</td>
<td>CD-ROM</td>
</tr>
<tr>
<td>transfer schedule</td>
<td>synchronous</td>
<td>tape</td>
</tr>
<tr>
<td></td>
<td>asynchronous</td>
<td>keyboard</td>
</tr>
<tr>
<td>sharing</td>
<td>dedicated</td>
<td>tape</td>
</tr>
<tr>
<td></td>
<td>sharable</td>
<td>keyboard</td>
</tr>
<tr>
<td>device speed</td>
<td>latency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>seek time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transfer rate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delay between operations</td>
<td></td>
</tr>
<tr>
<td>I/O direction</td>
<td>read only</td>
<td>CD-ROM</td>
</tr>
<tr>
<td></td>
<td>write only</td>
<td>graphics controller</td>
</tr>
<tr>
<td></td>
<td>read–write</td>
<td>disk</td>
</tr>
</tbody>
</table>
Characteristics of I/O Devices (Cont.)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
- For direct manipulation of I/O device specific characteristics, usually an escape / back door
  - Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register
- UNIX and Linux use tuple of “major” and “minor” device numbers to identify type and instance of devices (here major 8 and minors 0-4)

```bash
% ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```
Block and Character Devices

- **Block devices include disk drives**
  - Commands include read, write, seek
  - **Raw I/O**, **direct I/O**, or file-system access
  - Memory-mapped file access possible
    - File mapped to virtual memory and clusters brought via demand paging
  - DMA

- **Character devices include keyboards, mice, serial ports**
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
Network Devices

- Varying enough from block and character to have own interface
- Linux, Unix, Windows and many others include socket interface
  - Separates network protocol from network operation
  - Includes `select()` functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)
Clocks and Timers

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- **Programmable interval timer** used for timings, periodic interrupts
- `ioctl()` (on UNIX) covers odd aspects of I/O such as clocks and timers
Nonblocking and Asynchronous I/O

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs

- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with count of bytes read or written
  - `select()` to find if data ready then `read()` or `write()` to transfer

- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed
Two I/O Methods

(a) Synchronous

(b) Asynchronous

(requesting process)

user land

kernel

device driver

interrupt handler

hardware data transfer

(time)

(requesting process)

device driver

interrupt handler

hardware data transfer

(time)
**Vectored I/O**

- **Vectored I/O** allows one system call to perform multiple I/O operations.
- For example, Unix `readve()` accepts a vector of multiple buffers to read into or write from.
- This scatter-gather method is better than multiple individual I/O calls:
  - Decreases context switching and system call overhead.
  - Some versions provide atomicity:
    - Avoid for example worry about multiple threads changing data as reads / writes occurring.
Kernel I/O Subsystem

- **Scheduling**
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
  - Some implement Quality Of Service (i.e. IPQOS)

- **Buffering** - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”
  - **Double buffering** – two copies of the data
    - Kernel and user
    - Varying sizes
    - Full / being processed and not-full / being used
    - Copy-on-write can be used for efficiency in some cases
### Device-status Table

<table>
<thead>
<tr>
<th>Device</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>Idle</td>
</tr>
<tr>
<td>Laser Printer</td>
<td>Busy</td>
</tr>
<tr>
<td>Mouse</td>
<td>Idle</td>
</tr>
<tr>
<td>Disk Unit 1</td>
<td>Idle</td>
</tr>
<tr>
<td>Disk Unit 2</td>
<td>Busy</td>
</tr>
</tbody>
</table>

- **Request for Laser Printer**
  - Address: 38546
  - Length: 1372

- **Request for Disk Unit 2**
  - File: `xxx`
  - Operation: Read
  - Address: 43046
  - Length: 20000

- **Request for Disk Unit 2**
  - File: `yyy`
  - Operation: Write
  - Address: 03458
  - Length: 500
Common PC and Data-center I/O devices and Interface Speeds

- Keyboard
- USB 1.0
- Bluetooth
- USB 2.0
- 1G Ethernet
- SAS 15K RPM Disk
- USB 3.0
- SATA 3.0
- Thunderbolt
- 10G Ethernet
- USB 3.1
- 16G FibreChannel
- InfiniBand QDR 4x
- 40G Ethernet
- PCI Express Gen 3 x8
- InfiniBand HDR 4x
- 100G Ethernet
- PCIe Express Gen 3 x16

GB/s

0.001 0.002 0.006 0.060 0.125 0.204 0.625 0.750 1.250 1.250 1.250 2.000 5.000 5.000 8.000 12.000 12.500 16.000
Kernel I/O Subsystem

- **Caching** - faster device holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering

- **Spooling** - hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing

- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced – Solaris FMA, AIX
    - Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports
I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too
Use of a System Call to Perform I/O

1. Trap to kernel
2. Perform I/O
3. Return to calling thread
Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O
  - Windows uses message passing
    - Message with I/O information passed from user mode into kernel
    - Message modified as it flows through to device driver and back to process
    - Pros / cons?
UNIX I/O Kernel Structure

- file descriptor
  - per-process open-file table
    - system-wide open-file table
      - file-system record
        - inode pointer
        - pointer to read and write functions
        - pointer to select function
        - pointer to ioctl function
        - pointer to close function
      - networking (socket) record
        - pointer to network info
        - pointer to read and write functions
        - pointer to select function
        - pointer to ioctl function
        - pointer to close function
    - active-inode table
    - network-information table
  - user-process memory
Power Management

- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
  - Cloud computing environments move virtual machines between servers
    - Can end up evacuating whole systems and shutting them down
- Mobile computing has power management as first class OS aspect
For example, Android implements
- Component-level power management
  - Understands relationship between components
  - Build device tree representing physical device topology
  - System bus -> I/O subsystem -> {flash, USB storage}
  - Device driver tracks state of device, whether in use
  - Unused component – turn it off
  - All devices in tree branch unused – turn off branch
- Wake locks – like other locks but prevent sleep of device when lock is held
- Power collapse – put a device into very deep sleep
  - Marginal power use
  - Only awake enough to respond to external stimuli (button press, incoming call)

Modern systems use advanced configuration and power interface (ACPI)
firmware providing code that runs as routines called by kernel for device discovery,
management, error and power management
Kernel I/O Subsystem Summary

In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel:

- Management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot seek())
- File-system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization
- Power management of I/O devices

The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers.
Consider reading a file from disk for a process:

- Determine device holding file
- Translate name to device representation
- Physically read data from disk into buffer
- Make data available to requesting process
- Return control to process
Life Cycle of An I/O Request

1. Userland requests I/O.
2. System call
3. Kernel I/O subsystem:
   - Can already satisfy I/O request? (yes/no)
   - Yes: place data in return values or in process space.
   - No: send request to device driver, block process if appropriate.
4. Device driver:
   - Process request, issue commands to controller, configure controller to block until interrupt.
5. Interrupt handler:
   - Determine which I/O completed, indicate state changes to I/O subsystem.
   - Receive interrupt, store data in device-driver buffer if input, signal to unblock device driver.
6. Device controller:
   - Execute command, monitor device.
7. I/O complete, generate interrupt.
STREAMS

- **STREAM** – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond

- A STREAM consists of:
  - STREAM head interfaces with the user process
  - driver end interfaces with the device
  - zero or more STREAM modules between them

- Each module contains a read queue and a write queue

- Message passing is used to communicate between queues
  - **Flow control** option to indicate available or busy

- Asynchronous internally, synchronous where user process communicates with stream head
The STREAMS Structure

- User process
- Stream head
- Read queue
- Write queue
- Read queue
- Write queue
- Read queue
- Write queue
- Read queue
- Write queue
- Driver end
- Device

STREAMS modules
Performance

- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful
Intercomputer Communications

The diagram illustrates the process of intercomputer communication. It shows the flow of events from the sending system to the receiving system. The process begins with a user process sending a character, which triggers an interrupt handler. The interrupt handler calls the device driver, leading to a context switch to the kernel. The kernel then generates an interrupt, which is sent across the network. On the receiving system, the network daemon receives the packet, and the process continues with another context switch to the user process.
Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads
Device-Functionality Progression

- increased time (generations)
- increased efficiency
- increased development cost
- increased abstraction

new algorithm

application code

kernel code

device-driver code

device-controller code (hardware)

device code (hardware)
I/O Performance of Storage (and Network Latency)

- Memory Load/Store
  - CPU Caches
  - DIMM
  - DRAM
  - NVM

- Storage Read/Write
  - PCIe/NVMe
  - SSD
  - SAS
  - HDD

- Network Latency
  - ns
  - μs
  - ms
End of Chapter 12