Chapter 11 I/O Processing

11.1 Basic Issues in Device Management
11.2 A Hierarchical Model
11.3 I/O Devices
11.4 Device Drivers
   - Memory-Mapped vs Explicit Device Interfaces
   - Programmed I/O with Polling
   - Programmed I/O with Interrupts
   - Direct Memory Access (DMA)
11.5 Device Management
   - Buffering
   - Error Handling
   - Disk Scheduling
   - Device Sharing
11.6 The Abstract I/O Interface

Metric Units

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Exponent (10^n)</th>
<th>Prefix</th>
<th>Exponent (10^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milli</td>
<td>-3</td>
<td>Kilo</td>
<td>3</td>
</tr>
<tr>
<td>Micro</td>
<td>-6</td>
<td>Mega</td>
<td>6</td>
</tr>
<tr>
<td>Nano</td>
<td>-9</td>
<td>Giga</td>
<td>9</td>
</tr>
<tr>
<td>Pico</td>
<td>-12</td>
<td>Tera</td>
<td>12</td>
</tr>
<tr>
<td>Femto</td>
<td>-15</td>
<td>Peta</td>
<td>15</td>
</tr>
<tr>
<td>Atto</td>
<td>-18</td>
<td>Exa</td>
<td>18</td>
</tr>
<tr>
<td>Zepto</td>
<td>-21</td>
<td>Zetta</td>
<td>21</td>
</tr>
<tr>
<td>Yocto</td>
<td>-24</td>
<td>Yotta</td>
<td>24</td>
</tr>
</tbody>
</table>
Basic issues

- **I/O devices:**
  - communication devices
    - input only (keyboard, mouse, joystick)
    - output only (printer, display)
    - input/output (network card)
  - storage devices
    - input/output (disk, tape)
    - input only (CD-ROM)

- **main tasks of I/O system:**
  - present logical (abstract) view of devices
    - hide details of hardware interface
    - hide error handling
  - facilitate efficient use
    - overlap CPU and I/O
  - support sharing of devices
    - protection when device is shared (disk)
    - scheduling when exclusive access needed (printer)
A hierarchical model of I/O

Figure 11-1

- abstract I/O interface
  - block devices, character devices, network
- device-independent software
  - buffering, scheduling, caching
- device-dependent software
  - I/O drivers (supplied by device manufacturer)
Hierarchical Model of I/O System

- Device-Independent I/O Software
- Device Driver
- Hardware

I/O devices

- display monitors
  - character or graphics oriented
  
  Figure 11-2

- different data rates:
  - 25 x 80 characters vs 800 x 600 x 256
  - 30-60 times/sec
I/O devices

- keyboards
  - most common: QWERTY
  - very low data rate (<10 char/sec)

- pointing devices
  - mouse (optical, optical-mechanical)
  - trackball
  - joystick
  - low data rate (hundreds of bytes/sec)

I/O devices

- printers
  - line printers, dot-matrix, ink-jet, laser
  - low data rates
  - character-oriented

- scanners
  - digitize picture into bit map (similar to video RAM)
  - low data rates
I/O devices

- (floppy) disks
  - surface, tracks/surface, sectors/track, bytes/sector
  - all sectors numbered sequentially 0..(n-1): logical disk

Figure 11-3 (a,b)
**I/O devices**

- floppy disks
  - track skew
    - account for seek to next track to minimize latency

  *Figure 11-3 (c)*

- double-sided floppy
  - tracks with same diameter: cylinder
  - number all sector within cylinder consecutively to minimize seek

  *Figure 11-3 (d)*
Fig. 11-3: disk numbering
(c) with track skew

Fig. 11-3: disk numbering
(d) double sided
I/O devices

- hard disks
  - multiple surfaces
  - higher densities and data rates than floppy

![Figure 11-4](Image)

<table>
<thead>
<tr>
<th>floppy</th>
<th>hard disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes/sec</td>
<td>512-4096</td>
</tr>
<tr>
<td>sec/track</td>
<td>9, 15, 18, 36</td>
</tr>
<tr>
<td>tracks/surf</td>
<td>40, 80, 160</td>
</tr>
<tr>
<td># surf</td>
<td>1-2</td>
</tr>
<tr>
<td>seek</td>
<td>30-100 ms</td>
</tr>
<tr>
<td>rotation</td>
<td>400-700 rpm</td>
</tr>
<tr>
<td></td>
<td>3600-10,000 rpm</td>
</tr>
</tbody>
</table>

---

Save up to 90GB* of data!

The **NEW** Iomega® REV™ 35GB/90GB* drive is the first backup solution that combines the speed, reliability and ease-of-use of a hard drive with removable and low-cost capacity expansion.

- **SECURE** Password protect confidential data and encrypt your backups.
- **RELIABLE** Proven hard disk technology for ultimate reliability.
- **EXPANDABLE** Add capacity when you need it with most efficient Iomega REV disks.

**NEW** Iomega REV Drives

USB 2.0 External Drive: $399.99
ATAPI Internal Drive: $279.99

**Buy now.**
I/O devices

- optical disks
  - CD-ROM, CD-R (WORM), CD-RW
  - originally designed for music
  - data stored as continuous spiral, subdivided into sectors
  - constant linear speed (200-530 rpm)
  - higher storage capacity than magnetic disks: 0.66 GB/surface
I/O devices

- **data transfer rates of disks**
  - *sustained*: continuous data delivery
  - *peek*: transfer once r/w head is in place
    * depends on rotation speed and data density
    * 1 revolution passes over all sectors of 1 track
  - **Example: 7200 rpm, 100 sect/track, 512 B/sect**
    * 7200 rpm: 60,000/7200 = 8.3 ms/rev
    * 8.3/100 = 0.083 ms/sector
    * 512 bytes transferred in 0.083 ms: ~6MB/s

---

**Fig 11-5: Disk Characteristics**

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Floppy Disk</th>
<th>Hard Disk</th>
<th>CD ROM Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes per sector</td>
<td>512</td>
<td>512 - 4,096</td>
<td>2048</td>
</tr>
<tr>
<td>sectors per track</td>
<td>9, 15, 18, 36</td>
<td>100 - 400</td>
<td>333,000</td>
</tr>
<tr>
<td>tracks per surface (number of cylinders)</td>
<td>40, 80, 160</td>
<td>1,000 - 10,000</td>
<td>sectors per surface</td>
</tr>
<tr>
<td>number of surfaces</td>
<td>1 - 2</td>
<td>2 - 24</td>
<td>1 - 2</td>
</tr>
<tr>
<td>seek time - adjacent</td>
<td>3 - 5 ms</td>
<td>0.5 - 1.5 ms</td>
<td>NA</td>
</tr>
<tr>
<td>seek time - average</td>
<td>30 - 100 ms</td>
<td>5 - 12 ms</td>
<td>80 - 400 ms</td>
</tr>
<tr>
<td>rotational speed</td>
<td>400 - 700 rpm</td>
<td>3,600 - 10,000 rpm</td>
<td>(200 - 530)*1000 rpm</td>
</tr>
</tbody>
</table>
I/O devices

- magnetic tapes (reel or cartridge)
  - large storage capacity (GB)
  - data transfer rate: ~ 2 MB/sec
- networks (interface card)
  - Ethernet, token ring, slotted ring
    * controller implements protocol to accept, transmit, receive packets
  - modem
    * convert between analog and digital (phone lines)
    * character-oriented (like printer and keyboard)

Device drivers

- accept command from application
  - get/put char, read/write block, send/rec. packet
- interact with device controller (HW) to carry out command
- typical device controller interface: set of registers
  Figure 11-6
- Example: serial or parallel port on PC
  - generic driver reads/writes chars to registers
Fig 11-6: device controller interface

Device

Driver

opcode register

operand registers

busy register

status register

data buffer

write

read

read/write

Fig 11-7: device addressing (a) explicit

addresses

0

Main memory

\( n-1 \)

\( \text{dev}_0 \)

opcode

operand 0

operand 1

\( \ldots \)

\( \text{dev}_1 \)

opcode

operand 0

operand 1

\( \ldots \)
Device drivers

- memory-mapped vs explicit dev interface
  - similar idea to memory-mapped files
    Figure 11-7

- explicit: special I/O instruction:
  \[ \text{io\_store\ cpu\_reg, dev\_no, dev\_reg} \]

- memory-mapped: regular CPU instruction:
  \[ \text{store\ cpu\_reg, n} \]
  (n is a memory address)

Fig 11-7: device addressing

(b) memory-mapped addresses

0

Main
memory

n-1

n

opcode

operand 0
operand 1

device 0
controller

n+1

n+2

... ...

opcode

operand 0
operand 1

device 1
controller

... ...
Device Addressing

Store to memory vs. store to device interface

store cpu reg, k
io store cpu reg, dev no, dev reg

Polling:
Input:
i = 0;
do {
    write_reg(opcode, read);
    while (busy_flag == true) {...??...};
    mm in area[i] = data_buffer;
    increment i;
    compute;
} while (data_available)

Device IO

Polling:

Output:
i = 0;
do { compute;
data_buffer = mm_out_area[i];
increment i;
write_reg(opcode, write);
while (busy_flag == true) {...??...};
} while (data_available)
Programmed I/O with polling

- CPU is responsible for
  - moving every char to/from controller buffer
  - detecting when I/O operation completed
- protocol to input a character:

Figure 11-8
Programmed I/O with polling

• driver operation to input sequence of chars
  
  \[
  i = 0; \\
  \text{do } \{ \text{write\_reg}(\text{opcode}, \text{read}); \\
  \quad \text{while (busy\_flag == true) \{\ldots\}}; \\
  \quad \text{mm\_in\_area}[i] = \text{data\_buffer}; \\
  \quad \text{increment } i; \\
  \quad \text{compute;} \\
  \} \text{ while (data\_available)}
  \]

• what to do while waiting?
  – idle (busy wait)
  – some other computation
    * how frequently to poll? -- Figure 11-9
  – give up CPU
    * device may remain unused for a long time

• driver operation to output sequence of chars
  
  \[
  i = 0; \\
  \text{do } \{ \text{compute}; \\
  \quad \text{data\_buffer} = \text{mm\_in\_area}[i]; \\
  \quad \text{increment } i; \\
  \quad \text{write\_reg}(\text{opcode}, \text{write}); \\
  \quad \text{while (busy\_flag == true) \{\ldots\}}; \\
  \} \text{ while (data\_available)}
  \]

• what to do while waiting?
  – idle (busy wait)
  – some other computation
    * how frequently to poll? -- Figure 11-9
  – give up CPU
    * device may remain unused for a long time
Programmed I/O with interrupts

- CPU is responsible for
  - moving chars to/from controller buffer, but
- interrupt signal informs CPU when I/O operation completes
- protocol to input a character:

  Figure 11-10
**Fig 11-10: programmed IO with interrupts**

Diagram showing the interaction between the CPU, controller, device, and main memory. The diagram illustrates how data is transferred and the roles of various registers and buffers in the process.

**DMA**

- **Driver operation to input sequence of chars**
  
  ```
  write_reg(mm_buf, m);
  write_reg(count, n);
  write_reg(opcode, read);
  block to wait for interrupt;
  ```
  
  - writing opcode triggers DMA controller
  - DMA controller issues interrupt after n chars in memory

- **I/O processor (channel)**
  
  - extended DMA controller
  - executes I/O program in own memory
Device IO with Interrupt

Input:
  \[ i = 0; \]
  \[ \text{do } \{ \text{write\_reg}(\text{opcode}, \text{read}); \text{block to wait for interrupt;} \text{mm\_in\_area}[i] = \text{data\_buffer}; \text{increment } i; \text{compute}; \} \text{while (data\_available)} \]

Output:
  \[ i = 0; \]
  \[ \text{do } \{ \text{compute}; \text{data\_buffer} = \text{mm\_out\_area}[i]; \text{increment } i; \text{write\_reg}(\text{opcode}, \text{write}); \text{block to wait for interrupt;} \} \text{while (data\_available)} \]

Use of I/O Interrupts

Input:
  \[ i := 0; \]
  \[ \text{repeat} \]
  \[ \text{Read;} \]
  \[ \text{wait\_for\_interrupt;} \]
  \[ \text{in\_area}[i] := \text{in}; \]
  \[ \text{Increment } i; \]
  \[ \text{Compute} \]
  \[ \text{until all\_data\_in} \]

Output:
  \[ i := 0; \]
  \[ \text{repeat} \]
  \[ \text{Compute;} \]
  \[ \text{out} := \text{out\_area}[i]; \]
  \[ \text{Increment } i; \]
  \[ \text{Write;} \]
  \[ \text{wait\_for\_interrupt} \]
  \[ \text{until all\_data\_out} \]
Programmed I/O with interrupts

- Example: keyboard driver
  
  ```
  i = 0;
  do {  block to wait for interrupt;
    mm_in_area[i] = data_buffer;
    increment i;
    compute(mm_in_area[]);
  } while (data_buffer != ENTER)
  ```

- timing of interrupt-driven I/O

  Figure 11-11
  - more OS overhead but better device utilization

Fig 11-11: timing of interrupt-driven IO
Programmed I/O with interrupts

- compare polling with interrupts:
  
  ```c
  i = 0;
  do { write_reg(opcode, read);
      while (busy_flag == true) { ... };
      mm_in_area[i] = data_buffer;
      increment i;
      compute;
  } while (...)
  ```

  ```c
  i = 0;
  do { write_reg(opcode, read);
      block to wait for interrupt;
      mm_in_area[i] = data_buffer;
      increment i;
      compute;
  } while (...)
  ```

DMA

- CPU only initiates operation
- DMA controller transfers data directly to/from main memory
- interrupt when transfer completed
- protocol to input data using DMA:

  Figure 11-12
Device management

- **circular buffer**
  - when average times to fill and empty are comparable but vary over time: circular buffer absorbs bursts
  - producer and consumer each use an index
    - nextin gives position of next input
    - nextout gives position of next output
    - both are incremented modulo n at end of operation
Circular Buffer

User Process:
   do {
       buffer.remove(data);
       compute(data);
   } while (data_available)

Driver:
   do {
       write_reg(...);
       block to wait for interrupt;
       buffer.deposit(data_buffer);
   } while (data_available)
Device management

- device-independent techniques
- reasons for buffering
  - allows asynchronous operation of producers and consumers
  - allows different granularities of data
  - consumer or producer can be swapped out while waiting for buffer fill/empty

Device management

- single buffer operation
  Figure 11-14 (a)
- double buffer (buffer swapping)
  Figure 11-14 (b)
  - increases overlap
  - ideal when:
    time to fill = time to empty = constant
  - when times differ, benefits diminish
Double Buffer

**procedure** Get( I );

**begin**

while busy[ch] do; //wait
Read(ch, next(p_rec), buf[I]);
if i==1 then i := 2 else i := 1;
end

i := 2; // initial read
Read(ch, first(p_rec), buf[1]);
repeat
  Get( i );
  Compute (buf[I]);
until all_records_in;

---

Fig 11-14: buffering

(a) single buffer

(b) double buffer - buffer swapping
Buffer Swapping

Input:
   i = 0;
   b = 1;
   write_reg(mm_buf, buf[2]);
   write_reg(opcode, read);
   do { while (busy_flag == true); /* busy-wait */
       write_reg(mm_buf, buf[b]);
       write_reg(opcode, read);
       if (b == 1) b = 2; else b = 1;
       mm_in_area[i] = buf[b];
       increment i;
       compute;
   } while (data_available)

Fig. 11-15: Unix C-Lists

count = 104
rear
front
Device management

- error handling
  - persistent vs transient; SW vs HW
  - persistent SW error
    * repair/reinstall program
  - other errors: build in defense mechanisms
  - Examples:
    * transient SW errors: error correcting codes, retransmission
    * transient HW errors: retry disk seek/read/write
    * persistent HW errors: redundancy in storage media
Device management

- bad block detection and handling
  - block may be defective as a manufacturing fault or during use (a common problem)
  - parity bit is used to detect faulty block
  - the controller bypasses faulty block by renumbering
  - a spare block is used instead
  - two possible remappings:

Figure 11-17
* more work but contiguity of allocation preserved

Fig 11-17: handling of bad blocks
Device management

- **stable storage**
  - some applications cannot tolerate any loss of data (even temporarily)
  - stable storage protocols:
    * use 2 independent disks, A and B
    * write: write to A; if successful, write to B
    * read: read from A and B; if A!=B, go to recovery
    * recovery from media failure: A or B contains correct data; remap failed disk block
    * recovery from crash: if before writing A, B is correct; if after writing A, A is correct; recover

Fig 11-18: stable storage

(a) bad block
Device management

- RAID (redundant array of independent disks)
  - increased performance through parallel access
  - increased reliability through redundant data
  - maintain exact replicas of all disks

  Figure 11-19(a)

  * most reliable but wasteful
  - maintain only partial recovery information (e.g., error correcting codes)

  Figure 11-19(b)
Fig 11-19: (a) RAID – fully replication

\[ b_1 \quad b_2 \quad \ldots \quad b_n \]

Fig 11-19: (b) RAID -derived recovery information

\[ f(b_1, b_2, \ldots b_n) \]
Device management

- disk scheduling
  - minimize seek time and rotational delay
  - requests from different processes arrive concurrently:
    * scheduler must attempt to preserve locality
  - rotational delay:
    * order requests to blocks on each track in the direction of rotation:
      access in one rotation
    * proceed with next track on same cylinder

minimizing seek time: more difficult
- r/w arm can move in two directions
- minimize total travel distance
- guarantee fairness
- FIFO: simple, fair, but inefficient
  Figure 11-20 (a)
- SSTF: most efficient but prone to starvation
  Figure 11-20 (b)
- Scan: fair, acceptable performance
  Figure 11-20 (c)
Fig 11-20:  information

**FIFO**

Total distance traveled

- 23 cylinders

**SSTF**

- 14 cylinders

**Scan**

- 20 cylinders