Chapter 13:
I/O Systems

Zhi Wang
Florida State University
Content

• I/O hardware
• Application I/O interface
• Kernel I/O subsystem
• I/O 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 performance of I/O hardware and software
Overview

- I/O management is a major component of OS design and operation
  - important aspect of computer operation
    - I/O devices is the way computer to interact with user and other systems
  - I/O devices vary greatly
    - various methods to control them
    - performance varies
    - device drivers encapsulate device details; presents an uniform interface
  - new types of devices frequently emerges
I/O Hardware

- Incredible variety of I/O devices
  - storage, communication, human-interface
- Common concepts: signals from I/O devices interface with computer
  - **bus**: an interconnection between components (including CPU)
  - **port**: connection point for device
- **controller**: component that control the device
  - can be integrated to device or separate circuit board
  - usually contains processor, microcode, private memory, bus controller, etc
- I/O access can use **polling** or **interrupt**
A Typical PC Bus Structure

- Monitor
- Graphics controller
- Processor
- Bridge/memory controller
- Cache
- Memory
- IDE disk controller
- Disk
- Expansion bus interface
- Expansion bus
- Parallel port
- Serial port
- SCSI controller
- Disk
- SCSI bus
I/O Hardware

• Some CPU architecture has dedicated I/O instructions
  • e.g., x86: in, out, ins, outs

• Devices usually provide registers for data and control I/O of device
  • device driver places (pointers to) commands and data to register
  • registers include data-in/data-out, status, control (or command) register
  • typically 1-4 bytes, or FIFO buffer

• Devices are assigned addresses for registers or on-device memory

  • **direct I/O instructions**
    • to access (mostly) registers

  • **memory-mapped I/O**
    • data and command registers mapped to processor address space
    • to access (large) on-device memory (graphics)
<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 I/O operation:
  • busy-wait if device is busy (status register)
  • send the command to the device controller (command register)
  • read status register until it indicates command has been executed
  • read execution status, and possibly reset device status
• Polling requires busy wait
  • reasonable if device is fast; inefficient if device slow
Interrupts

- Polling requires busy-wait, inefficient use of CPU resource
- Interrupts can avoid busy-wait
  - device driver send a command to the controller, and return
  - OS can schedule other activities
  - device will interrupt the processor when command has been executed
  - OS retrieves the result by handling the interrupt
- Interrupt-based I/O requires context switch at start and end
  - if interrupt frequency is extremely high, context switch wastes CPU time
  - solution: use polling instead
    - example: NAPI in Linux enables polling under very high network load
Interrupt-Driven I/O Cycle

1. CPU executes checks for interrupts between instructions.
2. CPU receiving interrupt, transfers control to interrupt handler.
3. Input ready, output complete, or error generates interrupt signal.
4. Interrupt handler processes data, returns from interrupt.
5. CPU resumes processing of interrupted task.
## Intel Pentium Interrupt 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

• Interrupt is also used for exceptions
  • protection error for access violation
  • page fault for memory access error
  • software interrupt for system calls

• Multi-CPU systems can process interrupts concurrently
  • sometimes a CPU may be dedicated to handle interrupts
  • interrupts can also have CPU affinity
Direct Memory Access

- DMA transfer data directly between I/O device and memory
  - OS only need to issue commands, data transfers bypass the CPU
  - no programmed I/O (one byte at a time), data transferred in large blocks
  - it requires DMA controller in the device or system
- OS issues commands to the DMA controller
  - a command includes: operation, memory address for data, count of bytes…
  - usually it is the pointer of the command written into the command register
  - when done, device interrupts CPU to signal completion
Six Steps of 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
  - in Linux, devices can be accessed as files; low-level access with ioctl
- Device-driver layer hides differences among I/O controllers from kernel
  - each OS has its own I/O subsystem and device driver frameworks
  - new devices talking already-implemented protocols need no extra work
- 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
Kernel I/O Structure

The diagram illustrates the structure of the kernel I/O subsystem. It shows the hierarchy from hardware to software, with the kernel at the top. The hardware box contains SCSI devices, keyboard, mouse, and others. The software box includes SCSI device driver, keyboard device driver, mouse device driver, and others. The kernel I/O subsystem connects these components, facilitating communication between hardware and software.
## 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</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 graphics controller</td>
</tr>
<tr>
<td></td>
<td>write only</td>
<td>disk</td>
</tr>
<tr>
<td></td>
<td>read–write</td>
<td></td>
</tr>
</tbody>
</table>
Characteristics of I/O Devices

- Broadly, I/O devices can be grouped by the OS into
  - block I/O
  - character I/O (Stream)
  - memory-mapped file access
  - network sockets
- Direct manipulation of I/O device usually an escape / back door
  - Linux’s ioctl call to send commands to a device driver
Block and Character Devices

- Block devices access data in blocks, such as disk drives…
  - commands include read, write, seek
  - raw I/O, direct I/O, or file-system access
  - memory-mapped file access possible (e.g., memory-mapped files)
  - DMA
- Character devices include keyboards, mice, serial ports…
  - very diverse types of devices
Network Devices

- Varying enough from block and character to have own interface
  - very different from pipe, mailbox...

- Popular interface for network access is the **socket** interface
  - it separates network protocol from detailed network operation
  - some non-network operations are implemented as sockets
    - e.g., Unix socket
Clocks and Timers

- Clocks and timers can be considered as character devices
  - very important devices as they provide current time, elapsed time, timer
- Normal resolution about 1/60 second, some OS provides higher-resolution ones
Synchronous/Asynchronous I/O

• **Synchronous I/O** includes blocking and non-blocking I/O
  
  • **blocking I/O**: process suspended until I/O completed
    
    • easy to use and understand, but may be less efficient
    
    • insufficient for some needs
  
  • **non-blocking I/O**: I/O calls return as much data as available
    
    • process does not block, returns whatever existing data (read or write)
    
    • use select to find if data is ready, then use read or write to transfer data

• **Asynchronous I/O**: process runs while I/O executes,
  
  • I/O subsystem signals process when I/O completed via signal or callback
  
  • difficult to use but very efficient
Two I/O Methods

(a) synchronous

(b) asynchronous
Kernel I/O Subsystem

- **I/O scheduling**
  - to queue I/O requests via per-device queue
  - to schedule I/O for fairness and quality of service

- **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”
  - to improve performance (double buffering in video)
Kernel I/O Subsystem

- **Caching**: hold a copy of data for fast access
  - key to performance
  - sometimes combined with buffering
- **Spooling**: hold output 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
## 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
Sun Enterprise 6000 Device-Transfer Rates

- System Bus
- HyperTransport (32-pair)
- PCI Express 2.0 (×32)
- Infiniband (QDR 12X)
- Serial ATA (SATA-300)
- Gigabit Ethernet
- SCSI bus
- FireWire
- Hard disk
- Modem
- Mouse
- Keyboard

Transfer Rates Range:
- 0.00001 to 1E-1
Error Handling

- Some OSes try to recover from errors
  - e.g., device unavailable, transient write failures
  - sometimes via retrying the read or write
  - some systems have more advanced error handling
    - track error frequencies, stop using device with high error frequency
- Some OSes just return an error number or code when I/O request fails
  - system error logs hold problem reports
I/O Protection

• OS need to protect I/O devices
  • e.g., keystrokes can be stolen by a **keylogger** if keyboard is not protected
  • always assume user may attempt to obtain illegal I/O access
• To protect I/O devices:
  • define all I/O instructions to be privileged
    • I/O must be performed via system calls
  • memory-mapped I/O and I/O ports must be protected too
Use System Call to Perform I/O
Kernel Data Structures

- Kernel keeps state info for I/O components
  - e.g., open file tables, network connections, character device state
  - many data structures to track buffers, memory allocation, “dirty” blocks
    - sometimes very complicated
- Some OS uses message passing to implement I/O, e.g., Windows
  - message with I/O information passed from user mode into kernel
  - message modified as it flows through to device driver and back to process
UNIX I/O Kernel Structure

- 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
I/O Requests to Hardware

- System resource access needs to be mapped to hardware
- 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. User process initiates an I/O request via a system call.
2. The kernel checks if the I/O request can already satisfy the request.
3. If yes, the I/O completed, input data available, or output completed.
4. If no, the request is sent to the device driver, and a block process is initiated if appropriate.
5. The kernel I/O subsystem processes the request, issues commands to the controller, and configures the controller to block until interrupted.
6. Device controller commands are received.
7. The device controller monitors the device and generates an interrupt when the I/O is completed.
8. An interrupt handler receives the interrupt, stores data in the device-driver buffer if input, and signals the unblock device driver.
9. The I/O completed, generate interrupt.
Streams

- Stream is a full-duplex communication channel between a user-level process and a device in Unix systems

- A stream consists of:
  - **stream head** interfaces with the user process
  - **driver end** interfaces with the device
  - zero or more stream modules between them (stacked)
    - each module contains a read queue and a write queue
  - Message passing is used to communicate between queues
    - asynchronous internally, synchronous for user interface
Streams Structure
Performance

- I/O is a major factor in system performance:
  - CPU to execute device driver, kernel I/O code
  - context switches due to interrupts
  - data buffering and copying
    - network traffic especially stressful
Intercomputer Communications

Diagram showing the process of data transmission between two computer systems, highlighting the role of interrupts, network packets, and kernel interactions in the communication process.
Performance

- To improve performance
  - reduce number of context switches
  - reduce data copying
  - reduce interrupts by using large transfers, smart controllers, polling
  - use DMA
  - use smarter hardware devices
  - move user processes 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)

increased flexibility
End of Chapter 13