Bachelor of Computer Application Part-III

Paper : BCA-304

Operating Systems

Unit No. 2

Lesson Nos. :

2.1 : Memory Management
2.2 : Virtual Memory
2.3 : File Management
2.4 : Device Management - I
2.5 : Device Management - II

Department of Distance Education
Punjabi University, Patiala

MEMORY MANAGEMENT

Contents

2.1.0 Objectives
2.1.1 Introduction
2.1.1.1 Address Binding
2.1.1.2 Logical and Physical Address Space
2.1.1.3 Program Relocation
2.1.2 Storage Allocation and Management Techniques
2.1.2.1 Contiguous Storage Allocation
2.1.2.1.1 Fixed-partition contiguous storage allocation
2.1.2.1.2 Variable - partition contiguous storage allocation
2.1.2.2 Non-contiguous Storage Allocation
2.1.2.2.1 Paging
2.1.2.2.2 Segmentation
2.1.3 Keywords
2.1.4 Summary
2.1.5 Short Answer Type Questions
2.1.6 Long Answer Type Questions
2.1.7 Suggested Readings

2.1.0 Objectives

The intention of this lesson is to introduce the basic concepts of address binding, relocation and address space. After making you familiar with the terms like load time, compile time and run time, it moves on to discuss very important topic of contiguous and non-contiguous storage allocations. The storage talked about here is main memory. It covers fixed partition and variable partition as contiguous storage allocation techniques. It further discusses paging and segmentation as non-contiguous storage allocation techniques. Each technique has been introduced with the help of examples. Merits and demerits associated with each of the techniques have been mentioned in detail. This lesson serves as a base of the next lesson and therefore, it should be studied thoroughly before going through lesson 7.

2.1.1 Introduction

The memory management service of an operating system is one of the basic services, as it is needed:

- To ensure protection of different processes from each other (so that they do not interfere with each other's operations).
- To place the programs in memory (such that memory is utilized to its fullest and high degree of multiprogramming can be achieved.).
- Before knowing various memory allocation schemes, an idea about the various phases of program execution, address-binding, logical and physical address is must. The following subsections deal with the above said notion.

2.1.1.1 Address Binding

Just think about running a program and the steps involved in its execution. We know program is a set of instructions. Each instruction of the program is saved in RAM (Random Access Memory) when it is executed and thus each instruction has its location / address in the memory. Each instruction gets executed in a systematic manner known as instruction execution cycle. Instruction Execution Cycle has three steps:

- Fetch the instruction from memory
- Decode the instruction
- Get operands from the memory and then execute.

Thus, instruction gets executed and its output (if any) is stored (by the operating system) into memory.

The entire execution phase of a program is shown in figure 2.1.1. It divides the entire execution phase of a program into three parts:

1. Compilation phase
2. Loading phase
3. Run phase

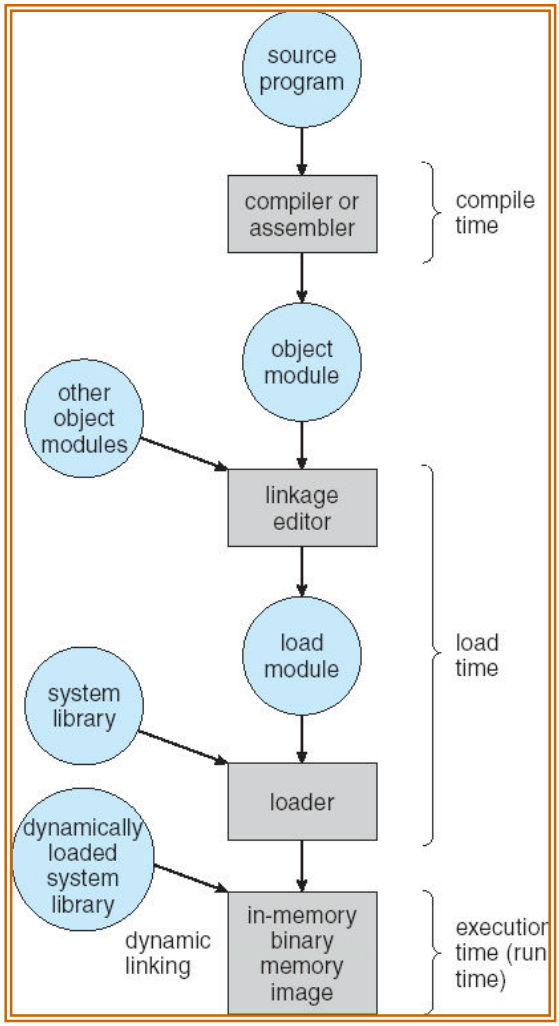


Figure 2.1.1 : Processing of a User Program

The problem that arises is how does the operating system know that where in the memory the results should be placed? Of course, the user whose program is getting executed must provide this. For example, if we have an instruction b = a+5; it implies that the output of a+5 should be stored in b. However, the user cannot tell where will 'b' be stored in the memory and where will 'a' get stored? But from user's point of view, he has tried to mention where the output should be stored. Such addresses used by a user in his/her programs are called symbolic addresses or logical addresses. Address binding is done to map these logical addresses to real physical addresses in memory. This binding of addresses of instructions and data to actual physical addresses can take place at following times during the course of execution of a process:

(i) Compile Time - Binding at compile time, generates absolute addresses. It requires that it must be known at the compile time itself, that where will a process reside in memory? It is most appropriate choice for small, simple systems (or dedicated hardware). The problem with this binding is that if there happens to be a change in the starting location of a process (in memory), then the entire process must be recompiled to generate the absolute addresses again. Figure 2.1.2 shows binding at compile time.

(ii) Load Time - In this case, the compiler generates relocatable addresses which are converted to absolute addresses at load time. Process cannot be moved around during execution. In this case, if there happens to be a change in the starting address of the process, it can be accomplished by statically relocating the code. This method is simple and is used in most of the cases. Figure 2.1.2 shows load time binding.

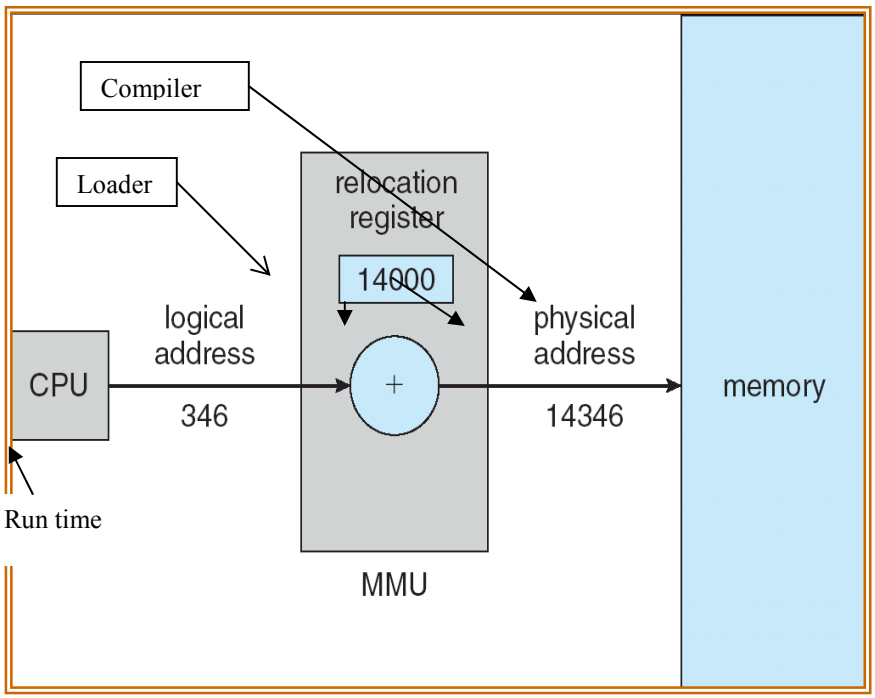


Figure 2.1.2: Address Binding Methods

(iii) Run Time - This method permits moving a process during run time from one memory segment to another. This method cannot be implemented until a good hardware support is available. This method can elongate the total run time of a process due to the movement of process at this time. Dynamic relocation is used to achieve it.

Figure 2.1.2 shows run time binding. Most commonly for global variables, the address binding occurs at load time whereas for local variables, the address binding takes place at run time.

2.1.1.2 Logical and Physical Address Space

A logical address is the address of an instruction or data as used by a program. A physical address is the effective memory address of an instruction or data i.e. it is the address obtained after binding of logical addresses has been done. The set of logical addresses used by a program constitutes the logical address space of the program. The set of physical addresses (in main memory) which the process actually occupies constitutes the physical address space of the process. These terms are used to formulate a virtual memory system. This system has been taken up in detail in the lesson 7 of this text. Figure 2.1.3 depicts the notion of logical and physical address space.

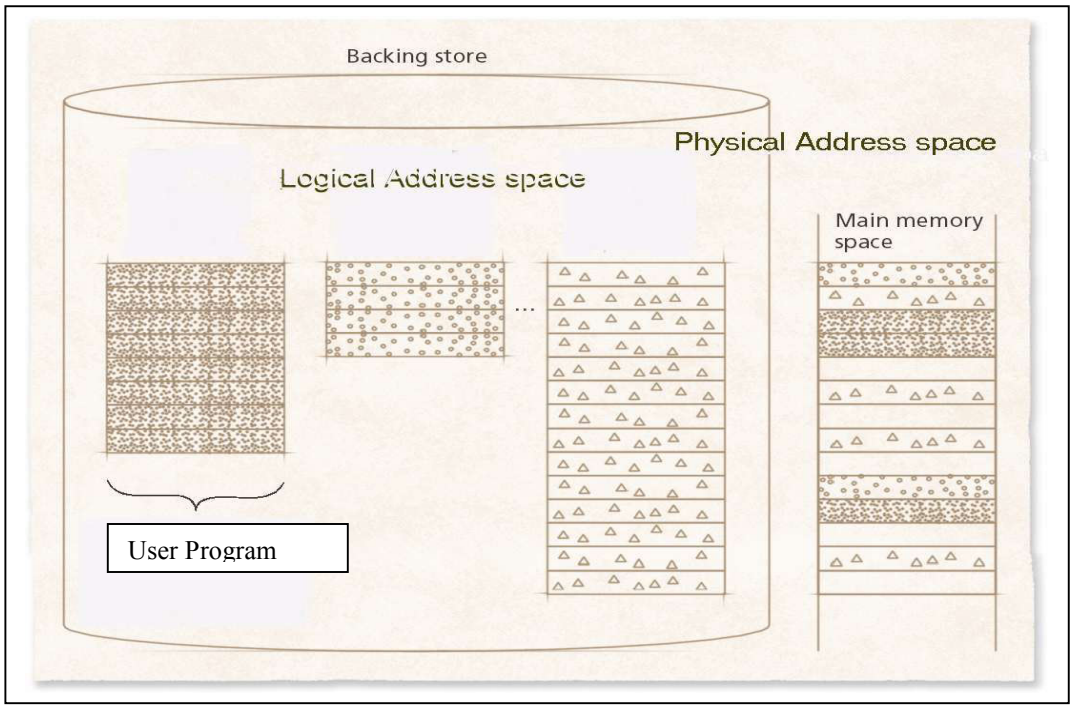


Figure 2.1.3 : Logical and Physical Address Space

2.1.1.3 Program Relocation

We know that a linker binds a program to a set of memory addresses starting on the load origin of the program. This makes the program address sensitive, i.e. the program can execute correctly if it occupies a memory area whose start address equals its load origin. To make a program execute correctly from any other memory area, it has to be relocated by changing all memory addresses used in it. Since an operating system cannot guarantee that programs would be allocated memory areas matching

their load origin, almost every program may need to be relocated. For this, some systems provide a special register in the CPU called a relocation register as an aid in program relocation. Every address used in the program is relocated as follows:

Effective address = Address used by the program + contents of relocation register

Memory management unit (a special hardware unit, which performs address binding, i.e. the mapping of logical addresses used by the program to physical addresses of the system's main memory) uses relocation. Consider the following example to have a clear picture of what relocation is all about?

Example: Consider a program P with a linked origin of 50000. Let the memory area allocated to it has the start address of 70000. The value to be loaded in the relocation register is determined as follows:

The value to be loaded in relocation register = Start address – Load origin
= 70000 – 50000 = 20000.

The figure shown below illustrates execution of an instruction with the address 61000. The instruction accesses a memory location with the link-time address of 65784. Since the relocation register contains 20000, the effective address is 65784 + 20000 = 85784. Hence, the actual memory reference is performed at the address 85784. Note that memory protection considerations should be applied to the address 85784. Thus, it should lie in the range of addresses defined by Base register and Limit register for memory protection purposes.

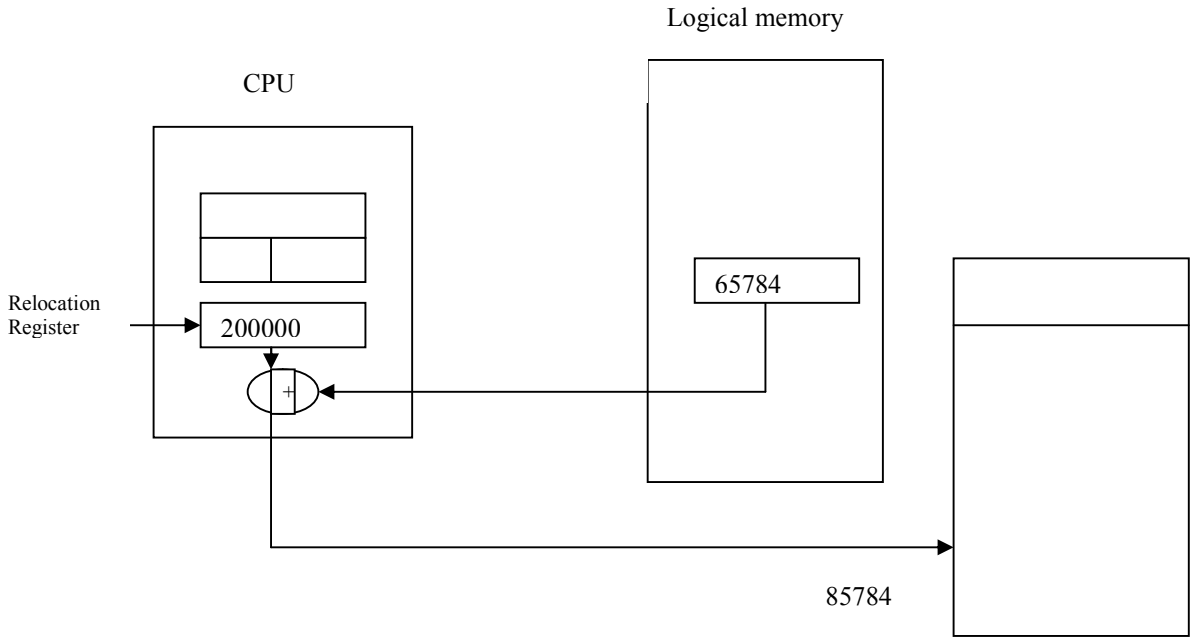


Figure 2.1.4: Relocation using relocation register

2.1.2 Storage Allocation and Management Techniques

Storage allocation can be of two types: (i) Contiguous storage allocation, and (ii) Non-contiguous storage allocation. Contiguous storage allocation implies that a program's data and instructions are assured to occupy a single contiguous memory area. Whereas, in non-contiguous storage allocation, a program's data and instructions may occupy non-contiguous areas of memory.

2.1.2.1 Contiguous Storage Allocation

Contiguous storage allocation method can be further subdivided into Fixed-partition storage allocation strategy and variable-partition storage allocation strategy.

2.1.2.1.1 Fixed-partition contiguous storage allocation

In continuation with the above said, consider that the primary memory is statically divided into N fixed regions or partitions, where regions R_i has N_i units of memory. Typically, N_i and N_j are different sizes, so processes with small address space use small partitions and processes with large address space use large partitions. This is known as fixed partition contiguous storage allocation. If a process requires n_k units of memory, it can be loaded into any region or partition R_i, where $N_i > n_k$. Upon allocation of the process, $N_i - n_k$ units of the primary memory are left unused, as, though it is allocated to the process but not mapped into its address space. This phenomenon where a part of the space allocated to a process is left unused is called Internal fragmentation. Figure 2.1.5 shows fixed partition contiguous storage allocation scheme. This scheme was used for early batch systems and it was good for those systems because in batch systems, space requirement of a process could be easily known before hand.

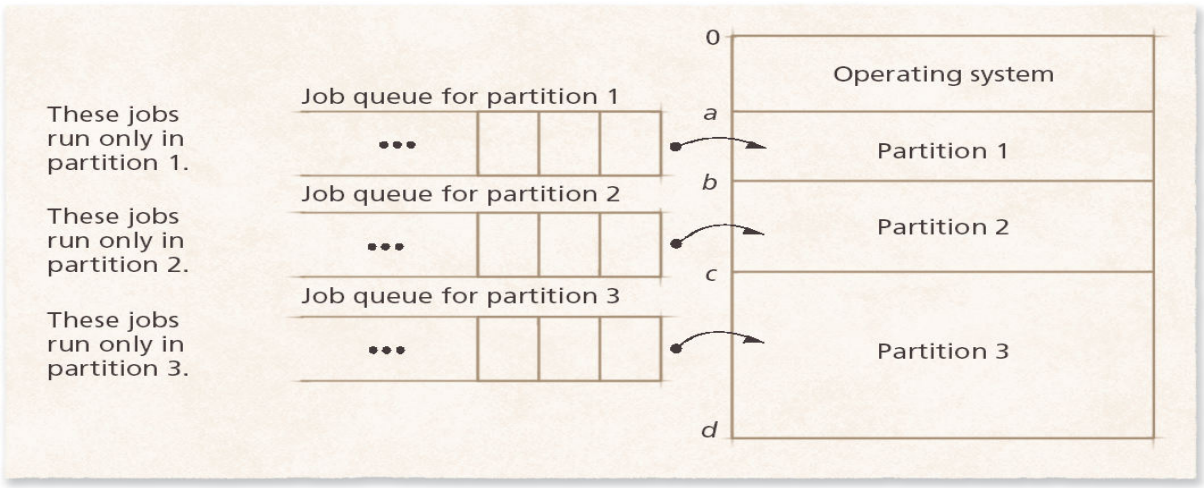


Figure 2.1.5 : Fixed partition contiguous storage allocation (Fixed-partition multiprogramming with absolute translation and loading).

2.1.2.1.2 Variable - partition contiguous storage allocation

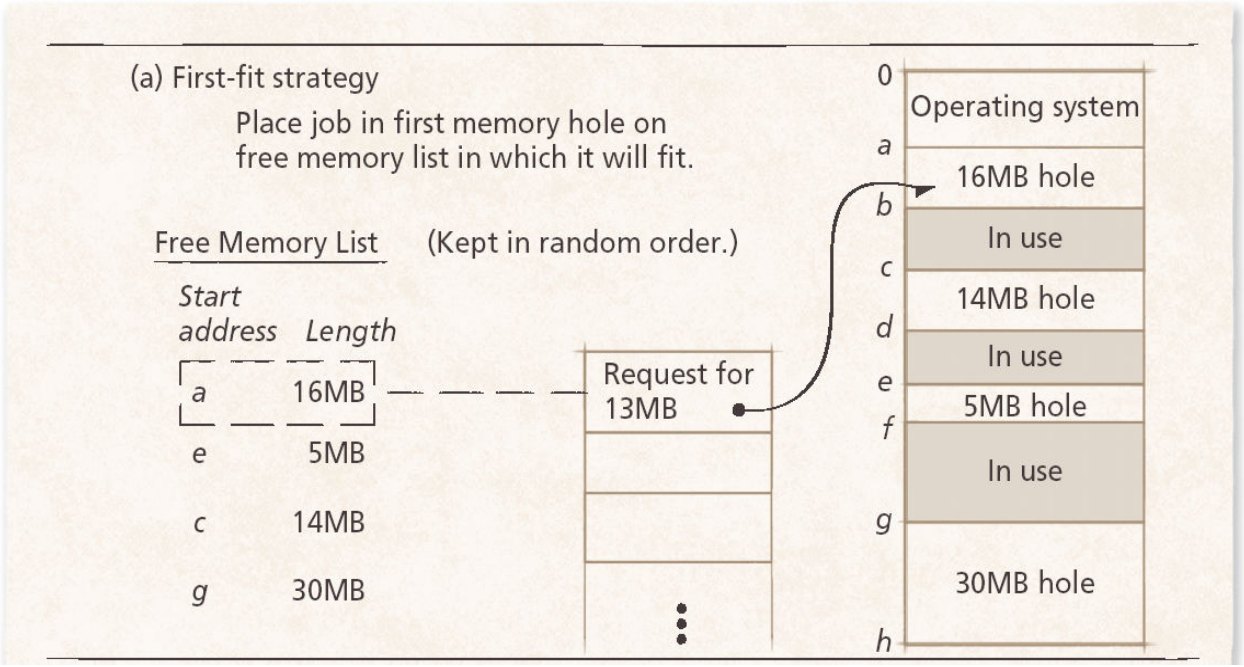
This notion is derived from parking of vehicles on the sides of streets where the one who manages to enter will get the space. Two vehicles can leave a space between them that cannot be used by any vehicle. This means that whenever a process needs memory, a search for the space needed by it, is done. If contiguous space is available to accommodate that process, then the process is loaded into memory. Some processes after finishing their execution will leave the memory and some new may enter into it. This phenomenon of entering and leaving the memory can cause the formation of unusable memory holes (like the unused space between two vehicles). This is known as External Fragmentation.

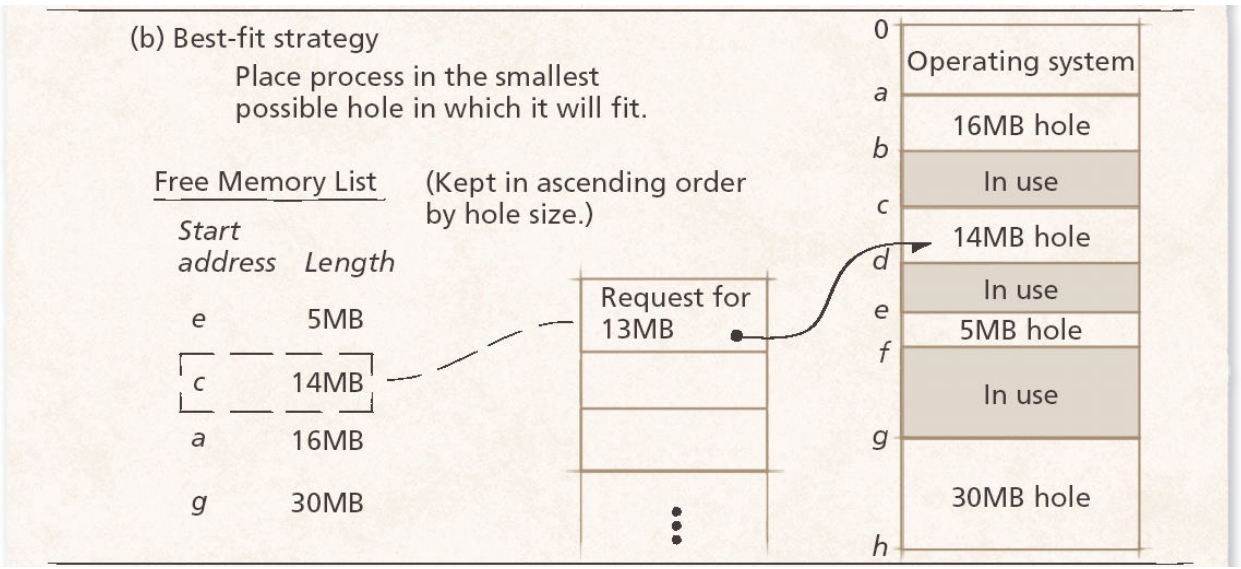
At one moment of time, memory can be considered to be a set of partitions (free space). There are three strategies that can be used to allocate memory to the existing set of processes. Suppose that a process p needs k units of memory then:

1. **Best - Fit -** chooses a partition that is smallest and whose size is greater than equal to k. It leaves small-sized unusable partitions or holes.
2. **Worst - fit -** chooses the largest partition and allocates it to process p. It can leave bigger unusable partitions.
3. **First - fit -** chooses the first partition whose size is greater than equal to k.

As far as the size of the unusable partitions created by these schemes are concerned, best-fit and first-fit schemes are better than worst - fit scheme. As far as time taken for searching a partition is concerned, first-fit scheme is better the other two. Consider a system shown in figure 2.1.6 and having free memory partitions of 5 MB, 14 MB, 16 MB, 30 MB at any instance of time. Let us analyze how would each of the first-fit, best - fit and worst-fit algorithms place a process of sizes 13 MB. It is clear from the figure 2.1.6 that best-fit is the best scheme.

The partitions left by these schemes contribute towards External fragmentation. A technique to deal with the external fragmentation problem is 'Compaction'.


(a) First-fit strategy
Place job in first memory hole on free memory list in which it will fit.
Free Memory List
(Kept in random order.)
Start address
Length
a 16MB
e 5MB
c 14MB
g 30MB
Request for 13MB
0
a
b
c
d
e
f
g
h
Operating system
16MB hole
In use
14MB hole
In use
5MB hole
In use
30MB hole


(b) Best-fit strategy
Place process in the smallest possible hole in which it will fit.
Free Memory List
(Kept in ascending order by hole size.)
Start address
Length
e 5MB
c 14MB
a 16MB
g 30MB
Request for 13MB
0
a
b
c
d
e
f
g
h
Operating system
16MB hole
In use
14MB hole
In use
5MB hole
In use
30MB hole

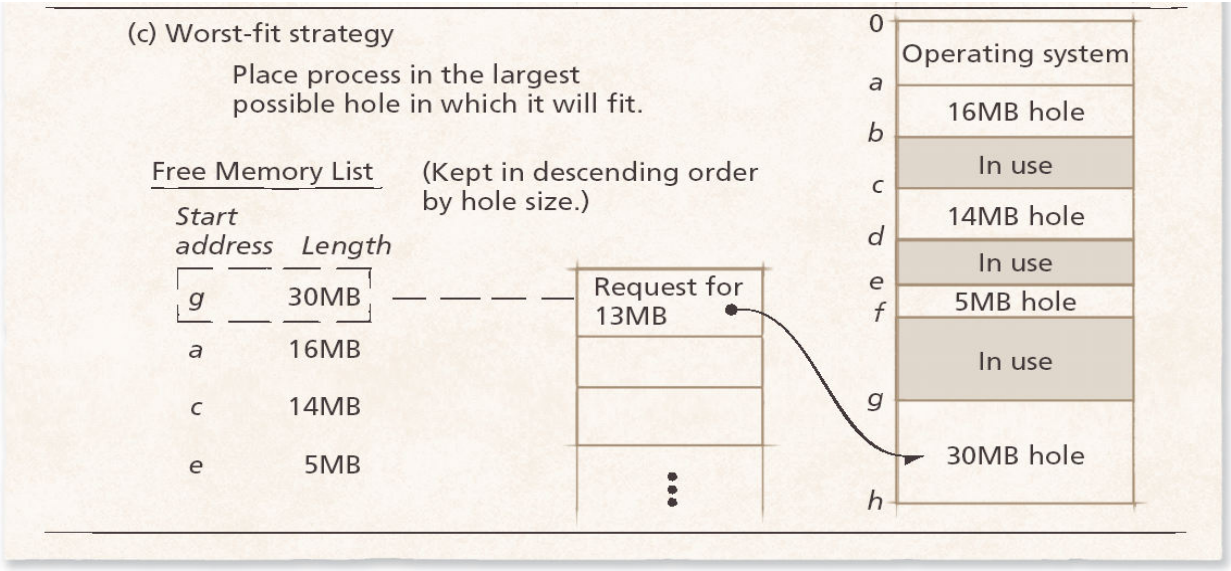
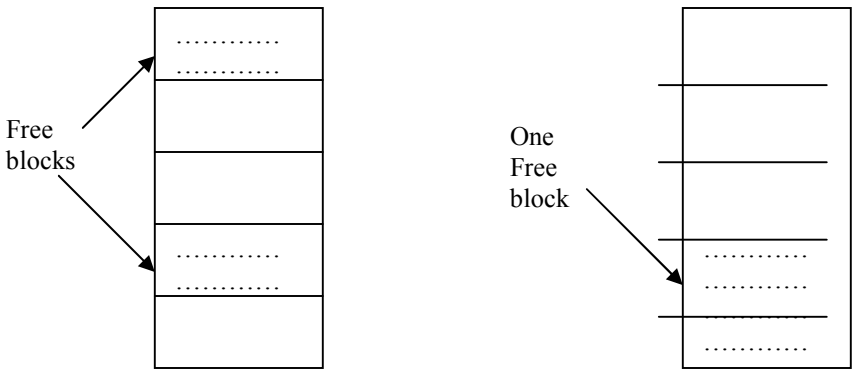


Figure 2.1.6 : Example of a)first fit, b)best fit and c)worst fit schemes

Figure 2.1.7 gives a clear picture of external fragmentation problem and its solution by compaction method. **Compaction** means to move the processes in the memory in such a way that scattered pieces of unused (free) memory can be placed together so that any other process demanding contiguous memory for it can use it.



Before compaction After compaction

Figure 2.1.7 : Memory Compaction

Compaction involves dynamic relocation of a program. This can be achieved quite simply in the computer systems using a relocation register (which contains the displacement factor of a process).

The problem with the compaction method is that it needs the relocation of addresses and that too dynamic relocation at the execution time of a process. Thus, Compaction is not possible if relocation is done statically (i.e. at load time).

The cost of compaction increases with the increase in the number of processes to be moved (relocated) in order to make use of the unused memory space. A strategy may have to be designed to decide direction (upwards or downwards) in which the processes should be moved to achieve the goal of compaction.

Figure 2.1.8 shows a table, which compares fixed-partition and variable - partition storage allocation methods.

Fixed Partition method	**Variable partition method**
OS decides partition size only once at system boot time	OS has to decide about partition size very often, almost every time when a new process is chosen by long term scheduler
Degree of Multiprogramming is fixed	Degree of multiprogramming will vary depending upon the size of processes being accommodated in the memory at one instance
Leads to internal fragmentation	No internal fragmentation is there. External fragmentation exists
IBM/360 DOS and OS/MFT operating systems used this approach	IBM OS/MFT used this approach also.

Figure 2.1.8 : A comparative study of fixed-partition and Variable-partition storage allocation methods

Before, we move on to Section 2.1.2.2 to discuss non-contiguous storage allocation, let us have a look on the advantages and disadvantages of contiguous storage allocation method.

Advantages

- It is a simple method because the entire process is allocated contiguous memory. This makes addressing (within a process) very simple.

Disadvantages

- There is no possibility of sharing data and instructions among the processes (which are loaded into entirely different portions of the memory).
- Degree of multiprogramming is dependent on the availability of a partition that can accommodate a process wholly. This reduces the number of ready processes in the memory.

2.1.2.2 Non-contiguous Storage Allocation

To resolve the problem of external fragmentation and to enhance the degree of multiprogramming to a greater extent, it was decided to sacrifice the simplicity of allocating contiguous memory to every process. It was decided to have a non-contiguous physical address space of a process so that a process could be allocated

memory wherever it was available. Figure 2.1.9 shows the non-contiguous memory allocation strategy. Paging and segmentation are such mechanisms with which a process can be efficiently allocated non-contiguous memory. These mechanisms have been discussed in the subsequent sections:

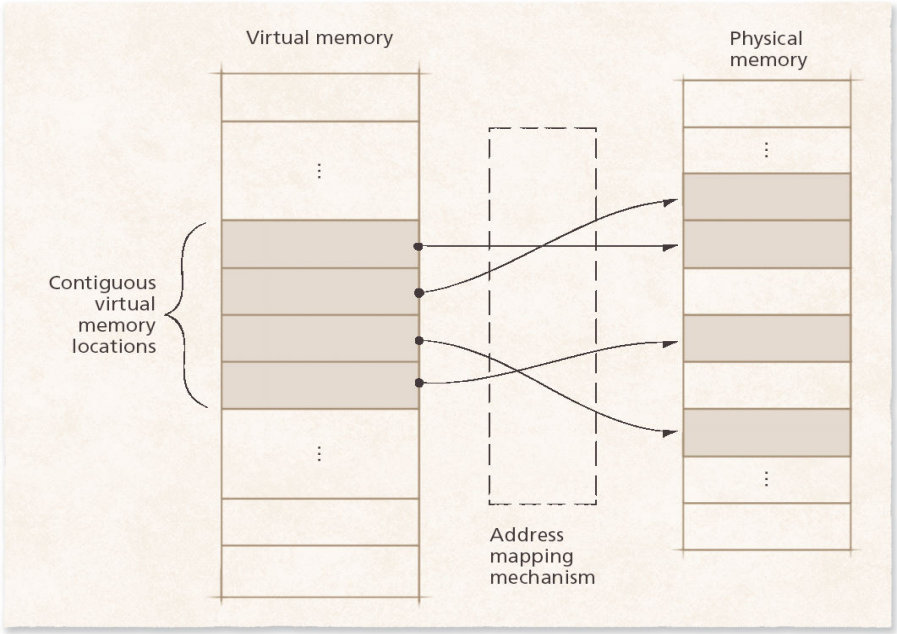


Figure 2.1.9 : Non-contiguous Storage Allocation

2.1.2.2.1 Paging

In this approach, physical memory is divided into fixed-size blocks called frames and the logical memory is divided into the fixed-sized blocks called pages. The size of a page is same as that of frame. The key idea of this method is to place the pages of a process into the available frames of memory, whenever, this process is to be executed. Figure 2.1.10 shows the key concept of paging.

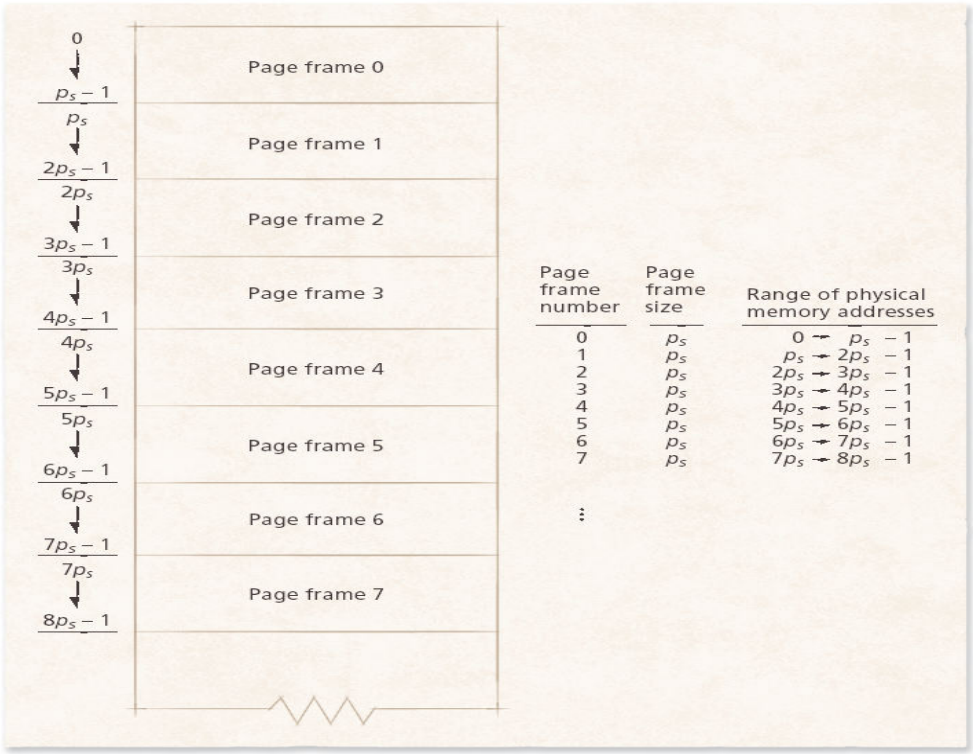


Figure 2.1.10 : Paging concept

Address Translation

The next very basic question comes to one's mind is that how will the logical address get converted to physical address in this case? The answer is mapping. The logical address is mapped to physical address using a page table contained by special hardware unit called memory management unit. This page table has one entry for each page of the process and indicates the status of each page i.e. whether it has been allocated space in the physical memory or not? If yes, which frame it is currently occupying?

Each table entry contains a "valid" bit which if set indicates that the associated page is in the process's logical address space otherwise it is not. It contains some other bits also. These other bits include:

Protection bit: This bit is used to flag the page as read-only or read/write or execute-only.

Modified bit: This bit, usually called the dirty bit, is set whenever the page is referenced by a write (store) operation i.e. when a page is modified.

Referenced bit: This bit is set whenever the page is referenced for any reasons, whether read, load or store.

Figure 2.1.11 shows the page table structure. The page size or the frame size is defined by the hardware and is always a power of 2. Can you guess why the page or frame size is always a power of 2? This is to make the division of a logical address into page number and page offset, easy. Logical address is divided into page number to specify which page is considered and page offset to specify the location of an instruction within a particular page. It will be clearer after you go through the following example: -

Let us consider that IBM/370 machine uses 24-bit logical addresses. The page size could be 2K or 4K bytes. When 2K byte pages are used, size of page = 2^6 bytes. This implies that 6 bits are adequate to address the bytes in the pages. The remaining 24 – 6 = 13 bits will be used as page number to specify the address of a particular page. Now, consider that the size of physical memory is 1 M bytes. This implies size of physical memory = 2^{20} bytes. A typical physical address will have 20-bits. As 6 bits are used to indicate the bytes in a page, the same number is used to address bytes in a frame (as pages and frames are of same size). The remaining 20 – 6 = 9 bits specify the address of an individual frame in the physical memory.

The above example can be generalized to state that if size of logical address space is $\mathbf{2^l}$ and size of each page is $\mathbf{2^p}$, then the high-order $\mathbf{l - p}$ bits of logical address specify the page number and the **p** lower-order bits designate the page offset i.e.

Logical address

Page No.	Page off set/displacement
l – p	p

The same holds true for the physical address space of 2^m size and having same frame size as the page size mentioned above.

Physical address

Frame No.	Frame off set/displacement
m – p	p

Figure 2.1.11 gives the complete picture of mapping a logical address to its corresponding physical address using page table.

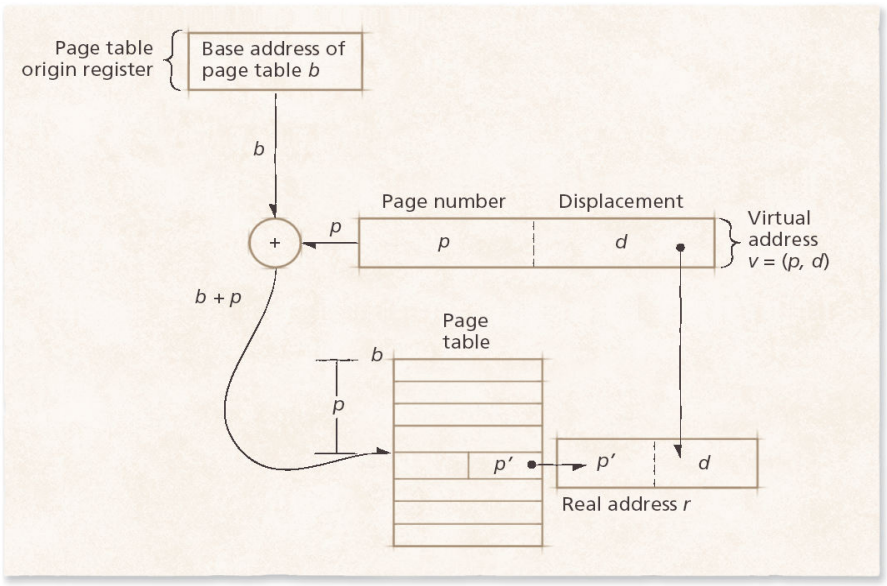


Figure 2.1.11 : Mapping a logical address(virtual address) to physical address(real address)

Example: A good paging example is being shown in figure 2.1.12.

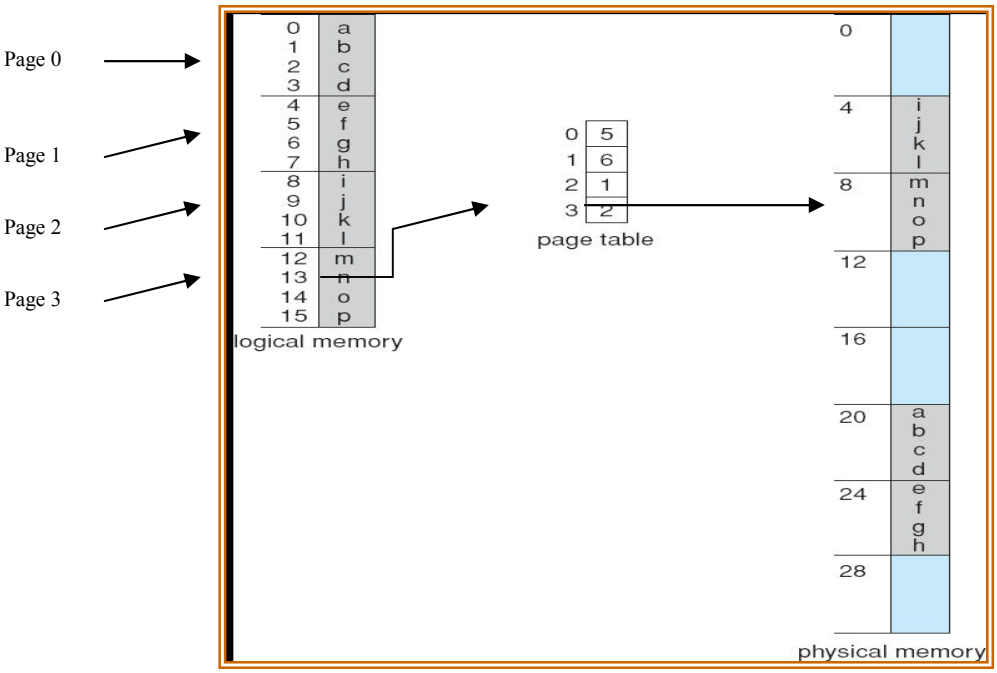


Figure 2.1.12 : Paging Example

It works as:

Translate logical address 13 = 0601 in binary

Offset = 01 in binary or 1 in decimal

Page no. = 06 in binary or 3 in decimal
Physical address = pageframe*pagesize + offset i.e. 2*4+1 = 9 (which is the real address of page containing character "n" in the main memory)

Problems with paging

1. Fast lookup in page table

It uses a page table per process for translating logical to physical address space. A basic issue associated with the use of page table is, that the look up in the page table should be very fast, as it is done on every memory reference (at least once per instruction being executed and often two or more times per instruction). The lookup is always done by special purpose hardware. Even, with special hardware, if the page table is stored in memory, the table look up makes each memory reference generated by the CPU cause two references to memory.

To make the look up time less, caching is used. It has been said that caching is the only technique in computer science used to improve performance. In this case, the specific cache device is called a translation look aside buffer (TLB). The TLB contains a set of entries, each of which contains a page number, the corresponding frame number, and the protection bits. There is special hardware to search the TLB for an entry matching a given page number. If the TLB contains a matching entry, it is found very quickly and nothing more needs to be done. Otherwise we have a TLB miss and have to fall back on one of the other techniques to find the translation. However, we can take that translation which we found with great difficulty and put it into the TLB so that we find it much more quickly the next time. The TLB has a limited size, so to add a new entry, we usually have to throw out an old entry. The usual technique is to throw out the entry that has not been used the longest. The reason this approach works so well is that most programs spend most of their time accessing a small set of pages over and over again. For example, a program often spends a lot of time in an "inner loop" in one procedure. Even if that procedure, the procedures it calls, and so on are spread over 40K bytes, 10 TLB entries will be sufficient to describe all these pages, and there will no TLB misses provided the TLB has at least 10 entries. This phenomenon is called locality of reference. In practice, the TLB hit rate for instruction references is extremely high. The hit rate for data reference is also good, but can vary widely for different programs. Figure 2.1.13 shows the use of TLB in paging. If the TLB performs well enough, it almost does not matter how TLB misses are resolved.

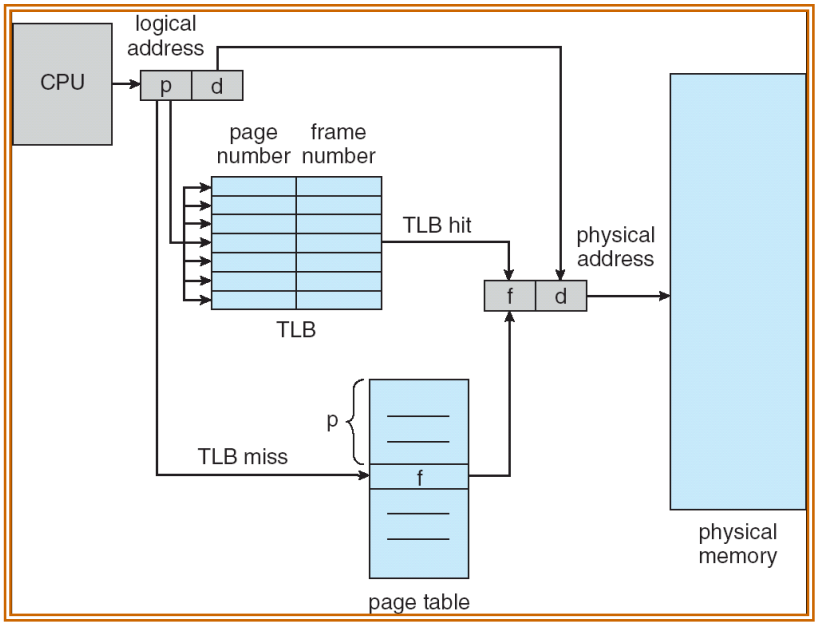


Figure 2.1.13: Address translation using TLB

Two processes may map the same page number to different frames. Since the TLB hardware searches for an entry by page number, there would be an ambiguity if entries corresponding to two processes were in the TLB at the same time. There are two ways to solve this problem. Some systems simply flush the TLB (set a bit in all entries marking them as unused) whenever they switch processes. This is very expensive, not because of the cost of flushing the TLB, but because of all the TLB misses that will happen when the new process starts running. An alternative approach is to add a process identifier to each entry. The hardware then searches on for the concatenation of the page number and the process id of the current process.

2. What to do when a page table gets large?

Suppose the page size is 4K bytes and a logical address is 32 bits long. Then the logical address would be divided into a 20-bit page number and a 12-bit offset (because 2^{12}= 4096 = 4K), so the page table would have to have 2^{20} = 1,048,576 entries. If each entry is 4 bytes long, that would use up 4 megabytes of memory. And each process has its own page table. Newer machines being introduced now generate 64-bit addresses. Such a machine would need a page table with 4,503,599,627,370,496 entries!

There are several different page table organizations used in actual computers.

* One approach is to put the page table entries in special registers. This was the approach used by the PDP-6 minicomputer introduced in the 1970's. Putting page table entries in registers helps make the Memory Management Unit run faster (the registers were much faster than main memory), but this approach has a downside as

well. The registers are expensive, so it works for very small page-table size. Also, each time the operating system wants to switch processes, it has to reload the registers with the page-table entries of the new process.

* A second approach is to put the page table in main memory. The (physical) address of the page table is held in a register. The page number of the logical address is added to this register to find the page table entry in physical memory. This approach has the advantage that switching among the processes is easy (all you have to do is change the contents of one register) but it means that every memory reference generated by the CPU requires two trips to memory. It also can use too much memory, as we saw above.
* A third approach is to use what is called an inverted page table. An ordinary page table has an entry for each page, containing the address of the corresponding frame (if any). An inverted page table has an entry for each frame, containing the corresponding page number. To resolve a logical address, the table is searched to find an entry that contains the page number. The good news is that an inverted page table only uses a fixed fraction of memory. For example, if a page is 4K bytes and a page-table entry is 4 bytes, there will be exactly 4 bytes of page table for each 4096 bytes of physical memory. In other words, less than 0.1% of memory will be used for page tables. This is by far the slowest of the methods, since it requires a time consuming search of the page table for each reference. A special hardware to search the table in parallel can be implemented. Figure 2.1.14 shows an inverted page table. 'pid', here, refers to the process identifier and "i" is the frame number.

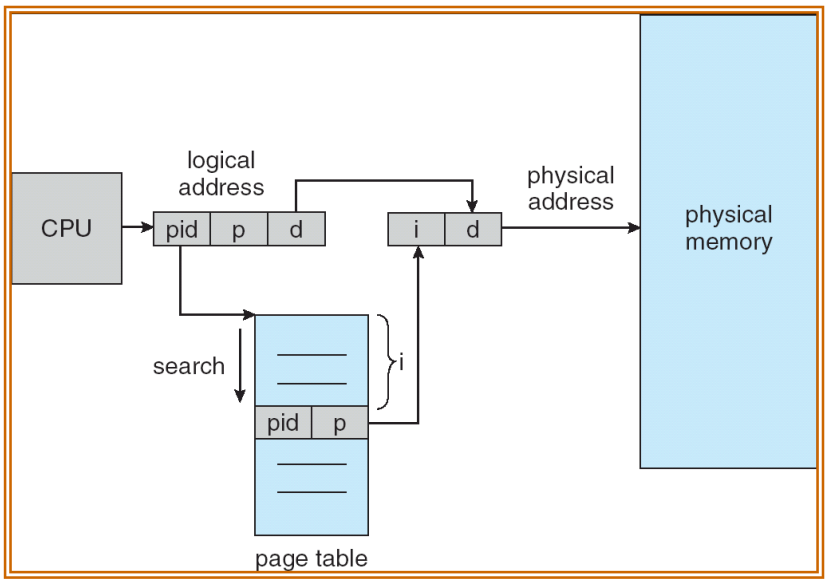


Figure 2.1.14: Inverted Page Table

4. Internal Fragmentation - Paging also suffers from this problem. This may occur when the process size is not an exact multiple of page size. Let us say that we have a page size of 4K and the size of logical address space of a process is 133K then it can be divided into 133 K / 4K = 34 pages. When accommodated in frames, the last frame will not be completely full. Figure 2.1.15 explains this. The extent of internal fragmentation depends on the size of frames or pages. If we keep very large page size, it causes more internal fragmentation. If we have very small page size, there will be a lot of pages and thus more overhead will be involved e.g. more disk I/O operations will be required to move a page from disk to memory. Similarly, number of page table entries will increase leading to an increase in the page table size.

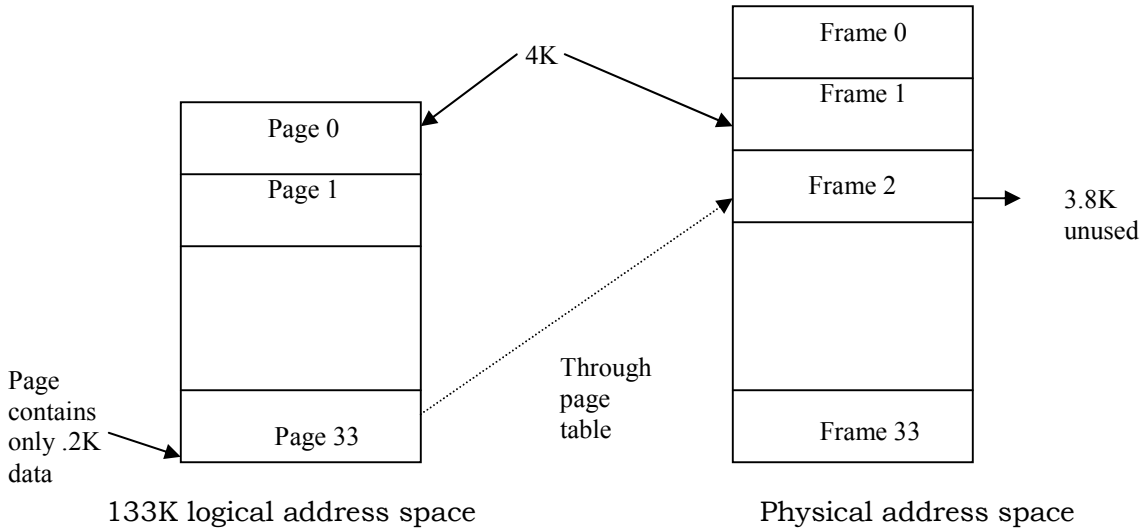


Figure 2.1.15: Simplified diagram to explain internal fragmentation in paging

Advantages of paging

1. As seen there is never a possibility of external fragmentation, it really achieves a higher degree of multiprogramming as compared to contiguous storage allocation strategies.
2. Sharing of common code among the processes is now possible. Single copy of some commonly used program such as compilers, editors, database systems can be kept in the physical memory. Many processes can share it, by making the page table of each of the processes such that it maps to the frames having shared data or programs in the physical memory. This means that many logical addresses will be mapped to one physical address.

2.1.2.2.2 Segmentation

As mentioned in the beginning of this section, segmentation is another technique for the noncontiguous storage allocation. It is different from paging as it

supports users' view of his program. We know that users (programmers) think of their program as a collection of routines, libraries and piece of data. The programmer interprets the logical address space of a program according to his own viewpoint rather than as an array of bytes.

For a programmer it might be more relevant to divide the logical address space of his program into variable sized segments (with respect to his view of main program, subroutines, data, etc.) than to divide it into fixed size pages. Such variable sized segments, which are a collection of logically related information, are the basis of segmentation technique. In this technique, logical address space is a collection of such segments. The logical address comprises of segment number and segment offset. Normally, the compilation of a user program leads to the construction of segments. Thus a compiler may create separate segments for the global variables, the procedure call stack (that stores parameters and return addresses), the code of each procedure and the local variables of each procedure. The loader would assign numbers to these segments.

Address Translation: Let us now see, how does the mapping take place between logical and physical address? It is done by memory management unit, which uses segment tables. A segment table contains an entry for each segment of a process. Entries in segment tables also contain:

- Pointer to start of a segment in physical memory (base)
- Size of segment (bound)
- Indication of whether or not a segment is currently in memory
- Protection Information indicating whether the process has permission to read, write or execute the segment

Figure 2.1.16 shows the use of segment table. The following steps are involved in address translation:

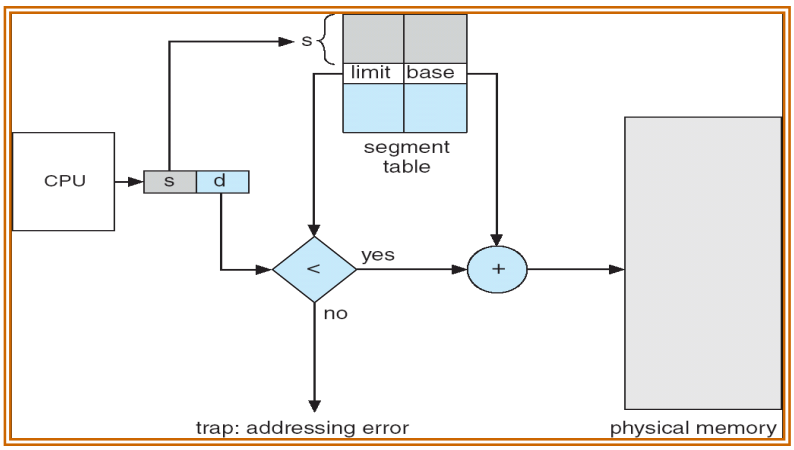


Figure 2.1.16: Address translation using segment table

1. Segment number is extracted from logical address by dividing the address into segment number and offset. If there are 2^n segments, then n higher order bits in the logical address space specify the segment number and the remaining bits give the offset.
2. If segment number is greater than the maximum number of segments, a trap is generated to the operating system. It indicates that an attempt has been made to access an illegal segment.
3. Otherwise, segment table entry is accessed:
   (i) If segment offset is greater than the segment size (i.e. value of bound), a trap indicating "offset out of range" will be generated.
   (ii) If process does not have permission to access that segment, a trap indicating "protection violation" will be generated.
4. Otherwise, physical address (segment base + offset) is returned.

Figure 2.1.17 shows an example of segmentation.

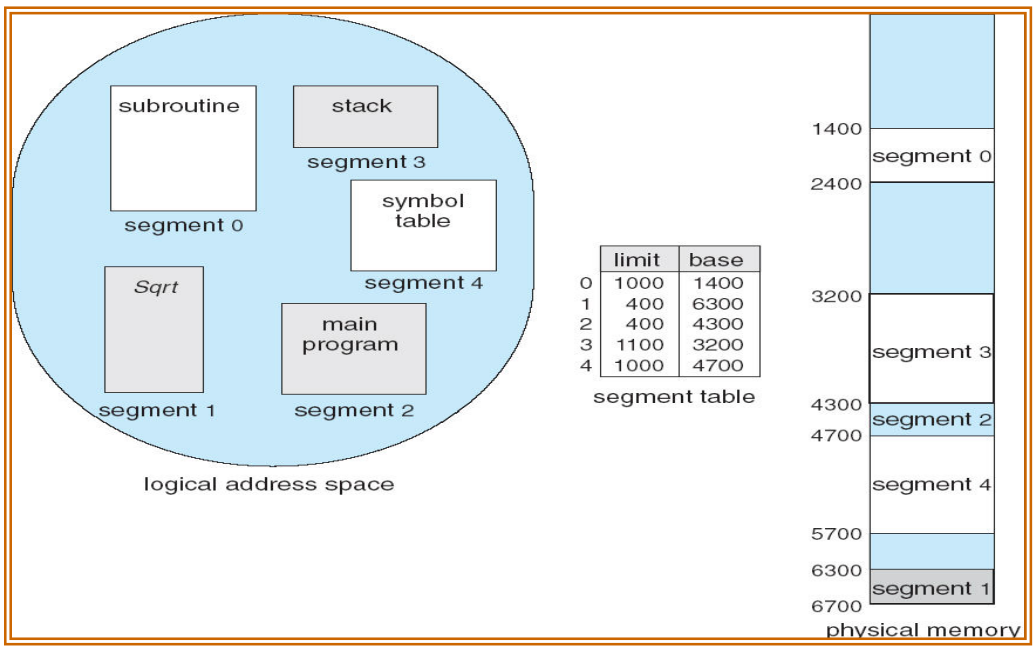


Figure 2.1.17: Segmentation Example

Problem with Segmentation

1. As with paging, this mapping requires two memory references per logical address, which slows down the computer system by a factor of two. Caching is the method used to solve this problem.
2. In cases, where there are more number of segments, segment table size will grow. Thus, it cannot be accommodated in any of the registers and has to be kept in memory. Keeping it in memory needs a good hardware support from the underlying architecture.

3. Segmentation is prone to external fragmentation. This may occur when all the blocks in the memory are too small to accommodate a segment. Compaction, if possible, may sometimes be applied to solve the problem of external fragmentation.

Advantages of segmentation

1. Protection bits are associated with segments, which check that attempt to write a read-only segment should fail. They can also be used to check that attempt to use an execute-only segment, as a data segment should be prohibited.
2. As with paging, sharing of code or data is possible even with the segmentation technique. Here, entries in two different segment tables can be made to point to a common physical location.

2.1.3 Keywords

Logical addresses- addresses used by a user in his/her programs are called symbolic addresses or logical addresses.

Physical address- actual physical addresses in main memory.

Address Binding-it is done to map these logical addresses to real physical addresses in memory.

Page- the logical memory is divided into the fixed-sized blocks called pages. The size of a page is same as that of frame.

Frame- physical memory is divided into fixed-size blocks called frames.

Segment- is a variable sized block which is a collection of logically related information.

Program Relocation-. To make a program execute independently of its location, it has to be relocated by changing all memory addresses used in it. Relocation register is used for the purpose.

Memory compaction- Compaction involves dynamic relocation of a program. This can be achieved quite simply in the computer systems using a relocation register.

Page table- It has one entry for each page of the process and indicates the status of each page i.e. whether it has been allocated space in the physical memory or not. If yes, it specifies which frame it is currently occupying.

Segment table- A segment table contains an entry for each segment of a process. Entries in segment tables also contain pointer to start of a segment in physical memory (base), size of segment (bound), indication of whether or not a segment is currently in memory, protection information indicating whether the process has permission to read, write or execute the segment.

Memory management unit- a special hardware unit, which performs address binding.

Translation look aside buffer- the specific cache device used to make paging faster.

Inverted page table- An inverted page table has an entry for each frame, containing the corresponding page number.

Internal fragmentation- The unused space of a memory block.

External fragmentation- phenomenon of entering and leaving the memory can cause the formation of unusable memory holes. This is called external fragmentation.

2.1.4 Summary

The memory manager is required to administer the executable memory, allocating it to different processes as needed. Address binding is a fundamental barrier to data movement. This is because the traditional program translation environment causes points in a program's address space to be bound to physical memory locations before the program begins to execute. Static binding inhibits the manager's ability to move an address space around in the memory. Dynamic binding can be done to achieve this. It uses hardware relocation register.

A program can be allocated contiguous or non-contiguous space in memory. Contiguous storage allocation method can be further subdivided into Fixed-partition storage allocation strategy and variable-partition storage allocation strategy. First-fit, best-fit or worst-fit strategies can be used for variable-partition storage allocation strategy. Paging and segmentation are the mechanisms with which a process can be efficiently allocated non-contiguous memory. Paging uses fixed size blocks in logical address space called pages and also same sized blocks in physical address space called frames. Paging suffers from internal fragmentation and segmentation suffers from external segmentation. Techniques like translation look aside buffer can be used to make look up of page table faster. Paging is the base of demand paging described in next lesson.

2.1.5 Short Answer Type Questions

Q.1 Explain address binding at compile time, load time and run time.
Q.2 How is relocation used in run-time address binding of a program?
Q.3 Differentiate between internal and external fragmentation.
Q.4 Compare fixed-partition allocation strategy with variable-partition scheme for contiguous memory allocation.
Q.5 Compare best -fit, worst- fit and first- fit allocation algorithms.
Q.6 What is compaction? How is it implemented?
Q.7 What are the problems associated with paging?
Q.8 Why is TLB (Translation look aside buffer) used in a paging scheme?
Q.9 List the advantages of paging.
Q.10 Do you think there is any necessity of fixed-partition and variable - partition schemes, in a uni-programming environment?
Q.11 How is protection ensured in paging mechanism?

2.1.6 Long Answer Type Questions

Q.1. Compare paging and segmentation techniques. List their relative advantages and disadvantages.

Q.2. Consider a logical address space of eight pages of 1024 words each, mapped on to a physical memory of 32 frames. Find out the number of bits in the logical address and the number of bits in the physical address.

Q.3. Describe using a diagram how a logical address consisting of 18 bits could be converted to a physical address where each page was 1K byte. How many pages would there be?

Q.4. Consider the following segment table:

Segment No.	Base	Bound	r w x
0	219	600	r w x
1	2300	14	r w x
2	90	100	r w –
3	1327	580	r w –
4	1952	96	r w –

What are the physical addresses for the following logical addresses?

0, 430
0, 10
2, 500
3, 400
4, 112

Where 0, 430 implies 430th instruction/data in 0th segment.

Q.5. Compare fixed-partition and variable - partition schemes of memory allocation.

2.1.7 Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts", Khanna Publishing Co., 2002.
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India.
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.

Web Resources :

www.personal.kent.edu/~rmuhamma/opsystem/os.html
www.wiley.com/college/silberschatz6e/0471417432/slides/slides.html

MANAGING VIRTUAL MEMORY

Contents

2.2.0 Objectives
2.2.1 Introduction
2.2.2 Swapping
2.2.3 Demand Paging and its Performance
2.2.3.1 Pure Demand Paging
2.2.4 Page Replacement
2.2.4.1 Page replacement algorithms
2.2.4.1.1 The optimal page replacement algorithm
2.2.4.1.2 The first-in, first-out (FIFO) page replacement algorithm
2.2.4.1.3 The second chance page replacement algorithm
2.2.4.1.4 The least recently used (LRU) page replacement algorithm
2.2.4.1.5 Simulating LRU in Software
2.2.4.1.6 The Working Set Page Replacement Algorithm
2.2.4.1.7 Summary of page replacement algorithms
2.2.5 Page Replacement Policies
2.2.6 Demand Segmentation
2.2.7 Keywords
2.2.8 Summary
2.2.9 Short Answer Type Questions
2.2.10 Long Answer Type Questions
2.2.11 Suggested Readings

2.2.0 Objectives

This lesson focuses on management of virtual memory. Virtual memory is an important concept with which every user program gets a feeling of being allocated the needed main memory in one go whereas actually, it is not like that. In reality, the operating system and user programs taken together may need total memory space, which exceeds the actually available one. Virtually, each user program seems to be accommodated in the main memory. Demand paging is used to implement the concept of virtual memory and therefore, has been described in detail. Page replacement algorithms (like FIFO, LRU, Optimal etc.) have been discussed with a mention of their advantages and disadvantages. Demand segmentation has been described in brief. Thus, the idea is to discuss the virtual memory management techniques.

2.2.1 Introduction

Today, operating systems and applications occupy space in excess of 200 MB! With the relatively substantial cost of main memory (system RAM) and most systems being limited to 64 MB or 78 MB of RAM how does it all fit in a comparatively small main memory? Well, actually, it does not. A virtual effect is however possible by virtue of something known as "Virtual Memory"

Virtual memory is a technique of executing program instructions that may not fit entirely in system memory. This is done by calling instructions as and when the application requires it. Virtual memory is implemented by using secondary storage to augment the main memory. Data is transferred from secondary to main storage as and when necessary and the data modified is written back to the secondary storage according to a predetermined algorithm.

Virtual memory, thus enhances the CPU utilization and throughput by executing as many programs as possible, almost simultaneously. Thus, it appears to the user that operating system and application programs are easily getting accommodated in small main memory of the system. This is actually a virtual effect because, now instead of holding the entire program, main memory holds only a portion of the program, which is currently being executed. Before moving on, here are a couple of things you should be familiar with:

Address translation done by Memory Management Unit in paging and segmentation techniques. Both these techniques have been discussed in detail in previous lesson.

Figure 2.2.1 shows the actual implementation of virtual memory. This diagram illustrates that (1) address translation of logical address to physical address should take place. (2) This should be followed by a check, if the page is present in the main memory, (3) the mapper enters this page in the available frame where this page is accommodated in the main memory, (4) if the mapper detects that requested page is not present in main memory. (5) A page fault occurs and the page must be read into a frame in main memory, from secondary storage, (hard disk). This is known as demand paging and is one way of implementing virtual memory. The logical address here is also called virtual address, sometimes. Another way of implementing virtual memory is demand segmentation. The following sections discuss these methods in detail.

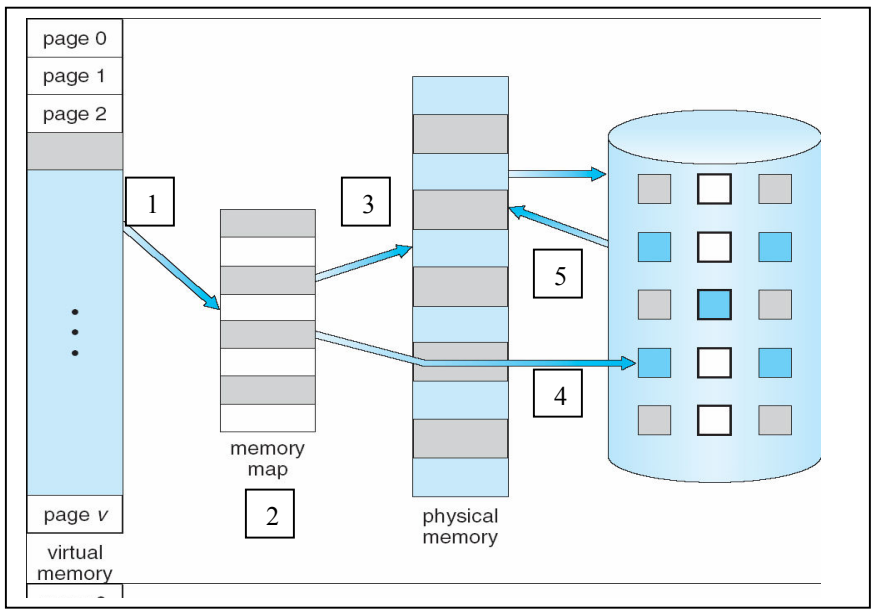


Figure 2.2.1: Virtual Memory implementation

2.2.2 Swapping

Swapping is the technique of temporarily removing inactive programs from the main memory of a computer system. An inactive program is one that is neither executing nor performing an I/O operation. Figure 2.2.2 illustrates swapping as used in a practical situation. There is usually a set of active programs in memory. Out of these only one executes on the CPU while others perform I/O. Whenever, an active program becomes inactive, the operating system begins to swap it (when system is falling short of main memory space). This involves writing the program's instructions and data areas onto the secondary storage, typically a disk. Once a program is completely swapped out, the memory area allocated to it becomes free. For example, this may happen in case of round robin scheduling. A process is swapped out when its time quantum finishes and later, it is brought into the memory for continued execution.

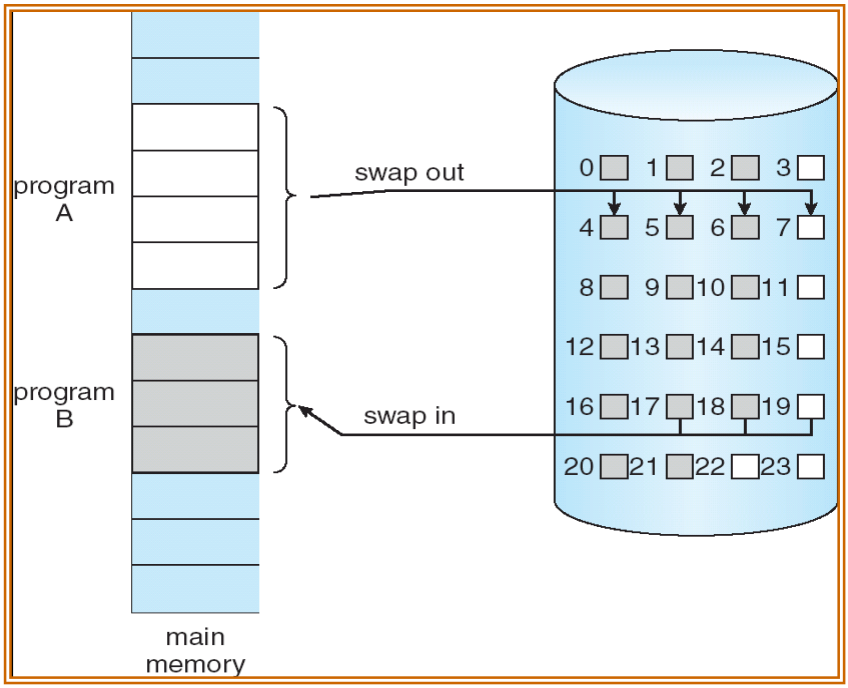


Figure 2.2.2: Swapping in practice

This may also happen when it is desired to place a higher-priority process in the memory. A lower priority process may be swapped out so that higher-priority process may be loaded and executed.

Swapping increases the operating system's overhead due to the need to perform swap-in and swap-out operations. DOS does not perform swapping but most other operating systems, including OS/2, Windows and UNIX perform swapping. Now, let us see how this concept becomes a key to virtual memory? As mentioned in the previous section, the idea of virtual memory concept is to accommodate as many application processes (in main memory) as possible so as to increase CPU utilization and throughput. This is possible, by keeping only those portions of these programs in memory, which are currently needed for their execution. The remaining portions can be picked up as and when needed (from the secondary storage like disk). This means that portions of these programs have to be swapped in to the memory (by the operating system) when they are needed for execution. This may also require putting the currently unused portions back on to the disk. This involves swapping out these portions to the disk.

2.2.3 Demand Paging and its Performance

It is almost similar to the concept of paging, which has been described in the previous chapter. In virtual memory system, demand paging is a type of swapping

in which pages of programs are not copied from disk to main memory until they are needed for execution.

In demand paging, virtual address (logical address) is accessed by CPU, the corresponding page number is looked up in the page table and if it shows that currently this page is not in main memory, then this page must be brought into the main-memory. This requires some hardware support to distinguish between pages that are in memory and those that are on disk.

A valid-invalid bit in the page table may be used to achieve this. If this bit for a page is invalid then either it is on disk or it is not within the logical address space of the process i.e. it do not belong to the process in whose context it is being accessed.

A **page fault** occurs when an invalid page is addressed. Page fault must be followed by swapping-in the page (demanded just now by the CPU) from disk to main-memory or a trap should be generated to the operating system if the page being demanded is not within the logical address space of the process. To determine whether the reference to the requested page is within the logical address space or not, an internal table may be consulted. Figure 2.2.3 shows how the page table looks when all the pages are not in main memory and figure 2.2.4 shows the steps involved in servicing a page fault.

These are:

1. Operating system looks at another table to decide:
   Invalid reference $\Rightarrow$ abort
   Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

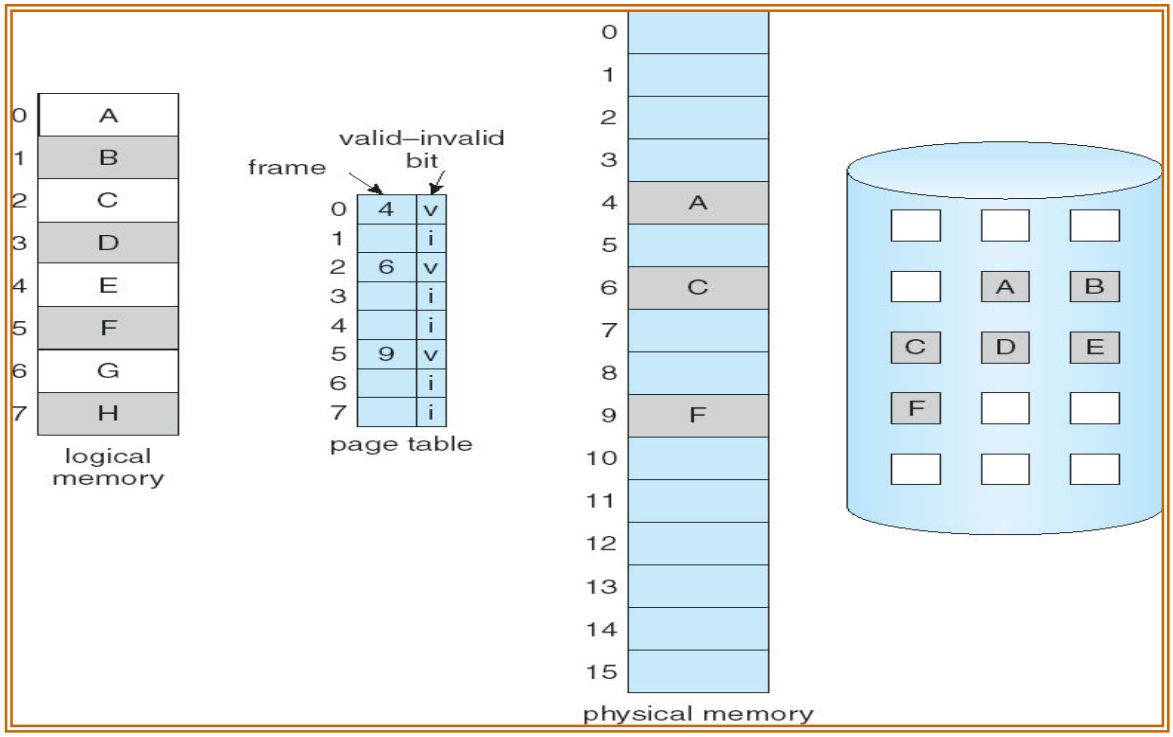


Figure 2.2.3: Page table when all the pages are not in main memory

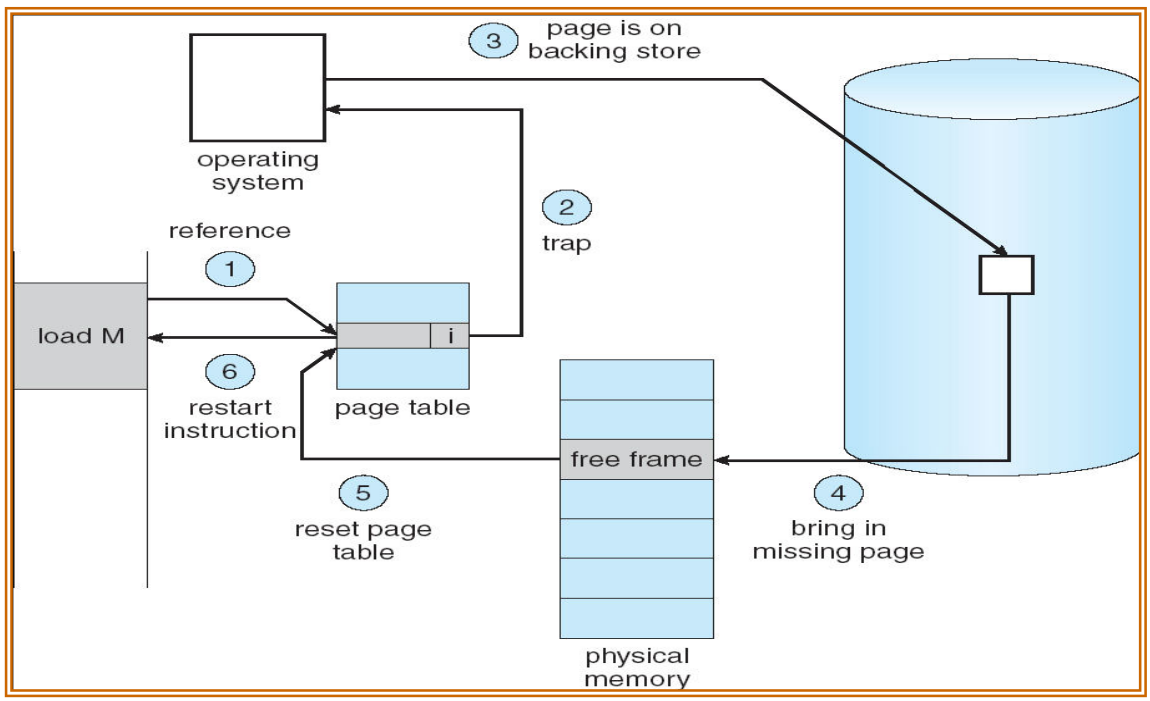


Figure 2.2.4: Servicing a page fault

2.2.3.1 Pure Demand Paging

Pure demand paging is the form of demand paging in which not even a single page is loaded into memory, initially. Thus, the very first instruction causes a page fault in this case. This kind of demand paging may significantly decrease the performance of a computer system by generally increasing the effective access time of memory. If ma is the memory access time and p is the probability of occurrence of a page fault. Then the effective access time is:

effective access time = (1 – p) x ma + p x page fault time

Example : Suppose the average page fault service time is 20 milli-seconds and memory access time is 200 nanoseconds. Suppose, we wish to have less than 10% degradation in the memory access when a page fault occurs, what must the page fault rate be less than?

This means that we want effective access time should be less than

$\left(\frac{10}{100} \times 200 + 200\right) \times 10^{-9}$ seconds i.e. 220×10^{-9} seconds.

Effective access time is obtained as:

$(1 - p) \times 200 \times 10^{-9} + p \times 20 \times 10^{-3}$

$= 2 \times 10^{-7} - 2 \times 10^{-7} p + 2 \times 10^{-2} p$

Now, $2 \times 10^{-7} + 1.99 \times 10^{-2} p$ should be less than 220×10^{-9}

$\Rightarrow$ Thus, $2 \times 10^{-7} + 1.99 \times 10^{-2} p < 220 \times 10^{-9}$

$\Rightarrow$ $1.99 \times 10^{-2} p < 2.2 \times 10^{-7} - 2 \times 10^{-7}$

$\Rightarrow$ $1.99 \times 10^{-2} p < .2 \times 10^{-7}$

$\Rightarrow$ $p < \frac{.2 \times 10^{-7}}{1.99 \times 10^{-2}}$

$\Rightarrow$ $p < \frac{.2 \times 10^{-5}}{1.99}$

$\Rightarrow$ $p < .10050 \times 10^{-5}$

$\Rightarrow$ $p < .00001$ approximately

This means that probability of occurring of a page fault should be less than .00001 so as to achieve a degradation of less than 10%. This implies that page fault rate should be kept as low as possible so that effective memory access time should not increase.

Page fault service time talked about in this example means the time it takes to serve the page fault. It usually includes the time taken to

1. Determine that the page being demanded is legal (reference is valid).
2. Read the page from disk.
3. Swap it in free space in memory if available. Otherwise make space available by swapping out some page of the same process or other process.
4. Restart the process, which has demanded this page.

2.2.4 Page Replacement

Once the main memory fills up, a page must be swapped out to make room for any pages to be swapped in. This is known as page replacement. The question arises which page should be swapped out of the memory? This is decided by various algorithms known as page replacement algorithms.

Figure 2.2.5 shows the need for page replacement and Figure 2.2.6 shows a generalized method to do it. The sequence of actions is labeled using numbers 1, 2, 3, etc.

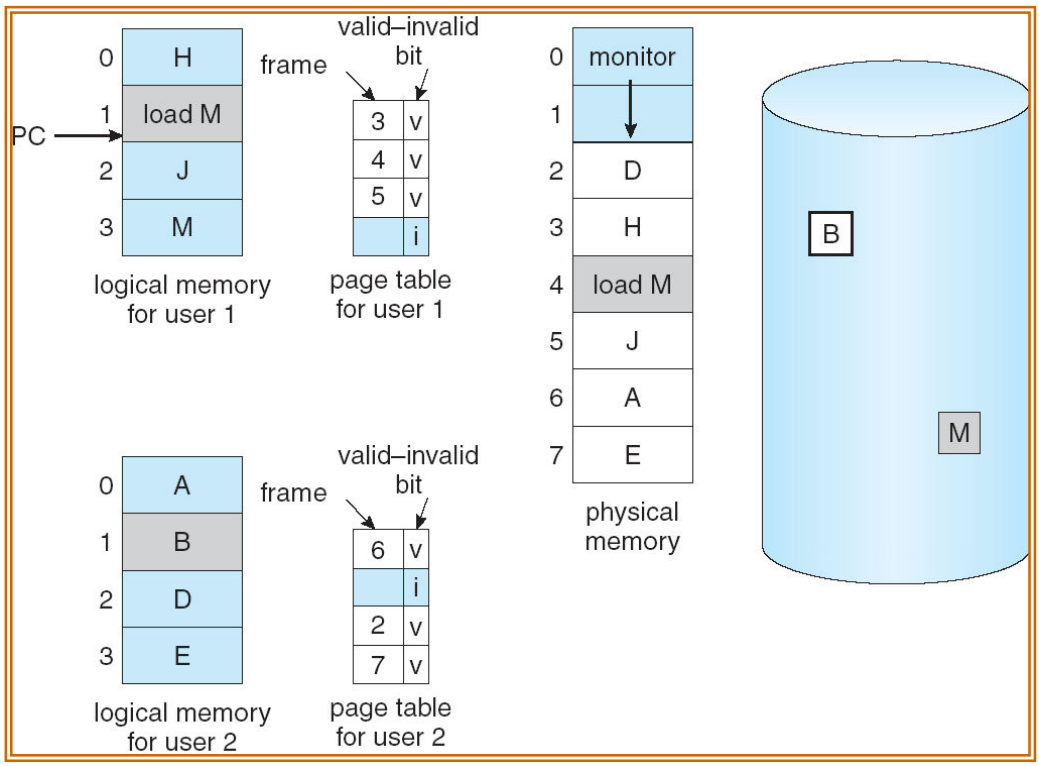


Figure 2.2.5: Need of Page replacement

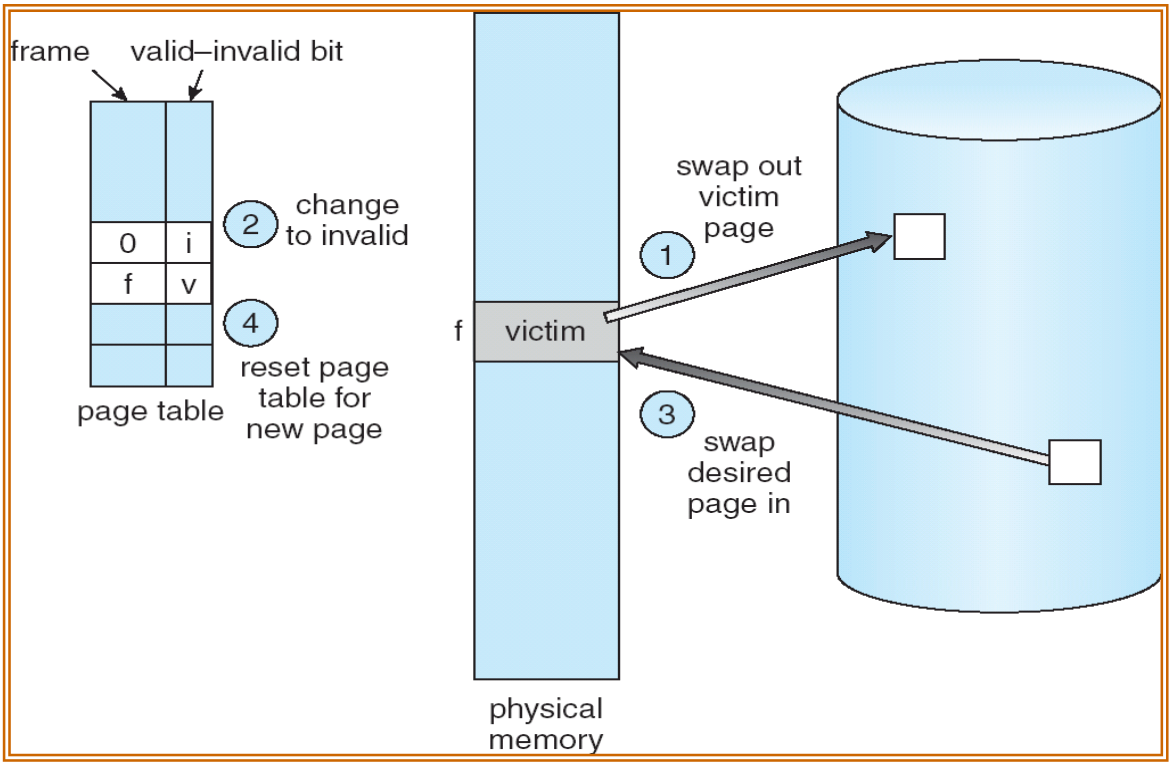


Figure 2.2.6: Page Replacement Process

Before, taking up each of the page replacement algorithm in detail, we analyze the general working of page replacement. This involves the movement of two pages. One is copied from disk to main memory (to serve page fault) and other from main memory to disk (to create free space). This is a time consuming operation.

To overcome this problem, a bit known as modify / dirty bit can be associated with a page. This bit is set whenever any word is written into this page and thus indicates that the page has been modified. Now, when a page is selected to be replaced, its modify bit is checked.

If it is set, it implies that this page was modified (when it was in main memory) and therefore, must be written back to the disk, otherwise this page must not be written back to the disk, because no changes have been made to it after it was read from the disk into the memory. The new demanded page (the one, which is swapped-in,) can now replace it as it is.

Let us now move on to discuss the page replacement algorithm.

2.2.4.1 Page replacement algorithms

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page to be removed has been modified while in memory, it must be rewritten to the

disk to bring the disk copy upto date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already upto date, so no rewrite is needed. The page to be read in just overwrites the page being replaced.

While it would be possible to pick a random page to replace at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in the near future, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental.

Following sections describe few important page replacement algorithms in detail.

2.2.4.1.1 The optimal page replacement algorithm

The best possible page replacement algorithm is easy to describe but impossible to implement. It goes like this. At the moment when a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction. Other pages may not be referenced until 10, 100, or perhaps 1000 instructions have been executed before that page is first referenced.

The optimal page algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, the former page should be removed, as it will not be used in the near future. Computers, like people, try to put off unpleasant events for as long as they can!

The only problem with this algorithm is that it is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next (we saw a similar situation earlier with the shortest job first scheduling algorithm - how can the system tell which job is shortest?) Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the second run by using the page reference information during the first run.

Although this method is useful for evaluating page replacement algorithms, it is of no use in practical systems.

2.2.4.1.2 The first-in, first-out (FIFO) page replacement algorithm

Another low-overhead paging algorithm is the FIFO (first-in, first-out) algorithm. To illustrate how this works, consider a supermarket that has enough shelves to display exactly k different products. One day, some company introduces a new product. It is an immediate success, so our finite supermarket has to get rid of one old product in order to stock the recently successful product.

One possibility is to find the product that the supermarket has been stocking the longest (i.e., something it began selling 70 years ago) and get rid of it on the grounds that no one is interested any more. In effect, the supermarket maintains a

linked list of all the products it currently sells in the order they were introduced. The new one goes on the back of the list; the one at the front of the list is dropped.

As a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the page at the head of the list, being the oldest one and the page at the tail, being the most recent arrival. On a page fault, the page at the head is removed and the new page is added to the tail of the list. When applied to stores, FIFO might be useful. When applied to computers, problem arises because the page although being oldest, might be the heavily used page and may be needed immediately in the memory. This will thus lead to quick page faults. FIFO in its pure form is rarely used because this algorithm suffers from Belady's anomaly. This is named after Belady who in 1970 found a problem with FIFO algorithm. The problem is when more frames are allotted to a process (to accommodate more pages of the process), it generates more page faults whereas it should be the other way round. Figure 2.2.7 shows this anomaly in a very descriptive way.

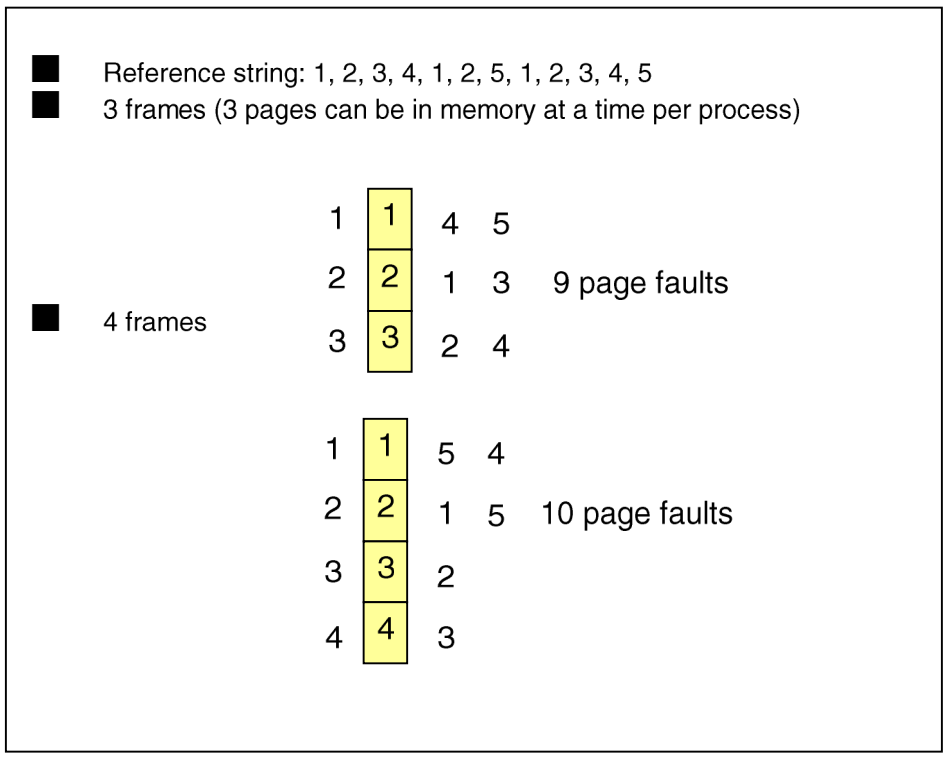


Figure 2.2.7: Belady's Anomaly

The newest page is on the top and the oldest migrates to the bottom like a queue. All of the page faults are indicated here in this figure. Notice the expected improvement in the number of page faults when three frames are used. But, most unexpectedly, see that adding fourth frame result in an increase in the number of

page faults whereas it should have been decreased. This is called Belady's anomaly and is the biggest disadvantage of FIFO algorithm. Figure 2.2.8 shows one more example of this algorithm with physical memory having three frames.

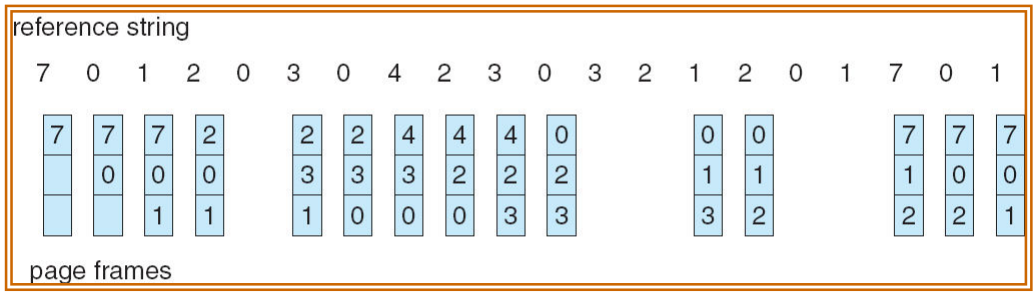


Figure 2.2.8: Another FIFO page replacement example

2.2.4.1.3 The second chance page replacement algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the Referenced bit (R) of the oldest page.

If it is 0, the page is both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues. This implies that this algorithm searches for an old page that has not been referenced. If a page is oldest but has been referenced, it will not be considered for replacement.

The operation of this algorithm, called second chance, is shown in Figure 2.2.9. In figure 2.2.9(a) we see pages kept on a circular queue and sorted by the time they arrived in memory.

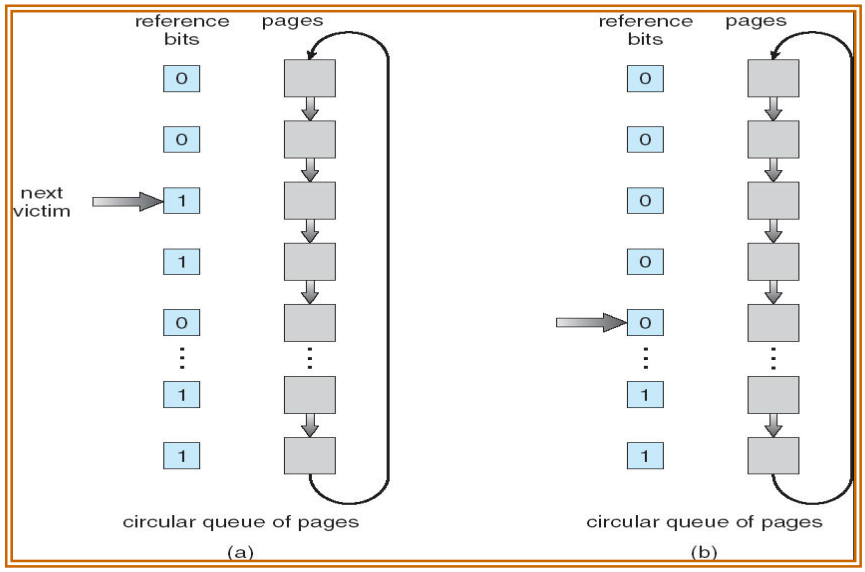


Figure 2.2.9: Operation of second chance page replacement algorithm

What second chance is doing is looking for an old page that has not been referenced in the previous clock interval. If all the pages have been referenced (i.e. have their R bit set) second chance degenerates into pure FIFO. Specifically, imagine that all the pages in figure 2.2.9(a) have their R bits set. One by one, the operating system moves the pointer to the pages at the end of the list, clearing the R bit each time it moves to the end of the list. Eventually, it comes back to the first page, which now has its R bit cleared. At this point the very first page is replaced. Thus, the algorithm always terminates. Figure 2.2.9(b) shows that after setting the reference bits to 0, the next victim is selected which already has its reference bit 0 and is not given any second chance.

2.2.4.1.4 The least recently used (LRU) page replacement algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called LRU (Least Recently Used) algorithm.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built).

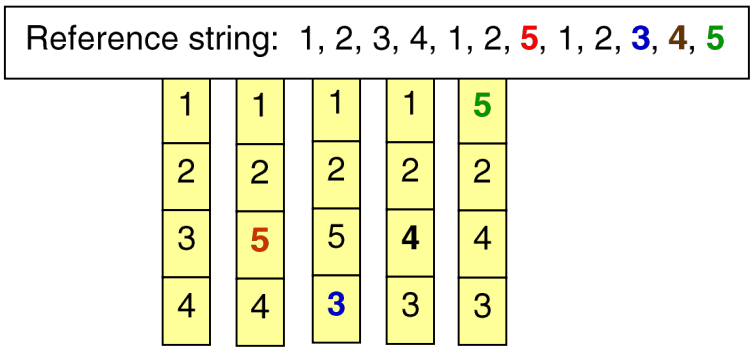


Figure 2.2.10: Example of LRU algorithm

2.2.4.1.5 Simulating LRU in Software

Although both of the previous LRU algorithms are realizable in principle, but only few machines have this hardware, so they are of little use to the operating system designer who is making a system for a machine that does not have this hardware. Instead, a solution that can be implemented in software is needed. One

possibility is called the NFU (Not Frequently Used) algorithm. It requires a software counter associated with each page, initially zero. In effect, the counters are an attempt to keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement.

The main problem with NFU is that it never forgets anything. For example, in a multi-pass compiler, pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass 1 pages. Consequently, the operating system will remove useful pages (of pass 2 and so on) instead of pages (of pass 1) no longer in use.

Another possibility of simulating LRU in software is to use stack of page numbers. Whenever a page is referenced, its number is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page number and the bottom is the LRU page number. This method may be time consuming when LRU page number is to be removed from the stack. However, this algorithm does not suffer from Belady's anomaly.

2.2.4.1.6 The Working Set Page Replacement Algorithm

In the purest form of demand paging, processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruction. Other page faults for global variables and the stack usually follow quickly. After a while, the process has most of the pages it needs and settles down to run with relatively few page faults.

Most processes exhibit a **locality of reference**, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multi-pass compiler, for example, references only a fraction of all the pages.

The set of pages that a process is currently using is called its working set. If the entire working set is in memory, the process will run without causing many page faults until it moves into another execution phase (e.g., the next pass of the compiler). If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly since executing an instruction takes a few nanoseconds and reading in a page from the disk typically takes 10 milliseconds. At a rate of one or two instructions per 10 milliseconds, it will take ages to finish. A process causing page faults every few instructions is said to be **thrashing**. Thus, thrashing refers to high paging activity of a process. This implies that a process is effectively spending more time in paging than executing therefore, frequent occurrence of thrashing can slow down the system and thus, it is not desirable.

In a multiprogramming system, processes are frequently moved to disk (i.e., all their pages are removed from memory) to let other processes have a turn at the CPU. The question arises is what to do when a process is brought back in again. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having 20, 100 or even 1000 page faults every time a process is loaded, is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault (page fault service time).

Therefore, many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the **working set model**. It is designed to greatly reduce the page fault rate. Loading the pages before letting processes run is also called pre-paging. Note that the working set changes over time.

It has been long known that most programs do not reference their address space uniformly, but that the references tend to cluster on a small number of pages. A memory reference may fetch an instruction, it may fetch data, or it may store data. At any instant of time, t, there exists a set consisting of all the pages used by the k most recent memory references. This set ws (k, t), is the working set. Because the k = 1 most recent references must have used all the pages used by the k > 1 most recent references and possibly others, ws (k, t), is a monotonically non-decreasing function of k. The limit of ws (k, t) as k becomes large is finite because a program cannot reference more pages than its address space contains, and few programs will use every single page.

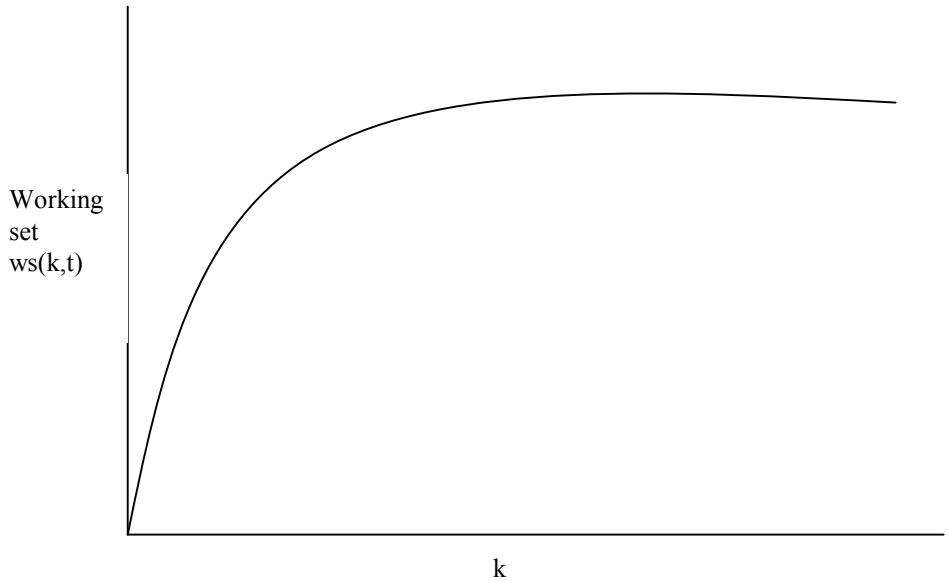


Figure 2.2.11: The working set is the set of pages used by the k most recent memory references. The function w (k, t) is the size of the working set at time t.

Figure 2.2.11 depicts the size of working set as a function of k. The fact that most programs randomly access a small number of pages, but that this set changes slowly in time explains the initial rapid rise of the curve and then the slow rise for large k. For example, a program that is executing a loop occupying two pages using data on four pages, may reference all six pages every 1000 instructions, but the most recent reference to some other page may be a million instructions earlier, during the initialization phase.

Thus, there exists a wide range of k values for which the working set is unchanged. Because the working set varies slowly with time, it is possible to make a reasonable guess as to which pages will be needed when the program is restarted on the basis of its working set when it was last stopped. Pre-paging consists of loading these pages before the process is allowed to run again.

To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. Having this information also immediately leads to a possible page replacement algorithm: When a page fault occurs, find a page not in the working set and replace it. To implement such an algorithm, we need a precise way of determining which pages are in the working set and which are not at any given moment of time.

As we mentioned above, the working set is the set of pages used in the k most recent memory references (some authors use the k most recent page references, but the choice is arbitrary). To implement any working set algorithm, some value of k must be chosen in advance. Once some value has been selected, after every memory reference, the set of pages used by the previous k memory references is uniquely determined.

Now let us look at a page replacement algorithm based on the working set. The basic idea is to find a page that is not in the working set and replace it. In figure 2.2.12 we see a portion of a page table for some machine. Because only pages that are in memory are considered as candidates for replacement, pages that are absent from memory are ignored by this algorithm. Each entry contains (at least) two items of information: the approximate time the page was last used and the R (Referenced) bit. The empty white rectangle symbolizes the other fields not needed for this algorithm, such as the frame number, the protection bits and the M (Modified) bit.

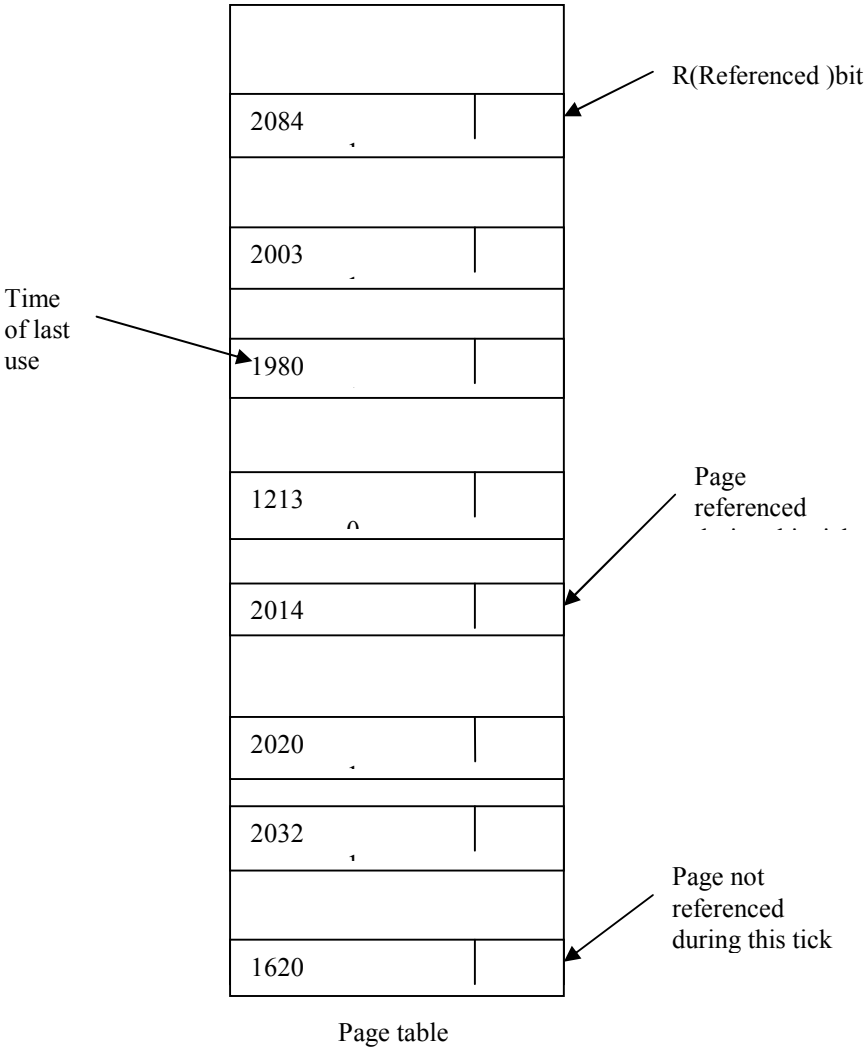


Figure 2.2.12 : The working set algorithm

The algorithm works as follows.

The hardware is assumed to set the R and M bits, whenever a page is referenced or modified (as we have discussed before). Similarly, a periodic clock interrupt is assumed to cause a software to run that clears the Referenced bit on every clock tick. On every page fault, the page table is scanned to look for a suitable page to be replaced.

As each entry is processed, the R bit is examined. If it is 1, the current virtual time is written into the *'*Time *of last use*' field in the page table, indicating that the page was in use at the time when the fault occurred. Since the page has been referenced during the current clock tick, it is clearly in the working set and is not a candidate for removal. (τ is assumed to span multiple clock ticks).

If R is 0, the page has not been referenced during the current clock tick and may be a candidate for removal. To see whether or not it should be removed, its age, that is, the current virtual time minus its *'Time of last use'* is computed and compared to τ. If the age is greater than τ, the page is no longer in the working set. It is replaced and the new page loaded here. The scan continues updating the remaining entries, however.

However, if R is 0 but the age is less than or equal to τ, the page is still in the working set. The page is temporarily spared, but the page with the greatest age (smallest value of *'Time of last use'*) is noted. If the entire table is scanned without finding a victim, that means that all pages are in the working set. In that case, if one or more pages with R = 0 is found and the one with the greatest age is replaced. In the worst case, all pages have been referenced during the current clock tick (and thus all have R=1), so one is chosen at random for removal, preferably the one whose modify bit is not set.

Besides the algorithms that have been discussed here, least frequently used (LFU) and most frequently used (MFU) are other algorithms that can be used for page replacement.

2.2.4.1.7 Summary of page replacement algorithms

We have now looked at a variety of page replacement algorithms. In this subsection we will briefly summarize them. The list of algorithms discussed is given in figure 2.2.13.

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark.
FIFO (First-in, First-out)	Might throw out important pages
Second chance	Big improvement over FIFO
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Working set	Somewhat expensive to implement

Figure 2.2.13 : Page replacement algorithms discussed in the text

2.2.5 Page Replacement Policies

So far, we have considered only 'local page replacement policy' in our page-replacement algorithms. What is this local page replacement policy? It means that when a process requests for a new page to be brought in and there are no free frames in the memory, we choose a frame allocated to only that process (and not of any other process), for replacement. For example, if a program "1" wants a page p14 to be loaded into the memory. This is done by choosing page p13 (of course using some page replacement algorithm) of same program "1".

Whereas in 'global replacement' policy, any frame from any other process can be replaced. For example, page p24 of a program "2" can be chosen to be replaced by p14, if global page replacement policy is used. Thus, here, the operating system has to decide the set of worst pages i.e. least referenced pages from all processes and then choose a page amongst them. In this case, process priorities and types also play a major role. Therefore, without going into any detail, we can say that the page replacement algorithms which use global page replacement policy, will be very difficult to implement and their discussion is outside the scope of this text.

2.2.6 Demand Segmentation

Demand paging is generally considered to be the most efficient virtual memory system but a significant amount of hardware is needed to implement it. The lack of this hardware may sometimes be a hindrance in the implementation of demand paging.

Demand segmentation is another method of implementing virtual memory. Demand segmentation allocates segments rather than pages to processes. It keeps track of these segments in segment descriptors / segment table that are similar to a page table and store information about the segment size, location and protection. As virtual memory provided with paging enables the execution of the process without it being totally in memory, similarly, with segmentation all the segments do not have to be in memory to execute.

The segment descriptor contains a valid bit in the same way as in paging to show if the segment is in memory.

If the segment is in main memory, the access continues unhindered. If the segment is not in memory, a trap to the operating system occurs (segment fault), just as a page fault occurs in demand-paging implementations, and the required segment is swapped in from secondary memory.

In order to make the selection of a segment to be replaced by another segment, a bit in the segment descriptor called accessed bit is used. An accessed bit services the same purpose as a reference bit in demand paging. It is set whenever any byte in the segment is either read or written.

When an invalid-segment trap occurs, the memory management routines first determine whether there is sufficient free memory space to accommodate the segment. If there is not enough room in memory, the least recently used segment is chosen for replacement and is written to disk. If the newly freed space is large enough to accommodate the requested segment, then the requested segment is read into the vacated segment, the segment descriptor is updated, and the segment is placed at the head of the queue. If the newly freed space is not that large, then the requested segment cannot be read and probably one more segment has to be replaced.

Demand segmentation has the disadvantage of high overheads and is not an optimal means for making best use of the resources of a computer system. It suffers from external fragmentation. The problem of external fragmentation can be reduced by taking the size of segment (to be brought into the memory) into consideration when the segment replacement decision takes place.

2.2.7 Keywords

Virtual Memory: it is an important concept with which every user program gets a feeling of being allocated the needed main memory in one go whereas actually, it is not like that. It is implemented through demand paging and demand segmentation.

Demand Paging: It is one way of implementing virtual memory. Whenever, an invalid page is addressed, it is put into the main memory from the secondary memory.

Page Fault: It occurs when an invalid page is addressed.

Pure Demand Paging: it is the form of demand paging in which not even a single page is loaded into memory, initially. Thus, the very first instruction causes a page fault in this case.

Demand Segmentation: it is another method of implementing virtual memory. It allocates segments rather than pages to processes. Segments are brought into the memory as and when desired.

Swapping: it is the technique of temporarily removing inactive programs from the main memory of a computer system.

Page Replacement: When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. This decision is made by using different page replacement algorithms.

Thrashing: It is referred to as high paging activity of a process.

2.2.8 Summary

This lesson has explained the concept of demand paging and demand segmentation for virtual memory management. In case of a page fault, if the main memory is full, one page is to be swapped out of it. This decision is taken by one of the various available page replacement algorithms. These are FIFO, LRU, optimal algorithms etc. Algorithm based on the working set model can also be considered for this decision. It greatly reduces the page fault rate as compared to other techniques.

2.2.9 Short Answer Type Questions

Q.1. Explain the term 'virtual memory'.

Q.2. How is swapping used in the implementation of virtual memory?

Q.3. How can a "dirty bit/modify bit" improve the performance of a virtual memory system?

Q.4. How can the performance of demand paging mechanism degrade?

Q.5. What is a page fault? How is it serviced?

Q.6. Explain demand paging with the help of an example.

Q.7. Is page fault a form of interrupt?

Q.8. What is Belady's anomaly? Does LRU algorithm for page replacement, suffer from this anomaly? Justify your answer with the help of an example.

Q.9. Consider that the pages are referenced in the following sequence:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.
How many page faults would occur for the following replacement algorithms, assuming one, two, three, and four frames?
1. LRU 2. FIFO

Q.10 If demand segmentation is used for virtual memory implementation, do you think that it will be necessary to consider "the size of the segment (in demand)" in the segment replacement algorithm.

Q.11 What is thrashing? What is the cause of thrashing? How can the system eliminate the problem of thrashing?

Q.12 Compare the different page replacement algorithms with respect to:
- Time complexity - Cost of implementation.

2.2.10 Long Answer Type Questions

Q.1. Explain the steps that need to be taken when a page fault occurs in a paged virtual memory system (assume there is a free frame available).

Q.2. Explain the steps that need to be taken when a page fault occurs in a paged virtual memory system (assume there is no free frame available). Hint: Page replacement will be required.

Q.3. List the advantages and disadvantages of page replacement algorithms.

2.2.11 Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts" , Khanna Publishing Co., 2002
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.

Web Resources :

www.personal.kent.edu/~rmuhamma/opsystems/os.html
www.wiley.com/college/silberschatz6e/0471417432/slides/slides.html

FILE MANAGEMENT

Contents

2.3.0 Objectives
2.3.1 Introduction
2.3.2 File Concept
2.3.2.1 File Naming
2.3.2.2 File Attributes
2.3.3 File Access Methods
2.3.4 Directory Structure
2.3.4.1 Hierarchical Directory Systems
2.3.4.2 Access paths
2.3.4.3 Directory Operations
2.3.5 File Protection
2.3.5.1 Access control
2.3.5.2 Other protection mechanism
2.3.6 File System Implementation
2.3.6.1 Contiguous allocation
2.3.6.2 Linked list allocation
2.3.6.3 Linked list allocation using an index
2.3.6.4 I-nodes
2.3.7 Free Space Management
2.3.7.1 Bitmap
2.3.7.2 Linked List
2.3.7.3 Grouping
2.3.7.4 Counting
2.3.8 Keywords
2.3.9 Summary
2.3.10 Short Answer Type Questions
2.3.11 Long Answer Type Questions
2.3.12 Suggested Readings

2.3.0 Objectives

The first part of this lesson introduces file: an abstract data type defined and implemented by the operating system. File concepts containing naming of files, file operations, file attributes have been discussed. File access methods - 'sequential'

and 'random' have been described and their comparison has been done in brief. Directory structures, directory operations and access paths have been explained in detail. Hierarchical directory structures have been discussed. A brief notion of file protection has been introduced. The second part of this lesson deals with file system implementation issues. It involves the discussion of various methods of allocating space to the files on the disk and management of free space on the disk.

2.3.1 Introduction

It becomes essential to store information for long-term so that it can be accessed at any time. As seen in earlier lessons, it is also essential to make data sharable among various processes. This information can be huge in size and therefore, must be accommodated on the appropriate storage devices. Thus, these are three essential requirements for long-term information storage. It indicates that information should be stored on disks and other external media in units called files. Processes, running in the system can read or write these storage media via files. Information stored in files must be persistent i.e. it should not be affected by process creation and termination.

Files are managed by the operating system. How are they structured, named, accessed, protected and implemented are few major considerations in operating system design. The part of operating system, which deals with files, is known as the file system.

2.3.2 File Concept

Physically, **A file is the smallest allotment of secondary storage device e.g. disks.** Logically a file is sequence of logical records i.e. a sequence of bits and bytes. A file has various attributes like name, type, location, size, protection, time and data of creation, user information etc. Let us look at various attributes from the user's point of view.

2.3.2.1 File Naming

The exact rules for file naming vary somewhat from system to system, but all operating systems allow strings of one to eight letters as legal file names. Thus 'ekta', 'walia', 'lecturer' are possible file names. Frequently digits and few special characters are also permitted therefore, file names like 'OS1', 'Sys_Prog' are also permitted. Similarly, some file systems distinguish between upper case and lower case letters, whereas others do not.

Almost every operating system supports two parts of a file name, where the two parts are separated by a period, as in 'ekta.c'. The part following the period is called the file extension, and usually indicates something about the file. The size of this extension can be upto 3 characters in MS-DOS whereas, UNIX permits that a file may have two or more extensions like 'ekta.c.z'., where '.z' indicates that a file has been compressed or zipped. Figure 1 shows some typical file extensions.

Extension	Meaning
.bak	Backup file
.bas	Basic source program
.bin	Executable binary program
.c	C source program
.dat	Data file
.doc	Document file
.hlp	Text for HELP command
.obj	Object file(output of compiler)
.txt	General text file

Figure 1: Some typical file extensions

File extensions are important to know because they indicate a particular software needed to open a particular file e.g. a C compiler will compile only '.c' files and not any other file. Looking at the file "ekta.c", the user can identify that this is a program file in 'C' language and can be opened and compiled using a 'C' compiler.

2.3.2.2 File Attributes

A file has a name and data. Other than this, the operating system can associate other information like date and time of file creation, file's current size etc. with a file. These are known as attributes of a file. Figure 2 shows a list of possible attributes. The list of these attributes varies from system to system.

File Preparation

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. The following list shows the most common file operations available via system calls:

1. **CREATE**: A blank file is created. The purpose of this call is to create a new blank file with the specified attributes.
2. **DELETE**: The purpose of this system call is to delete this file from the disk when it is not needed.
3. **OPEN**: Before using a file either for reading or writing, it must be opened. The purpose of this call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on subsequent calls. This is a very important concept, which relates to caching.
4. **CLOSE**: When a file is no longer accessed, it must be closed using this system call.
5. **READ**: When a file is only to be read, this call is used. The location of the data to be read must be specified. The size of data to be read must also be specified.

6. **WRITE**: Data can be written to file with this system call. The data is written at the current position in the file and if the current position is at the end of the file, then the file size increases automatically.
7. **APPEND**: This call is a restricted form of WRITE. It can only add data to the end of the file.
8. **SEEK**: This call is used to reposition the current pointer to a specific place in the file. After this call succeeds the pointer is placed at the desired location and data is read or written there, using another system call (like READ or WRITE).
9. **GET ATTRIBUTES**: Processes often need to know the attributes of some files to do their work. For example, a process might want to display the id of the person who created the file. Therefore, it can use this system call to get the attributes of that file.
10. **SET ATTRIBUTES**: Using this system call, attributes of a file can be set. For example, a user may want to set its read-only flag to 1, so that it can only be read by the users and thus, cannot be written.
11. **RENAME**: A user may need to change the name of a file very often. Therefore, this functionality is provided by RENAME system call.

Protection, password, creator and owner attributes are basically used to protect a file from unauthorized access. Some systems provide the privilege to lock a file with some particular password. Only those users who know this password can then access this file. The flags mentioned in figure 2, are used to enable or disable some specific property of a file. A file can be made read-only, hidden, system-file, and so-on by setting its read only flag to 1, its hidden flag to 1, its system flag to 1 and so on. The archive flag is a bit that keeps track of whether the file has been backed up or not? Once the backup of a file has been taken, its backup bit is set to 0. Operating system sets this bit to 1 whenever a change takes place in the contents of this file, so that it should again be backed up the user, using a back up program. Record length, key position and key length fields are only present in files whose records can be searched using a key.

The various time attributes keep track of when the file was created, most recently accessed and most recently modified. This information may be necessary to know that if the source file has been modified after the creation of the corresponding object file, then it needs to be compiled again. Maximum size attribute is used by some systems, which require the maximum size to be specified when the file is created.

Field	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	Identity of person who created the file
Owner	Current owner of the file
Read only flag	0 for read/write, 1 for read only
Hidden flag	0 for normal file, 1 for do not display in listings
System flag	0 for normal file, 1 for system file
Archive flag	0 has been backed up, 1 for needs to be backed up
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time file was created
Time of last access	Date and time file was last accessed
Time of last change	Date and time file was last changed
Current size	Number of bytes in the file
Maximum size	Maximum size file may grow to

Fig 2: Some possible file attributes

2.3.3 File Access Methods

Files can be categorized broadly into two types depending upon their access methods. Sequential files are those, which are read sequentially (one record, then another and so on) in an order, starting at the beginning to the end of the file. Sequential files can be rewound so that they can be read as often as needed. However, records in such files cannot be read out of order e.g. reading of 34th record followed by 5th record and then 1st record is not possible with sequential files. Sequential files are convenient when the storage media is serial-access e.g. magnetic tape rather than direct-access e.g. magnetic disk.

Sequential Access: in this access method, data records are retrieved in the same order in which they have been stored on the disk. The organization is such that the length of each record is fixed and can not be changed. The data contents can only be replaced by fresh contents. The read and write operations are performed sequentially on file contents. This access method is the simplest one as the disk head moves in one direction to access the mentioned data records. It need not jump randomly to forward and backward disk blocks during every next retrieval. Magnetic tapes stores information in sequential manner and like wise the retrieval is performed. Printing and scanning devices provide efficiency in their performance while accessing data records in an order. In case of select or update operations

where random access to data records is required, the sequential access gives inefficient results.

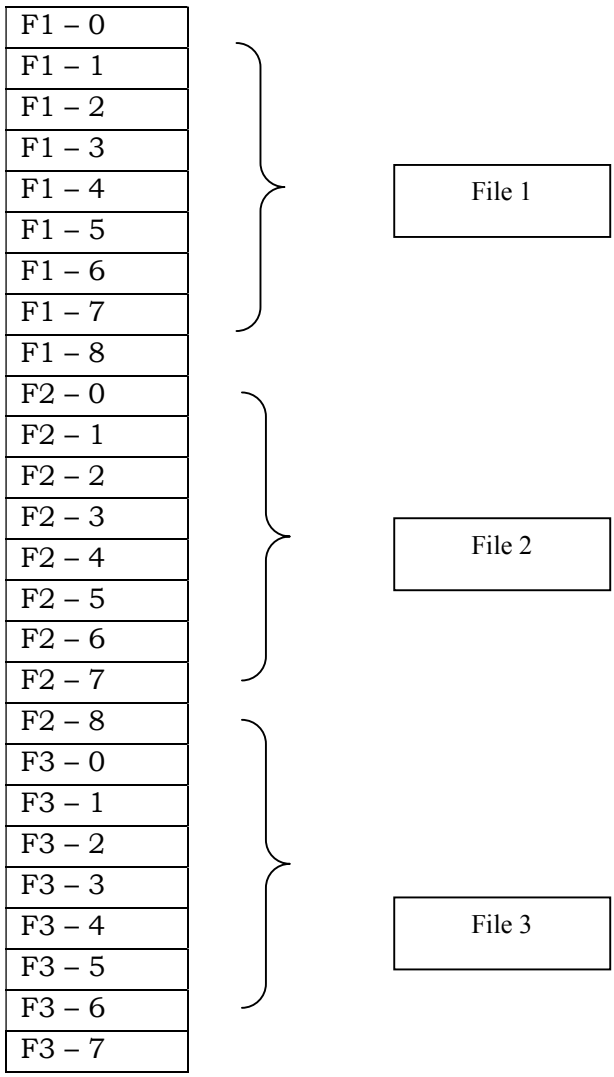


Fig 3: Sequential access

Files whose records can be read in any order are called random access files. Random access files are basically a collection of records stored on a disk and these records are numbered 1,2,3 and so on and therefore, can be referred to by their numbers instead of their position. Such files should be necessarily be stored on direct access media like disk.

Random access files are essential for many applications, for example, data base systems. In a banking application, a customer may want to look up his current balance. This can be easily done by locating this customer's record using

his account number as a key, rather than sequentially reading the records for thousands of other customers before this customer's record is located and read.

It is worth mentioning here that not all operating systems support both sequential and random access for files. Some systems allow only sequential file access, others allow only random access. Some systems require that a file should be defined as sequential or random when it is created, so that it can be accessed in the way it has been declared.

Random Access: In case of random access the record is searched from the disk based on its direct address information. The technique used is Hashing. In hashing every record is associated with a key number to preprocess the address calculation. Hash function is used to obtain absolute address of a particular record. No any further search algorithm is required to fetch the mentioned data record. The only restriction here is that hash table used to store entries must be of fixed size and comparable to the number of data records so that all data records got uniformly distributed among all the table entries. E.g. suppose there are 10 records and hash table size is also 10. Every record will be assigned an address value from 1-10 based on the hash function result. Similarly with the help of hash function the data values will be retrieved at any moment of time. The complexity involved in this method is the selection of hash function. A hash function should be such that makes uniform distribution of records among the provided memory space. Also hash function must not generate similar address values for more than one data records. The situation where more than one records are assigned same address is called collision state. The moment record count is limited or small the hashing technique proves to be very beneficial for query processing, but as the number of records increases the complexity of hash function increases to provide maximum unique address distribution among available records. There are many schemes provided to handle collisions. The important technique used in this concern is chaining. In chaining, a linked list is attached to every hash table entry and similar address data records are made to get added up to that list. Another technique is Rehashing, where the hash address obtained is further processed with another hash function to generate final hash address.

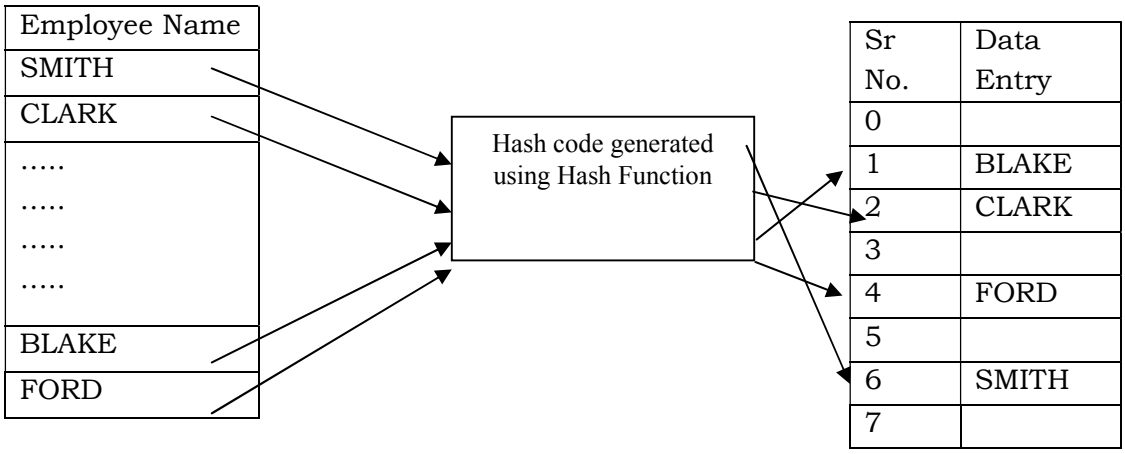


Fig 4: Random access

Index Access method: Indexed file approach is helpful with multiple attribute fields like in database files. In these files, every field is associated with an index key. While querying data the index key is kept in the memory and related records are fetched from the disk. These indexing keys help distinguishing one record from the other. Indexes are managed in various ways like single level index and multilevel index. These are also called dense and sparse index respectively. In single level, every record has an index value associated with it. These index values can get associated with primary or secondary index key. In case of multi level index, a group of entries have one index key associated and rest of the entries in that group are further being indexed based on another key criteria. Indexes require limited memory space for execution as only index key values are required at a moment in memory. In other cases whole data file need to be in main memory. This result in wastage of memory area allocated to a process. While query processing generally we deal with just one third of the whole contents and rest of the memory space occupied is always a waste. Indexed access method lies between sequential and random access and provide benefit in case of large size of record files.

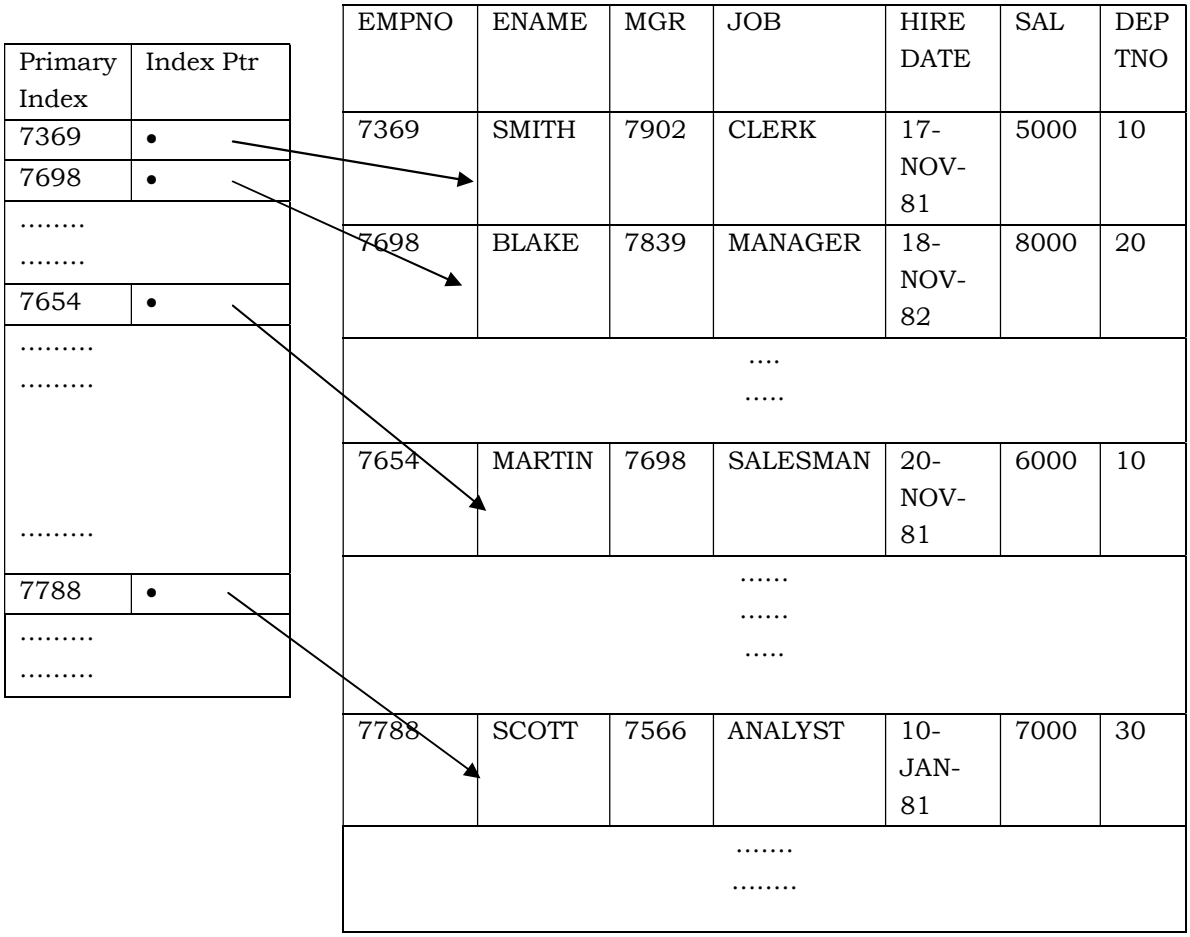


Fig 5: Index Access Method

2.3.4 Directory Structure

To keep track of files, the file systems normally provide directories, which, in many systems, are themselves treated as files. A directory contains information about files. Thus, it is central to the functioning of the file system. A directory is used as a means to group the files owned by a user. This notion of grouping can be extended to permit grouping of files according to some user-defined criteria. This section focuses upon directories, their properties, organization and the operations that can be performed on them.

2.3.4.1 Hierarchical Directory Systems

A directory typically contains a number of entries, one per file. One possibility is that each entry contains the file name, the file attributes, and the disk address where the file is stored. Another possibility is that a directory entry holds the file name and a pointer to another data structure where the attributes and disk

addresses are found. Both of these systems are commonly used. Figure 6 depicts the two ways to represent a directory.

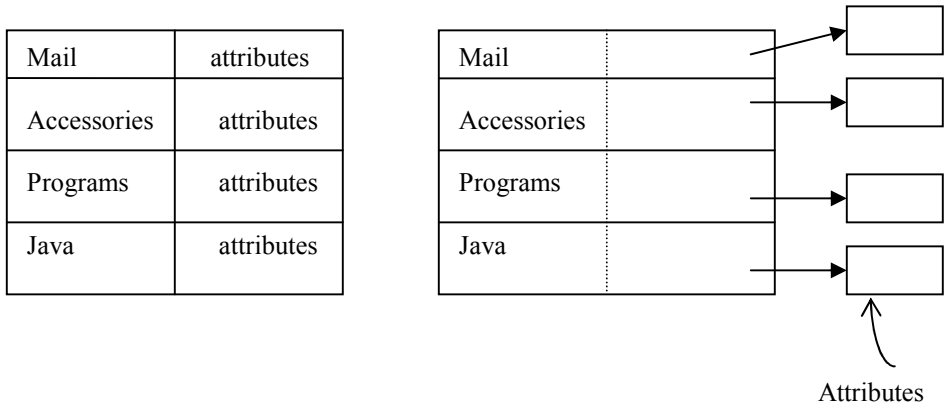


Figure 6: Directories (a) Attributes in the directory entry (b) Attributes elsewhere

When a file is opened, the operating system searches the directory structure until it finds the name of the file to be opened. It then extracts the attributes and disk addresses, either directly from the directory entry or from the data structure pointed to and puts them in a table in main memory. All subsequent references to the file use the information that has been fetched in the main memory.

The number of directories varies from system to system. The simple design is for the system to maintain a single directory containing all the files of all the users, as illustrated in Figure 7. In this case, however, If there are many users, and they choose the same file names, conflicts and confusion will make the system unworkable. This system model is used only by the most primitive operating systems.

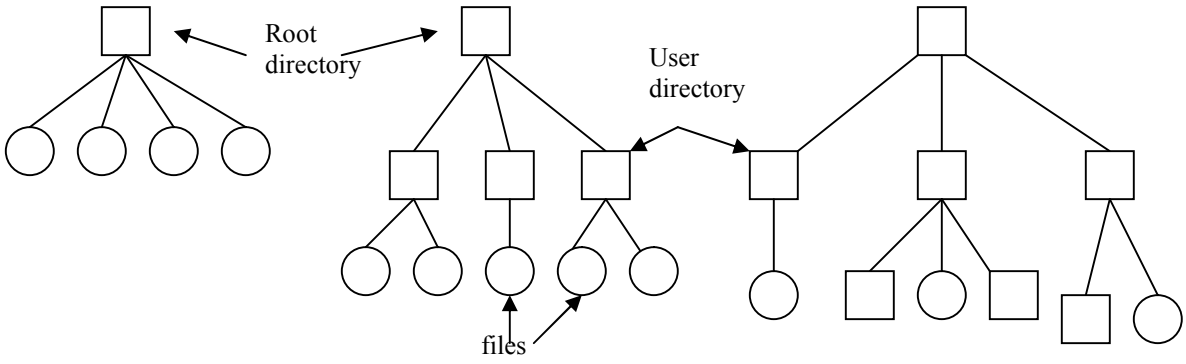


Figure 7 : Three file system designs (a) Single directory shared by all users;(b) One directory per user; (c) Arbitrary tree per user. The letters indicate name of files or directories

An improvement on the idea of having a single directory for all files is to have one directory per user as shown in figure 7(b). This design eliminates name conflicts among users, but is not very satisfactory for users with many files. It is quite common for users to want to group their files together in logical ways. For example, I may wish to have a collection of the files related to operating system concepts) to be consulted by me while writing this book). Similarly, I may wish to have a collection of files containing assignments submitted by my students learning Java or a collection of files containing the source code of my project work etc.

A general hierarchy (i.e., a tree of directories) is thus needed. With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Figure 7 (c). Here the directories A, B, and C contained in the root directory each belong to a different user and two (B and C) of them have subdirectories for two different projects.

2.3.4.2 Access paths

When the file system is organized as a directory tree, some way is needed to specify file names uniquely. Two methods are possible for specifying access paths;

(1) Absolute path name: It is a listing of the directories and files from the root directory to the intended file. For example, the path 'c:/ windows/ programs/ spss.exe' means that the root directory contains a subdirectory 'windows', which further contains a subdirectory 'programs', that contains an executable "spss.exe".

Absolute path names are always unique as they start the specifications from the root directory. Mostly '/' is used as a separator in the path name string. Some operating systems use '\' or '>' as path name separators.

(2) Relative path name: This uses the concept of current directory (also known as working directory). A user can specify a particular directory as his current working directory and all the path names instead of being specified from the root directory are specified relative to the working directory. For example, if the current working directory is '\user\curr', then the file whose absolute path is '\user\curr\ekta' can be referred simply as 'ekta'. Thus, relative path name specifies the access path relative to the current working directory. What happens when a program needs to access a specific file without regard to what the working directory is? This can lead to errors if relative path names are used. For example, if a program wants to open an instance of Microsoft Word, then it should specify the location of its exe by specifying absolute path name instead of relative path name, because the working directory is not known in advance. However, the absolute path name will always work, no matter, what the working directory is.

Most operating systems that support a hierarchical directory system have two special entries "•" and "•• " in every directory. They depict the current directory and the parent of the current directory respectively. Let us see how are they used. Refer to figure 8, which shows a directory tree in UNIX operating system. A certain process having '\user\jim' as its current working directory can use •• to go up the tree. For example, it can copy file '\user\ekta\Java1' to its own directory using the shell command: -

cp •• \ekta\java1

The first path instructs the system to go upwards (to the 'user' directory), and then to go down to the directory 'ekta' to find the file java1 there.

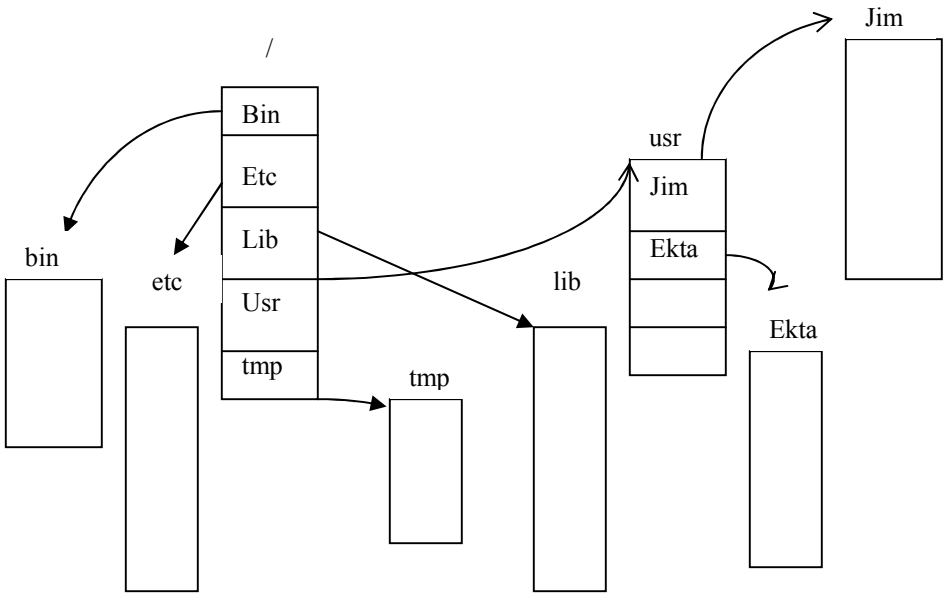


Figure 8: A UNIX directory tree

The second argument of this command specifies the current directory. Thus, this cp command copies the file 'java1' to the current working directory. This command is equivalent to

cp \user\ekta\java1 •

The use of dot here saves the user from the trouble of typing 'java1' again as the second argument.

Links

The access path information described so far has been based on the parent child relationship in the file system directory hierarchy. This can make access paths long and cumbersome to write. A link is a directed connection in the file system hierarchy between two existing files. A link is thus an ordered triple: -

(<from_filename>, <to_filename>, <linkname>) where <from_filename> is a directory, <to_filename> is a directory or a file and <linkname> is a symbolic name for the link. Once a link is established, the <to_filename> can be accessed as it were a child of <from_filename> i.e. as if it were a file named <linkname> in the directory <from_filename>. Figure 9 shows the implementation of a link. It considers A to be a user directory and establishes a link named mylink between the directory A and the file 'ekta' whose access path is /A/Projn/notesonos/ekta. Note that the existence of links makes the directory structure a *generalized graph* rather than a *tree*. A file may have multiple access paths in the file system. In fact, access paths may even contain loops.

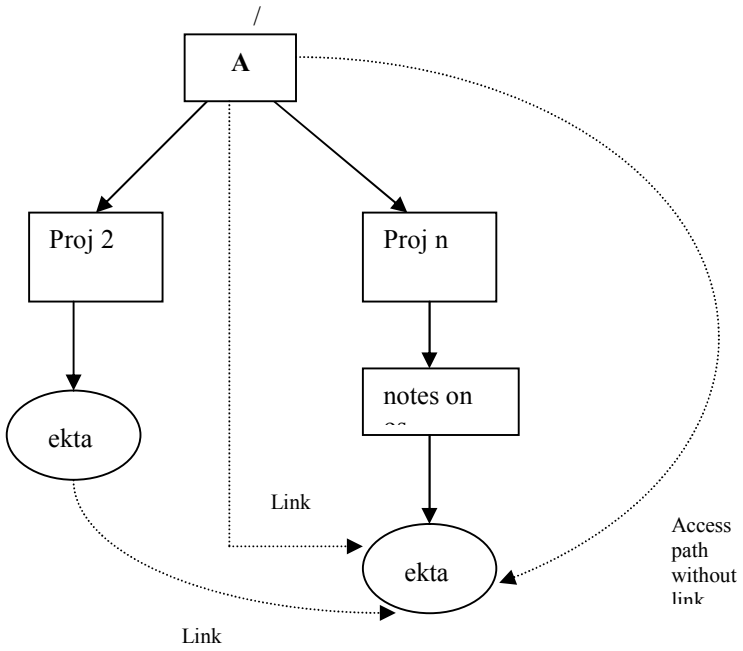


Figure 9: Generalized graph structure using links

The figure 9 shows that the link permits \A\proj n\notes on os\ekta to be accessed by the name \A\link1.

2.3.4.3 Directory Operations

Many systems calls are allowed for managing directories. They vary from system to system. This part of text discusses some common system calls used to manipulate directories.

1. **CREATE**: A directory is created. It is empty except for dot (·) and dotdot (··), which are put there automatically by the system.
2. **DELETE**: A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty, as these usually

cannot be deleted. In order to delete a directory having subdirectories, first the subdirectories should be deleted.

3. **OPENDIR**: Directories can be opened for reading it. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened.
4. **CLOSEDIR**: When a directory has been read, it should be closed to free up internal table space in main memory.
5. **READDIR**: This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual READ system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, READDIR always returns one entry in a standard format, no matter which of the possible directory structures are being used.
6. **RENAME**: Directories can be renamed just like files.
7. **LINK**: This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories.
8. **UNLINK**: A directory entry is removed. If the file being unlinked is present only in one directory, it is removed from the file system. If it is present in multiple directories, only the path name specified is removed and the others remain as such.

2.3.5 File Protection

File systems often contain information that is highly valuable to their users. Protecting this information against unauthorized usage is therefore, a major concern of all the file systems. In the following sub-sections we will look at a variety of issues concerned with file protection.

2.3.5.1 Access control

The access to a particular file can be controlled by limiting the types of file access that can be made. Following are the few file operations that can be controlled:

* Read - Read a file
* Write - Write the file
* Append - Append a file
* Delete - Delete a file
* List - List the name and attributes of a file
* Rename - Rename a file
* Edit - Changing the contents of a file
* Copy - Make a copy of a file.

One way to control these operations in through access control lists (ACLs) and groups. An access list may be associated with each file and directory. This list may contain the user name and the types of access allowed for each user. The operating system checks this access control list (associated with a file) whenever a user requests an access to a particular file. If that user is listed for the requested access, the access is allowed. Otherwise, user is denied access to the file.

The main problem with access controls list (ACLs) is their length. If we want to allow every-one to read a file, we must list all users with read access. To overcome this problem many systems classify the users of a file into three types:

1. **Owner** - The user who created the file.
2. **Group** - A set of users who are sharing the file and need similar access.
3. **Universe** - All remaining users in the system constitute universe.

Thus, if a person has created a file, he/she is owner of the file. If the owner has given execute-only permission (on his file) to some set of users, then this set of users constitute a group and all users in this group have execute-only permission for the file. All the other possible users will fall into the universe category and probably may be given read-only access to the file. UNIX operating system uses this method to provide protection to the files. Access control lists should be created and maintained by some administrator, file owner or any other manager of the organization.

2.3.5.2 Other protection mechanism

Files can be protected by a password. The owner of a file can control its access by assigning a password. Thus, only those users who know the password, can access a particular file.

Disadvantages of this scheme are:

1. If a separate password is associated with each file, the user will have to remember so many passwords and the scheme will become impractical.
2. If only one password is used for protecting all files, then cracking of one password by an unauthorized user will enable him to access all the files.

This problem can be dealt by associating a password with a subdirectory rather than with an individual file.

2.3.6 File System Implementation

So far, we have discussed the users' view of a file system. Let us now look at implementer's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like and similar interface issues. Implementers are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. Let us examine the file implementation issues here.

The key issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different systems. In this section, we will examine a few of them.

2.3.6.1 Contiguous allocation

The simplest allocation scheme is to store each file as a contiguous block of data on the disk. Thus, on a disk having blocks size 1k, a 25k file would be allocated 25 consecutive blocks. This scheme has two significant advantages:

(1) It is simple to implement because keeping a track of where a file's blocks are, is reduced to remembering one number only. This number is the disk address of the first block occupied by the file.

(2) Performance is good because entire file can be read from the disk in a single operation (just because it is stored contiguously).

Disadvantages of this scheme are:

(1) It is not feasible unless the maximum file size is known at the time of file creation. If it is not known, the operating system does not know how much disk space should be reserved.

(2) Fragmentation of the disk may occur as a result of this scheme. Therefore, space is wasted that might otherwise have been used. Compaction cannot be applied as a solution to this problem, because compaction of the disk is expensive.

2.3.6.2 Linked list allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Figure 10. The first word of each block is used as a pointer to the next one. The rest of the block is used for storing data.

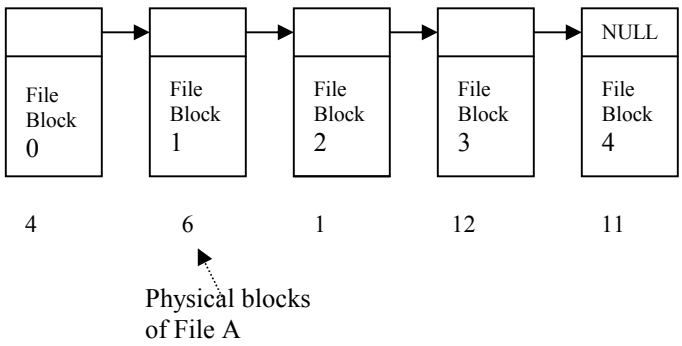


Figure 10: Storing a file as a linked list of disk blocks

Advantages of this scheme are:

(1) Unlike contiguous allocation, every disk block can be utilized by this method. Thus there is least possibility of disk fragmentation.

(2) The directory entry can merely store the disk address of the first block and this can be used to find the rest of the blocks occupied by the file.

This scheme however, suffers from the following disadvantages:

(1) Although reading a file sequentially is easy, but random access becomes extremely slow.

(2) Pointer takes up space in each disk block and therefore, the amount of data storage in a block is no longer a power of two, which otherwise would have been a power of two. This is essential because most programs read and write data in blocks whose size is a power of two.

2.3.6.3 Linked list allocation using an index

This scheme is an attempt to remove the disadvantages of the linked list allocation. Here, instead of having a pointer, an index is maintained. For the figure 10, where the linked allocation of two files namely A and B has been shown, table or index like the one shown in Figure 11 can be used. Using the table of figure 11, for file A we can start with the block 4 and follow the linked list all the way to the end. The same way for file B, we can start from block 3 and follow the linked list.

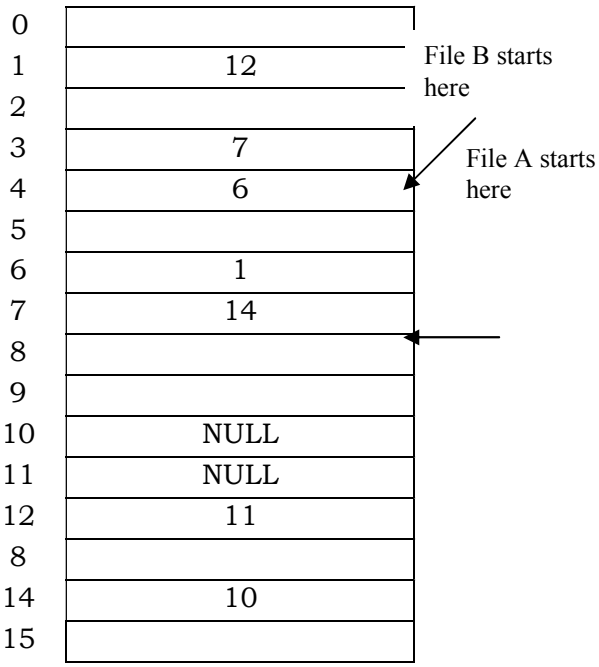


Figure 11: Linked list allocation using a table in main memory

Advantages of this scheme are:

1. The entire block is available for data. No space is consumed by pointers etc.
2. Random access is easier. This is so because, no doubt, the chain has to be followed to locate a particular part of a file, but because the index is in

memory, the seek operations do not involve disk references as in the previous case.

3. Here also, the directory entry can merely store the disk address of the first block and this can be used to find the rest of the blocks occupied by the file. This method is used by MS-DOS for disk allocation.

Disadvantages of this scheme are:

1. The entire index or table will have to be kept in man memory for all the time to make it work. This may prove to be hazardous if the disk is large and thus the table or index is also large.
2. Looking up for an entry in such a large index will also become very time consuming.

2.3.6.4 I-nodes

This method for keeping track of which block belongs to which file, is used by UNIX operating system. In this scheme, each file is associated with a little table called an i-node (index node), which lists the attributes and disk addresses of the file's blocks. I-nodes are shown in figure 12.

This scheme is implemented as follows:

1. For small files, the first few disk addresses and all the necessary information are stored in the i-node itself. Whenever a file is opened, all this information is fetched from disk to main memory.
2. For somewhat larger file, one of the addresses in the i-node is the address of a disk block called single indirect block. This block contains additional disk addresses.
3. For even larger file, the i-node may contain another address which is the address of a disk block called a double indirect block. This block contains the addresses of the blocks containing the lists of single indirect blocks. Each of these single indirect blocks point to a few hundred data blocks. Similarly, a triple indirect block can also be used.

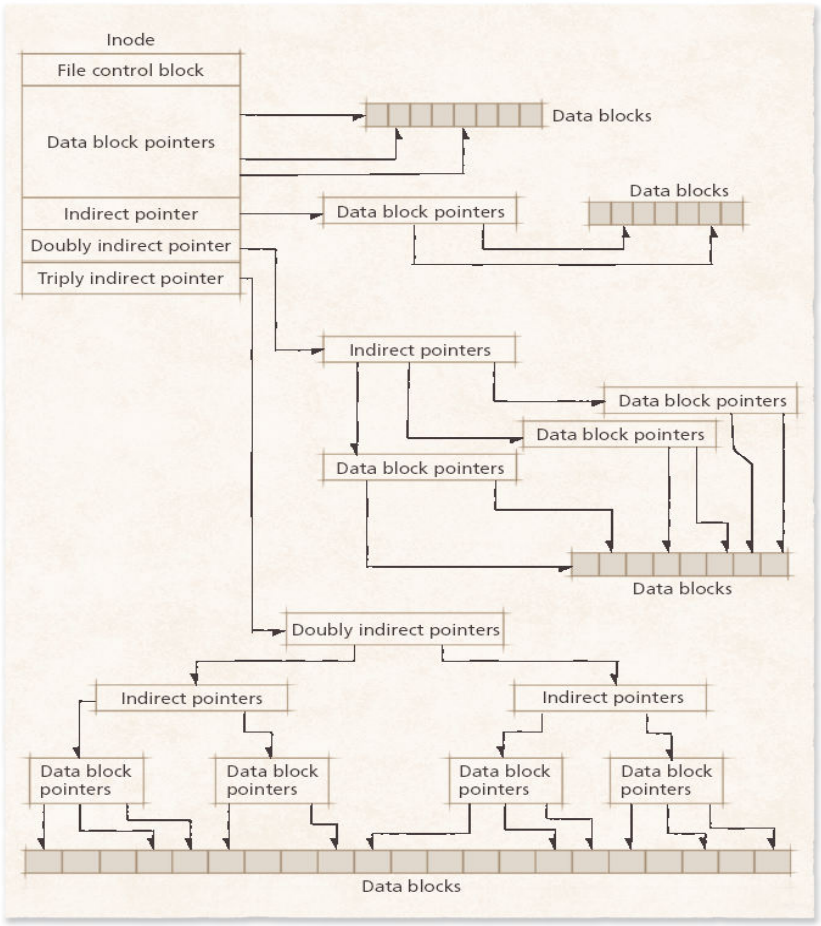


Figure 12 : An i-node

2.3.7 Free Space Management

It becomes necessary to reuse the disk space created after deleting the files so that new files can use it. The operating system maintains a free-space list to keep track of free disk space. Whenever, a new file is created, the free-space list is searched for the required amount of space. This free space is allocated to the newly created file and its entry is then removed from the free-space list. When a file is deleted, the disk space freed up by it is then added to the free-space list.

Let us now understand the various possible methods of implementing free-space list:

2.3.7.1 Bit map

The free space list is implemented as a bit map. Every bit represents a block on the disk. The bit for a block is 1 if it is free and it is 0 if the block is allocated. This bit map is stored on specific place on the disk. The free-space bit map is as shown in figure 10.

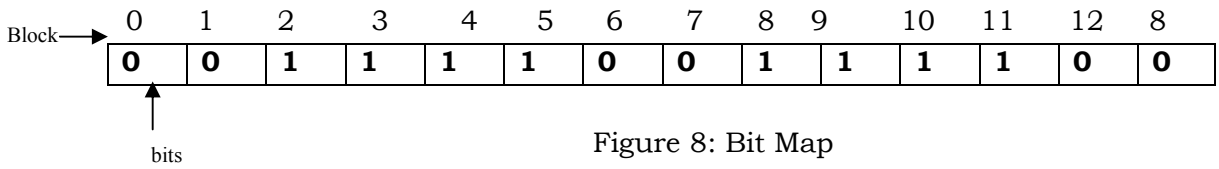


Figure 8: Bit Map

The figure 10 indicates that the blocks 2, 3, 4, 5, 8, 9, 10, 11 are free and rests of the blocks are allocated.

Advantages of this approach are:

1. It is simple and efficient to find the first free block or n consecutive free blocks on the disk. To find the first free block, the operating system checks sequentially each word in the bit map to see whether that value is not 0. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block.

Disadvantages of this approach are:

1. This may require special hardware support (bit operation) to find the first '1' in a word if it is not zero.
2. Bit maps are inefficient unless they are kept in main memory and are written to disk occasionally for recovery needs. If the disk size is large, the corresponding bit map is large and thus keeping it in the main memory is not possible. Consider that the disk size is 1 GB $\Rightarrow 2^{30}$ bytes and the block size is 2^{12} bytes then the number of blocks will be $2^{30}/2^{12} = 2^{18}$ and corresponding to each block, a bit will be there in the bit map. This implies that size of bit map will be 2^{18} bits = 256 KB.

2.3.7.2 Linked list

This approach maintains a linked list of all the free disk blocks. The first free block in the list can be pointed out by a head pointer, which is kept in a special location on the disk. Figure 14 shows the linked list of free blocks.

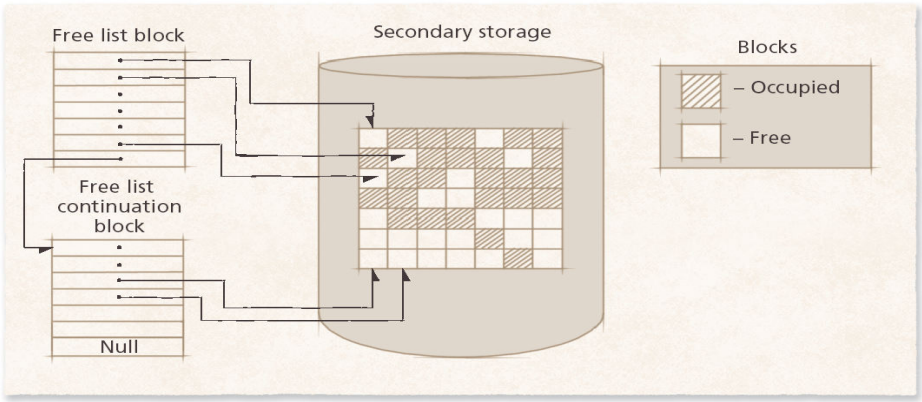


Figure 14: Linked free space list on disk

Advantages of this approach are:

1. Whenever a file is to be allocated a free block, the operating system can simply allocate the first block in the free space-list and move the head pointer to the next free block in the list.

Disadvantages of this approach are:

1. Traversing the free space list will be very time consuming; each block will have to be read from the disk, which is read very slowly as compared to the main memory.
2. Pointers used in the linked list will consume disk space.

2.3.7.3 Grouping

Another approach is to store the addresses of n free blocks in the first free block. The last block contains the addresses of other n free blocks and so on. This means that first n-2 blocks of these blocks are actually free (as the first and last blocks of the group will contain addresses). The advantage of this approach over linked list approach is that by accessing only one block, the addresses of a large number of free blocks can be known at once.

2.3.7.4 Counting

Another approach takes into consideration that generally, several contiguous blocks may be allocated or freed simultaneously, particularly when contiguous allocation algorithm has been used for allocating disk space. Thus, instead of keeping a list of addresses of n free blocks, it is more convenient to keep the address of the first free blocks and the number n of free contiguous blocks that follow the first block. Therefore, each entry in the free space list then consists of a disk address and a count. This may make the overall list shorter in size.

2.3.8 Keywords

Access path: When the file system is organized as a directory tree, some way is needed to specify file names uniquely. Two methods are possible for specifying access paths- absolute and relative path names.

Absolute path name: It is a listing of the directories and files from the root directory to the intended file.

Relative path name: A user can specify a particular directory as his current working directory and all the path names instead of being specified from the root directory are specified relative to the working directory.

Links: it is a directed connection in the file system hierarchy between two existing files.

Access control: The access to a particular file can be controlled by limiting the types of file access that can be made. The methods to control these operations in through access control lists (ACLs) and groups.

i-node: In UNIX operating system, each file is associated with a little table called an i-node (index node), which lists the attributes and disk addresses of the file's blocks.
Directory: A directory structure is an organized form of groups of files.

2.3.9 Summary

File is a unit of data storage in computer system. Files are used to store data and programs for various applications. We are provided with different types of data representations and like wise the files to store them. Files are stored in permanent storage so that user can retrieve, modify or delete it at its convenience. Also access modes must be defined for particular user at specific files to keep them protected from unauthorized access. These operations regarding file handling must be managed by operating system.

Files are basically kept in different forms depending upon the type of data contents in it, like word documents are stored with (.doc) extension, web pages are managed as .html, C programs with .C etc. the type of file tells us that how the contents have been stored init. E.g. In database files data has been managed in the form of Tables with Rows and columns. Every row represents one record of an object while column represents their attribute fields. Collectively the file gives information about the object and its properties. With every file management system we can create, delete, modify or move a file. File handling system must keep the access control of various users on data contents. The access rights are divided into two categories: Read or Write. With read access, user can check the contents but with write access user can even modify the contents.

A directory structure is an organized form of groups of files. Directories are used to store set of files under one name. Directories keep the file attributes like file name, type, content size, access mode etc. With the help of directories, similar files can be placed under one heading. Directories are managed in Tree or Graph Structures.

2.3.10 Short Answer Type Questions

Q.1. What is the difference between absolute and relative path name of a file?
Q.2. Compare sequential and random file access methods with respect to their usefulness in today's applications.
Q.3. What is the necessity of different directories for different users?
Q.4. Explain various file allocation methods and compare them with respect to space and time consumed by each of them.
Q.5. What is i-node? How is it is better than simple indexed allocation method?
Q.6. Explain the purpose of links with the help of an example.
Q.7. If you have to maintain a railway reservation application, what file organization will you choose for it and why?

2.3.11 Long Answer Type Questions

Q.1. We have learnt in this lesson that fragmentation on a disk could be eliminated by compaction. But disks do not have relocation or base registers (such as those that are used for memory compaction), so how can the files be relocated? Give three reasons why relocation of files on disks is often avoided.

Q.2. We have learnt in this lesson that bit maps are inefficient unless the entire map is kept in main memory (and is written to disk occasionally for recovery needs). Do you think there is a possibility of inconsistency between the bit map loaded into the main memory and that on the disk? If yes, how will it be ensured that such inconsistency should not occur.

(Hint: situation like bit for block i is 1 in memory and is 0 on the disk should be avoided. Solution is to set bit for block i to 1 on disk. Allocate this block i and set bit for block i to 1 in memory).

2.3.12 Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts" , Khanna Publishing Co., 2002
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.

Web Resources :

www.personal.kent.edu/~rmuhamma/opsystems/os.html
www.wiley.com/college/silberschatz6e/0471417432/slides/slides.html

Device Management-1

Contents

2.4.0 Objectives
2.4.1 Device Management
2.4.2 Device Management functions
2.4.3 Basics
2.4.3.1 Interrupts
2.4.3.2 Trap
2.4.3.3 I/O subsystem
2.4.4 I/O system organization
2.4.5 Direct I/O with polling
2.4.6 Interrupt driven I/O
2.4.7 Interrupts versus Polling
2.4.8 Keywords
2.4.9 Summary
2.4.10 Short Answer Type Questions
2.4.11 Long Answer Type Questions
2.4.12 Suggested Readings

2.4.0 Objectives

The aim of this lesson is to describe one of the most important work of an operating system i.e. device management. Whatever I/O devices are attached to a computer system must be managed by the operating system. Their allocation to various processes and their uses can vary**.** Before going to various details, the lesson discusses the most important concepts of interrupts, traps and I/O subsystem. The allocation of devices in shared, dedicated and virtual manner has been given. Abstraction is again the most important thing needed by a programmer to easily access the devices. The direct I/O, direct I/O with polling and interrupt driven I/O approaches have been described in detail. An exhaustive comparison of interrupts and polling techniques has been presented.

2.4.1 Device Management

It is the part of operating system responsible for directly manipulating the hardware devices. It provides the first level abstraction of these resources that will be used by applications to perform I/O. There are various approaches by which device management can abstract I/O operations. These are:

1. Direct I/O- It requires the CPU software to explicitly transfer data to and from the controller's data registers.
2. Direct I/O with polling- The device management software polls the device controller status register to detect completion of the operation.
3. Interrupt driven I/O-Interrupts are used to simplify the software's responsibility for detecting operation completion.
4. Memory mapped I/O-Device addressing simplifies the interface, resulting in widespread use in contemporary systems. This technique can be used with the polling/interrupt strategy. The device management software polls the device controller status register to detect completion of the operation.

Before proceeding further, let us discuss the various device management functions.

2.4.2 Device Management Functions

As the two main jobs of a computer are I/O and processing, therefore, it becomes essential to know the role of an operating system in managing and controlling the I/O operations and I/O devices.

The device management functions that must be performed by an operating system are:

1. Track the status of each device (such as tape drives, disk drives, printers, plotters and terminals).
2. Use algorithms to decide which process will get a device and for how long.
3. Allocate the devices.
4. Deallocate the devices at two levels:
   - At command level, when I/O command has been executed and the device has been temporarily released.
   - At process level, when process has terminated and the device has been permanently released.

Allocating Devices

The devices can be managed and allocated by an operating system in three possible ways:

- Dedicated
- Shared
- Virtual

Dedicated devices

An operating system can use a device in a dedicated manner by assigning it to only one process at a time, such device will remain assigned to only one process at a time and will serve that process till it terminates. There are certain devices that can be managed and used in this manner only. Examples of such devices are tape drives, plotters etc. These devices demand this kind of allocation scheme, as it is difficult for them to be shared.

The disadvantages of this scheme are:

1. It can be a very inefficient scheme especially when device is not used 100 percent of time. System may not always get full utilization of that device.
2. Another obvious disadvantage of this scheme is that the device cannot be shared among different user processes.

Shared devices

In this scheme, the operating system can assign a device in such a way that it can be shared among several processes. Examples of the devices that can be shared include printers, hard disks etc. Several processes can share them at same time by interleaved execution of their requests.

However, the device manager must carefully control this interleaving of process requests. In case, two requests from two different processes arrive at the same time, then such conflicts must be resolved with the help of predetermined policies which can decide that which request should be handled first.

This may be done partially by software (the I/O scheduler and traffic controller) or entirely by hardware (as in some computer with very sophisticated channels and control units). Policy for establishing which process' request is to be satisfied first might be based on a priority list or on the system output (for example, by choosing whichever request is nearest to the current position of the head of the disk).

The advantage of this scheme is that the device utilization reaches almost 100 percent if it is shared among several processes and thus kept busy almost at every instance.

Virtual devices

Some devices that would normally have to be dedicated (e.g. card readers, printers) may be converted into shared devices through techniques such as SPOOLing (Simultaneous Peripheral Operations On Line). For example, a SPOOLing program can read and copy all card input onto a disk at high speed. Later, when a process tries to read a card, the SPOOLing program intercepts the request and converts it to a read from the disk. Since several users may easily share a disk, we can easily convert a dedicated device to a shared device, changing one card reader into many "virtual" card readers. This technique is equally applicable to a large number of peripheral devices, such as teletypes, printers, and most dedicated slow input/output devices.

Another example is; Printers are converted to sharable devices through a SPOOLing program that re-routes all printing requests to the disk. The disk sends the output to the printer (for printing) only when all of the other requests that have been lined up before this request have been finished.

But to the user who has just now given a request for printing, it appears that this system has its own dedicated printer attached to it because the moment he gives

printing command, and the SPOOLing occurs and he can resume his work. Thus, this technique can convert one printer into several "virtual" printers, thus improving both its performance and use.

We further discuss the need for having virtual devices, the old solutions to meet the requirements and the new concept of virtual system.

Need for virtual devices

Devices such as card readers, punches, and printers present two serious problems that hamper effective device utilization.

1. Each of these devices performs best when there is a steady, continuous stream of requests at a specified rate that matches its physical characteristics (e.g., 2000 cards per minute read rate, 2000 lines per minute print rate, etc.). If a program attempts to generate requests faster than the device's performance rate, the program must wait for a significant amount of time. On the other hand, if a program generates requests at a much lower rate, the device is idle much of the time and is substantially underutilized.
2. These devices must be dedicated to a single program at a time. Thus, since most programs perform some input and output, we would require as many card readers and printers as we have programs being multiprogrammed. This would normally be uneconomical because most programs use only a fraction of the device's capacity. Many other devices, such as plotters, graphic displays, and typewriter terminals, have similar problems.

2.4.3 Basics

2.4.3.1 Interrupts

The events can be signaled by the occurrence of an interrupt or trap. Trap and Interrupt are closely related terms with a marginal difference. Interrupts serve a very important purpose - if there were no mechanism of interrupts, one would have to manually check in the main program whether the timers have over flown, whether we have received another character via the serial port, or if some external event had occurred. Besides making the main program ugly and hard to read, such a situation would make our program inefficient. Since we would be wasting precious "instruction cycles" checking up for events that usually do not happen.

2.4.3.2 Trap

Trap is actually a software-generated interrupt caused either by an error (for example division by zero, invalid memory access etc.), or by a specific request for an operating system service generated by a user program. Trap is sometimes called Exception.

The hardware or the software can generate these interrupts. When the interrupt or trap occurs, the hardware therefore, transfers control to the operating system which first preserves the current state of the system by saving the current

CPU registers contents and program counter's value. After this, the focus shifts to the determination of which type of interrupt has occurred.

For each type of interrupt, separate segments of code (called interrupt service routine) in the operating system determine what action should be taken and thus the system keeps on functioning by executing computational instructions, I/O instructions, storage instructions etc.

Modern operating systems are interrupt driven i.e. if there are no processes to execute, no I/O devices to service, no users whom to respond then operating system will be idle and wait for something to happen.

2.4.3.3 I/O Sub-system

Because I/O devices vary so widely in their functions and speed, a variety of methods are needed to control them. These methods form the I/O subsystem of the kernel, which separates the rest of the kernel from the complexity of managing I/O devices.

Look at the figure 1, which shows the components of an I/O subsystem.

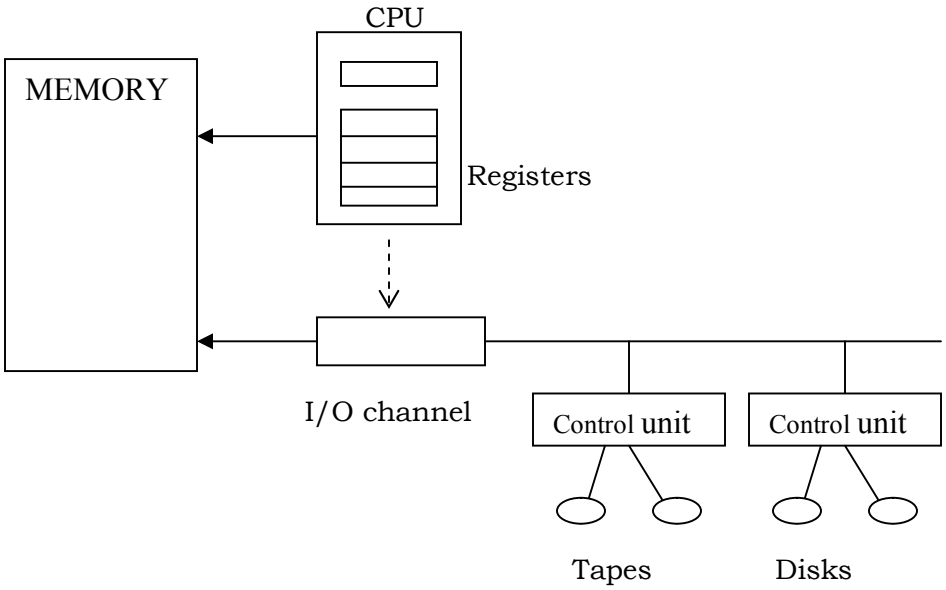


Figure 1: Components of the I/O Subsystem

The two basic components of this subsystem are:

- I/O channels
- Control units
- Device Controllers

I/O Channels

They are used to provide a path or a channel for the data to flow between I/O devices and the main memory. I/O channels use channel programs that specify action to be performed by devices and control the transmission of data between main

memory and control units. They are required to resolve the disparity in speed of the I/O devices and the CPU. They are in fact, specialized processing units intended to operate the I/O devices. Since these units may be simple, specialized and not too fast they are generally much less expensive than a conventional CPU. If all the input and output are performed via the channel, the CPU is free to perform its high-speed computations without wasting time on slow operations such as reading cards. It is also possible for the CPU to operate several channels simultaneously, therefore, many card readers, tape drives and printers can function at the same time. However, to obtain these benefits, the operating system may have to supervise these operations.

Types of I/O Channels

Basically I/O channels come in two types: Selectors and Multiplexers. In both cases, multiple devices can be connected to each channel.

Selector Channel - It can service only one of the devices at a time - i.e. one device is selected for service at one time. These channels are normally used for high-speed I/O devices such as magnetic disks and drums. Thus each I/O request is usually completed quickly and then another device is selected for I/O.

Multiplexer Channel - It is capable of servicing many devices, simultaneously. It is able to accomplish this only for slow I/O devices such as card readers, cardpunches and printers.

A third type of I/O channel which is a compromise solution that allows multiple channel programs for high-speed devices to be active on the same I/O channel, is Block Multiplexer Channel. The block multiplexer channel performs one channel instruction for one device and then, automatically, switches to perform an instruction for another device and so on. The block multiplexer channel resembles the selector as at any given time, it services only one device and it also resembles the multiplexer as it does not have to wait until the entire channel program is completed before it can service another device.

I/O Control Units

Due to the high cost of channels, there are fewer channels than devices. The channels, must, therefore, be switched from one device to another. This switching of devices can be done by an I/O control unit, which interprets signal, sent by channel for switching from one device to another.

It can be shared by many similar type of I/O devices (such as tapes, disks etc.) and can interpret the commands associated with these devices. As shown in figure 1, there is a control unit that manages devices like card readers, a control unit for tapes and a control unit for disks etc.

Device Controller: The electronic component of an I/O unit is called the device controller or adapter. On personal computers, it takes the form of a printed circuit card that can be inserted into an expansion slot.

2.4.4 I/O system organization

In modern operating systems, device management is implemented either through the interaction of *device driver(*They are distinct "black boxes" type system software that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works) and interrupt service routine(called interrupt driven I/O) or wholly within a device driver if interrupts are not used (called direct I/O with polling). Fig 2 summarizes the units involved in I/O operations for both approaches.

An application process uses a device by issuing commands and exchanging data with the device management (device driver).

Device driver two major responsibilities:

- Implement communication application programming interface(API) that abstract the functionality of the device
- Provide device dependent operations to implement functions defined by the API

• APIs should be similar across different device drivers, reducing the amount of information an application programmer needs to know to use the device i.e. the abstractions implemented by various device drivers should be as much alike as possible for all drivers.

• Since each device controller is specific to a particular device, the device driver implementation will be device specific, to
  - Provide correct commands to the controller
  - Interpret the controller status register (CSR) correctly
  - Transfer data to and from device controller data registers as required for correct device operation

  The first requirement of device management is to provide software that meets these two responsibilities.

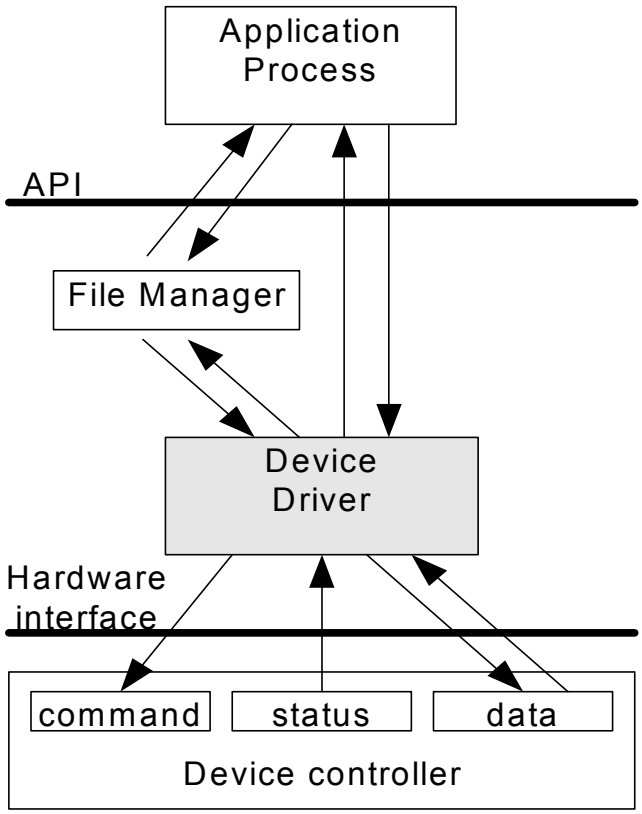


Fig 2: Device Management Organization

2.4.5 Direct I/O with polling

Direct I/O refers to the method of accomplishing I/O whereby the CPU is responsible for transferring the data between the machine's primary memory and the device controller data registers. While managing the I/O, the device manager may poll the device or use interrupts to detect the operation's completion. This section concentrates on the most basic technique; that is; the CPU starts the device and then polls the status register to determine when the operation has completed.

As shown in Fig 3, the steps to perform an input operation are as follows:

1. Application process requests a read operation
2. The device driver queries the CSR to determine whether the device is idle; if device is busy, the driver waits for it to become idle
3. The driver stores an input command into the controller's command register, thus starting the device
4. The driver repeatedly reads the content of CSR to detect the completion of the read operation
5. The driver copies the content of the controller's data register(s) into the main memory user's process's space.

Each I/O operation requires that the software and hardware coordinate their operations to accomplish the desired effect. In direct I/O polling, this coordination is accomplished by encapsulating the software part of the interactions with the device controller hardware wholly with the device driver.

Disadvantages

It is difficult to achieve high CPU utilization, since the CPU must constantly check the controller status. As a result, CPU cycles are used to repeatedly test the controller interface while the device is busy. These wasted CPU cycles can be utilized for some other process.

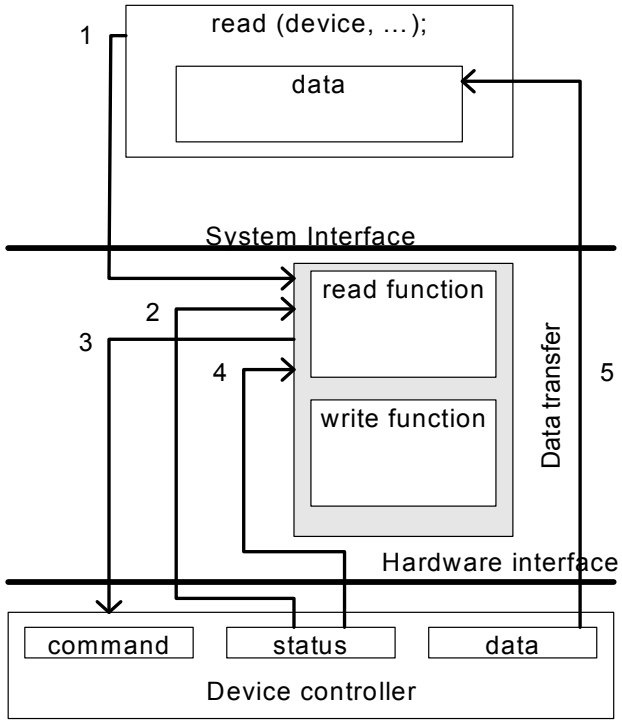


Fig 3: Polling I/O Read operation

As shown in Fig 4, the steps to perform an output operation are as follows:

1. The application process requests a write operation.
2. The device driver queries the CSR to determine if the device is idle; if busy, it will wait to become idle.
3. The device driver copies data from user space memory to the controller's data register(s).
4. The driver stores an output command into the command register, thus starting the device.
5. The driver repeatedly reads the CSR to determine when the device completed its operation.

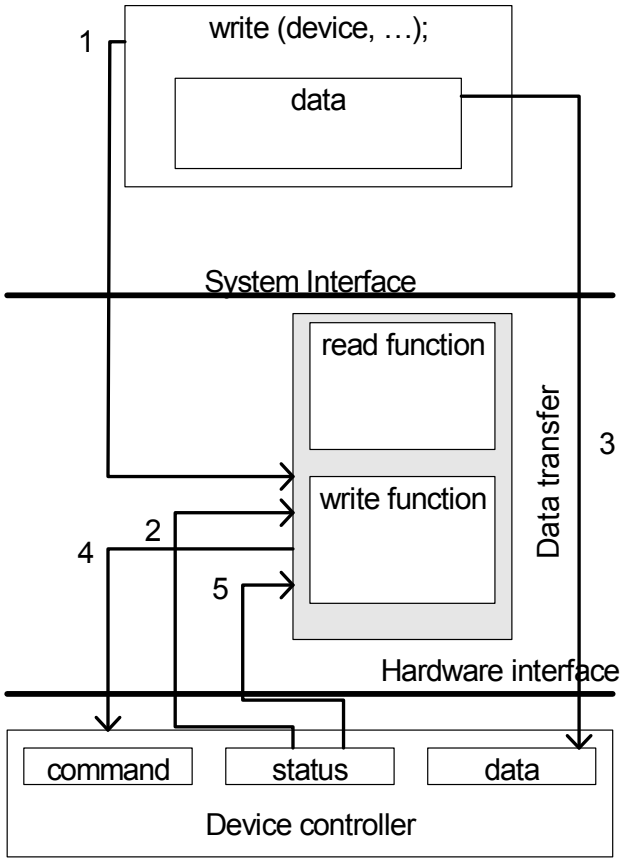


Fig 4: Polling I/O Write operation

Each I/O operation requires that the software and hardware coordinate their operations to accomplish the desired effect. In direct I/O polling, this coordination is accomplished by encapsulating the software part of the interactions with the device controller hardware wholly within the device driver.

However, with this approach it is generally difficult to achieve highly effective CPU utilization, since the CPU must constantly check the controller status. As a result, CPU cycles are used to repeatedly test the controller interface while the device is busy. In a multiprogrammed system, these wasted CPU cycles could be used by another process. Because the CPU is used by other processes in addition to the one waiting for the I/O to complete, multiprogramming may result in wrong detection of I/O completion. This can be remedied through the use of interrupts.

2.4.6 Interrupt driven I/O

In a multiprogramming system the wasted CPU time (in polled I/O) could be used by another process; because the CPU is used by other processes in addition to

the one waiting for the I/O operation completion, in multiprogramming system may result a sporadic detection of I/O completion; this may be remedied by use of interrupts.

The motivation for incorporating interrupts into the computer hardware is to eliminate the need for the device driver to constantly poll the controller status register.

Instead of polling, the device controller "automatically" notifies the device driver when the operation has completed.

As shown in Fig 9, the following are the steps for performing an input instruction in a system by using interrupts:-

1. The application process requests a read operation.

2. The device driver queries the CSR to find out if the device is idle; if busy, then it waits until the device becomes idle.

3. The driver stores an input command into the controller's command register, thus starting the device.

4. When this part of the device driver completes its work, it saves information regarding the operation it began in the device status table; this table contains an entry for each device in system.

9. The device completes the operation and interrupts the CPU, therefore causing an *interrupt handler* to run.

6. The interrupt handler determines which device caused the interrupt; it then branches to the *device handler* for that device.

7. The device driver retrieves the pending I/O status information from the device status table.

8a, 8b. The device driver copies the content of the controller's data register(s) into the user process's space.

9. The device handler returns the control to the application process (knowing the return address from the device status table).

The output operation behaves similarly. From the viewpoint of the application process, the activity has the same semantics as an ordinary procedure call. However, the time to execute the program considerably shorter than it is for a polling system, depending on the time ratio of computing, the I/O and the timeliness with which the software processes poll the device. This added delay in a polling system stems from the accumulation of delays between the time that the device finishes the operation and the time that the executing program observes this event and continues its normal execution.

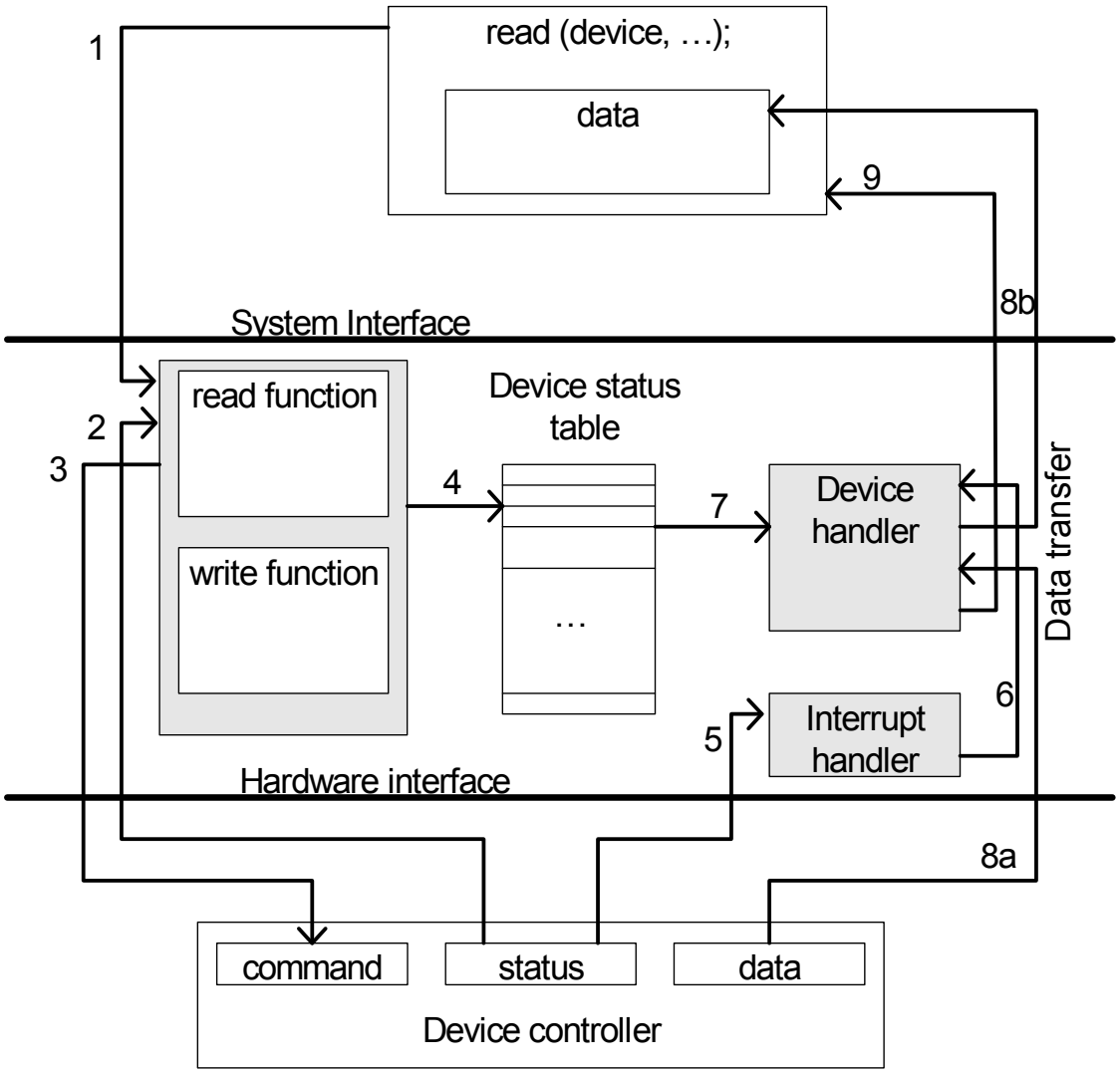


Fig 5: Interrupt-I/O operation

The software interface to an I/O device usually enables the operating system to execute alternative processes when any specific process is waiting for I/O to complete, while presenting serial execution semantics for an individual process. This means that the serial programs have simple I/O semantics that allow the programmer to treat read and write operations as sequential operations. When programmers use a read statement in a program, they know the read instruction has completed before the next instruction is executed.

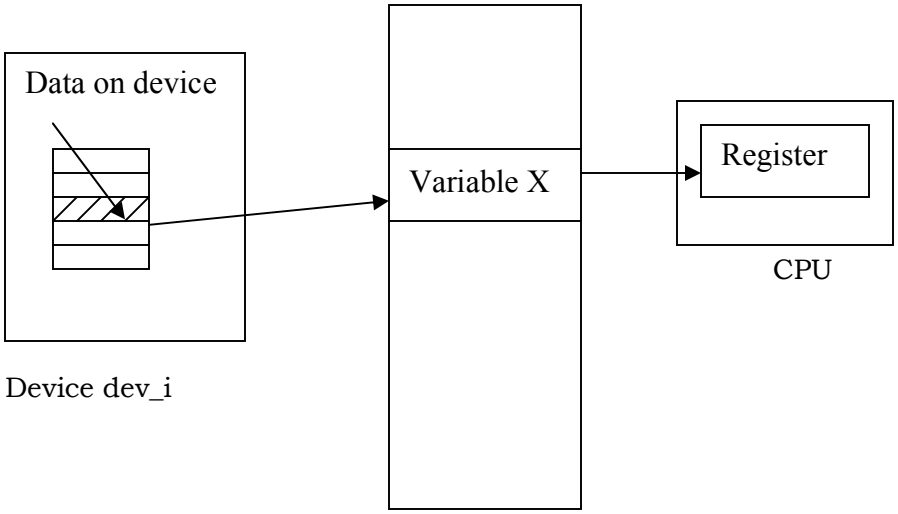
Consider the following code:

```
…
read (deviceX, “%d”, x);
```

```
y = f(x);
….
```

Fig 6 shows the situation after the read system call has started the deviceX device but before the operation has completed.

If the CPU was to execute the assignment (y = f(x)) before the I/O operation has completed, it will result in an error; thus, the operating system explicitly blocks the process that is executing the sample code, until the I/O read call is completed.



Sequential operation

```
…
read (deviceX, “%d”, x);
y = f(x);
….
```

Overlapped Operation

```
…
startRead(deviceX, “%d”, x);
…                                   …
While(stillReading( ));
y = f(x);
….
```

Fig 6: Overlapping the operation of a Device and the CPU

More complex semantics could allow the programmer to initiate the read operation –i.e. to start the device and continue processing. This would require that the programmer have some way to determine that the read operation had completed before referencing the variables that are receiving the results of that operation.

In the conventional approach, the device driver interface implements a standard procedure call mechanism for accomplishing I/O. The application program calls a procedure to request the I/O operation. When the procedure returns, the operation will have been completed. Operation of an individual process and its I/O is serialized.

Even though an individual process may not be able to take the advantage of the overlap of the CPU and the I/O operation, the operating system can switch the CPU to another process whenever one process invokes an I/O operation. Thus, the overall system performance can be improved due to the resulting overlap, although the individual process will still execute sequentially across the processor and I/O device. This requirement for serialization within a process means that the process management part of the operating system must become involved in I/O operations. This ensures that an I/O call will result in the calling process yielding control of the CPU to another application process. When the I/O completes, the original process can be rescheduled.

2.4.7 Interrupts versus polling

In general, the time to execute a process can be broken down to the time spent on computation $time_{compute}$, the time spent on I/O operations, $time_{device}$, and the time the process spends determining when each I/O operation is complete., $time_{overhead}$, as follows:-

$time_{total} = time_{compute} + time_{device} + time_{overhead}$

In an I/O device manager using polling, $time_{overhead}$ is the accumulated amount of time after a device completes an operation but before the polling loop has determined that the completion has occurred ($time_{polling}$). This is generally only a few instruction execution times.

In a system with interrupts, $time_{overhead}$, is calculated as follows:-

$time_{overhead} = time_{handler} + time_{ready}$

where $time_{handler}$ is the accumulated time required to execute the interrupt handler and device handler routines, and $time_{ready}$ is the accumulated time the process waits to use the CPU after it has completed its I/O but another process is using the CPU.

Polling is normally superior from the point of view of the individual process, since normally

$time_{polling} < time_{handler} + time_{ready}$

However, consider the effect of both the approaches on the overall performance of the system, rather than this turnaround time for one process. Suppose three processes are to be executed on a system. Process 1 requires $time_{total1}$ to complete, process 2 requires $time_{total2}$ to complete, and process 3 requires $time_{total3}$ to complete. In a polling system, process 1 might run to completion before process 2 is started and process 2 would run to completion before process 3 is started. The total time to execute all three processes with polling would be:

$time_{totalp} = time_{total1}+time_{total2}+time_{total3}$

In a system with interrupts, multiprogramming can make good use of the CPU by processes 2 and 3 when process 1 is conducting I/O. Ideally,

$time_{device1} <= time_{compute2}$

$time_{device2} <= time_{compute3}$

$time_{device3} <= time_{compute1}$

meaning that the total time to execute the three processes in a system with interrupts is

$time_{total1} = time_{compute1}+ time_{compute2} + time_{compute3} + time_{overhead}$

where the overhead is the sum of the overhead times of the individual processes. The average time to finish a process is $time_{total1}$ divided by 3. Hence, the average time to execute a process is much less with interrupts than it is with polling.

2.4.8 Keywords

Dedicated device: A device used in a dedicated manner by getting assigned to only one process at a time, and serving that process till it terminates.

Shared device: a device is assigned in a way that it can be shared among several processes

Virtual device: a dedicated device can be converted into shared device using the SPOOLing concept. The resulting device is known as virtual device.

Interrupts: the device alerts the CPU by activating one of the control lines known as interrupt-request line.

Trap: It is actually a software-generated interrupt caused either by an error (for example division by zero, invalid memory access etc.), or by a specific request for an operating system service generated by a user program.

I/O channels: They are used to provide a path or a channel for the data to flow between I/O devices and the main memory.

Control units: They interpret signal, sent by channel for switching from one device to another.

Interrupt-service routine: The routine which is executed in response to an interrupt request is called the interrupt-service routine.

Direct I/O: It requires the CPU software to explicitly transfer data to and from the controller's data registers.

Direct I/O with polling: The device management software polls the device controller status register to detect completion of the operation.

Interrupt driven I/O: Interrupts are used to simplify the software's responsibility for detecting operation completion.

Device driver: It is a part of the device manager used by an application processto call for an I/O operation.

Device Controller: The electronic component of an I/O unit is called the device controller or adapter. On personal computers, it takes the form of a printed circuit card that can be inserted into an expansion slot.

2.4.9 Summary

The device management functions are used to track the status of each device (such as tape drives, disk drives, printers, plotters and terminals).They are also used to describe the algorithms to decide which process will get a device and for how long. Its other two main concerns are allocation and de-allocation of the devices. It also provides the first level abstraction of the resources that will be used by applications to perform I/O. Direct I/O refers to the method of accomplishing I/O whereby the CPU is responsible for transferring the data between the machine's primary memory and the device controller data registers. While managing the I/O, the device manager may poll the device or use interrupts to detect the operation's completion. Polling keeps the CU extremely busy without giving it any chance of multiprogramming, whereas, the average time to execute a process is much less with interrupts than it is with polling.

2.4.10 Short Answer Type Questions

Q.1. "The average time to execute a process is much less with interrupts than it is with polling". Is the statement true of false? Justify your answer.

Q.2. List the major device management functions that must be performed by an operating system.

Q.3. Explain the functions of I/O channels and control units.

Q.4. What were the factors that led to the development of virtual devices? How did the concept of virtual devices evolve?

Q.9. List two problems that force some operating system to use the technique of device dedication.

Q.6. List three problems introduced by allowing users to share a device. What makes a disk more sharable than a tape?

Q.7. Compare the characteristics of selector and multiplexer channels.

Q.8. Define "trap".

Q.9. What is the name of the routine used to serve an interrupt?

Q.10. What is the function of a device controller?

2.4.11 Long Answer Type Questions

Q.1. Compare the "Direct I/O with polling" technique with "Interrupt driven I/O" technique. With the help of an example, explain the disadvantages of polling.

Q.2. With the help of a diagram, explain the steps involved in reading data from an input device using "direct I/O with polling" and "interrupt driven I/O".

Q.3. With the help of a diagram explain the device management organization.

Q.4. With the help of a diagram, explain the various components of I/O-subsystem.

2.4.12 Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts" , Khanna Publishing Co., 2002
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.

Web Resources :

www.personal.kent.edu/~rmuhamma/opsystems/os.html
www.wiley.com/college/silberschatz6e/0471417432/slides/slides.html

Device Management-2

Contents

2.5.0 Objectives
2.5.1 Memory Mapped I/O
2.5.2 Direct Memory Access
2.5.2.1 DMA Controllers
2.5.2.2 DMA applications
2.5.2.3 DMA implementations
2.5.2.4 Cache Coherency Problem
2.5.3 Buffering
2.5.3.1 Hardware Buffering
2.5.3.2 Double Buffering
2.5.3.3 Multiple Buffers
2.5.4 Device Drivers
2.5.4.1 Introduction
2.5.4.2 Built-in drivers
2.5.4.3 Usage of device drivers
2.5.4.4 Loading device drivers
2.5.4.5 Structure of a device driver
2.5.4.6 Classes of Devices and Modules
2.5.4.7 Device Manager Design
2.5.5 Disk scheduling algorithms
2.5.5.1 First in first out
2.5.5.2 Shortest Seek Time First (SSTF)
2.5.5.3 SCAN
2.5.5.4 C-SCAN
2.5.5.5 LOOK
2.5.5.6 C-LOOK
2.5.6 Keywords
2.5.7 Summary
2.5.8 Short Answer Type Questions
2.5.9 Long Answer Type Questions
2.5.10 Suggested Readings

2.5.0 Objectives:

The aim of this lesson is to describe other forms of I/O not discussed in the previous lesson. Memory-mapped I/O, direct memory access and concept of buffering which make the I/O faster and sophisticated have been discussed in detail. The most important system software used to run devices i.e. device driver is discussed. The general concept and the design and implementation aspects of drivers have been elaborated. The lesson ends with detailed discussion of disk scheduling algorithms which describe how the I/O requests pertaining to a disk are handled by an operating system.

2.5.1 Memory Mapped I/O

Memory-mapped I/O uses a section of memory for I/O. The idea is simple. Instead of having "real" memory (i.e., RAM) at that address, place an I/O device. Thus, communicating to an I/O device can be the same as reading and writing to memory addresses devoted to the I/O device. The I/O device merely has to use the same protocol to communicate with the CPU as memory uses.

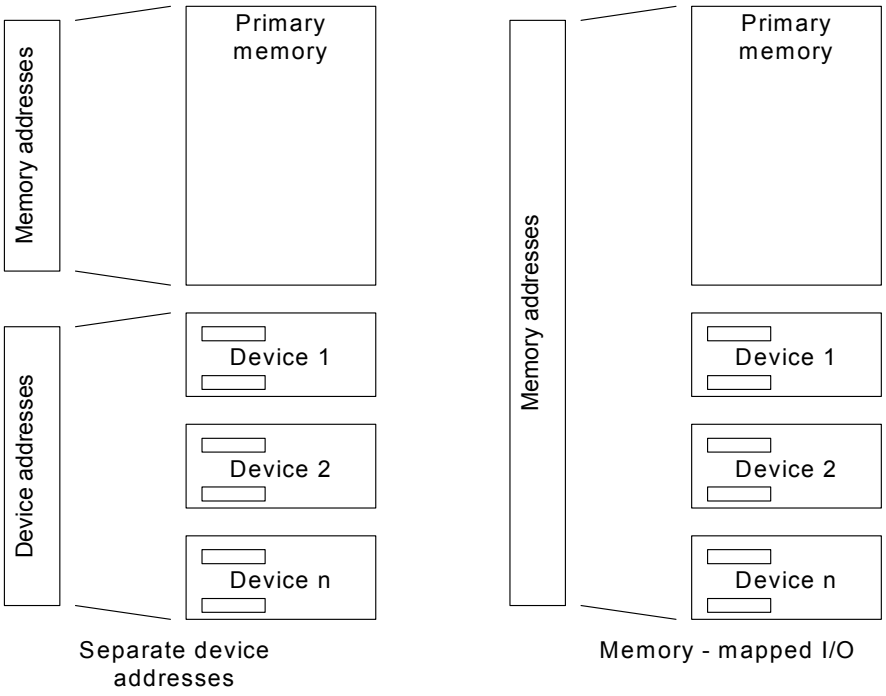


Fig 1(a) Separate Device Addresses (b) Memory-mapped I/O

An I/O device is managed by having software read/write information from/to the controller's registers. The computer designer must decide which instructions will be included in the machine repertoire to manipulate each controller's registers.

Traditionally, the machine instruction set includes special I/O instructions to accomplish this task. For example, to perform the I/O operations an instruction set might include instructions such as the following:
Each instruction refers to the device's address through a unique identifier.
input device_address
Causes the read command to be placed in command register of the given controller (at the specified address)
output device_address
Causes the write command to be placed in the command register of the given controller
copy_in CPU_register, device_address, controller_reg
copy_out CPU_register, device_address, controller_reg
Copy into or out of a CPU register from /to the designated data register of the I/O controller identified by device_address address.
test CPU_register, device_address
It copies the contents of the designated CSR to a CPU register.
Fig 7 describes the memory mapped approach, contrasted with the traditional approach.
With separate device addresses, as shown in Fig 1(a),each component in a device has a two component address such as *(i,j)*, where *i* is the device address and *j* is the address of the command, status, or data registers with device *i*.
For example, an assembly language statement such as:-

copy_in R3, 0x012, 4

This will cause the machine to copy the contents of data register 4 in the controller with address 0x012 into CPU register R3.

In memory mapped I/O approach shown in Fig 1(b), devices are associated with logical primary memory addresses rather than having a specialized device address. Each component of the device that is referenced using software is assigned a normal memory address. For example, device 0x0012 might have a block of addresses from 0xFFFF0120 to 0xFFFF012F to reference the device's command, status, and 14 data registers. A memory mapped I/O instruction to accomplish the same task as copy_in instruction might be:

Load R3, 0xFFFF0124

Memory mapped I/O reduces the number of instruction types in the processor. This happens because memory load/store instructions can be used to interact with the device's registers in a memory mapped I/O system.

2.5.2 Direct Memory Access

DMA is a method of transferring data from the computer's RAM to another part of the computer without processing it using the CPU. While most data that is input or output from our computer is processed by the CPU, some data does not

require processing, or can be processed by another device. In these situations, DMA can save processing time and is a more efficient way to move data from the computer's memory to other devices.

For example, a sound card may need to access data stored in the computer's RAM, but since it can process the data itself, it may use DMA to bypass the CPU. Video cards that support DMA can also access the system memory and process graphics without needing the CPU.

As shown in Fig 2(a), in Traditional I/O polling approach, CPU transfer data between the controller data registers and the primary memory and during the output operations, device driver copies data from the application process data area to the controller; vice versa for input operations. In the interrupt driven I/O approach, the interrupt handler is responsible for the transfer task.

As shown in Fig 2(b), in case of DMA, the I/O operation has to be initiated by the driver. DMA hardware enables the data transfer to be accomplished without using the CPU at all. The DMA controller must include an address register (and address generation hardware) loaded by the driver with a pointer to the relevant memory block; this pointer is used by the DMA hardware to locate the target block in primary memory.

2.5.2.1 DMA Controllers

The DMA controllers are special hardware - now embedded into the chip in modern integrated processors - that manage the data transfers and arbitrate access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles. Transfers are initiated when the DMA controller is notified of the need to move data to the memory by some event (keyboard press or mouse click, for examples). The controller asserts a DMA request signal to the CPU to use the system bus. The CPU completes its current operation and yields control of the bus to the DMA controller via a DMA acknowledge signal. The controller then reads and writes data and controls signals as if it is the CPU, which at that instant is tri-stated (idled). Upon completion of the transfer, DMA controller de-asserts the DMA request signal and the CPU in turn removes its DMA acknowledge signal and resumes control of the bus. DMA is implemented in computer bus architectures to speed up computer operations and allow multitasking. Normally, the CPU will be fully occupied in any read/write operation; enabling DMA allows reading/writing data in the internal memory, external memory and peripherals without CPU involvement, thus making the processor available for other tasks. This ensures streamlined operations, as movement of data to/from memory is one of the most common computer operations and freeing the CPU of this overhead can lead to a significant improvement in performance.

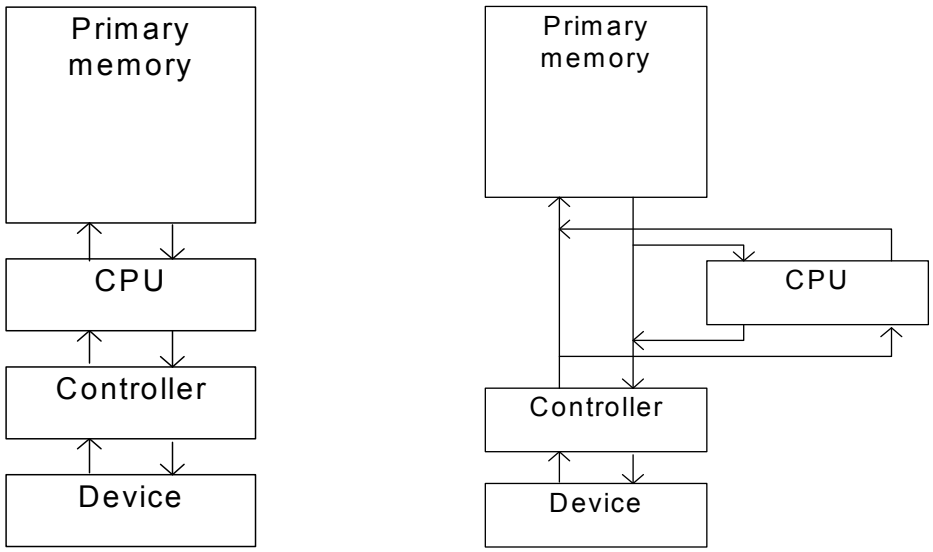


Fig 2(a): Traditional I/O (b) DMA

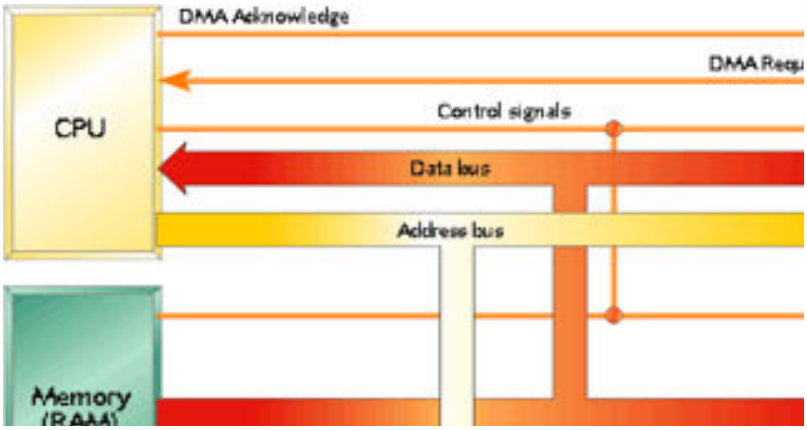


Fig 3: A DMA controller shares the processor's memory bus

2.5.2.2 DMA applications

DMA is useful in real-time computing applications where critical operations must be done concurrently. Stream processing is another application of DMA, where transfer and data processing are done simultaneously. Many hardware systems use DMA including floppy and disk drive controllers, graphics cards, network cards, sound cards and graphics processing units.

A DMA transfer essentially copies a block of memory from one device to another. While the CPU initiates the transfer, it does not execute it. For so-called "third party" DMA, as is normally used with the ISA bus, the transfer is performed by a DMA controller which is typically part of the motherboard chipset. More advanced

bus designs such as PCI typically use bus mastering DMA, where the device takes control of the bus and performs the transfer itself.

A typical usage of DMA is copying a block of memory from system RAM to or from a buffer on the device. Such an operation does not stall the processor, which as a result can be scheduled to perform other tasks. DMA transfers are essential to high performance embedded systems. It is also essential in providing so-called zero-copy implementations of peripheral device drivers as well as functionalities such as network packet routing, audio playback and streaming video.

2.5.2.3 DMA implementations

Synchronous DMA moves a byte or word at a time between system memory and a peripheral. After completing each transfer, the DMA asks the I/O port to signal when the latter is ready for another transaction. In this set-up, the DMA and the CPU shares the bus cycles, with the DMA winning any contest for system bus control.

Burst Mode DMA assumes that both the destination and source can take transfers as quickly as the controller can make them. The CPU sets up the controller, and after a signal from the I/O port, the entire data is copied to the destination. The DMA controller has sole access to the system bus during the transfer which is very rapid compared to synchronous DMA.

Flyby DMA, which is not supported by all controllers, puts out the source or destination address, then initiates a simultaneous read and write cycle. Flyby transfers are very fast as the read cycle and write cycle are compressed to a single cycle. Flyby can support both burst and synchronous types of transactions.

2.5.2.4 Cache coherency problem

DMA can lead to cache coherency problems. Imagine a CPU equipped with a cache and an external memory, which can be accessed directly by devices using DMA. When the CPU accesses location X in the memory, the current value will be stored in the cache. Subsequent operations on X will update the cached copy of X, but not the external memory version of X. If the cache is not flushed to the memory before the next time a device tries to access X, the device will receive a stale value of X. Similarly, if the cached copy of X is not invalidated when a device writes a new value to the memory, then the CPU will operate on a stale value of X.

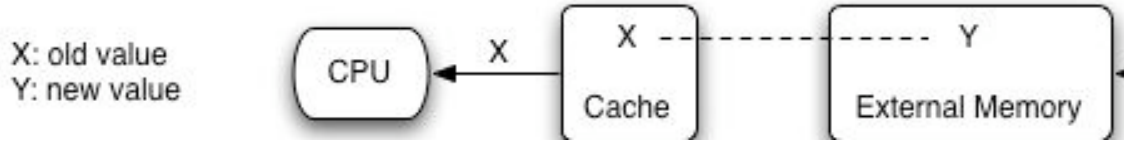


Fig 4: Cache Coherency Problem

2.5.3 Buffering

Buffering is the technique by which the device manager can keep slower I/O devices busy during times when a process is not requiring I/O operations.

Input buffering is the technique of having the input device read information into the primary memory before the process requests it.

Output buffering is the technique of saving information in memory and then writing it to the device while the process continues execution

2.5.3.1 Hardware Buffering

Consider a simple character device controller that reads a single byte form a modem for each input operation, as shown in Fig 5(a).

In normal operation, the read occurs and the driver passes a read command to the controller; the controller instructs the device to put the next character into one-byte data controller's register; the process calling for byte waits for the operation to complete and the retrieves the character from the data register.

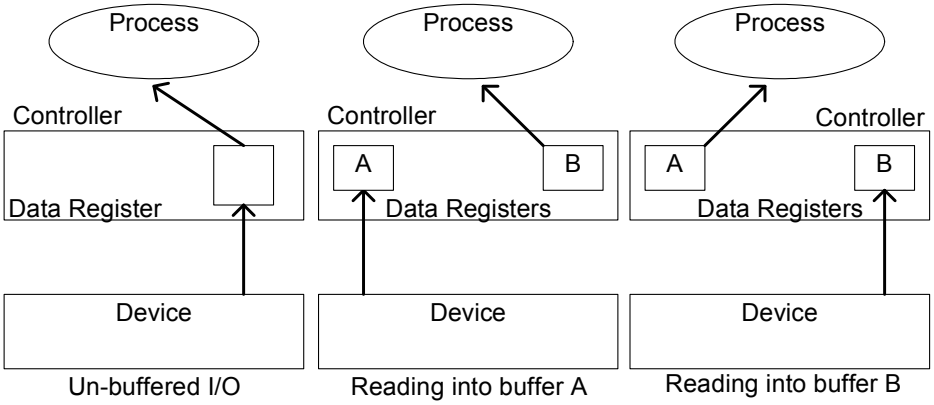


Fig 5(a): Unbuffered I/O (b) Reading into buffer A (c) Reading into buffer B

In Fig 5(b) and 5(c), we can see how a hardware buffer can be added to the controller to substantially decrease the amount of time the process has to wait for a character if the controller buffers it ahead of time. In Fig 5(b), the next character to be read by the process has already been placed into the data register B by the controller. The device is currently reading the next character from the device and placing it in data register A, even though the program has not yet called for the read operation. In Fig 5(c), the process requests the character previously read into controller data register A, so the device is started on a read operation to fill buffer B.

2.5.3.2 Double Buffering

Buffering or single buffering just explained, is affected by buffer underrun and buffer overflow. Double buffering is a better solution because it is possible to use two separate buffers in parallel, so that while B1 is read B2 can be written, and while B2 is read B1 can be written.

The easiest way to explain how double buffering works is to take a real world example. It is a nice sunny day and you have decided to get the paddling pool out, only you can't find your garden hose. You'll have to fill the pool with buckets. So you

fill one bucket (or buffer) from the tap, turn the tap off, walk over to the pool, pour the water in, walk back to the tap to repeat the exercise. This is analogous to single buffering. The tap has to be turned off while you "process" the bucket of water.

Now consider how you would do it if you had two buckets. You would fill the first bucket and then swap the second in under the running tap. You then have the length of time it takes for the second bucket to fill in order to empty the first into the paddling pool. When you return you can simply swap the buckets so that the first is now filling again, during which time you can empty the second into the pool. This can be repeated until the pool is full. It is clear to see that this technique will fill the pool far faster as there is much less time spent waiting, doing nothing, whilst buckets fill. This is analogous to double buffering. The tap can be on all the time and does not have to wait whilst the processing is done.

In computer science the situation of having a running tap that cannot be, or should not be, turned off is common (such as a stream of audio). It is also usual that computers prefer to deal with chunks of data rather than a stream. In such situations double buffering is often employed.

As shown in Fig 6(a and b), the hardware buffering technique can also be applied at the controller –driver level. This is generally called double buffering, since there are two buffers in the system.

One buffer is for the driver to store the data while waiting for the higher layers to read it. The other buffer is to store data from the lower level module. This technique can be used for the block-oriented devices (buffers must be large enough to accommodate a block of data). Example of such devices is magnetic tape drives

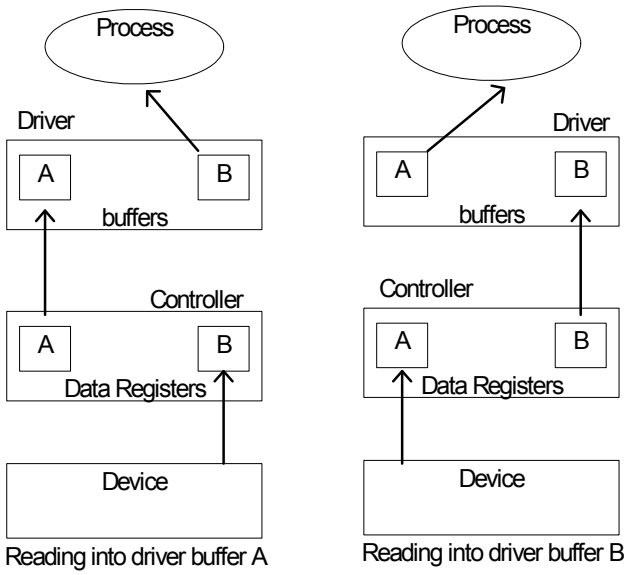


Fig 6(a) Reading into driver buffer A (b) Reading into driver buffer B

2.5.3.3 Multiple Buffers

As shown in Fig 7, the number of buffers is extended from two to n. The data producer (controller in read operations, CPU in write operations) is writing into buffer i while the data consumer (the CPU in read operations, the controller in write operations) is reading form buffer j. In this configuration, buffers j+1 to n-1 and 0 to i-1 are full. This is known as *circular buffering* technique. In this buffering technique, the producer cannot pass the consumer because it would overwrite buffers before they had been consumed. The producer can only fill up to buffer j-1 while data in buffer j is waiting to be consumed. Similarly, the consumer cannot pass the producer because it would be reading information before it was placed into the buffer by the producer.

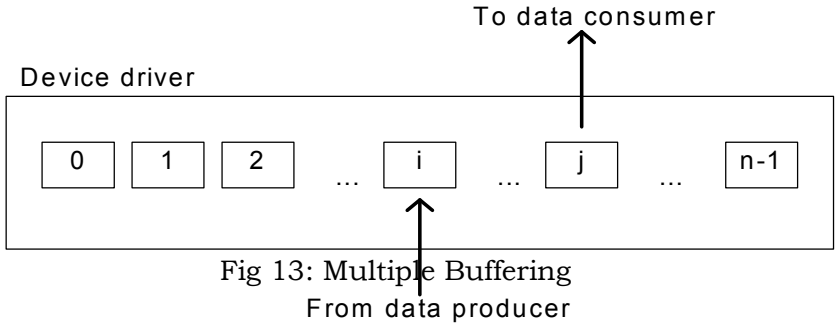


Fig 13: Multiple Buffering

10.4 Device Drivers

2.5.4.1Introduction

They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and "plugged in" at runtime when needed.

Device drivers are sets of procedures that are used to communicate with the hardware on the computer. The task performed by device drivers is in this respect similar to software interrupts. However, while each interrupt routine has a different calling strategy, all device drivers have a standard method of operation. This makes it possible to write programs that interface with arbitrary devices, either monitor, printer or disk drives.

Device drivers are not like normal programs since they cannot be run from the DOS command line. Instead they must be loaded in the CONFIG.SYS file at boot-up time using a device command.

Device drivers are, for all intents and purposes, a collection of routines in memory. Consider what happens when a file is opened from a high-level language. First the relevant sub-service of the DOS interrupt 21h is called to open the file. This then sends a sequence of commands to the relevant device driver. The device driver opens the file, and communicates with it using BIOS commands to access the disk. Thus the data requests get filtered from the program to DOS to the device driver to BIOS, which ultimately uses hardware commands to communicate directly with the hardware.

2.5.4.2 Built-in drivers

DOS has a number of built-in drivers to communicate with the console (keyboard+screen), printer, disk-drives, etc. The device drivers are given specific names.

Name of device	Name of driver
Console	CON:
Printer	LPT1: LPT2: LPT3:
Serial Ports	COM1: COM2: COM3: COM4:
Null device	NUL:
Disk drive A	A:
Disk drive B:	B:
Disk drive C:	C:

The console device gets input from the keyboard and sends output to the screen. The printer drivers send output only to the printer; the number of the printer port is specified in after LPT. The NULL device simply ignores all output to it and generates no input.

2.5.4.3 Usage of device drivers

When the programmer uses DOS interrupt calls, the relevant device drivers are automatically used. File commands can access just about any device but the more specific commands use particular devices. The screen output DOS commands explicity use the CON: device while the printer output DOS commands probably use the PRN: or LPT devices.

Since the use and installation of device drivers is so obscure, a little experiment may help to show their presence. A standard device driver called ANSI.SYS is shipped with most versions of DOS. This driver is a replacement for the standard console driver. It allows the user of DOS to move the cursor, clear the screen, and set colours (among other uses) using standard output commands. With this driver installed, multi-colour DOS prompts can be generated just by setting the PROMPT command appropriately. Particular sequences of characters sent to the

screen will be intercepted by the driver and interpreted to change the display characteristics. If the driver were not installed, these sequences will be printed on the screen.

2.5.4.4 Loading device drivers

Device drivers are loaded in memory in the order in which they are encountered in CONFIG.SYS file. DOS maintains a linked list of these drivers, with new drivers being added to the head of the list. When a device driver is accessed, the list is searched from the head. The latest driver is found first and is subsequently used.

2.5.4.5 Structure of a Device Driver

Device Header
Strategy procedure
Interrupt procedure
Command processing

The device header is a formatted table of information that the OS needs to set up and link in the device driver properly. The strategy and interrupt procedures are called by the OS. The rest of the driver is composed of routines that can be called within the driver. The device driver is not simply called to perform a particular task. Instead, a device Request Header is formed in memory. This is a fixed-format table of information specifying what the device driver is expected to do.

The address of the request header is then passed as a parameter, in the ES:BX register pair, to the strategy procedure. The strategy procedure stores this address within the body of the device driver and returns control to the operating system. The operating system then calls the interrupt procedure without any parameters. The interrupt procedure is expected to retrieve the address of the request header from where the strategy procedure had saved it. The request header specifies what is to be done and the interrupt procedure will carry out this task.

The reason for this delayed two-step process is that it allows for future expansion into multi-tasking environments with process scheduling and prioritizing.

2.5.4.6 Classes of Devices and Modules

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a *char module*, a *block module* or a *network module*. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendibility. The three classes are:

Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the *open*, *close*, *read*, and *write* system calls. The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of file system nodes, such as */dev/tty1* and */dev/lp0*.The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using *mmap* or *lseek*.

Block devices

Like char devices, block devices are accessed by file system nodes in the */dev* directory. A block device is a device (e.g., a disk) that can host a file system. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a file system node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an *interface* is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are usually, designed around the transmission and receipt of packets.

A network driver knows nothing about individual connections; it only handles packets. Not being a stream-oriented device, a network interface isn't easily mapped to a node in the file system, as */dev/tty1* is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the file system. Communication between the kernel and a network device driver is completely different from that used with char

and block drivers. Instead of *read* and *write*, the kernel calls functions related to packet transmission.

There are other ways of classifying driver modules that are orthogonal to the above device types. In general, some types of drivers work with additional layers of kernel support functions for a given type of device. For example, one can talk of universal serial bus (USB) modules, serial modules, SCSI modules, and so on. Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as a char device (a USB serial port, say), a block device (a USB memory card reader), or a network device (a USB Ethernet interface).Other classes of device drivers have been added to the kernel in recent times, including FireWire drivers and I2O drivers. In the same way that they handled USB and SCSI drivers, kernel developers collected class-wide features and exported them to driver implementers to avoid duplicating work and bugs, thus simplifying and strengthening the process of writing such drivers.

In addition to device drivers, other functionalities, both hardware and software, are modularized in the kernel. One common example is file systems. A file system type determines how information is organized on a block device in order to represent a tree of directories and files. Such an entity is not a device driver, in that there's no explicit device associated with the way the information is laid down; the file system type is instead a software driver, because it maps the low-level data structures to high-level data structures. It is the file system that determines how long a filename can be and what information about each file is stored in a directory entry. The file system module must implement the lowest level of the system calls that access directories and files, by mapping filenames and paths (as well as other information, such as access modes) to data structures stored in data blocks. Such an interface is completely independent of the actual data transfer to and from the disk (or other medium), which is accomplished by a block device driver. If you think of how strongly a Unix system depends on the underlying file system, you'll realize that such a software concept is vital to system operation. The ability to decode file system information stays at the lowest level of the kernel hierarchy and is of utmost importance; even if you write a block driver for your new CD-ROM, it is useless if you are not able to run *ls* or *cp* on the data it hosts. Linux supports the concept of a file system module, whose software interface declares the different operations that can be performed on a file system inode, directory, file, and super block. It's quite unusual for a programmer to actually need to write a file system module, because the official kernel already includes code for the most important file system types.

Storage devices implement persistent storage in a computer system, meaning that information is placed in a storage device can be preserved after the computer is turned off and until it is turned back on again. These devices can be further classified based on how they are accessed.

- Sequentially accessed storage devices
- Randomly accessed devices

Sequentially accessed storage devices

Sequentially accessed storage devices physically store the blocks on the recording medium in a linear sequence. Bytes may or may not be stored linearly within the block. The read/write interface to the device precludes programmers from ever really knowing exactly how bits and bytes are physically stored within a block. A read operation returns a block of bytes from the device, and a write operation copies a block of bytes to the device. The prominent example of such storage device is magnetic tape.

Randomly accessed devices

Such devices allow a driver to access the blocks on the device in an arbitrary order. This capability exacts a small, but measurable, performance penalty for accessing blocks stored at physically distant locations on the recording surface. Magnetic disks and optical disks are examples of randomly accessed devices.

2.5.4.7 Device Manager Design

Designing the device manager involves invoking controller specific I/O operations while satisfying three constraints:

- Create an API that implements the I/O functions available to the device, still compliant with interfaces implemented by other drivers; this is called device driver interface.
- Achieve correct coordination among the application processes, drivers and device controllers.
- Optimize the overall machine performance with correct driver strategies.

Device driver interface

Each operating system defines architecture for its device management system. These designs are different from operating system to operating system; thus there is no universal organization. Each operating system has two major interfaces to the device manager:

- The driver API
- The interface between a driver and the operating system kernel

Driver Application Programming Interface

It provides a set of functions that a programmer can call to manage a device. A device is generally used for communications or storage.

The device manager should do the following:

- Must track the state of the device: when it is idle, when is being used and by which process.
- Must maintain the information in the device status table.
- May maintain a device descriptor to store other information about the device.

- Must provide open/close functions to allow initiate/terminate of the device's use.
    - open – allocates the device and initializes the tables and the device for use
    - close – releases dynamic tables entries and releases the device
- *Must provide read/write* functions to allow the programmer to read/write from/to the device A consistent way to do these operations across all the devices is not possible; so a concession by dividing the devices into *classes* is made:
    - such as character devices or block devices
    - Sequential devices or randomly accessed devices
- Must provide *ioctl*(I/O control) function to allow programmers to implement device specific functions.

Driver Kernel Interface

The device driver must execute privileged instructions when it starts the device; this means that the device driver must be executed as part of the operating system rather than part of a user program. The driver must also be able to read/write info from/to the address spaces of different processes, since same device driver can be used by different processes.

There are two ways of dealing with the device drivers

- Old way: driver is part of the operating system, to add a new device driver, the whole Operating System must have been complied.
- Modern way: driver's installation is allowed without re-compilation of the operating System by using reconfigurable device drivers; the operating System dynamically binds the operating System code to the driver functions.

Reconfigurable device drivers

Modern operating systems simplify driver installation by using reconfigurable device drivers. Such a system allows one to add a device driver to the operating system without recompiling the operating system. This reconfiguration is accomplished by designing the operating system so that it dynamically binds the operating system code to the driver functions. For example, in Fig 14, an indirect table for device i points at driver modules for open, read, and other functions on the interface. A reconfigurable device driver has to have a fixed, standardized API. Because the device driver is added to the kernel after the kernel has been compiled, the kernel also provides an interface to provide an interface to allow the device driver to allocate/de-allocate space for buffers, manipulate tables in the kernel, etc.

The operating system uses an indirect reference table to access the different driver entry points, based on the device identifier and function name. That is, the kernel provides an API as a part of the operating system interface. When a process performs a system call, the kernel passes the call onto the device driver via the

indirect reference table. When the driver is installed, information for the indirect reference table is provided to the operating system so that it can read the information at runtime.

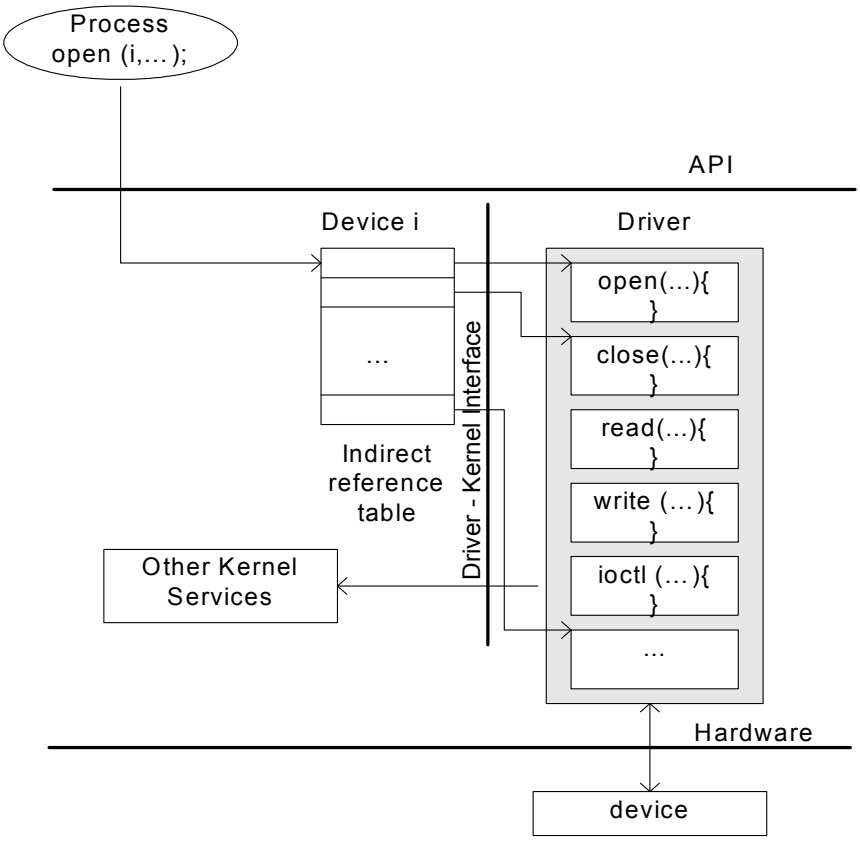


Fig 8: Reconfigurable Device Drivers

2.5.5 Disk Scheduling

In most systems, there are many processes that may be running simultaneously. Often, many processes request I/O operation from/to the hard disk. The algorithm used to select which I/O request is going to be satisfied first is called "disk scheduling algorithm". In order to be able to understand and evaluate these algorithms, one should be familiar with the way hard disk works. As we know the disk structure, a brief idea of working of disk will be sufficient here. It goes like this - for each I/O request, first, a head must be selected. This is done electronically and the time it takes is not significant. Then the head is moved over the destination track. After that, the disk is rotated to position the desired sector under the head. Finally, the I/O operation is performed. Arm movements and disk rotations (as discussed earlier) are the points where the delay occurs.

There are two objectives for any disk-scheduling algorithm:

1. Minimizing the response time i.e. the average time that a request must wait before it is satisfied.
2. Maximize the throughput i.e. the average number of requests satisfied per unit of time.

There are many disk-scheduling algorithms. The following have been discussed here:

First-in-first-out (FIFO), Shortest Seek Time First (SSTF), SCAN, Circular Scan (C-SCAN), LOOK, Circular Look (C-LOOK).

2.5.5.1 First-in-first-out (FIFO)

This is the simplest algorithm. It processes the I/O requests in the same order as they arrive. This technique improves the response time as a request gets response in fair amount of time. However, the throughput is not efficient. It involves a lot of random head movements and disk rotations. Figure 9(a) shows the cylinder request queue and Fig 9(b) shows the result of applying FIFO algorithm. This algorithm is used only in small systems where I/O efficiency is not very important.

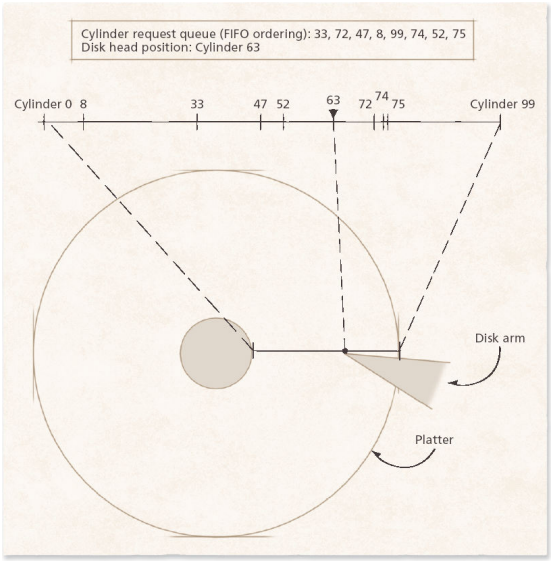


Figure 9(a): I/O request queue

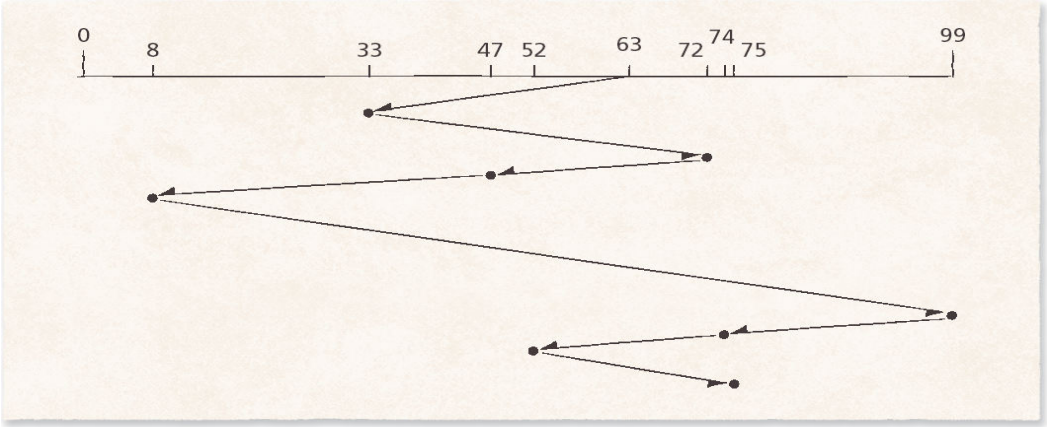


Fig 9 (b): FIFO algorithm

2.5.5.2 Shortest Seek Time First (SSTF)

After each request is received, the disk controller selects the request that needs to access a track/sector, which is closest to the current location of the head. The throughput in this case is much better than in FIFO. However, some process may have to wait for a long time until its request(s) are satisfied, if new requests with shorter seek time keep arriving. Thus, it may cause starvation of some requests. Figure 10 shows an example of this scheduling algorithm.

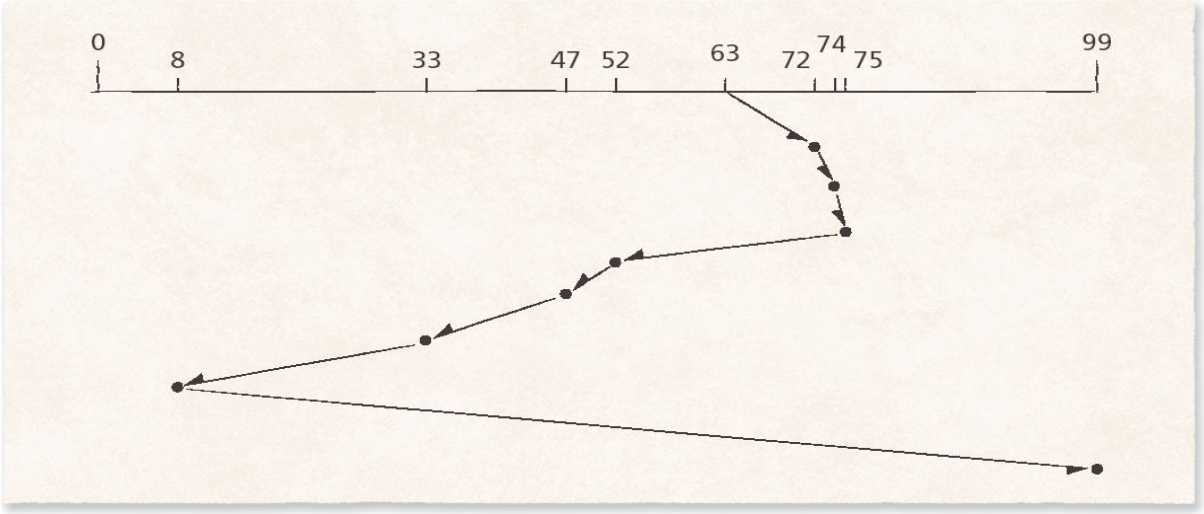


Figure 10: SSTF Algorithm

2.5.5.3 SCAN

In this algorithm, the head always constantly moves from the most inner cylinder to the outer cylinder, then it changes its direction back towards the center. As the head moves, if there is a request for the current disk position, it is satisfied.

The throughput is better than FIFO. The SCAN algorithm is fairer than SSTF as far as starvation of requests is considered. An example of this algorithm is shown in figure 11.

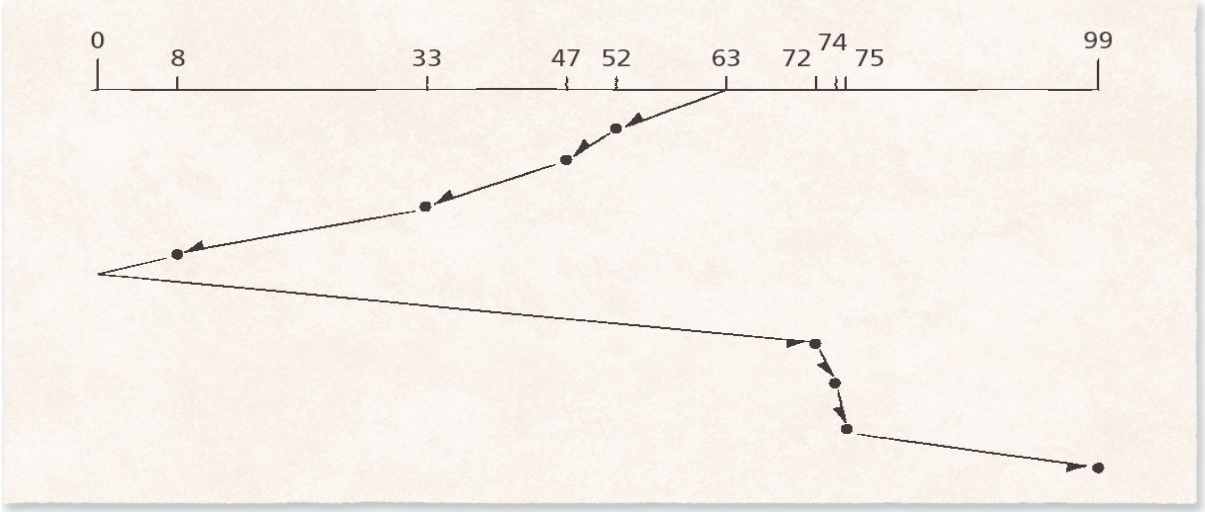


Figure 11: SCAN Algorithm

2.5.5.4 Circular Scan (C-SCAN)

This is an improvement over SCAN algorithm. In SCAN, the most outer and most inner cylinders have less opportunity to be accessed than the ones in the middle (as the middle is covered twice when the head moves from inner to outer and then outer to inner cylinders). C-SCAN eliminates this by satisfying requests only when the head moves in one direction and not satisfying any requests when it moves back. Figure 12 shows an example of this algorithm.

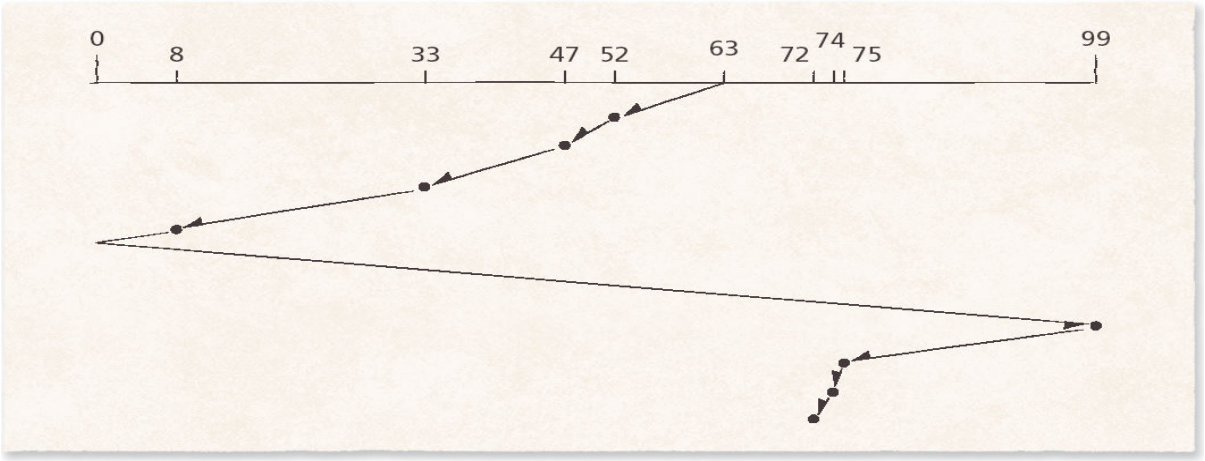


Figure 12: C-SCAN Algorithm

2.5.5.5 LOOK

As in SCAN, requests are served when the head moves in both directions. The difference is that LOOK uses information about the requests to change the direction of the head. When it knows that there are no requests beyond the current point, it changes the direction of the head. This improves both the throughput and the response time. Figure 13 shows an example of this algorithm.

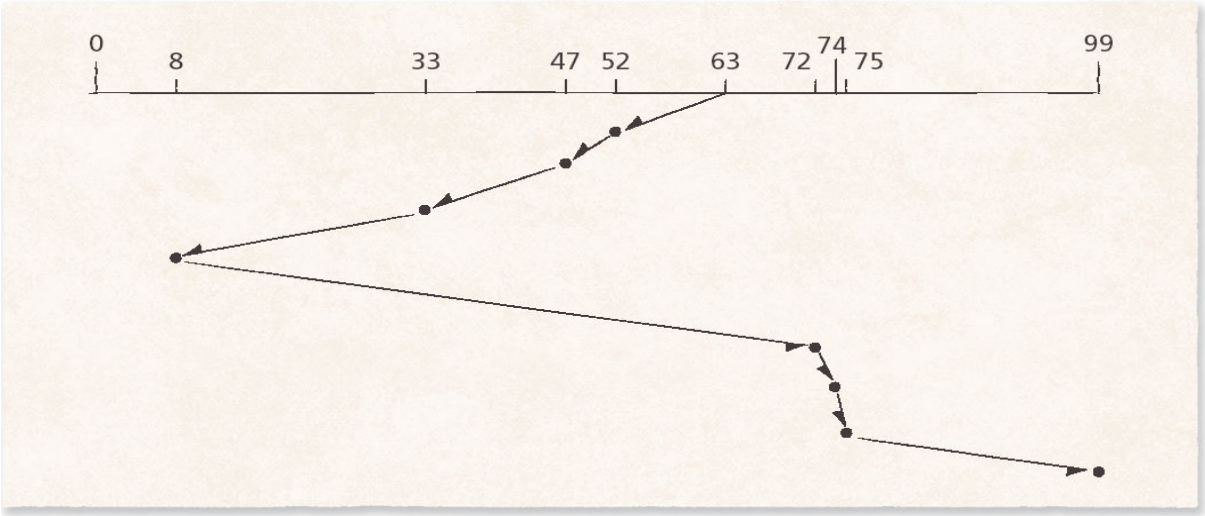


Figure 13: LOOK Algorithm

2.5.5.6 Circular Look (C- LOOK)

This is a variation of LOOK where requests are satisfied only when the head moves outwards, as in C-SCAN. Thus, no request is satisfied when the head moves inwards after determining that no requests are there beyond the current point.

In general, there is a trade off between the throughput and the response time. Some algorithms focus more on the throughput. Others give more importance to the response time. The type of the system and the type of I/O requests, in terms of size and frequency, would determine the most suitable algorithm. A good algorithm would find the golden point somewhere in between the throughput and the response time. For a complicated system, a combination of more than one algorithm may be necessary to provide fair efficiency for all users.

2.5.6 Keywords

DMA controllers- special hardware embedded into the chip in modern integrated processors - that manage the data transfers and arbitrate access to the system bus.

Direct Memory Access (DMA) - It is a method of transferring data from the computer's RAM to another part of the computer without processing it using the CPU.

Memory-mapped I/O- It uses a section of memory for I/O which makes communication to an I/O device same as reading and writing to memory addresses devoted to the I/O device.

Input buffering- It is the technique of having the input device read information into the primary memory before the process requests it.

Output buffering- It is the technique of saving information in memory and then writing it to the device while the process continues execution.

Double buffering-It uses two separate buffers in parallel.

Device Drivers-These are sets of procedures that are used to communicate with the hardware on the computer.

Disk scheduling algorithm- It is used to select which I/O request for a disk is going to be satisfied first.

2.5.7 Summary

Memory-mapped I/O uses a section of memory for I/O. Thus, communicating to an I/O device can be the same as reading and writing to memory addresses devoted to the I/O device. Buffering is a traditional technique used to increase performance through the overlap of execution. Double buffering and multiple buffering further improve the I/O performance by using more buffers for I/O. Device drivers are sets of procedures that are used to communicate with the hardware on the computer. The task performed by device drivers is in this respect similar to software interrupts. However, while each interrupt routine has a different calling strategy, all device drivers have a standard method of operation. For a character device, a driver usually implements at least the *open*, *close*, *read*, and *write* system calls. Block drivers have a completely different interface to the kernel than drivers for the character devices. Modern operating systems simplify driver installation by using reconfigurable device drivers. Disk scheduling algorithms are used to minimize response time and maximize throughput. These are based on first come first served basis or based on priority like distance from the current head position. The type of the system and the type of I/O requests, in terms of size and frequency, would determine the most suitable algorithm.

2.5.8 Short Answer Type Questions

Q.1. Which disk scheduling algorithms never lead to starvation?

Q.2. Define “device drivers”.

Q.3. What are reconfigurable device drivers?

Q.4. What is the difference between double buffering and multiple buffers?

Q.5. Compare the performance of SCAN and C-SCAN disk scheduling algorithms.

Q.6. What are the advantages of memory mapped I/O?

Q.7. List major applications of DMA.

Q.8. Compare the characteristics of block and character devices.

Q.9. Can the device drivers be run from DOS command line like any other program?

Q.10. Which parameters are optimized by disk scheduling algorithms?

2.5.9 Long Answer Type Questions

Q.1. Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order is: -
86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130.
Starting from the current head position, what is the total distance that the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithms?

a) FCFS b) SSTF
c) SCAN d) LOOK
e) C-SCAN

Q.2. Explain why SSTF scheduling tends to favour middle cylinder over the innermost and outermost cylinders.

Q.3. With the help of a diagram, explain the working of Direct Memory Access.

Q.4. Describe how DMA can lead to cache coherency problems.

2.5.10 Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts" , Khanna Publishing Co., 2002
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.

Web Resources :

www.personal.kent.edu/~rmuhamma/opsystems/os.html
www.wiley.com/college/silberschatz6e/0471417432/slides/slides.html

DEPARTMENT OF DISTANCE EDUCATION
PUNJABI UNIVERSITY, PATIALA

STUDENT'S RESPONSE-SHEET

Roll No.....
Class : BCA Part-III

Paper : BCA – 304
Operating Systems

ACADEMIC SESSION : 2018-2019

RESPONSE SHEET NO.1 and 2

Date of receipt of lesson.....

Marks obtained.....

Date of submission of Response-Sheet by the student....

Date & Signature of the Examiner
.....

No. of pages attached.....

Date of receipt in the Department
.....

Name & address of the student below in BLOCK LETTERS:
.....
.....
.....

Short answer type Questions Max Marks = 40

(Attempt any 5 from the following) 5 x 4 = 20 marks

1. List the contents of PCB.
2. What is CPU scheduling ?
3. What is compaction ? How is it implemented ?
4. Explain the term "virtual memory".
5. Explain the concept of process.
6. What is the difference between absolute and relative path name of a file ?
7. What is file management ?
8. Compare best-fit, worse-fit and first-fit allocation algorithms.

Long answer type Questions

(Attempt any 2 from the following) 2 x 10 = 20 marks

1. What do you mean by operating system ? What are the various services provided by the operating system ?
2. What is deadlock ? What are the necessary conditions for deadlock?
3. Compare paging and segmentation techniques. List their relative advantages and disadvantages.
4. List the advantages and disadvantage of page replacement algorithms.

Please send this Response-Sheet along with your answers to : The Deputy Registrar, Department of Distance Education, Punjabi University, Patiala-147002.