Operating Systems  
(1DT020 & 1TT802)  

Lecture 14  

I/O Systems  

May 19, 2009  

Léon Mugwaneza  

http://www.it.uu.se/edu/course/homepage/os/vt09
Goals

- Input / Output systems
  - Hardware Access
  - Device Drivers

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)
Requirements for I/O

• Without I/O, computers are useless
• Thousands of devices, each slightly different OS
  – should offer standard interfaces to applications
• Devices unreliable: media failures and transmission errors
  – How can we make them reliable?
• Devices unpredictable and/or slow
  – How can we manage them if we don’t know what they will do or how they will perform?
• Some operational parameters:
  – Byte/Block
    » Some devices provide single byte at a time (e.g. keyboard)
    » Others provide whole blocks (e.g. disks, networks, etc)
  – Sequential/Random
    » Some devices must be accessed sequentially (e.g. tape)
    » Others can be accessed randomly (e.g. disk, cd, etc.)
  – Polling/Interrupts
    » Some devices require continual monitoring
    » Others generate interrupts when they need service
Modern I/O Systems

- Monitor
- Graphics controller
- Bridge/memory controller
- Cache
- IDE disk controller
- Expansion bus interface
- Keyboard
- Expansion bus
- Parallel port
- Serial port
I/O Device Examples and Speeds

- **I/O Speed:** bytes transferred per second

<table>
<thead>
<tr>
<th>Device</th>
<th>Behavior</th>
<th>Partner</th>
<th>Data rate (Kbytes/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>Input</td>
<td>Human</td>
<td>0.01</td>
</tr>
<tr>
<td>Mouse</td>
<td>Input</td>
<td>Human</td>
<td>0.02</td>
</tr>
<tr>
<td>Voice output</td>
<td>Output</td>
<td>Human</td>
<td>5.00</td>
</tr>
<tr>
<td>Floppy disk</td>
<td>Storage</td>
<td>Machine</td>
<td>50.00</td>
</tr>
<tr>
<td>Laser Printer</td>
<td>Output</td>
<td>Human</td>
<td>100.00</td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>Storage</td>
<td>Machine</td>
<td>10,000.00</td>
</tr>
<tr>
<td>Wireless Network</td>
<td>I or O</td>
<td>Machine</td>
<td>10,000.00</td>
</tr>
<tr>
<td>Graphics Display</td>
<td>Output</td>
<td>Human</td>
<td>30,000.00</td>
</tr>
<tr>
<td>Wired LAN Network</td>
<td>I or O</td>
<td>Machine</td>
<td>125,000.00</td>
</tr>
</tbody>
</table>

- **Device Rates vary over many orders of magnitude**
  - from mouse to Gigabit LAN: *7 orders of magnitude*!
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices
The Goal of the I/O Subsystem

• Provide Uniform Interfaces, Despite Wide Range of Different Devices
  – This code works on many different devices:
    ```c
    FILE fd = fopen("/dev/something","rw");
    for (int i = 0; i < 10; i++) {
        fprintf(fd,"Count %d\n",i);
    }
    close(fd);
    – Why? Because code that controls devices ("device driver") implements standard interface.

⚠️ We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
Want Standard Interfaces to Devices

• **Block Devices**: e.g. disk drives, tape drives, DVD-ROM
  – Access blocks of data
  – Commands include `open()`, `read()`, `write()`, `seek()`
  – Raw I/O or file-system access
  – Memory-mapped file access possible

• **Character Devices**: e.g. keyboards, mice, serial ports, some USB devices
  – Single characters at a time
  – Commands include `get()`, `put()`
  – Libraries layered on top allow line editing

• **Network Devices**: e.g. Ethernet, Wireless, Bluetooth
  – Different enough from block/character to have own interface
  – Unix and Windows include **socket** interface
    » Separates network protocol from network operation
    » Includes `select()` functionality
  – Usage: pipes, FIFOs, streams, queues, mailboxes
How Does User Deal with Timing?

• **Blocking Interface: “Wait”**
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data

• **Non-blocking Interface: “Don’t Wait”**
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing

• **Asynchronous Interface: “Tell Me Later”**
  - When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
Main components of Intel Chipset: Pentium 4

- **Northbridge:**
  - Handles memory
  - Graphics

- **Southbridge: I/O**
  - PCI bus
  - Disk controllers
  - USB controllers
  - Audio
  - Serial I/O
  - Interrupt controller
  - Timers
How does the processor actually talk to the device?

- CPU interacts with a **Controller**
  - Contains a set of **registers** that can be read and written
  - May contain memory for request queues or bit-mapped images

- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - **I/O instructions**: in/out instructions
    » Example from the Intel architecture: `out 0x21, AL`
  - **Memory mapped I/O**: load/store instructions
    » Registers/memory appear in physical address space
    » I/O accomplished with load and store instructions
Example: Memory-Mapped Display Controller

- **Memory-Mapped:**
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the “frame buffer”) changes image on screen
    - Addr: 0x8000F000—0x800FFFF
  - Writing graphics description to command-queue area
    - Say enter a set of triangles that describe some scene
      - Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - Say render the above scene
      - Addr: 0x0007F004

- Can protect with page tables
Transfering Data To/From Controller

- **Programmed I/O:**
  - Each byte transferred via processor in/out or load/store
  - **Pro:** Simple hardware, easy to program
  - **Con:** Consumes processor cycles proportional to data size

- **Direct Memory Access:**
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly

- **Sample interaction with DMA controller (from book):**

```plaintext
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
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
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
```
A Kernel I/O Structure
Device Drivers

• **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  – Supports a standard, internal interface
  – Same kernel I/O system can interact easily with different device drivers
  – Special device-specific configuration supported with the `ioctl()` system call

• **Device Drivers typically divided into two pieces:**
  – **Top half**: accessed in call path from system calls
    » Implements a set of **standard, cross-device calls** like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    » This is the kernel’s interface to the device driver
    » **Top half will start** I/O to device, may put thread to sleep until finished
  – **Bottom half**: run as interrupt routine
    » Gets input or transfers next block of output
    » May wake sleeping threads if I/O now complete
Life Cycle of An I/O Request

User Program
- request I/O
- system call
- can already satisfy request?
  - yes
  - transfer data (if appropriate) to process, return completion or error code
  - kernel I/O subsystem
- no
  - send request to device driver, block process if appropriate
  - kernel I/O subsystem

Kernel I/O Subsystem
- device driver
- determine which I/O completed, indicate state change to I/O subsystem

Device Driver Top Half
- process request, issue commands to controller, configure controller to block until interrupted
- device-driver commands

Device Driver Bottom Half
- interrupt handler
  - receive interrupt, store data in device-driver buffer if input, signal to unblock device driver

Device Hardware
- monitor device, interrupt when I/O completed
- device controller
- I/O completed, generate interrupt
I/O Device Notifying the OS

• The OS needs to know when:
  – The I/O device has completed an operation
  – The I/O operation has encountered an error

• I/O Interrupt:
  – Device generates an interrupt whenever it needs service
  – Handled in bottom half of device driver
    » Often run on special kernel-level stack
  – Pro: handles unpredictable events well
  – Con: interrupts relatively high overhead

• Polling:
  – OS periodically checks a device-specific status register
    » I/O device puts completion information in status register
    » Could use timer to invoke lower half of drivers occasionally
  – Pro: low overhead
  – Con: may waste many cycles on polling if infrequent or unpredictable I/O operations

• Actual devices combine both polling and interrupts
  – For instance: High-bandwidth network device:
    » Interrupt for first incoming packet
    » Poll for following packets until hardware empty
Summary

• I/O Devices Types:
  – Many different speeds (10 bytes/sec to GBytes/sec)
  – Different Access Patterns: block, char, net devices
  – Different Access Timing: Non-/Blocking, Asynchronous

• I/O Controllers: Hardware that controls actual device
  – CPU accesses through I/O instructions, load/store to special physical memory
  – Report results through interrupts or a status register that processor looks at occasionally (polling)

• Device Driver: Device-specific code in kernel