4. Input/Output

Friday, May 1, 2020 6:18 PM

I/O devices

I/O device classification

Block devices

Ví dụ: ls -l /dev

```
brw-rw----    1 root     floppy     2,   0 Aug 24  2000 fd0
brw-rw----    1 root     disk       3,   0 Aug 24  2000 hda
```

The devices input/output data by blocks, each block has a specific address. A common block size is 512B to 64KB.

Character devices

Ví dụ: ls -l /dev

```
crw-rw----    1 root     root      10,   3 Aug 24  2000 atimouse
crw-r-----    1 root     sys       14,   4 Aug 24  2000 audio
```

The devices input/output data by a stream of characters, no data structure and without addresses, therefore there is no seek operation.

Exception

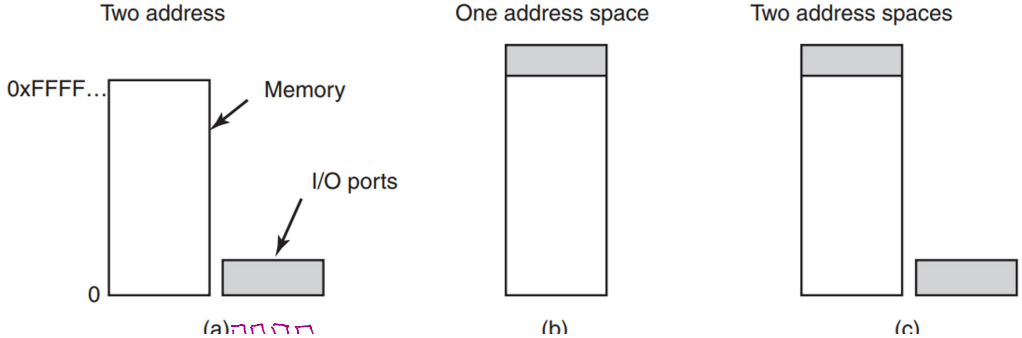
- Magnetic tapes: operate sequentially but input/output by blocks
- Clocks: neither belong to block nor character devices

Device controllers

I/O devices consist of 2 parts: the mechanic part and electronic part. The electronic part is called device controller or adapter. Function of a device controller is to convert bit streams into data blocks or character streams and error handling.
The OS uses device drivers to exchange data with device controllers.

Controlling I/O devices using CPU

A device controller has a set of registers. CPU uses these registers to command device controllers, such as to identify the device state or request for data etc.



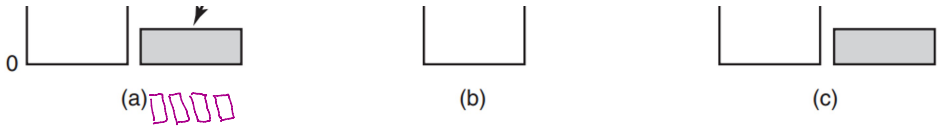


Figure 5-2. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

Using ports for I/O (isolated I/O)

Separate from the memory address space, a different addressing scheme is used for the I/O port space. CPU uses a set of registers as a channel for data exchange with I/O devices through specific ports

- IN register, port
- OUT port, register

Note that IN/OUT commands are at the machine code level, programmers have to embed this assembly code into a program written in high level language.

Memory-mapped I/O

The set of registers for I/O is mapped into the memory space. Access to these registers can be done as memory operations.

Pros:

- Easy to input/output using high level language
- To preven users from directly accessing I/O devices, the OS can map the I/O addresses to the virutal address space outside the range that users can access
- As an I/O device may have a separate set of registers located at different memory pages, the OS can share the devices to different users with different permissions by setting the page table entries point to those memory pages
- Accessing a register and accessing the memory are actually the same operation, there is no need to do it in 2 steps as in port I/O: first uing IN to read from a port into memory and then read the memory to get the value. For example

```
LOOP: TEST PORT_4     // check if por t 4 is 0
      BEQ READY       // if it is 0, go to ready
      BRANCH LOOP     // otherwise, continue testing
READY:
```

```
Loop : IN ax, 4
       Test ax
       Beq ready
       branch Loop
Ready:
```

Cons:

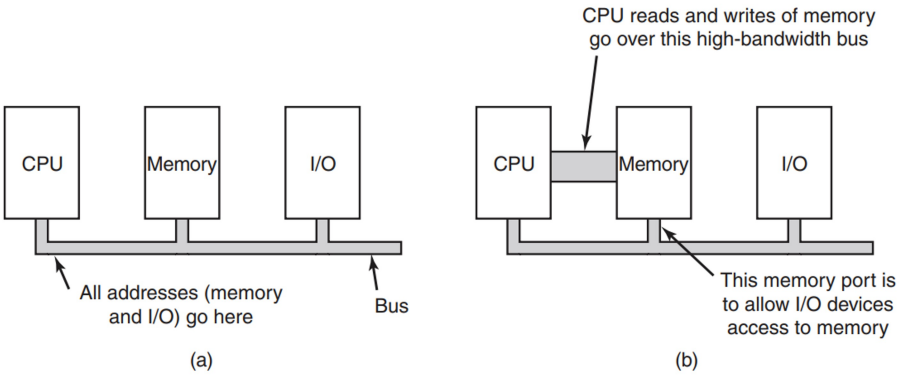
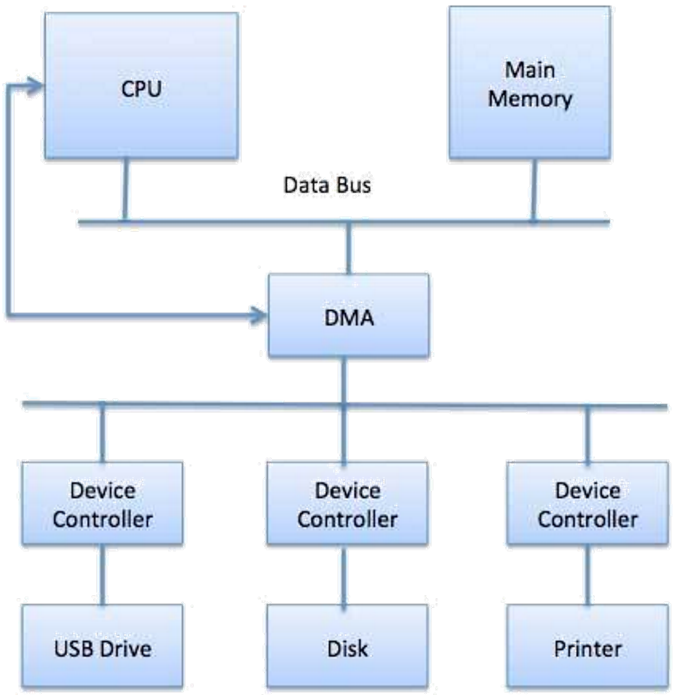


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

- Caching for I/O address space needs to be eleminated. This requires support from both hardware and OS
- As the same addressing scheme is used, when there is a memory access, both memory module and I/O devices need to check for which range the address belongs to. Modern PCs have a high speed bus between CPU and memory, therefore I/O devices may not receive the requested address. Solutions

- If the address is not in the memory, transfer the request to I/O devices through other buses
- Or putting a snooping device on the memory bus to pass all addresses to I/O devices. However, the snooping device is often not capable of processing at the speed of the memory
- Or having the memory controller to filter out addresses. The difficulty is the memory controller has to figure out all address ranges at boot time.

Direct Memory Acess



DMA (Direct Memory Access) allows I/O devices transfer data directly to memory without CPU intervention. An interrupt signal is created only after a whole data block has been in memory. DMA is often a separate device, serving for all I/O devices.

DMA controller

DMA controller (DMAC) has a set of registers that CPU and read from or write to, including registers for memory address, number of bytes and some control registers.

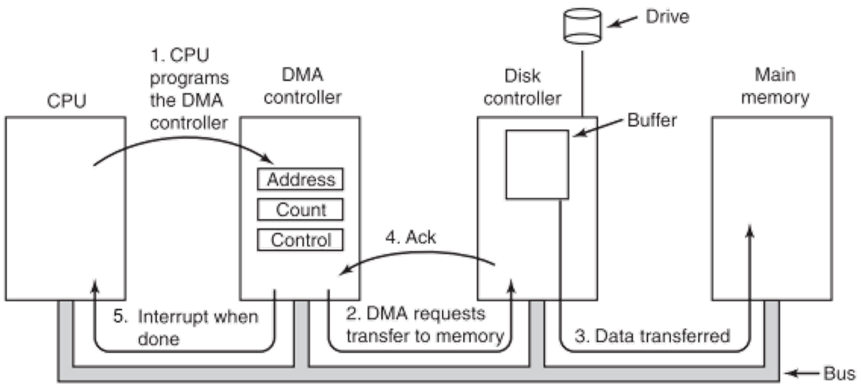


Figure 5-4. Operation of a DMA transfer.

Below are the steps of DMAC operation in case of reading from a disk

- Step 1: CPU send requests to DMAC via registers to specify the memory address, size fo data, then also request the disk controller to read data in a private buffer and verify the checksum. When the data is ready, DMAC can start.
- Step 2: DMAC sens a request to the disk controller for reading data. The disk controller does not care whether the request is from DMAC or CPU.
- Step 3: With the memory address is sent through the address bus, the disk controller writes data directly into the memory. Another solution is to have the disk controller sending a data word to DMAC, then DMAC writes the data word into the memory. This solution requires another bus cycle but it allows direct copy from a device to another device.
- Step 4: After the data transfer is complete, an acknowledgement is sent to DMAC. DMAC increases the memory address, decreases the data size and repeats from step 2.
- Step 5: After the whole data block transfer is done, DMAC generates an interrupt to give CPU to an application process for further processing.

Without DMAC, CPU will have to process every single byte from the disk controllerr and write it to the memory.

There are 2 data transfering modes in DMAC

- Block mode / burst mode: DMAC locks the bus, sends all data to the memory and unlocks the bus. This mode is efficient but causes great delay to CPU
- Word-by-word mode / cycle stealing: DMAC takes idle bus cycles to send data word by word, thus creates low delay.

Interrupts

Interrupt is a signal to inform OS to temporarily suspend the current process and let CPU to handle an I/O event. Interrupt helps OS to use CPU more efficiently, avoiding the situation where a process keeps CPU just to check whether data is readly.

Interrupt handling steps

1. A process sends an request to the I/O device. The process is moved into blocked state, waiting for I/O data.
2. When I/O data is ready, the device sends a signal to the interrupt controller
3. If the interrupt controller is available (i.e no interrupt event with higher priority), it will inform CPU by setting on signal on a CPU pin.
4. The interrupt controller sends the device number to CPU via bus, so CPU knows data from which device is ready
5. When CPU is ready to handle the interrupt, values of current registe including instruction pointer are stored on the stack, then CPU switches to the interrupt handling code. After handling the interrupt, the OS restores values of the saved regissters and continues to run the previously suspended process.

Precise vs Imprecise interrupts

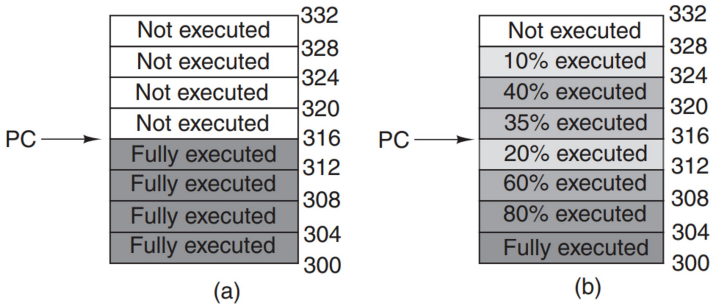
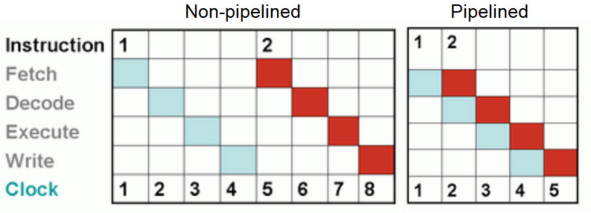


Figure 5-6. (a) A precise interrupt. (b) An imprecise interrupt.

An interrupt is called precise if after its execution, the computer state can be clearly defined. More specifically

- The program counter (PC, or instruction pointer) is saved in a known place
- All instructions before PC have completed
- No instruction beyond PC has finished
- Execution state of the instruction pointed to by PC is known

An interrupt is imprecise when PC does not create a clear boundary between the completed and unfinished instructions. Therefore when the interrupt occurs, the computer is not in a defined state. The reason is because modern CPUs have pipeline and superscalar architecture. In the pipeline model, the execution of an instruction is divided into phases: fetching, decoding, executing and writing down the result. This is to improve CPU performance. In the superscalar model, a complex instruction can be decomposed into micro instructions and run independently, may be out of order.



A superscalar CPU often has to save a large amount of current state data into the stack and restore them after that.
Some CPUs have a convention that I/O interrupts are precise and software interrupts can be imprecise, as after an software exception there is no need to continue the crashed application. Intel x86 CP Us use precise interrupts to be compatible with previous CPU versions. When there is an interrupt request, CPU has to determine the time to complete certain instructions, and after that the remain running instructions should not impact too much to the computer state. This mechanism increases CPU complexity considerably.

Principles of I/O software

Programmed I/O

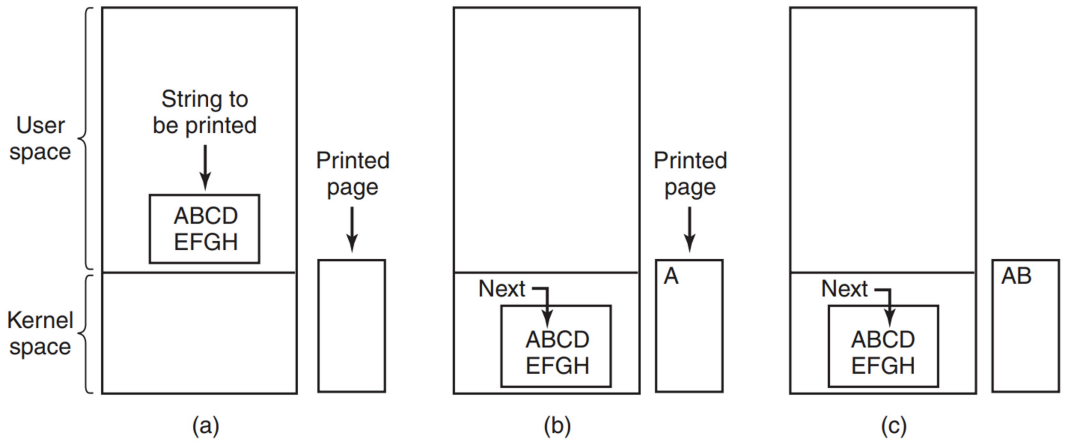


Figure
(a): A program opens a printer for writing, makes a system call to print a string
(b): OS copies the string into a kernel buffer, waits for the printer to be ready then OUT the 1st character.
(c): OS waits for the printer to be ready again, OUT the next character. Repeat the process until all characters are printed.

```
copy_from_user(buffer, p, count);
for (i=0; i<count; i++) {
    while (*printer_status_reg != READY);
    *printer_data_register = p[i];
}
return_to_user( );
```

```
/* p is the kernel buffer */
/* loop on every character */
/* loop until ready */
/* output one character */
```

Cons: I/O devices are slow, therefore CPU is always in the waiting state for printing the next character.

Interrupt-driven I/O

```
copy_from_user(buffer,p,count);
enable_interrupts();
while (*printer_status_reg!=READY);
*printer_data_register = p[0];
scheduler();
```

(a): print system call

```
if (count==0) {
    unblock_user();
} else {
    *printer_data_register=p[i];
    count = count-1;
    i = i+1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b) interrupt procedure

A process makes a system call and moves to the suspended state until the whole string is printed.
Figure (a): OS copies the string into a kernel buffer. CPU waits until the printer is ready and print the 1st character. CPU calls the scheduler and moves on another process.
Figure (b): When the printer is ready for the next character, it generates an interrupt. Figure (b) illustrates the code for interrupt handling. After each printed character, CPU returns to the suspended process.

Cons: Too many interrupts are generated. Time for saving and restoring the computer state affects system performance.

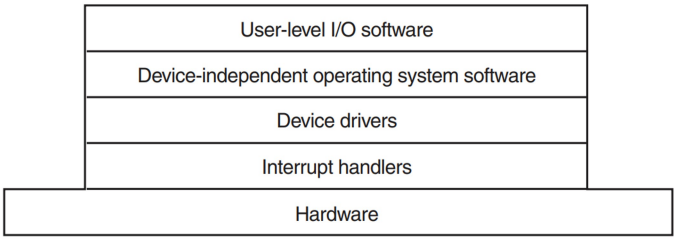
I/O using DMA

```
copy_from_user(buffer,p,count);
set_up_DMA_controller();
scheduler();

(a): print system call
```

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();

(b) Interrupt procedure
```

OS copies the string into a kernel buffer, sends a request to DMAC and call the scheduler to move CPU to another process.
DMAC acts like an I/O program (programmed I/O) to print each character. When the printing is done, DMAC generates an interrupt. The interrupt handler will wake up the suspended process to resume its job.

I/O software layers



Interrupt handlers

Handling data in an asynchronous way so the layers above can be programmed in a synchronous way, easier to write.

Solution: using semaphore or messaging

- down() wait for I/O data synchronization
- When I/O data is ready, an interrupt is generated and up() operation will wake up the process

Device drivers

The layer is to receive high-level I/O requests and translate them into commands to device controllers. This layer code is device dependent and helps upper layers to be device independent.

This layer knows specific device information, such as tracks, cylinders, diskarms, spinning speed etc. in case of hard disks.

Device-independent OS software

This layer provides following services:

- A common interface for all devices of the same genre: it provides a set of functions independent to devices of a specific manufacturer, avaialble to upper layers.
- Device naming:
  - Create a symbolic link between the device and a name
  - I-node has 2 fields: major device number to indicate the device number, and minor device number to specify the unit to be read or written
- Device protection:

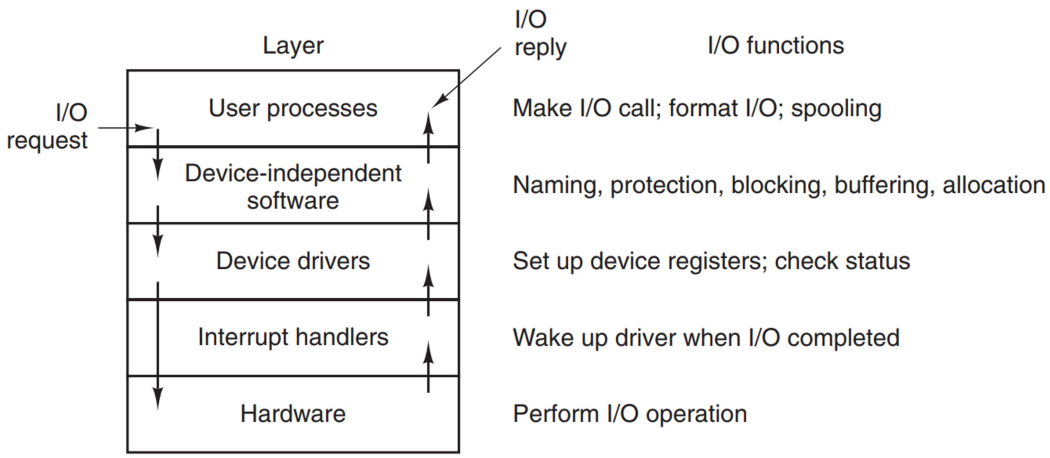
    - As a device is mapped to a file, device access permission is managed the same way as files
    - Access permission: 3 categories user, group, other, and 3 bits RWX for each category
- Provide a logical data block size independent to devices
- Caching: using buffer for caching data to improve I/O performance
- Allocate storage space for block devices such as for disks, data blocks can be managed using a bitmap or a linked list, together with different allocation strategies
- Allocate and release I/O devices: processes, that want to get access to a device, need to open the device file. If the device is not ready, the open operation will fail
- Error reporting: ask the device driver to try again, after a certain failures, report the error to the upper layer

User-level I/O software

This layer contains a set of library functions, available to user access. The code of these functions are included into user's programs at the compilation time or linked dynamically at runninng time. Many of these functions are to provide more user-friendly interface and make calls to other functions at the lower layers.

Notes: Working with a priviledged device requires using a spooler (queue). Otherwise a process opens the device but may forget or delay to close the file and that will seriously affect other processes waiting for turn.

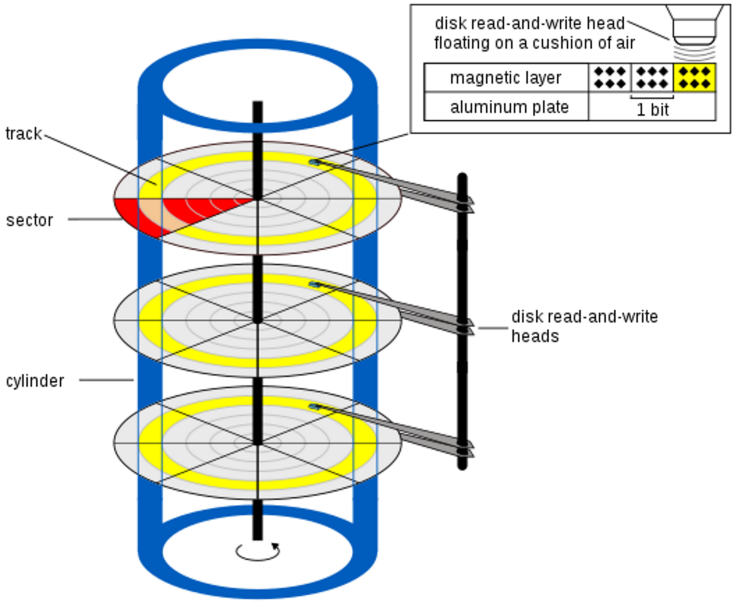
The I/O architecture can be summarized in the following diagram



Hard disks

Basic concepts

Block address



A disk is divided into tracks. Each track is divided into sectors. Tracks of the same diameter form a cylinder. A harddisk plattter can store data on both surfaces. Each surface is read/written by a disk head.
A data block is specified by its coordinate of (cylinder, head, sector).

Zone formatting

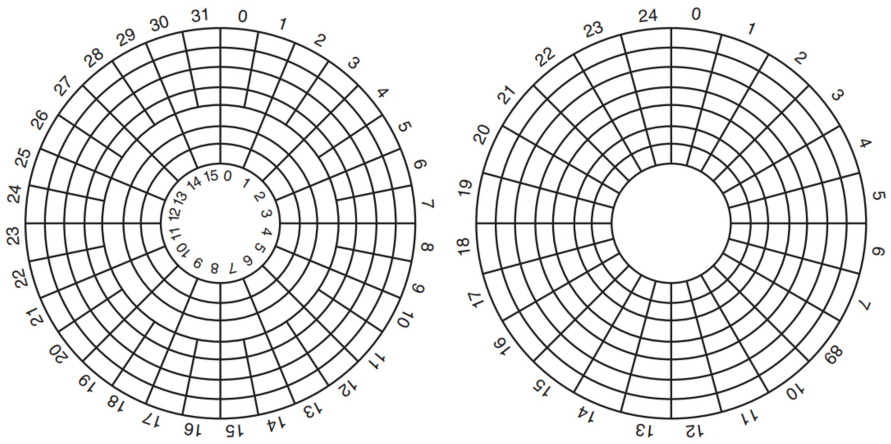


Figure 5-19. (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

Modern harddisks are physically formatted into zones (figure a). Outer zones have more sectors than inner zones. For example, WD 3000 HLFS harddisk has 16 zones, an outer zone has 4% of sectors more than its inner zone.
From the OS point of view, the disk is logically formatted as in figure (b).

Seek operation
Operating of seeking a data block can be performed in parallel on different disk surfaces. But it is not possible to perform parallel read/write operations.
Overlapped seek: while the disk controller is seeking on one surface, it can perform another seek on a different surface.

Harddisk performance

Harddisk performance depends on

- Seek time (move the head to the right track)
- Rotation time (wait until the right sector comes)
- Data transmission speed

Sector layout problem

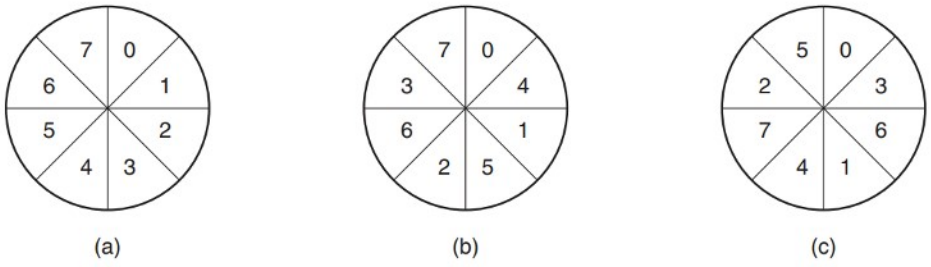


Figure 5-23. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

The purpose is to minimize the time wating for the next sector. Logically, content of a files should be stored on contingous sectors. However, formatting physical sectors contingously does not decrease rotation waiting time. It is because after a block reading operation, the disk controller needs some time to process the buffer, while the disk head has passed the beginning of the next sector.
Solution: format the sectors interleaved with each other, possibly single interleaving or double interleaving.

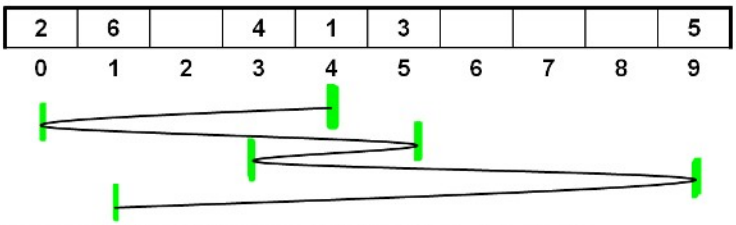
Disk-arm scheduling problem

Given a multiple platter HDD, with read/write requests coming from processes kept in a queue. The problem is to optimize the moving distance of disk heads in seek operation so that when a read/write operation is done on a platter, the disk head on another platter is already ready on the right track.
As seek operation is independent on platters, the problem on multiple platter can be reduced to the problem on a single platter surface.

Hàng đợi: 4, 0, 5, 3, 9, 1

$\Sigma = 4 + 5 + 2 + 6 + 8 = 25$

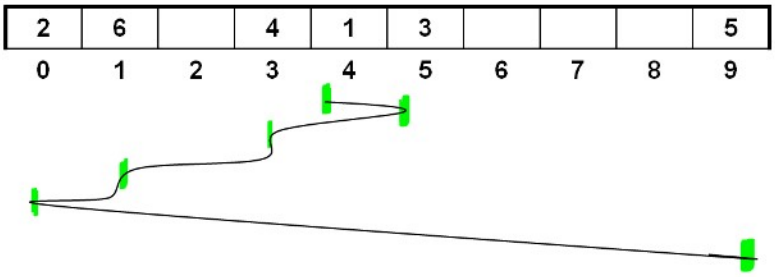
First Come - First Serve



The seek order is the sequence of requests in the queue.

Cons: no optimization of disk head movement

Shortest Seek First



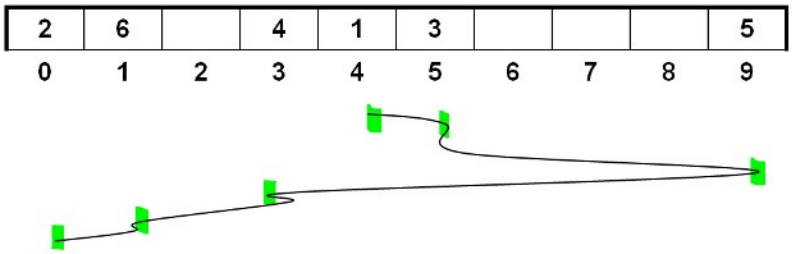
Hàng đợi: 4, 0, 5, 3, 9, 1

$\Sigma = 1 + 2 + 2 + 1 + 9 = 15$

The seek order is the next nearest track (nearest neighbour algorithm).

Cons: requests of blocks at the edges are often delayed.

Elevator algorithm



Hàng đợi: 4, 0, 5, 3, 9, 1

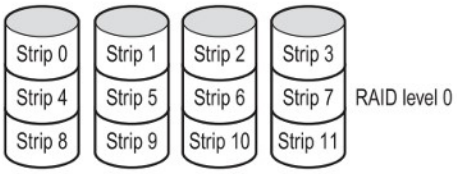
$\Sigma = 1 + 4 + 6 + 2 + 1 = 14$

Remember the current disk head direction and serve all requests on the way. If there are requests on the opposite direction, switch the current direction to the opposite.

RAID

Redundant Array of Inexpensive/Independent Disks (RAID) is a method of using a collection of inexpensive disks to deliver better performance than an expensive disk. There are different RAID levels, from 0 - 6. RAID levels are not hierachical.

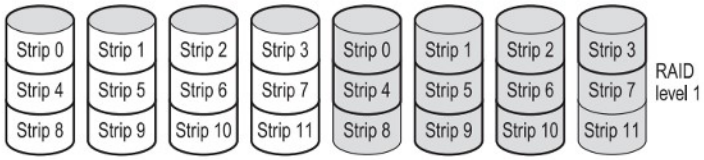
RAID level 0



Data is split into strips of k sectors each. Strips are written in a round-robin fashion on different disks.
Remarks:

- Pros: Increase read/write performance (parallel read/write is possible)
- Cons: Higher probability of data loss due to disk failure

RAID level 1

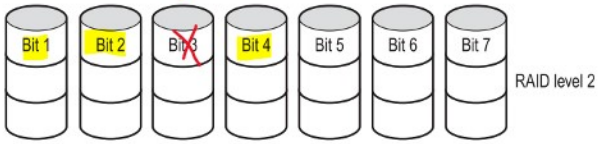


Similar to RAID 0 but there is a shadow copy of all data disks for the backup purpose.

For the write operations, each data strip has to be written on both group of disks (data and backup). Read operations can be done on any group of disks.
Remarks:

- Read operations give good performance as can be done on both group of disks
- Write operations give worse performance as replication on both groups of disks is required.
- Better fault tolerance

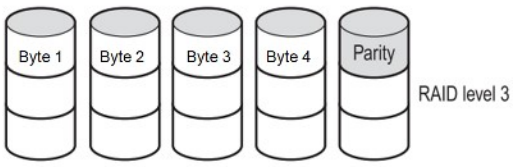
RAID level 2



Using Hamming code for error correction.
As in the figure, 4 data bits are inserted with error correction bits at position 1, 2, 4 to make a total 7 bit string. These 7 bits are spreaded on different disks, one bit on a disk. In reality, a 32 bit word will be inserted with 6 additional bits and written on 38 different disks in parallel.
Remarks:

- I/O bandwidth at the bit level is much higher(32 times higher for 32 bit words) but the number of separate I/O requests per second it can handle is no better than for a single drive
- Fault tolerance to one single disk failture
- All disks must have the same configuration and work in synchronization
- Require redundant disks for storing error correction bits. As in the case of 32 bit word, 19% of the disks is for error correction storage. Also special hardware is required for the error correction calculation.

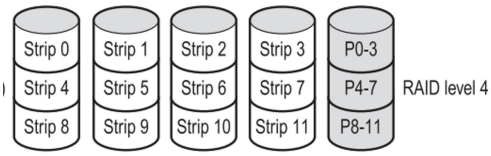
RAID level 3



Using byte level stripping and a separate disk for storing parity bits. A single disk failure does not disrupt the data. At the bit level, missing bits on the failed disk can be calculated from other disks and the parity disk.
Remarks:

- Tolerance to one disk failure
- All disks must have the same configuration and work in synchronization
- Consecutive read/write operations give good performance but random read/write operations result the same performance as with a single drive

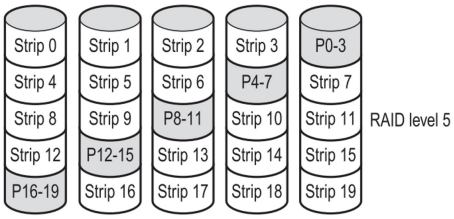
RAID level 4



Similar to RAID level 3 but using block level strippping.
Remarks:

- High read performance as data blocks can be read in parallel on different disks
- Does not require disks to work in synchronization
- Write performance is not really good: there must be at least 2 reads and 2 writes to be performed
    - When content of a block is changed, we need to read all other disks to re-calculate the checksum and write to the parity disk
    - Improvement: read the old content of the changed block + the corresponding block on the parity disk, re-calculate the checksum and write back to the parity disk (2 reads + 2 writes)
- Performance bottle neck on the parity disk

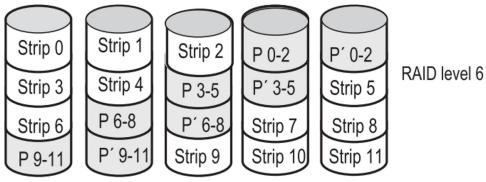
RAID level 5



Parity blocks are stored on all disk in a round-robin fashion.
Remarks:

- Can handle a single disk failture as in RAID level 4, a bit more complicated
- Avoid performance bottle neck problem to the parity disk

RAID level 6



Similar to RAID level 5 but having an additional parity block, stored on a different disk.

Assume we have n data blocks, 2 parity blocks P and P' on n+2 disks. Each data block consists of k bits. P and P' are calculated as follows:

- $\mathrm{P} = \mathrm{D}_0 \oplus \mathrm{D}_1 \oplus \cdots \oplus \mathrm{D}_{\mathrm{n}-1}$
- $\mathrm{P}' = \mathrm{D}_0 \oplus \mathrm{Shift}(\mathrm{D}_1) \ldots \oplus \mathrm{Shift}^{\mathrm{n}-1}(\mathrm{D}_{\mathrm{n}-1})$, where $Shift(\mathrm{d}_0\mathrm{d}_1 \ldots \mathrm{d}_{\mathrm{k}-1}) = \mathrm{d}_1 \ldots \mathrm{d}_{\mathrm{k}-1}\mathrm{d}_0$

RAID level 6 can recover data even in case of 2 disk failure. As the case n=3 illustrated in the figure.

If a data block (block 2) and parity block P are lost, the lost data block can be recovered as follows

- Calculate $\mathrm{P}' \oplus \mathrm{D}_0 \oplus \mathrm{Shift}(\mathrm{D}_1) = \mathrm{Shift}^2(\mathrm{D}_2)$
- Shift left 2 bits to get data block D_2

If 2 data blocks are lost (say, block 1 and 2), we do as follows

- Calculate $A = \mathrm{P} \oplus \mathrm{D}_0 = \mathrm{D}_1 \oplus \mathrm{D}_2$ (1)
- Calculate $\mathrm{B} = \mathrm{P}' \oplus \mathrm{D}_0 = \mathrm{Shift}(\mathrm{D}_1) \oplus \mathrm{Shift}^2(\mathrm{D}_2)$ (2)
- At the bit level, (1) and (2) form a system of 2k equation with 2k variables. Solving these equations we can recover D_1 và D_2

Remarks:

- Support up to 2 disk failure
- Not applicable when $n > k$ as the Shift will repeat and can not create 2k linear independent equations.

Error handling

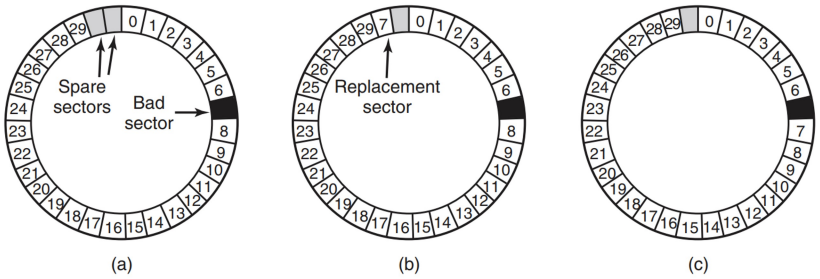
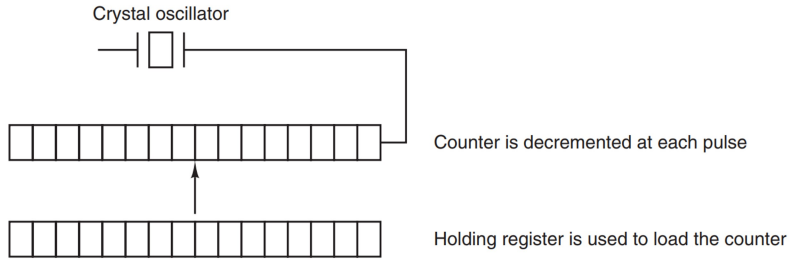


Figure 5-26. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

At the time of manufacture, if a bad sector can be replaced by a backup sector (b), or the disk can be formatted at the low level to skip the bad sector. OS and users do not realize this fix.
If bad sectors happen during operation, OS will gather them into a hiden file, therefore these blocks will never be accessible from the user.

Clocks

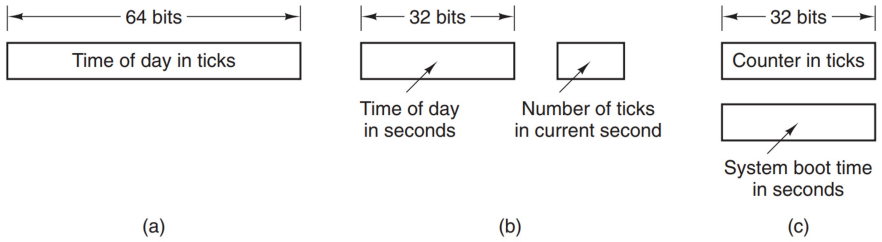
Programmable clocks



A programmable clock operates as follows

- Value of the holding register is copied to the counter register
- After each clock tick, the counter is decreased by 1
- When the counter is 0, the clock generates an interrupt, then repeat the above steps

Maintaining time



There are 3 ways of maintaining time in computers
(a) Use a 64-bit register to store the number of clock ticks since a predefined time. The cost is high (for 32-bit computers) as the register has to be increased many times in a second. If using only a 32-bit register, and the clock rate is 60Hz, then the register will be overflowed after 2 years.
(b) Use a 32-bit register to count the number of seconds since a predefined time, and a second register to count the number of clock ticks in a second. After a second, the former register is increased by one. Unix systems set the initial time at 0:00 am 1/Jan/1970 UTC, and Windows at 0:00 am 1/Jan/1980 UTC. Unix systems using signed integer register will overflow by 2038, and by 2106 in case of unsigned interger.
(c) Keep the number of clock ticks since the system boot time, using a 32-bit register. The boot time is read from a backup clock or typed in by the user.

CPU accounting

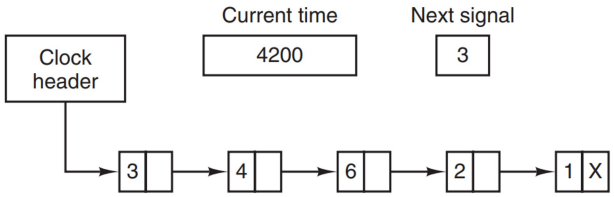
The system scheduler needs to carry out CPU accounting to guarantee that processes can only be executed during the allocated CPU time.

The most accurate accounting is to use a separate register for each process. When an interrupt occurs, these registers are saved and being restored after the interrupt is finished.

A practical approach is to keep a counter in the process information table. After each clock tick, the counter is increased by 1. As we do not save these counters before an interrupt is executed, the interrupt handling time is counted into the

counter of a process.

Timer



Timer can be set by using a count down queue till the execution time.

Each element in the queue is attached to a count down starting from the previous element. When the value of Next signal variable is decreased to 0, the 1st element of the queue is removed and the process attached to it is executed. Count down value of the next element is loaded into Next signal variable.