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 of I/O hardware and its complexity

• Provide details of 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
  • 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
I/O Hardware (Cont.)

• 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

   - device driver initiates I/O
   - CPU executing checks for interrupts between instructions
   - CPU receiving interrupt, transfers control to interrupt handler
     - interrupt handler processes data, returns from interrupt
     - CPU resumes processing of interrupted task

2. I/O controller

   - initiates I/O
   - input ready, output complete, or error generates interrupt signal
   - 3
   - 4
   - 5
   - 6
   - 7
# Intel Pentium Processor Event-Vector Table

<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>
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 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
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 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
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>software</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI devices</td>
<td>kernel I/O subsystem</td>
</tr>
<tr>
<td>keyboard</td>
<td>SCSI device driver</td>
</tr>
<tr>
<td>mouse</td>
<td>keyboard device driver</td>
</tr>
<tr>
<td>...</td>
<td>mouse device driver</td>
</tr>
<tr>
<td>PCI bus</td>
<td>...</td>
</tr>
<tr>
<td>floppy</td>
<td>PCI bus device controller</td>
</tr>
<tr>
<td>device</td>
<td>floppy device driver</td>
</tr>
<tr>
<td>controller</td>
<td>ATAPI device driver</td>
</tr>
<tr>
<td></td>
<td>ATAPI device controller</td>
</tr>
</tbody>
</table>

- **SCSI device driver**
- **Keyboard device driver**
- **Mouse device driver**
- **...**
- **PCI bus device controller**
- **Floppy device driver**
- **ATAPI device driver**
# 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 random</td>
<td>modem CD-ROM</td>
</tr>
<tr>
<td>transfer schedule</td>
<td>synchronous asynchronous</td>
<td>tape keyboard</td>
</tr>
<tr>
<td>sharing</td>
<td>dedicated sharable</td>
<td>tape keyboard</td>
</tr>
<tr>
<td>device speed</td>
<td>latency seek time transfer rate delay between operations</td>
<td>CD-ROM graphics controller disk</td>
</tr>
<tr>
<td>I/O direction</td>
<td>read only</td>
<td>CD-ROM graphics controller disk</td>
</tr>
<tr>
<td></td>
<td>write only</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read-write</td>
<td></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
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

Synchronous

Asynchronous
Vectored I/O

- **Vectored I/O** allows one system call to perform multiple I/O operations
- For example, Unix `readv()` accepts a vector of multiple buffers to read into or write from
- This scatter-gather method 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

- device: keyboard
  status: idle

- device: laser printer
  status: busy

- device: mouse
  status: idle

- device: disk unit 1
  status: idle

- device: disk unit 2
  status: busy

- 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
Sun Enterprise 6000 Device-Transfer Rates
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
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**
- **open-file table**
- **per-process**
- **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
- **kernel memory**
- **active-inode table**
- **network-information table**
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
Power Management (Cont.)

• 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)
I/O Requests to Hardware Operations

• 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

- request I/O
  - system call
  - can already satisfy request?
    - yes
    - transfer data (if appropriate) to process, return completion or error code
    - no
    - send request to device driver, block process if appropriate
  - process request, issue commands to controller, configure controller to block until interrupted
  - device-controller commands
  - monitor device, interrupt when I/O completed
  - device controller
  - interrupt
  - receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
  - determine which I/O completed, indicate state change to I/O subsystem
- I/O completed, input data available, or output completed
  - return from system call
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
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
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

Device code (hardware)
Device controller code (hardware)
Device driver code
Kernel code
Application code

Increased flexibility

New algorithm