COS 318: Operating Systems

I/O Device Interactions and Drivers

Jaswinder Pal Singh
Computer Science Department
Princeton University

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

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

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

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
    - Special I/O instructions (w. port addr.) or memory mapped I/O
I/O Hierarchy

- As with memory, fast I/O with less "capacity" near CPU, slower I/O with greater "capacity" further away.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
</table>

Memory Bus (proprietary)

- General I/O Bus (e.g., PCI, I/O)

Peripherals Bus (e.g., IDE, SATA, USB)

A typical PC bus structure

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>1.2Bytes/sec</td>
</tr>
</tbody>
</table>

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
  - May also poll to wait for device to complete its work
- **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

**Interrupt Handling Revisited/Refined**

- **Save more context**
- **Mask interrupts if needed**
- **Set up a context for interrupt service**
- **Set up a stack for interrupt service**
- **Acknowledge the interrupt controller, enable it if needed**
- **Save context to PCB**
- **Run the interrupt service**
- **Unmask interrupts if needed**
- **Possibly change the priority of the process**
- **Run the 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)

**Example of disk**
- A simple disk adaptor
  - Status register (ready, ...)
  - DMA command
  - DMA memory address and size
  - DMA data buffer
- **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>200–20F</td>
<td>game controller</td>
</tr>
<tr>
<td>2F0–2FF</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>3F0–3FF</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

- 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
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 turn off
- Device driver types
  - Character: variable sized data transfer
  - Block: fixed sized block data transfer
  - Terminal: character driver with terminal control
  - 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
  - Separates network protocol from network operation
- 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 (require system services)
- `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**
  - Calling process waits for I/O call to return before doing anything
  - Blocking 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
  - Nonblocking I/O
    - Return as much data (and count of it) as available right away

- **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

- 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

- Switch to Kernel context
- Driver initiates DMA read
- Copy to User buf
- Unblock
- Interrupt
- Do other work
- aio_return
- Copy to User buf
- Complete
- Interrupt
- 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

Why Buffering in Kernel?

• Speed mismatch between the producer and consumer
  - Character device and block device, for example
  - Adapt different data transfer sizes (packets vs. streams)
• DMA requires contiguous physical memory
  - I/O devices see physical memory
  - User programs use virtual memory
• Spooling
  - Avoid deadlock problems
• Caching
  - Reduce I/O operations

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 driver more secure
- How much of OS code is device drivers?

Next: Kernel I/O Subsystem

I/O Software Stack

Next: Kernel I/O Subsystem

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)
  - To maintain “copy semantics”
    - Copy data from user buffer to kernel buffer

- 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

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

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

Life cycle of an I/O request
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

From User Request 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

Another example: blocked read w. DMA
- A process issues a read call which executes a system call
- System call code checks for correctness and 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
- Controller performs DMA data transfer, blocks the process
- Device generates an interrupt on completion
- Interrupt handler stores any data and notifies completion
- Move data from kernel buffer to user buffer and wakeup blocked process
- User process continues

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