COS 318: Operating Systems

I/O Device Interactions and Drivers

Jaswinder Pal Singh and a Fabulous Course Staff
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Topics
- So far:
  - Management of CPU and concurrency
  - Management of main memory and virtual memory
- Next: Management of the I/O system
  - Interacting with I/O devices
  - Device drivers
  - Storage Devices
- Then, File Systems
  - File System Structure
  - Naming and Directories
  - Efficiency/Performance
  - Reliability and Protection

Input and Output
- A computer
  - Computation (CPU, memory hierarchy)
- Move data into and out of a system (between I/O devices and memory hierarchy)
- Challenges with I/O devices
  - Different categories with different characteristics: storage, networking, displays, keyboard, mouse ...
  - Large number of device drivers to support
  - Device drivers run in kernel mode and can crash systems
- Goals of the OS
  - Provide a generic, consistent, convenient and reliable way to access I/O devices
  - Achieve potential I/O performance in a system

Revisit Hardware
- Compute hardware
  - CPU cores and caches
  - Memory
  - I/O
  - Controllers and logic
- I/O Hardware
  - I/O bus or interconnect
  - I/O device
  - I/O controller or adapter
    - Often on parent board
    - Cable connects it to device
    - Often using standard interfaces: IDE, SATA, SCSI, USB, FireWire...
    - Has registers for control, data signals
    - Processor gives commands and/or data to controller to do I/O
I/O Hierarchy
- As with memory, fast I/O with less "capacity" near CPU, slower I/O with greater "capacity" further away

![I/O Hierarchy Diagram]

Performance Characteristics
- Overhead
  - CPU time to initiate an operation
- Latency
  - Time to transfer one bit
  - Overhead + time for 1 bit to reach destination
- Bandwidth
  - Rate at which subsequent bits are transferred or reach destination
  - Bits/sec or Bytes/sec
- In general
  - Different transfer rates
  - Abstraction of byte transfers
  - Amortize overhead over block of bytes as transfer unit

<table>
<thead>
<tr>
<th>Device</th>
<th>Transfer rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>10Bytes/sec</td>
</tr>
<tr>
<td>Mouse</td>
<td>100Bytes/sec</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10GE NIC</td>
<td>1.2GBytes/sec</td>
</tr>
</tbody>
</table>

A typical PC bus structure

Interacting with Devices
- A device has an interface, and an implementation
  - Interface exposed to external software, typically by device controller
  - Implementation may be hardware, firmware, software
- Mechanisms
  - Programmed I/O (PIO)
  - Interrupts
  - Direct Memory Access (DMA)
**Programmed I/O**

- **Example**
  - RS-232 serial port

- **Simple serial controller**
  - Status registers (ready, busy, ...)
  - Data register

- **Output**
  - CPU:
    - Wait until device is not "busy"
    - Write data to "data" register
  - Tell device "ready"

- **Device**
  - Wait until "ready"
  - Clear "ready" and set "busy"
  - Take data from "data" register
  - Clear "busy"

---

**Polling in Programmed I/O**

- **Wait until device is not “busy”**
  - A polling loop

- **Advantages**
  - Simple

- **Disadvantage**
  - Slow
  - Waste CPU cycles

- **Example**
  - If a device runs 100 operations / second, CPU may need to wait for 10 msec or 10,000,000 CPU cycles (1Ghz CPU)

---

**Interrupt-Driven Device**

- **Allows CPU to avoid polling**

- **Example: Mouse**

- **Simple mouse controller**
  - Status registers (done, int, ...)
  - Data registers (ΔX, ΔY, button)

- **Input**
  - Mouse:
    - Wait until "done"
    - Store ΔX, ΔY, and button into data registers
    - Raise interrupt
  - CPU (interrupt handler)
    - Clear "done"
    - Move ΔX, ΔY, and button into kernel buffer
    - Set "done"
    - Call scheduler

---

**Another Problem**

- **CPU has to copy data from memory to device**
- **Takes many CPU cycles, esp for larger I/Os**

- **Can we get the CPU out of the copying loop, so it can do other things in parallel while data are being copied?**
Direct Memory Access (DMA)

1. The DMA device driver is told to transfer disk data to buffer at address X.
2. The DMA controller tells the DMA controller to transfer data to buffer at address X.
3. The DMA controller initiates DMA transfer.
4. The DMA controller sends each byte to DMA controller.

Example of disk

- DMA Write
  - CPU:
    - Wait until DMA device is "ready"
    - Clear "ready"
    - Set DMAWrite, address, size
    - Set "start"
    - Block current thread/process
  - Disk adaptor:
    - DMA data to device (size--; address++)
    - Interrupt when "size == 0"
  - CPU (interrupt handler):
    - Put the blocked thread/process into ready queue
    - Disk: Move data to disk

Where Are these I/O "Registers"?

- Explicit I/O "ports" for devices
  - Accessed by privileged instructions (in, out)
- Memory mapped I/O
  - A portion of physical memory for each device
  - Advantages
    - Simple and uniform
    - CPU instructions can access these "registers" as memory
  - Issues
    - These memory locations should not be cached. Why?
    - Mark them not cachable
- Both approaches are used

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>200-2FF</td>
<td>game controller</td>
</tr>
<tr>
<td>2FF-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>3FF-3F7</td>
<td>diskette drive controller</td>
</tr>
<tr>
<td>3FF-3FF</td>
<td>serial port (primary)</td>
</tr>
</tbody>
</table>
I/O Software Stack

User-Level I/O Software

Device-Independent
OS software

Device Drivers

Interrupt handlers

Hardware

I/O Interface and Device Drivers

Device driver

Device controller

Device

Device controller

Device

Operating System

Hardware

I/O Interface and Device Drivers

- I/O system calls encapsulate device behaviors in
generic classes
- Device-driver layer hides differences among I/O
controllers from kernel
- Devices vary in many dimensions
  - Character-stream or block
  - Sequential or random-access
  - Sharable or dedicated
  - Speed of operation
  - Read-write, read only, or write only

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</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</td>
<td></td>
</tr>
<tr>
<td>I/O direction</td>
<td>read only</td>
<td>CD-ROM graphics controller disk</td>
</tr>
</tbody>
</table>
What Does A Device Driver Do?

- Provide “the rest of the OS” with APIs
  - Init, Open, Close, Read, Write, …
- Interface with controllers
  - Commands and data transfers with hardware controllers
- Driver operations
  - Initialize devices
  - Interpret outstanding requests
  - Manage data transfers
  - Accept and process interrupts
  - Maintain the integrity of driver and kernel data structures

Device Driver Operations

- Init (deviceNumber)
  - Initialize hardware
- Open(deviceNumber)
  - Initialize driver and allocate resources
- Close(deviceNumber)
  - Cleanup, deallocate, and possibly turnoff
- Device driver types
  - Character: variable sized data transfer
  - Terminal: character driver with terminal control
  - Block: fixed sized block data transfer
  - Network: streams for networking

Character and Block Interfaces

- Character device interface (keyboard, mouse, ports)
  - read(deviceNumber, bufferAddr, size)
    - Reads “size” bytes from a byte stream device to “bufferAddr”
  - write(deviceNumber, bufferAddr, size)
    - Write “size” bytes from “bufferAddr” to a byte stream device
- Block device interface (disk drives)
  - read(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from “deviceAddr” to “bufferAddr”
  - write(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from “bufferAddr” to “deviceAddr”
  - seek(deviceNumber, deviceAddress)
    - Move the head to the correct position
    - Usually not necessary

Network Devices

- Different enough from the block & character devices to have own interface
- Unix and Windows/NT include socket interface
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)
Clocks and Timers

- Provide current time, elapsed time, timer
- if programmable interval time used for timings, periodic interrupts
- ioctl (on UNIX) covers odd aspects of I/O such as clocks and timers

Unix Device Driver Entry Points

- init()
- Initialize hardware
- start()
  - Boot time initialization
- open(dev, flag, id) and close(dev, flag, id)
  - Initialization resources for read or write and release resources
- halt()
  - Call before the system is shutdown
- intr(vector)
  - Called by the kernel on a hardware interrupt
- read(...) and write() calls
  - Data transfer
- poll(pri)
  - Called by the kernel 25 to 100 times a second
- ioctl(dev, cmd, arg, mode)
  - special request processing

Synchronous and Asynchronous I/O

- Synchronous I/O
  - Read() or write() will block a user process until its completion
  - Easy to use and understand
  - OS overlaps synchronous I/O with another process
  - Blocking versus non-blocking variants

- Asynchronous I/O
  - Process runs while I/O executes
  - Let user process do other things before I/O completion
  - I/O completion will notify the user process

Synchronous Blocking Read

Application  ->  Kernel  ->  HW Device

- Application calls syscall
- Switch to Kernel context
- Driver Initiates DMA read
- DMA read
- Interrupt
- Copy to User buf
- Unblock
- Switch to User context
- Unblock
- return
Synchronous Blocking Read
- A process issues a read call which executes a system call
- System call code checks for correctness and buffer cache
- If it needs to perform I/O, it will issue a device driver call
- Device driver allocates a buffer for read and schedules I/O
- Initiate DMA read transfer
- Block the current process and schedule a ready process
- Device controller performs DMA read transfer
- Device sends an interrupt on completion
- Interrupt handler wakes up blocked process (make it ready)
- Move data from kernel buffer to user buffer
- System call returns to user code
- User process continues

Asynchronous Read
- Application
  - `aio_read`: begin asynchronous read
  - `aio_write`: begin asynchronous write
  - `aio_cancel`: cancel asynchronous read/write requests
  - `aio_error`: retrieve Asynchronous I/O error status
  - `aio_fsync`: asynchronously force I/O completion, and sets errno to ENOSYS
  - `aio_return`: retrieve status of Asynchronous I/O operation
  - `aio_suspend`: suspend until Asynchronous I/O completes
  - `lio_listio`: issue list of I/O requests
- Kernel
  - Switch to Kernel context
- HW Device
  - Driver initiates DMA read
  - Copy to User buf
- DMA read
  - Interrupt
  - Do other work
  - `aio_return`
  - `incomplete`
  - Complete

Asynchronous I/O
POSIX P1003.4 Asynchronous I/O interface functions:
(available in Solaris, AIX, Tru64 Unix, Linux 2.6,…)
- `aio_read`: begin asynchronous read
- `aio_write`: begin asynchronous write
- `aio_cancel`: cancel asynchronous read/write requests
- `aio_error`: retrieve Asynchronous I/O error status
- `aio_fsync`: asynchronously force I/O completion, and sets errno to ENOSYS
- `aio_return`: retrieve status of Asynchronous I/O operation
- `aio_suspend`: suspend until Asynchronous I/O completes
- `lio_listio`: issue list of I/O requests

Other Device Driver Design Issues
- Statically install device drivers
  - Reboot OS to install a new device driver
- Dynamically download device drivers
  - No reboot, but use an indirection
  - Load drivers into kernel memory
  - Install entry points and maintain related data structures
  - Initialize the device drivers
Dynamic Binding of Device Drivers

- Indirection
  - Indirect table for all device driver entry points
- Download a driver
  - Allocate kernel memory
  - Store driver code
  - Link up all entry points
- Delete a driver
  - Unlink entry points
  - Deallocate kernel memory

Issues with Device Drivers

- Flexible for users, ISVs and IHVs
  - Users can download and install device drivers
  - Vendors can work with open hardware platforms
- Dangerous
  - Device drivers run in kernel mode
  - Bad device drivers can cause kernel crashes and introduce security holes
- Progress on making device drivers more secure
- How much of OS code is device drivers?

I/O Software Stack

Kernel I/O subsystem: “Scheduling”

- Some I/O request ordering via per-device queue
- Some OSes try fairness
Kernel I/O subsystem (contd.)

- Buffering - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch (e.g., packets in networking)

- How to deal with address translation?
  - I/O devices see physical memory, but programs use virtual memory
  - E.g. DMA may require contiguous physical addresses

- Caching - fast memory holding copy of data
  - Reduce need to go to devices, key to performance

- Spooling - hold output for a device
  - If a device can serve only one request at a time, i.e., printing
  - Used to avoid deadlock problems

Life cycle of an I/O request

- Error handling
  - OS can recover from disk read, device unavailable, transient write failures
  - Most return an error no. or code when I/O request fails
  - System error logs hold problem reports

- 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 locations must be protected too

Kernel data structures

- State info for I/O components, including open file tables, network connections, character device state

- Many complex data structures to track buffers, memory allocation, "dirty" blocks

- Some use object-oriented methods and message passing to implement I/O
Summary

- **IO Devices**
  - Programmed I/O is simple but inefficient
  - Interrupt mechanism supports overlap of CPU with I/O
  - DMA is efficient, but requires sophisticated software

- **Synchronous and Asynchronous I/O**
  - Asynchronous I/O allows user code to perform overlapping

- **Device drivers**
  - Dominate the code size of OS
  - Dynamic binding is desirable for many devices
  - Device drivers can introduce security holes
  - Progress on secure code for device drivers but completely removing device driver security is still an open problem

- **Role of device-independent kernel software**