Operating Systems
2010/2011

Input/Output Systems– part 1 (ch13)

Shudong Chen
Objectives

- Discuss the principles of I/O hardware and its complexity
- Explore the structure of an operating system’s I/O subsystem
- Provide details of the performance aspects of I/O hardware and software
- Discuss operating-system services provided for mass storage --- RAID
Requirements

• I/O devices to deal with -
  – many different types
    • keyboard, screen, disk, temperature sensor, ....
  – very different speeds
  – specific (brand-specific) approaches

• Abstract, coherent view on input and output devices
  – efficient
  – facilitate sharing

• Requires
  – a classification
    • user (API) level: a standardized approach, per class
    • OS level: an encapsulation of brand-specific issues
      – share common functionality (e.g. buffer management)
      – standardize on OS-device interface (driver)
  – .... two places of abstraction: OS-driver and OS-user!
Agenda

- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- Streams
- Performance (disk scheduling)
- RAID
I/O Hardware

- **Common concepts**
  - **port**
    - a connection point between devices and a computer system
      - e.g., serial port, parallel port
  - **bus**
    - a set of wires and a defined protocol that specifies a set of messages that can be sent on the wires
      - daisy chain: devA -> devB -> devC -> a PC port
        - operated as a bus
  - **controller**
    - a collection of electronics that can operate a port, a bus, or a device
A Typical PC Bus Structure
I/O Hardware (Cont.)

• How does a controller communicate with a processor or a device?
  – one or more registers in the controller for data and control signals
  – these registers constitute the software / hardware interface
  – the processor reads and writes bit patterns in these registers
  – a device can call corresponding interfaces
Controller register example: disk controller

- An I/O port consists of four registers
  - **Data-in**
    - read by the host to get input
  - **Data-out**
    - written by the host to send output
  - **Status:**
    - contains bits that can be read by the host
    - bits indicate states
      - command completed
      - data ready for transfer
      - errors occurred
  - **Control:**
    - written by a host to start a command or to change the mode of a device
      - set certain bit for communication manners: full-duplex, half-duplex
      - set a bit to enable parity checking
Controller interface example

- **Opcode**
  - written by the driver to describe the operation it wishes the controller to carry out
  - operation code, e.g., read, write

- **Operands**
  - parameters of opcode
  - its number depends on the specific opcode
    - seek cylinder: the cylinder number as the sole operand
    - read disk: the track number with the current cylinder and the sector within the track

- **Busy, Status**
  - indicates the state of the controller, e.g.,
    - busy, readiness, errors

- **Buffer registers**
  - to operate the data bytes to be transferred to or from the device
  - the buffer size varies, e.g.,
    - a single byte: a value typed in on keyboard
    - thousands of bytes: for disks

- **Notice:** this is a quite complex example!
  - used by block-oriented devices (disks)
  - character-oriented devices have no opcode or operand registers
**IO instructions vs. Memory mapped**

- **Two communication ways:**
  - **Direct I/O instructions to control devices**
    - specifying the transfer of a byte or word to an I/O port address
  - **Memory-mapped I/O**
    - registers are mapped into the address space of the processor
    - CPU executes I/O requests using standard data-transfer instructions to read and write the registers
    - e.g., memory-mapped region used by a graphics controller
      - The process sends output to the screen by writing data to this region
      - The controller generates the screen image by reading this memory contents
      - **Advantages:**
        » Simple technique
        » Writing bytes to the graphic memory is faster than issuing I/O instructions
    - **Disadvantages:**
      » Memory-mapped device registers is vulnerable
      » Protected memory can reduce the risk
IO instructions vs. Memory mapped (Cont.)

- Memory mapped I/O
  - registers *(device interface)* are mapped onto certain memory locations
    - notice that this requires that part of the virtual memory space is consistently mapped to the same physical frame
  - it is possible for a user process to access a device directly
    - ask the OS to assign *a particular physical frame* to a virtual memory segment
      - the *mmap()* call in Posix

- I/O instructions
  - the instruction set is extended with special opcodes
    - numbered devices, part of the instruction
    - no interference with virtual memory
    - not possible to map device in user space
Different Instruction Set

• (a) the CPU must distinguish between main memory and device addresses, using a different instruction format for each
  – to store the content of a given CPU register \textit{cpu\_reg} into a given memory location \(k\)
    • \textit{store cpu\_reg, k}
  – to copy the content of the same CPU register to a controller’s register
    • \textit{io\_store cpu\_reg, dev\_no, dev\_reg}

• (b) the same instruction format for both main memory and devices
  • \textit{store cpu\_reg, k}
  • \textit{store cpu\_reg, n}
  • (device0 is mapped to address \(n\))

\begin{itemize}
\item Forms of device addressing
\item (a) Explicit
\item (b) memory-mapped
\end{itemize}
Communication between host & controller

- Host and controller coordinates by handshaking

- Assumptions:
  - 2 bits are used to coordinate the producer-consumer relationship between the host and the controller
    - *set* a bit: to write a 1 into the bit
    - *clear* a bit: to write a 0 into the bit
  - Controller indicates its state through the *busy* bit in the *status* register
    - *set busy* when it is busy working
    - *clear busy* when it is ready to accept the next command
  - Host signals its wishes via the *command-ready* bit in the *command* register
    - *set command-ready* when a command is available for the controller to execute
Basic handshaking between host & controller

- Handshaking example (output operation) --- the following loop is repeated for each byte
  1. The host repeatedly reads the busy bit until that bit becomes clear
  2. The host sets the write bit in the command register and writes a byte into the data-out register
  3. The host sets the command-ready bit
  4. When the controller notices that the command-ready bit is set, it sets the busy bit
  5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device
  6. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished

CPU is busy-waiting or polling
Polling

• Determines state of device by reading the registers
  – busy (status register)
  – error (status register)
  – command-ready (command registers: data-in, data-out)

• *Busy-wait* cycle to wait for I/O from device

• When the controller and the device are fast, this method is reasonable.

• Polling becomes inefficient when it is attempted repeatedly
  – yet rarely finds a device to be ready for service
  – while other CPU processing remains undone
Interrupt-Driven I/O Cycle

• The controller notifies the CPU when the device becomes ready for service.
  – CPU *Interrupt-request line* triggered by I/O device
  – *Interrupt handler* receives interrupts
Interrupt-controller hardware

• The basic interrupt mechanism enables the CPU to respond to an asynchronous event.

• Modern computer hardware needs to provide 3 more features
  – The ability to defer interrupt handling during critical processing
  – An efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt
  – Multilevel interrupt to distinguish between high- and low-priority interrupt

• Interrupt-controller hardware
  – CPU has two interrupt request lines
    • Nonmaskable: reserved for events such as unrecoverable memory errors
    • Maskable: ignore or delay some interrupts
  – Interrupt vector to dispatch interrupt to correct handler
    • contains the memory addresses of specialized interrupt handlers
    • to reduce the need of a handle to search all possible interrupt sources
    • based on priority, some are nonmaskable
  – Interrupt priority levels
    • Enable a high-priority interrupt to preempt the execution of a low-priority interrupt
# 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>
Example interaction: read data from file

- Interaction is named: **system call**
- Execute C statement: \( \text{status} = \text{read}(\text{fd}, \text{buffer}, \text{ nbytes}) \)
  - read \( \text{nbytes} \) bytes from \( \text{fd} \), storing it in \( \text{buffer} \)
  - is part of a program running on top of the OS (e.g. banking program)

- **memory**: kernel space/user space
  - kernel space only accessible with processor in kernel mode

- **parameters**: either via registers or via memory
  
  1-3: pushing parameters
  4: call library function
  5: put code for \text{read} in reg.
  6: trap (Linux@x86: int 0x80): switch mode and call trap handler
  7: handler calls read function handler
  8: handler performs read actions (a.o. store data at address). Here suspension of the calling process may occur if data needs to come from an io device.
  9-11: control back to caller
Direct Memory Access

- Used to avoid *programmed I/O (PIO)* for large data movement
  - PIO
    - A general-purpose processor
    - Watches status bits
    - Feeds data into a controller register one byte at a time

- Requires *DMA controller*
  - DMA controller
    - A special-purpose processor
    - Bypasses CPU to transfer data directly between I/O device and memory
DMA via separate DMA controller

- The host writes a DMA command block into memory (separate DMA controller)
  - A pointer to the source of a transfer
  - A pointer to the destination of the transfer
  - A count of the number of bytes to be transferred
- CPU writes the address of this block to the DMA controller, then goes on with other work
- DMA controller proceeds to operate the memory bus directly
  - Placing addresses on the bus to perform transfers
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
Communication between CPU & controller

- The sharing & interference problem coming from concurrency in controller and CPU is *identical* to earlier studied problems of communication & synchronization

  - *polling by OS: busy waiting*
    - (time triggered) polling, possibly with lowered frequency for efficiency
    - CPU directs discrete I/O actions, initiates them and observes completion

  - *interrupt service routine, executed on behalf of controller:*
    - uses condition synchronization (or action synchronization, if only the order matters) with processes waiting on its result
    - interrupt service routine operates “concurrently” with all other activities
    - variables in the driver and OS are shared
    - device informs CPU about completion/readiness and other events

  - *DMA*
    - similar, but now for entire transfer only one interrupt
    - device capable of moving data to and from memory independently; just needs initialization
Agenda

- I/O Hardware
- **Application I/O Interface**

- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- Streams
- Performance (disk scheduling)
- RAID
Requirements

• I/O devices to deal with -
  – many different *types*
    • keyboard, screen, disk, temperature sensor, ....
  – very different *speeds*
  – specific (brand-specific) approaches

• Abstract, coherent view on input and output devices
  – efficient
  – facilitate sharing

• Requires
  – a *classification*
    • user (API) level: a standardized approach, per class
    • OS level: an *encapsulation* of brand-specific issues
      – share common functionality (e.g. buffer management)
      – standardize on OS-device interface (driver)
  – .... two places of abstraction: OS-driver and OS-user!
Application I/O Interface

• I/O system calls encapsulate device behaviors in generic classes
  – to hide hardware difference from applications
  – called device drivers in kernel module
  – each generic class (driver) is accessed through a standardized set of instructions (interface)

• Device-driver layer hides differences among I/O controllers from kernel making the I/O subsystem independent of the hardware
  – to simplify the job of operating-system developer
  – to benefit hardware manufacturers when designing new devices
    • either make new devices compatible with an existing host controller interface
    • or write device drivers to interface the new hardware to popular OS
A layered kernel I/O structure

<table>
<thead>
<tr>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel I/O subsystem</td>
<td>SCSI device controller</td>
</tr>
<tr>
<td></td>
<td>keyboard device controller</td>
</tr>
<tr>
<td></td>
<td>mouse device controller</td>
</tr>
<tr>
<td></td>
<td>PCI bus device controller</td>
</tr>
<tr>
<td></td>
<td>floppy device controller</td>
</tr>
<tr>
<td></td>
<td>ATAPI device controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel</td>
<td>SCSI device driver</td>
</tr>
<tr>
<td></td>
<td>keyboard device driver</td>
</tr>
<tr>
<td></td>
<td>mouse device driver</td>
</tr>
<tr>
<td></td>
<td>PCI bus device driver</td>
</tr>
<tr>
<td></td>
<td>floppy device driver</td>
</tr>
<tr>
<td></td>
<td>ATAPI device driver</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI devices</td>
<td>SCSI device controller</td>
</tr>
<tr>
<td>keyboard</td>
<td>keyboard device controller</td>
</tr>
<tr>
<td>mouse</td>
<td>mouse device controller</td>
</tr>
<tr>
<td>...</td>
<td>PCI bus device controller</td>
</tr>
<tr>
<td>PCI bus</td>
<td>floppy device controller</td>
</tr>
<tr>
<td>floppy-disk drives</td>
<td>ATAPI device controller</td>
</tr>
</tbody>
</table>

Johan J. Lukkien, Shudong Chen, shudong.chen@tue.nl
TU/e Computer Science, System Architecture and Networking
## Varied dimensions 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></td>
<td>asynchronous</td>
<td></td>
</tr>
<tr>
<td>sharing</td>
<td>dedicated</td>
<td>tape</td>
</tr>
<tr>
<td></td>
<td>sharable</td>
<td>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</td>
<td></td>
</tr>
<tr>
<td></td>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>I/O direction</td>
<td>read only</td>
<td>CD-ROM graphics</td>
</tr>
<tr>
<td></td>
<td>write only</td>
<td>controller disk</td>
</tr>
<tr>
<td></td>
<td>read–write</td>
<td></td>
</tr>
</tbody>
</table>
Classical classifications

- Block devices (e.g., tape unit, disk drive, CD)
  - communicate blocks of data
  - commands include read, write, seek (‘random access’)  
  - raw I/O (database) or file-system access

- Character-stream devices (e.g., keyboard, mouse, sensors)
  - communicate bytes (or: some smallest units)
  - commands include get, put, not seekable

- Network devices
  - similar to stream-oriented devices
  - need to establish connection first
    - two roles: client (seeks contact) and server (exposes contact)
  - provide a network I/O interface
    - socket interface in Unix and Windows NT
  - includes select functionality
    - to return information about
      - sockets with a to-be-received packet or a to-be-sent packet
    - to eliminate the polling and busy-waiting
  - separates network operation from network protocol
    - encapsulate network behaviors
    - still can use underlying network hardware and protocol stack
‘Classical classification’: not enough

• Remember:
  – only provide a new function within the OS if building that function on top of the OS represents a (significant) performance loss

• Example: accessing sound and graphics hardware
  – requires highly efficient access (gaming!)
  – not really covered by the standard interface: \textit{read()} / \textit{write()}
    • e.g. playing sound through streaming
    • reading and writing frame buffers, for graphics
Solution: memory mapping

- Use *memory mapping*: the device controller interface is mapped into user accessible memory
  - works for all devices
  - requires memory-mapped I/O
  - optimization not easy
    - particularly, interrupts are cumbersome (often disallowed)
    - requires OS support for application doing optimization
  - no protection
  - no device-type standardization

- Actually, general resources can be made available through memory mapping
  - e.g. files
Posix call: `mmap()`

```c
void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

Example:
```
mem_area = mmap (WhereIWantIt, len, protection, flags, fd, offset);
```

- `fd[offset..offset+length-1]` is mapped to `start[0..length-1]` (all in bytes)
- `prot` concerns admitted operations
  - read, write, read/write, execute etc.
- `flags` concerns sharing (across processes)
- `fd` is a file descriptor. It is the result from `open()`
Uses of `mmap()`

- If `fd` refers to a normal file
  - The file is directly accessed by accessing memory

- If `fd` refers to `/dev/kmem`
  - A piece of memory (e.g., a mapped device) can be directly written

- If `fd` refers to `/dev/rhda0`
  - The disk becomes accessible through memory references

- If `fd` is the result from `shm_open()` (shared memory open)
  - `shm_open` creates a piece of kernel memory for sharing with other processes
  - The `mmap()` makes this available in user space
‘Classical classification’: not enough (Cont.)

• Direct access to device drivers
  – an escape (*back door*)
  – passes arbitrary commands from an application to a driver transparently
  – e.g., `ioctl(int d, int request, …)` (on Unix, Linux)
    • manipulates the underlying device parameters of special files
    • `d` is an open file descriptor
    • `request` is a device-dependent request code number
    • the third argument is an *untyped pointer* to an arbitrary data structure in memory
      – either an integer value, possibly unsigned (going to the driver)
      – or a pointer to data (either going to the driver, coming back from the driver, or both)

• Clocks and Timers
  – `ioctl` covers this odd aspects of I/O
  – provide current time, elapsed time, timer
  – *programmable interval timer* used for timings, periodic interrupts
    • set to wait a certain amount of time, then generate an interrupt,
    • once or repeatedly
Blocking and Nonblocking I/O

- **Blocking** - process suspended until I/O completed
  - easy to use and understand
  - used by most operating systems for application interface
  - insufficient for some needs
    - physical I/O actions are generally asynchronous --- taking unpredictable amount of time

- **Nonblocking** - I/O call returns as much as available
  - user interface, e.g.,
    - receives keyboard and mouse input while processing and displaying data on screen
  - implemented via multi-threading
    - some threads perform blocking system call
    - while others continue executing
  - returns quickly with count of bytes read or written

- **Asynchronous** - process runs while I/O executes
  - An alternative way of nonblocking I/O
  - Difficult to use
  - I/O subsystem signals process when I/O completed
Two I/O Methods

- **Synchronous:**
  - The execution of the application is suspended.
  - With received system call values, it resumes execution.

- **Asynchronous:**
  - The system call returns immediately, without waiting for the I/O to complete.
  - The application continues to execute.
  - The completion of the I/O at some future time is communicated to the application.
Layered view on i/o

- File system
- Virtual memory management
- Block-oriented device management
- Stream-oriented device management
- Network interface

Driver level

- CD-ROM driver
- Hard disk driver
- Keyboard driver
- Printer driver
- Network driver

Raw device

- CD-ROM controller
- Hard disk controller
- Keyboard controller
- Printer controller
- Network controller

Driver standardizes device interface

Kernel API

Listen/accept/bind/connect/send/receive

Listen/accept/bind/connect/send/receive

Johan J. Lukkien, Shudong Chen, shudong.chen@tue.nl

TU/e Computer Science, System Architecture and Networking
Low-level I/O intelligence hierarchy

• Direct control
  – CPU in full control, including timing

• Direct control delegated to special processor inside the device, the controller
  – Programmable I/O actions
    • CPU directs discrete I/O actions, initiates them and observes completion
  – Interrupts
    • device informs CPU about completion/readiness and other events
  – DMA
    • device capable of moving data to and from memory independently; just needs initialization
  – Programmed I/O (“I/O channel”)
    • device executes I/O program found in main memory (e.g. the ‘MOVE machine’ or advanced DMA controllers)

• From DMA onwards, there can be a separate module besides the CPU for controlling the devices
  – e.g. a general DMA controller for controlling other devices
Software approaches

• Direct control
  – real-time tasks, time-triggered

• Direct control delegated to special processor inside the device, the *controller*
  – Programmable I/O actions
    • (time triggered) polling, possibly with lowered frequency for efficiency
  – Interrupts
    • suspend i/o process – wakeup through interrupt handler *to continue with next i/o event*
  – DMA
    • similar, but now for entire transfer only one interrupt
  – Programmed I/O ("I/O channel")
    • similar, but the behavior of the i/o processor is variable through a program in main memory
Exercises

• Ch13 – 2, 3, 6, 8, 9, 10, 12, 15