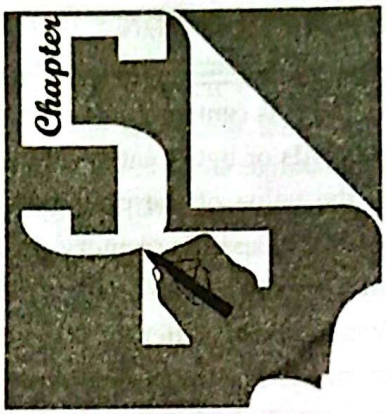


MEMORY MANAGEMENT

CHAPTER OUTLINE

After comprehensive study of this chapter, you will be able to:

- Introduction, Memory hierarchy, Logical versus Physical Address space,
- Memory management with Swapping: Memory Management with Bitmaps and with Linked List;
- Memory Management without Swapping,Contiguous-Memory Allocation: Memory protection, Memory Allocation, Fragmentation (Inter Fragmentation, External Fragmentation);
- Non-Contiguous Memory Allocation, Fixed Partitioning vs. Variable Partitioning,
- Relocation and Protection, Coalescing and Compaction.
- Background, Paging, Structure of Page Table: Hierarchical Page Table, Inverted Page Table, Shared page table
- Block mapping Vs. Direct Mapping, Demand Paging, Page Replacement and Page Faults, Page Replacement Algorithms: FIFO, OPR, LRU, SCP
- Some Numerical Examples on page Replacement, Trashing, Segmentation, Segmentation with paging.

INTRODUCTION

Memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated or what they are for. Accordingly, we can ignore how a memory address is generated by a program. We are interested in only the sequence of memory addresses generated by the running program.

MEMORY HIERARCHY

The memory in a computer can be divided into five hierarchies based on the speed as well as use. The processor can move from one level to another based on its requirements. The five hierarchies in the memory are registers, cache, main memory, magnetic discs, and magnetic tapes. The first three hierarchies are volatile memories which mean when there is no power, and then automatically they lose their stored data. Whereas the last two hierarchies are not volatile which means they store the data permanently. A memory element is the set of storage devices which stores the binary data in the type of bits. In general, the storage of memory can be classified into two categories such as volatile as well as non- volatile.

The memory hierarchy design in a computer system mainly includes different storage devices. Most of the computers were inbuilt with extra storage to run more powerfully beyond the main memory capacity. The following memory hierarchy diagram is a hierarchical pyramid for computer memory. The designing of the memory hierarchy is divided into two types such as primary (Internal) memory and secondary (External) memory.

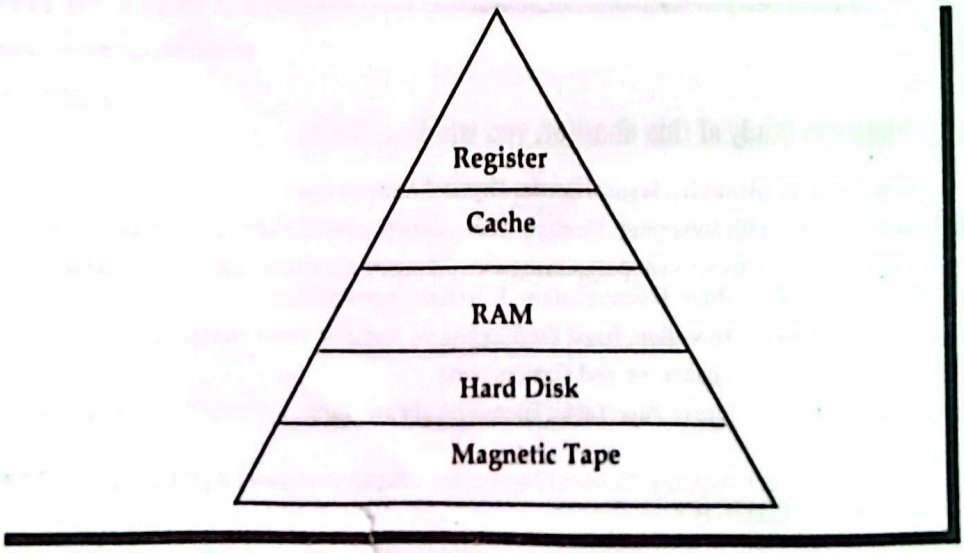


Fig 5.1: Memory Hierarchy

Registers

Usually, the register is a static RAM or SRAM in the processor of the computer which is used for holding the data word which is typically 64 or 128 bits. The program counter register is the most important as well as found in all the processors. Most of the processors use a status word register as well as an accumulator. A status word register is used for decision making, and the accumulator is used to store the data like mathematical operation. Usually, computers like complex instruction set computers have so many registers for accepting main memory, and RISC- reduced instruction set computers have more registers.

Cache Memory

Cache memory can also be found in the processor, however rarely it may be another IC (integrated circuit) which is separated into levels. The cache holds the chunk of data which are frequently used from main memory. When the processor has a single core then it will have two or more cache levels rarely. Present multi-core processors will be having three, 2-levels for each one core, and one level is shared.

Main Memory

The main memory in the computer is nothing but, the memory unit in the CPU that communicates directly. It is the main storage unit of the computer. This memory is fast as well as large memory used for storing the data throughout the operations of the computer. This memory is made up of RAM as well as ROM.

Magnetic Disks

The magnetic disks in the computer are circular plates fabricated of plastic otherwise metal by magnetized material. Frequently, two faces of the disk are utilized as well as many disks may be stacked on one spindle by read or write heads obtainable on every plane. All the disks in computer turn jointly at high speed. The tracks in the computer are nothing but bits which are stored within the magnetized plane in spots next to concentric circles. These are usually separated into sections which are named as sectors.

Magnetic Tape

This tape is a normal magnetic recording which is designed with a slender magnetizable covering on an extended, plastic film of the thin strip. This is mainly used to back up huge data. Whenever the computer requires accessing a strip, first it will mount to access the data. Once the data is allowed, then it will be unmounted. The access time of memory will be slower within magnetic strip as well as it will take a few minutes for accessing a strip.

Advantages of Memory Hierarchy

The need for a memory hierarchy includes the following.

- Memory distributing is simple and economical
- Removes external destruction
- Data can be spread all over
- Permits demand paging & pre-paging
- Swapping will be more proficient

LOGICAL VERSUS PHYSICAL ADDRESS SPACE

Logical Address

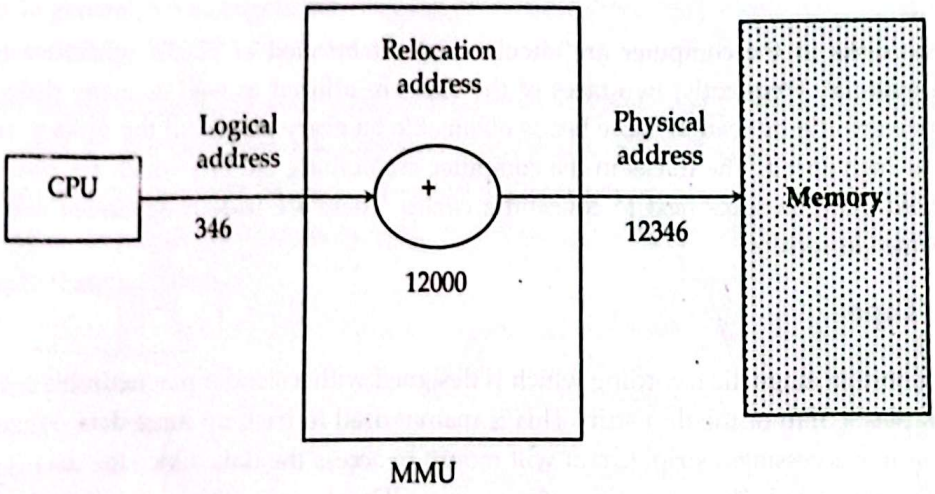
Address generated by CPU while a program is running is referred as Logical Address. The logical address is virtual as it does not exist physically. Hence, it is also called as Virtual Address. This address is used as a reference to access the physical memory location. The set of all logical addresses generated by a programs perspective is called Logical Address Space.

The logical address is mapped to its corresponding physical address by a hardware device called Memory-Management Unit. The address-binding methods used by MMU generates identical logical and physical address during compile time and load time. However, while run-time the address-binding methods generate different logical and physical address.

Physical Address

Physical Address identifies a physical location in a memory. MMU (Memory-Management Unit) computes the physical address for the corresponding logical address. MMU also uses logical address computing physical address. The user never deals with the physical address. Instead, the physical address is accessed by its corresponding logical address by the user.

The user program generates the logical address and thinks that the program is running in this logical address. But the program needs physical memory for its execution. Hence, the logical address must be mapped to the physical address before they are used.



Key Differences between Logical and Physical Address

- The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program. On the other hand, the physical address is a location that exists in the memory unit.
- The set of all logical addresses generated by CPU for a program is called Logical Address Space. However, the set of all physical address mapped to corresponding logical addresses is referred as Physical Address Space.

- The logical address is also called virtual address as the logical address does not exist physically in the memory unit. The physical address is a location in the memory unit that can be accessed physically.
- Identical logical address and physical address are generated by Compile-time and Load time address binding methods.
- The logical and physical address generated while run-time address binding method differs from each other.
- The logical address is generated by the CPU while program is running whereas; the physical address is computed by the MMU (Memory Management Unit).

MEMORY MANAGEMENT WITH SWAPPING

Swapping is mechanisms in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

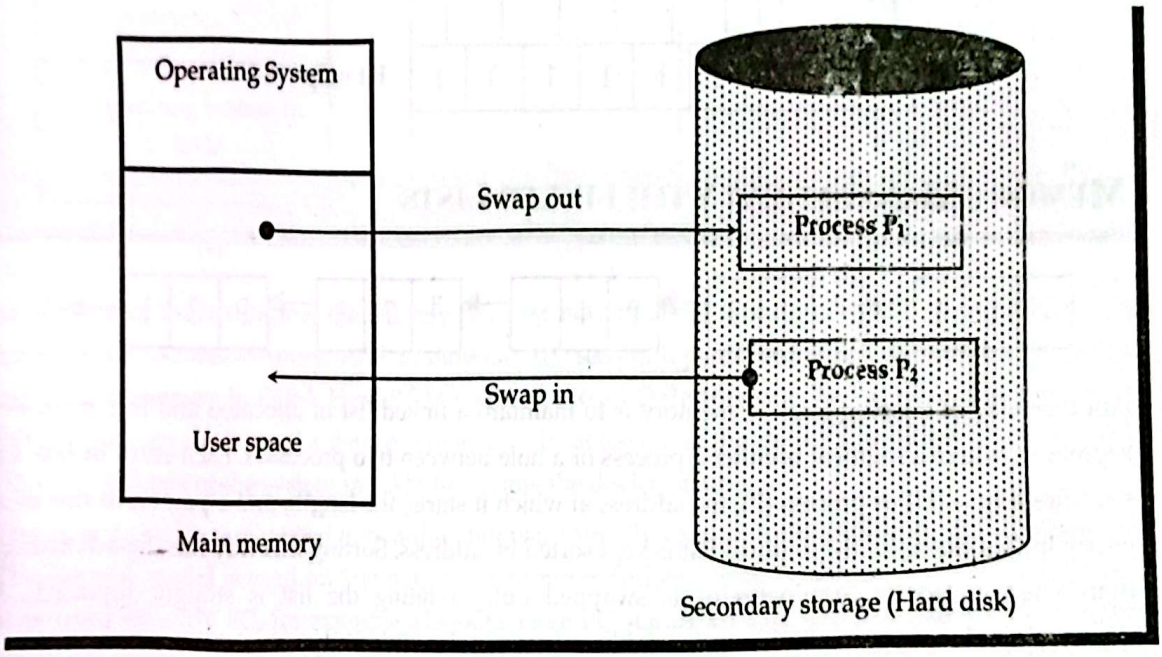


Fig 5.2: Swapping of two processes using a disk as a backing store

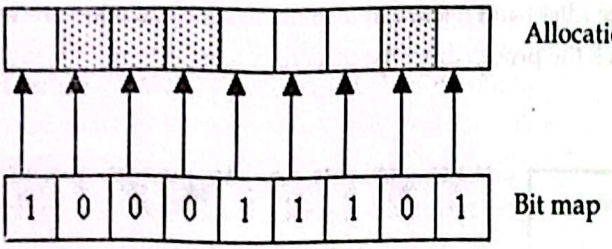
For improving the performance of the system we use the concept of swapping. In the swapping the processes those are on waiting state and those are on suspend or temporary suspend will be stored from outside the memory locations so that the speed of process will be high. In this the process those are waiting for some input and output are transferred to the physical memory from they are running and the processes those are ready for the execution will be execute by the CPU. When there is a situation to perform swapping, then we use the sweeper. The sweeper is used for:

- Selecting which process to be out
- Selecting which process to be in
- Providing the memory space to the processes those are newly entered.

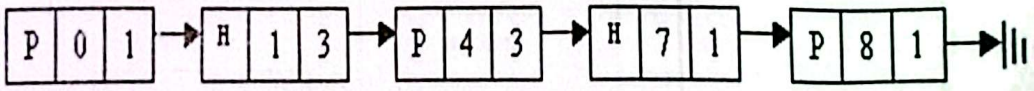
MEMORY MANAGEMENT WITH BITMAPS

Under this scheme the memory is divided into allocation units and each allocation unit has a corresponding bit in a bit map. If the bit is zero, the memory is free. If the bit in the bit map is one, then the memory is currently being used.

The main decision with this scheme is the size of the allocation unit. The smaller the allocation unit, the larger the bit map has to be. But, if we choose a larger allocation unit, we could waste memory as we may not use all the space allocated in each allocation unit. The other problem with a bit map memory scheme is when we need to allocate memory to a process. Assume the allocation size is 4 bytes. If a process requests 256 bytes of memory, we must search the bit map for 64 consecutive zeroes. This is a slow operation and for this reason bit maps are not often used.



MEMORY MANAGEMENT WITH LINKED LISTS



Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry. In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straight forward. A terminating process normally has two neighbors (except when it is at the very top or very bottom of memory).

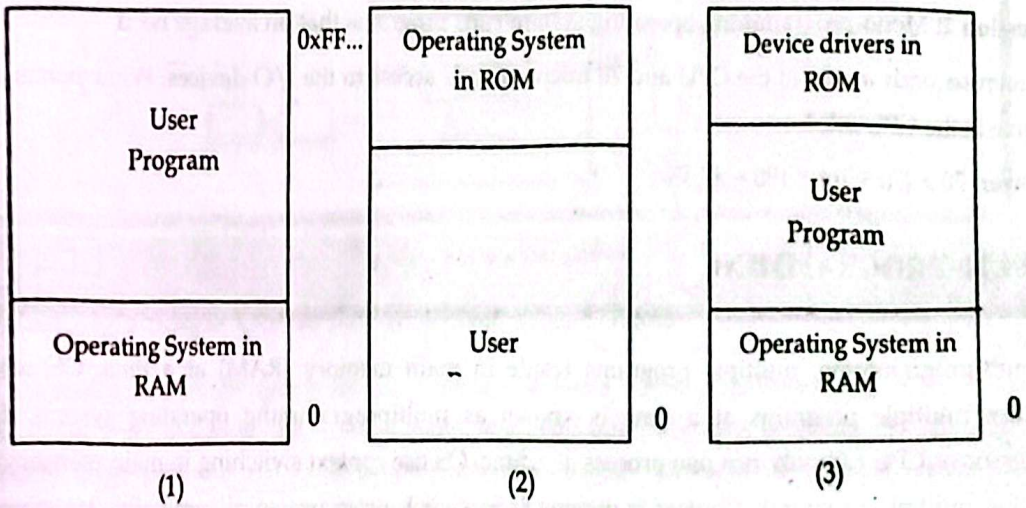
MEMORY MANAGEMENT WITHOUT SWAPPING

Memory Management is the process of controlling and coordinating computer memory, assigning portions known as blocks to various running programs to optimize the overall performance of the system. Here entire process remains in memory from start to finish and does not move. The sum of the memory requirements of all jobs in the system cannot exceed the size of physical memory.

MONO-PROGRAMMING

In mono-programming, memory contains only one program at any point of time. In case of mono-programming, when CPU is executing the program and I/O operation is encountered then the program goes to I/O devices, during that time CPU sits idle. Thus, in mono-programming CPU is not effectively used i.e. CPU utilization is poor. In mono-programming, most of the memory capacity is dedicated to a single program; only a small part is needed to hold the operating system. In this configuration, the whole program is in memory for execution. When the program finishes running, the program area is occupied by another program.

On this theme, there are three variations as shown in the figure given below:



As shown in the first figure, the OS may be at the bottom of the memory in RAM. Here RAM stands for Random Access Memory. And as shown in the second figure given above, the OS may be at the top of the memory in ROM. Here ROM stands for Read Only Memory. And as shown in the last or third figure given above, the device drivers may be at the top of the memory in a Read Only Memory (ROM), and rest of the system in RAM lies below the device drivers.

The first model is rarely used now-a-day, but was formally used on mainframes and minicomputers. The second model is used on few palmtop computers and embedded systems. And the third model was used by early PC, for example MS-DOS. Here PC stands for Personal Computer and MS-DOS stands for Microsoft-Disk Operating System. In that model, BIOS was the portion of the system in Read Only Memory (ROM). Here BIOS stands for Basic I/O System. Only one process can be running at a time when the system is organized in this way. As soon as the user types a command, the OS copies the requested program from disk to memory and then executes it. The OS displays a prompt character and waits for a new command when the process finishes. When it receives a command, then it loads a new program into memory, overwriting the first one.

Example of mono-programming

- Batch processing in old computers and mobiles
- The old operating system of computers
- Old mobile operating system

The characteristics of Uniprogramming are as follows

- Uniprogramming allows only one program to be present in memory at a time.
- The resources are provided to the single program that is present in the memory at that time.
- Since only one program is loaded the size is small as well.

Question 1: A computer has a mono-programming operating system. If the size of memory is 64 MB and the memory reserved part for the operating system is 4 MB, what is the maximum size of program that can be run by this computer?

Answer: 64 - 4 = 60 MB

Question 2: Mono-programming operating system runs programs that on average need

10 microseconds access to the CPU and 70 microseconds access to the I/O devices. What percentage of time is the CPU idle?

Answer: 70 / (70 + 10) × 100 = 87.5%

MULTI-PROGRAMMING

In multiprogramming, multiple programs reside in main memory (RAM) at a time. OS which handles multiple programs at a time is known as multiprogramming operating system. One processor or CPU can only run one process at a time. OS use context switching in main memory for running multiple programs. Context switching is to switch programs so all programs are given a suitable amount of time. OS can handle only a limited number of programs. If we run many programs on the computer or mobile then the computer becomes very slow or unresponsive.

Today, almost all computer systems allow multiple processes to run at the same time. When multiple processes running at the same time, means that when one process is blocked waiting for input/output to finish, another one can use the CPU. Therefore, multiprogramming increases the central processing unit utilization. Now-a-day, both network servers and client machines have the ability to run multiple processes at the same time. To achieve multiprogramming, the simplest way is just to divide memory up into n partitions (normally unequal partitions).It can be implemented in following two ways:

- Dedicate partitions for each process (Absolute translation)
- Maintaining a single queue (Re-locatable translation)

Dedicate Partitions for each Process (Absolute translation)

Here separate input queue is maintained for each partition. Processes are translated with absolute assemblers and compilers to run only in specific partition. The main problems of this process is that if a process is ready to run and its partition is occupied then that process has to wait even if other partitions are available. Wastage of storage.

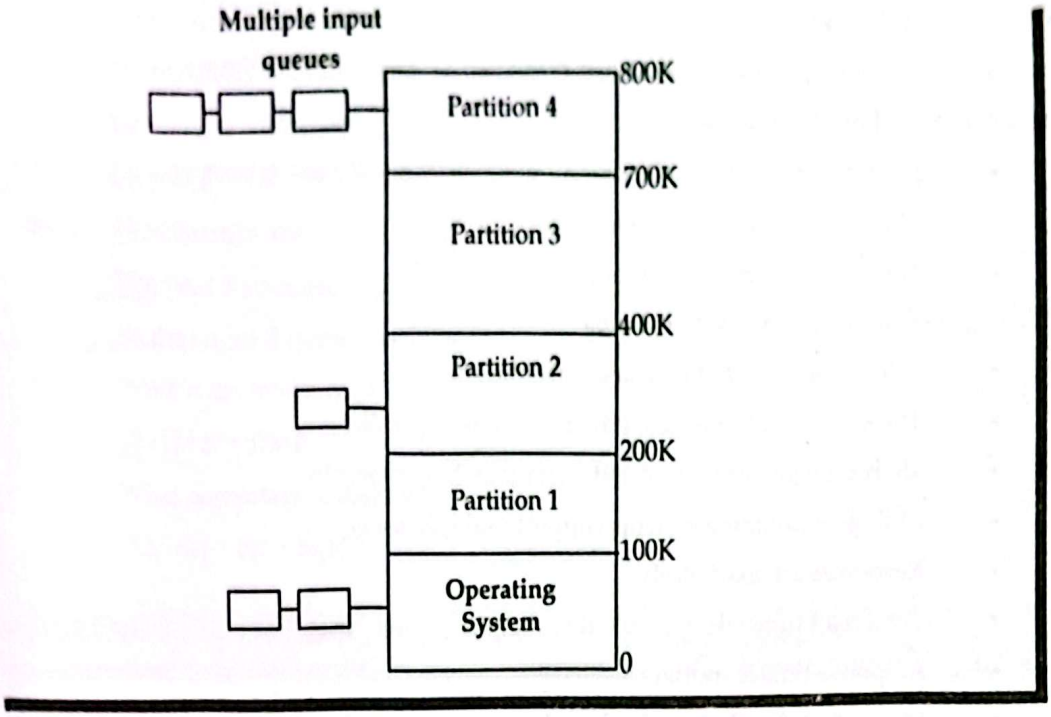


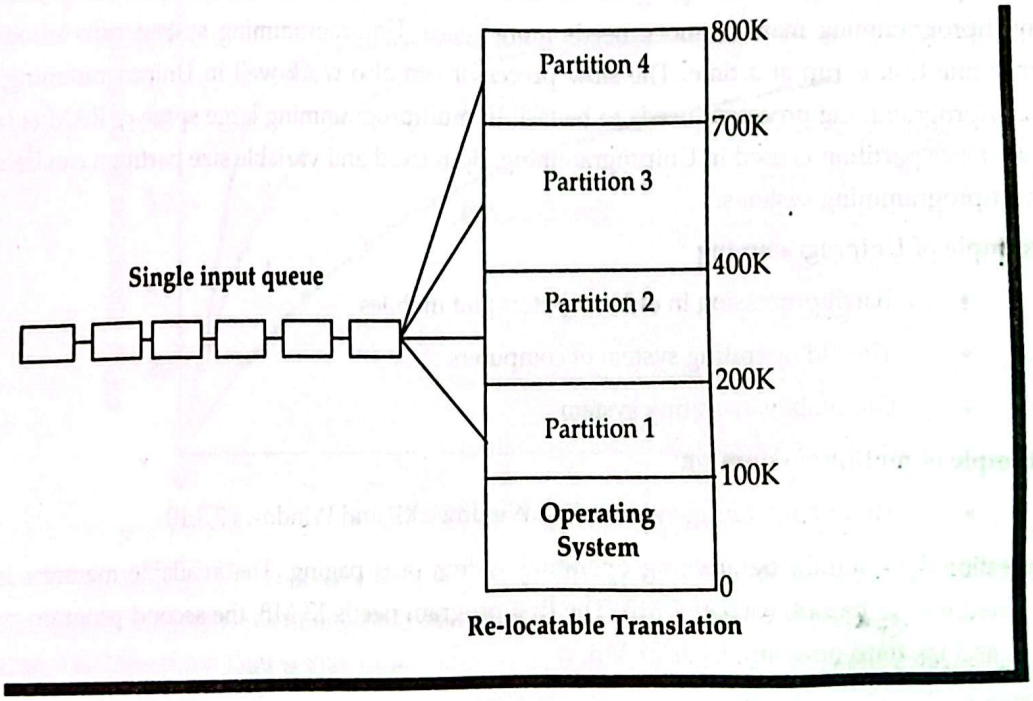
Fig 5.3: Absolute Translation

Maintaining a Single Queue (Re-locatable translation)

Here we maintain a single queue for all processes. When a partition becomes free, the processes closest to the front of queue that fits in it could be loaded into the empty partition and run. Not to waste the large partition for small process, another strategy to search the whole input queue whenever a partition becomes free and pick the largest process that fits.

Problems

- Eliminate the absolute problems but implementation is complex.
- Wastage of storage when many processes are small.



Example of multiprogramming

- Modern operating systems like Windows XP and Windows 7,8,10 etc.

Characteristics of multiprogramming

- Multiple programs can be present in the memory at a given time.
- The resources are dynamically allocated
- The size of the memory is larger comparatively

Advantages of multiprogramming systems

- CPU is used most of time and never become idle
- The system looks fast as all the tasks runs in parallel
- Short time jobs are completed faster than long time jobs
- Multiprogramming systems support multiply users
- Resources are used nicely
- Total read time taken to execute program/job decreases
- Response time is shorter
- In some applications multiple tasks are running and multiprogramming systems better handle these type of applications

Disadvantages of multiprogramming systems

- It is difficult to program a system because of complicated schedule handling
- Tracking all tasks/processes is sometimes difficult to handle
- Due to high load of tasks, long time jobs have to wait long

MONO-PROGRAMMING VS. MULTI-PROGRAMMING

In Uniprogramming only one program sits in main memory so it has a small size. But in the case of multiprogramming main memory needs more space. Uniprogramming system runs smoothly as only one task is run at a time. The slow processor can also work well in Uniprogramming but in multiprogramming processor needs to be fast. In multiprogramming large space of RAM is needed. Fixed size partition is used in Uniprogramming. Both fixed and variable size partition can be used in multiprogramming systems.

Example of Uniprogramming

- Batch processing in old computers and mobiles
- The old operating system of computers
- Old mobile operating system

Example of multiprogramming

- Modern operating systems like Windows XP and Windows 7,8,10

Question 1: A multiprogramming operating system uses paging. The available memory is60 MB divided into 15 frames, each of 4 MB. The first program needs 13 MB, the second program needs 12 MB, and the third program needs 27 MB.

a. How many frames are used by the first / second / third program?

P1 = 13/4= 3.4 = 4 frames

P2 = 12 / 4 = 3 frames

P3 = 27 / 4 = 6.75 = 7 frames

b. How many frames are un-used?

The used frames are 4 + 3 + 7 = 14 = 15- 14= 1

So the un-used frame is only one

c. What is the total memory used?

13+12+27= 52MB

d. What percentage of memory is used?

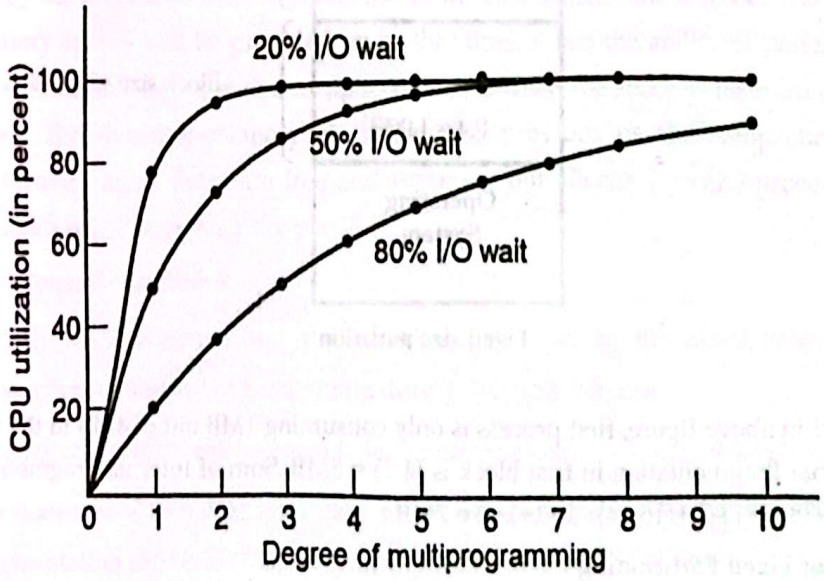
(52/ 60) × 100 = 86.67%

MODELING MULTIPROGRAMMING

If we have five processes that use the processor twenty percent (20%) of the time (spending eighty percent doing I/O) then we should be able to achieve 100% CPU utilization. Of course, in reality, this will not happen as there may be times when all five processes are waiting for I/O. However, it seems reasonable that we will achieve better than 20% utilization that we would achieve with mono-programming.

We can build a model from a probabilistic viewpoint. Assume that a process spend p% of its time waiting for I/O. With n processes in memory the probability that all n processes are waiting for I/O (meaning the CPU is idle) is p^n. The CPU utilization is then given by;

CPU utilization=1-p^n



We can see that with an I/O wait time of 20%, almost 100% CPU utilization can be achieved with four processes. If the I/O waits time is 90% then with 10 processes, we only achieve just above 60% utilization. The important point is that, as we introduce more processes the CPU utilization raises.

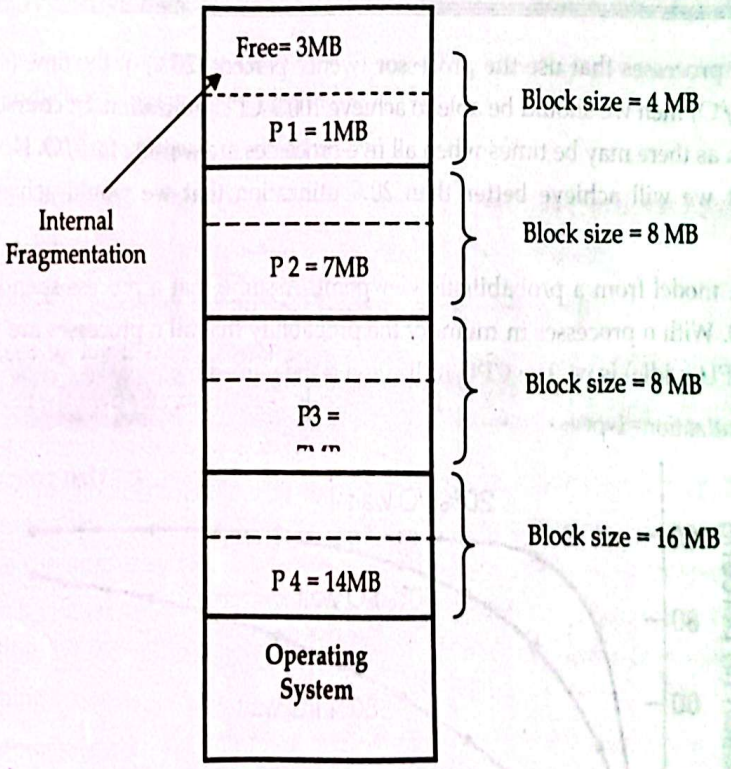
MULTIPROGRAMMING WITH FIXED AND VARIABLE PARTITIONS

There are two Memory Management Techniques: Contiguous, and Non-Contiguous. In Contiguous Technique, executing process must be loaded entirely in main-memory. Contiguous Technique can be divided into:

- Fixed (or static) partitioning
- Variable (or dynamic) partitioning

Fixed (or static) Partitioning

Dividing the main memory into a set of non-overlapping blocks is known as fixed partition. This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM is fixed but size of each partition may or may not be same. As it is contiguous allocation, hence no spanning is allowed. Here partition are made before execution or during system configure.



Fixed size partition

As illustrated in above figure, first process is only consuming 1MB out of 4MB in the main memory. Hence, Internal Fragmentation in first block is (4-1) = 3MB. Sum of Internal Fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB.

Advantages of Fixed Partitioning

- **Easy to implement:** Algorithms needed to implement Fixed Partitioning are easy to implement. It simply requires putting a process into certain partition without focusing on the emergence of Internal and External Fragmentation.

- **Little OS overhead:** Processing of Fixed Partitioning require lesser excess and indirect computational power.

Disadvantages of Fixed Partitioning

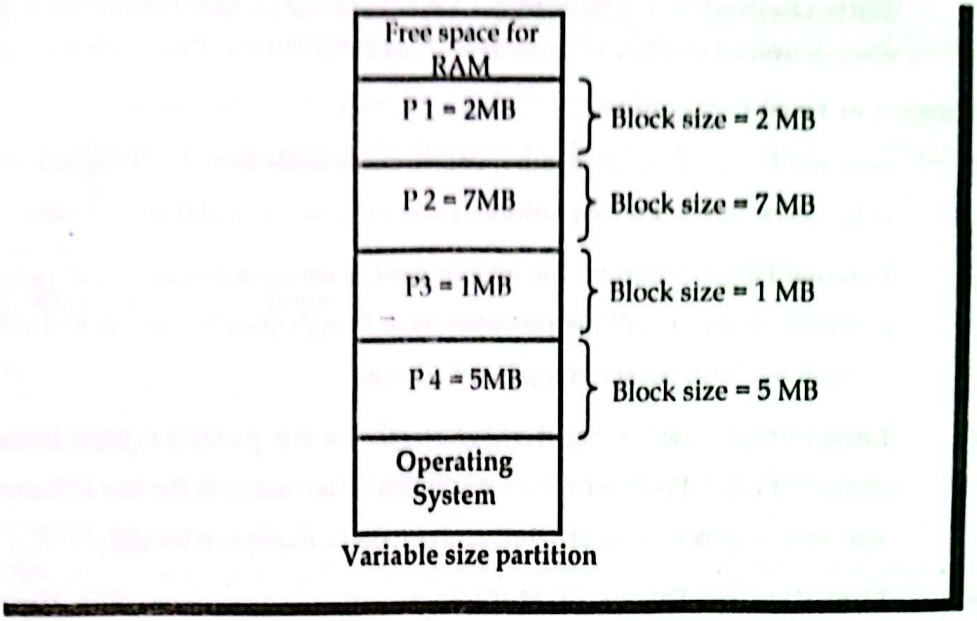
- **Internal Fragmentation:** Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.
- **External Fragmentation:** The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form (as spanning is not allowed).
- **Limit process size:** Process of size greater than size of partition in Main Memory cannot be accommodated. Partition size cannot be varied according to the size of incoming process's size. Hence, process size of 32MB in above stated example is invalid.
- **Limitation on Degree of Multiprogramming:** Partition in Main Memory is made before execution or during system configure. Main Memory is divided into fixed number of partition. Suppose if there are n_1 partitions in RAM and n_2 are the number of processes, then n2<= n1 condition must be fulfilled. Number of processes greater than number of partitions in RAM is invalid in Fixed Partitioning.

Variable (or dynamic) Partitioning

In this when an execution request of a process has to be made, then the memory is partition according to the size which is needed by the processes so that there will not be the internal and external fragmentation. In this when a process requested for some memory then the needed memory will be allocated by the process so that there will not be the case when some memory spaces will be the left. The memory spaces will be provided up to that time, when the ability of performing the number of programs doesn't comes to end or up to that time when the space of the hard disk never comes to an end. The dynamic memory allocation also provides us the compaction. In the compaction the memory areas those are free and those are not allocated by the process, will be combined and make a single large memory part.

Characteristics of dynamic partitions

- Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure.
- The size of partition will be equal to incoming process.
- The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilization of RAM.
- Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.



Variable size partition

Advantages of Variable Partitioning

- **No Internal Fragmentation:** In variable Partitioning, space in main memory is allocated strictly according to the need of process, hence there is any case of internal fragmentation. There will be no unused space left in the partition.
- **No restriction on degree of multiprogramming:** More number of processes can be accommodated due to absence of internal fragmentation. A process can be loaded until the memory is not empty.
- **No limitation on the size of the process:** In Fixed partitioning, the process with the size greater than the size of the largest partition could not be loaded and process cannot be divided as it is invalid in contiguous allocation technique. Here, in variable partitioning, the process size can't be restricted since the partition size is decided according to the process size.

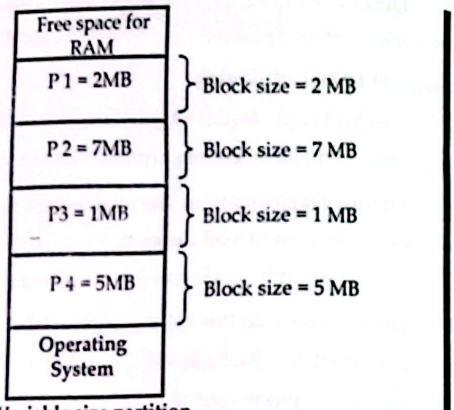
Disadvantages of Variable Partitioning

- **Difficult implementation:** Implementing variable Partitioning is difficult as compared to Fixed Partitioning as it involves allocation of memory during run-time rather than during system configure.
- **External fragmentation:** There will be external fragmentation in spite of absence of internal fragmentation.

RELOCATION AND PROTECTION

Relocation

Relocation is the process of assigning load addresses for position-dependent code and data of a program and adjusting the code and data to reflect the assigned addresses.Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program. Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization. Suppose 1st instruction of a program jumps at absolute address 200 within the exe file and it is loaded into partition 1 started at 100K. Jump should be performed at 100K+200K. This problem is called relocation problem.



Variable size partition

itioning

gmentation: In variable Partitioning, space in main memory is allocate
ng to the need of process, hence there is any case of interna
There will be no unused space left in the partition.

n degree of multiprogramming: More number of processes can b
lue to absence of internal fragmentation. A process can be loaded unt
t empty.

n the size of the process: In Fixed partitioning, the process with h
the size of the largest partition could not be loaded and process cann
t is invalid in contiguous allocation technique. Here, in variable
process size can't be restricted since the partition size is decide
rocess size.

titioning

entation: Implementing variable Partitioning is difficult as compare
ng as it involves allocation of memory during run-time rather than
figure.

tation: There will be external fragmentation in spite of absence o
tion.

TECTION

gning load addresses for position-dependent code and data of
and data to reflect the assigned addresses.Programmers typically
r programs will be resident in main memory at the time of execution
ses need to be able to be swapped in and out of main memory
ilization. Suppose 1st instruction of a program jumps at absolute
and it is loaded into partition 1 started at 100K. Jump should
roblem is called relocation problem.

it can be minimized by using relocation as the program is loaded into memory, modify the instructions accordingly. Address locations are added to the base location of the partition to map to the physical address. There are two types of relocations:

- **Static Relocation:** Program must be relocated before or during loading of process into memory. Program must always be loaded into same address space in memory, or relocator must be run again.
- **Dynamic Relocation:** Process can be freely moved around in memory. Virtual-to-physical address space mapping is done at run-time.

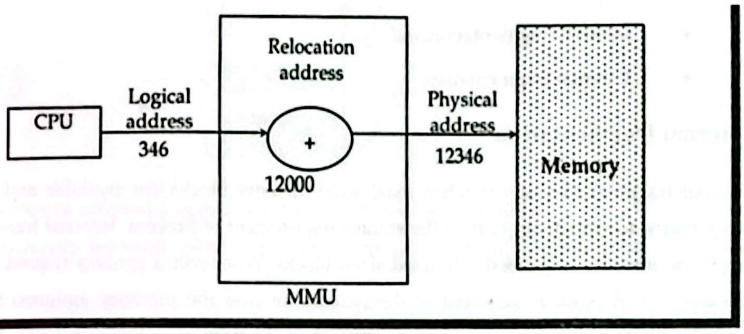


Fig 5.4: Memory relocation

Protection

Memory protection is a way to control memory access rights on a computer, and is a part of most modern instruction set architectures and operating systems. The main purpose of memory protection is to prevent a process from accessing memory that has not been allocated to it. This prevents a bug or malware within a process from affecting other processes, or the operating system itself. An attempt to access unwonted memory results in a hardware fault, called a segmentation fault or storage violation exception, generally causing abnormal termination of the offending process. Memory protection for computer security includes additional technique such as address space layout randomization and executable space protection.

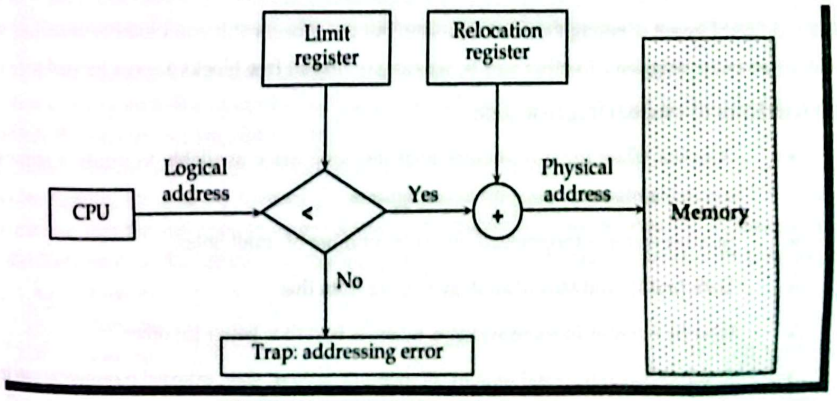


Fig 5.5: Memory protection and relocation

FRAGMENTATION

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation. Fragmentation is a condition that occurs when we dynamically allocate the RAM to the processes, then many free memory blocks are available but they are not enough to load the process on RAM. There are two types of fragmentations:

- Internal fragmentation and
- External fragmentation

Internal fragmentation

Internal fragmentation occurs when fixed sized memory blocks are available and a process gets a block that is too much larger than the storage requirement of process. Internal fragmentation occurs when the memory is divided into fixed sized blocks. Whenever a process request for the memory, the fixed sized block is allocated to the process. In case the memory assigned to the process is somewhat larger than the memory requested, then the difference between assigned and requested memory is the internal fragmentation.

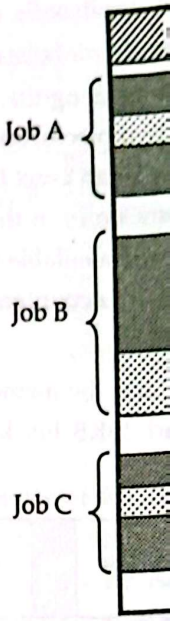
This leftover space inside the fixed sized block cannot be allocated to any process as it would not be sufficient to satisfy the request of memory by the process. Let us understand internal fragmentation with the help of an example. The memory space is partitioned into the fixed-sized blocks of 18,464 bytes. Let us say a process request for 18,460 bytes and partitioned fixed-sized block of 18,464 bytes is allocated to the process. The result is 4 bytes of 18,464 bytes remained empty which is the internal fragmentation.

External Fragmentation

Total free RAM space is enough to load a process but the process still can't load because free blocks of RAM are not contiguous. In other words, we can say that all free blocks are not located together.

Characteristics of external fragmentation

- It exists when there is enough total memory space available to satisfy a request, but available memory space is not contiguous.
- Storage space is fragmented into large number of small holes.
- Both first fit and best fit strategies suffer from this.
- First fit is better in some systems, whereas best fit is better for other.
- Depending on the total amount of memory storage, size, external fragmentation may be minor or major problem.



Comparison between internal fragmenta

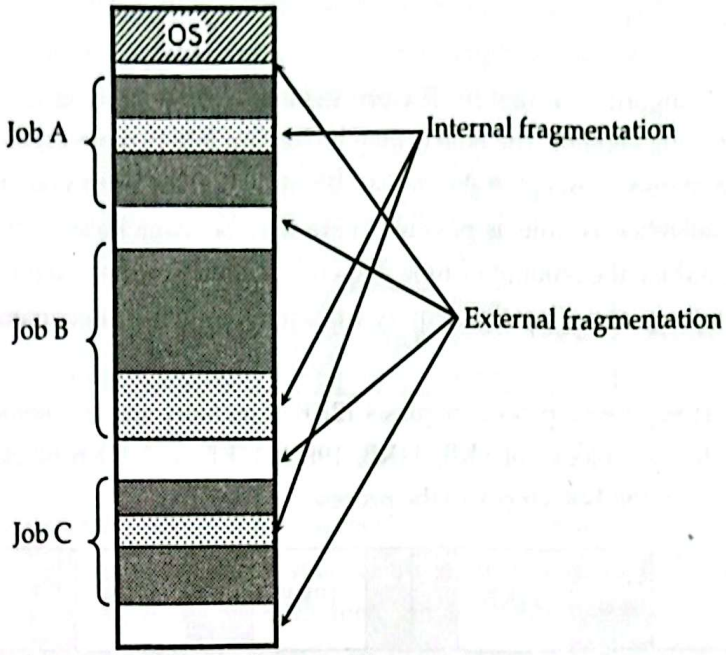
The major differences between internal below:

Internal fragmentation
occurs when fixed sized memory allocated to the processes.
When the memory assigned to the slightly larger than the memory reque process this creates free space in the block causing internal fragmentation.
The memory must be partitioned in sized blocks and assign the best fit b process.

MEMORY ALLOCATION STRA

Memory allocation is the process of assi receives memory from the operating sys to satisfy the requests for smaller blocks. When the processes and holes are kept allocate memory for a newly created p We assume that the memory manager methods are used to allocate memory fo

- First Fit
- Next Fit
- Best Fit
- Worst Fit
- Quick Fit



Comparison between internal fragmentation and external fragmentation

The major differences between internal fragmentation and external fragmentation are tabulated below:

Internal fragmentation	External fragmentation
It occurs when fixed sized memory blocks are allocated to the processes.	It occurs when variable size memory spaces are allocated to the processes dynamically.
When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.

MEMORY ALLOCATION STRATEGIES

Memory allocation is the process of assigning blocks of memory on request. Typically the allocator receives memory from the operating system in a small number of large blocks that it must divide up to satisfy the requests for smaller blocks.

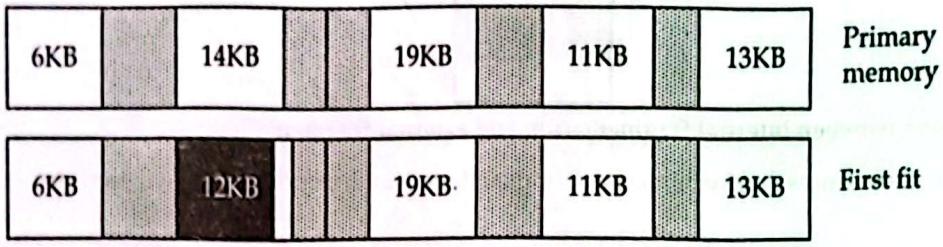
When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate. Following popular methods are used to allocate memory for a process in the memory management with linked list:

- First Fit
- Next Fit
- Best Fit
- Worst Fit
- Quick Fit

First Fit

The simplest algorithm is first fit. The process manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request.

For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The first-fit strategy will allocate 12KB of the 14KB block to the process.

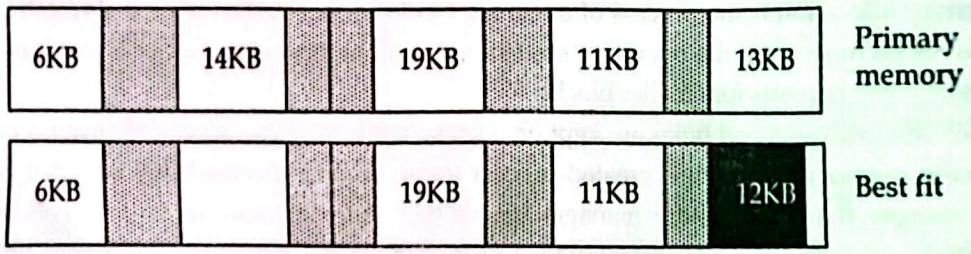


Next Fit

It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

Best Fit

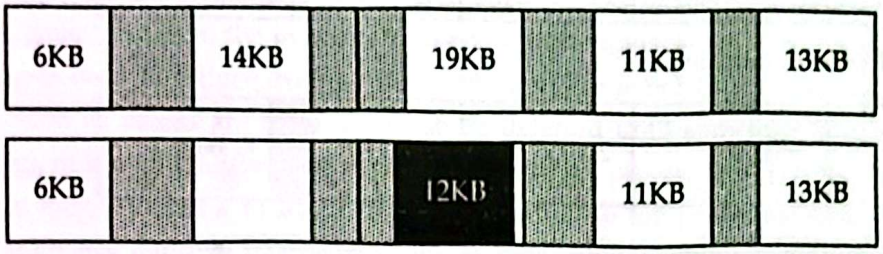
Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed. The allocator places a process in the smallest block of unallocated memory in which it will fit. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.



Worst Fit

Always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either. The memory manager places a process in the largest block of unallocated memory available. The idea is that this placement wil

create the largest hold after the allocations, thus increasing the possibility that compared to best fit; another process can use the remaining space. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

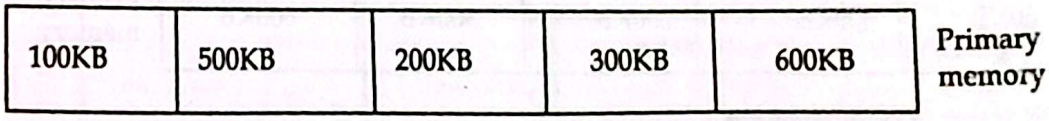


Quick Fit

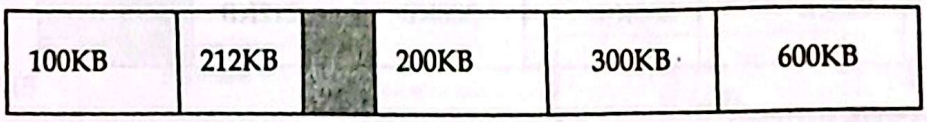
It maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes; the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of say, 21 KB, could either be put on the 20-KB list or on a special list of odd-sized holes. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

Numerical Problem 1: Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-Fit, Best-Fit, and Worst-Fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

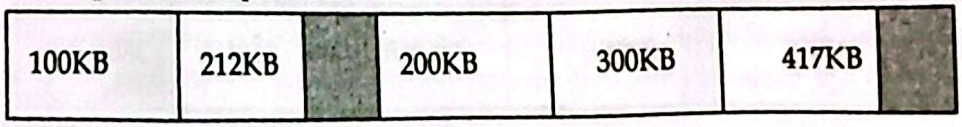
Answer: For First-Fit algorithm



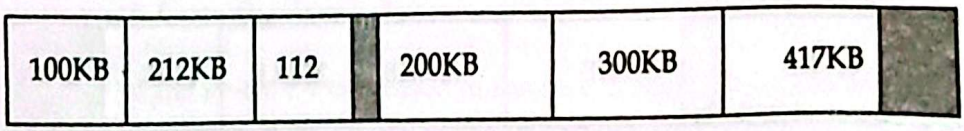
1. 212K is put in 500K partition



2. 417K is put in 600K partition

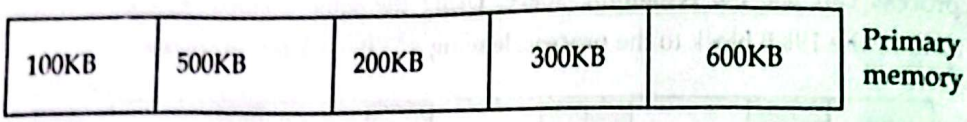


3. 112K is put in 288K partition (new partition 288K = 500K - 212K)

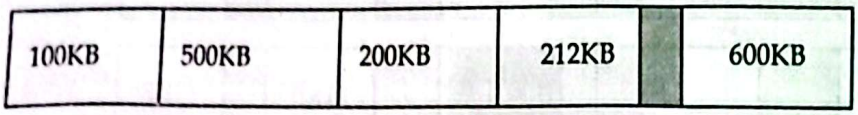


4. 426K must wait

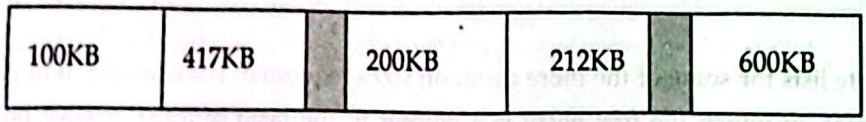
For Best-Fit algorithm



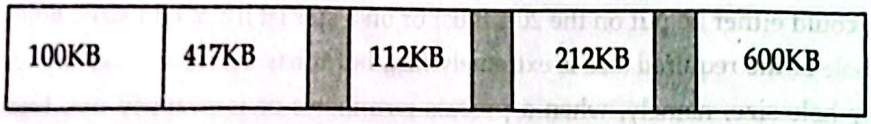
1. 212K is put in 300K partition



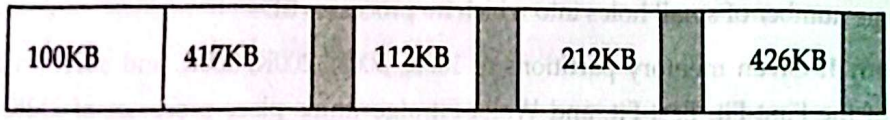
2. 417K is put in 500K partition



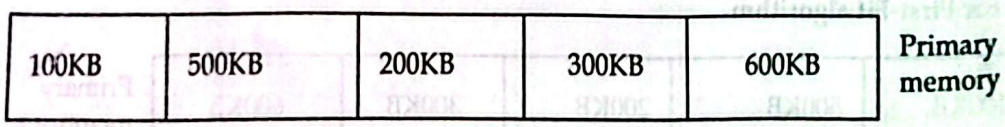
3. 112K is put in 200K partition



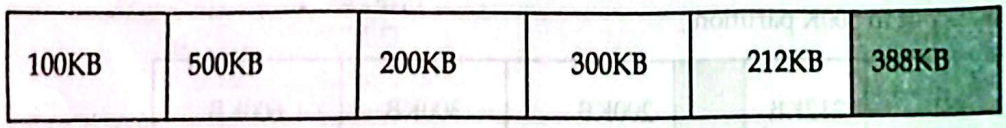
4. 426K is put in 600K partition



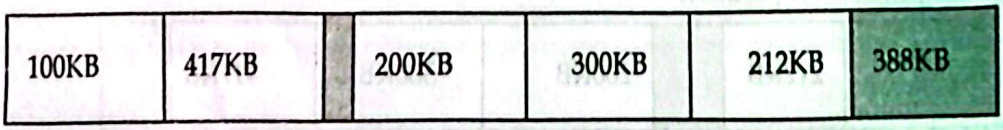
For Worst-Fit algorithm



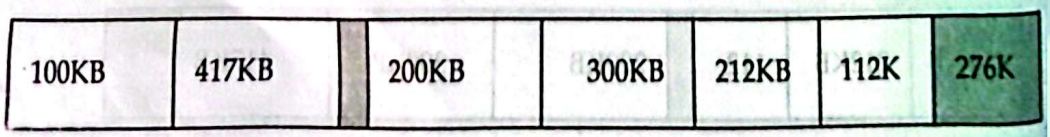
1. 212K is put in 600K partition



2. 417K is put in 500K partition



3. 112K is put in 388K partition



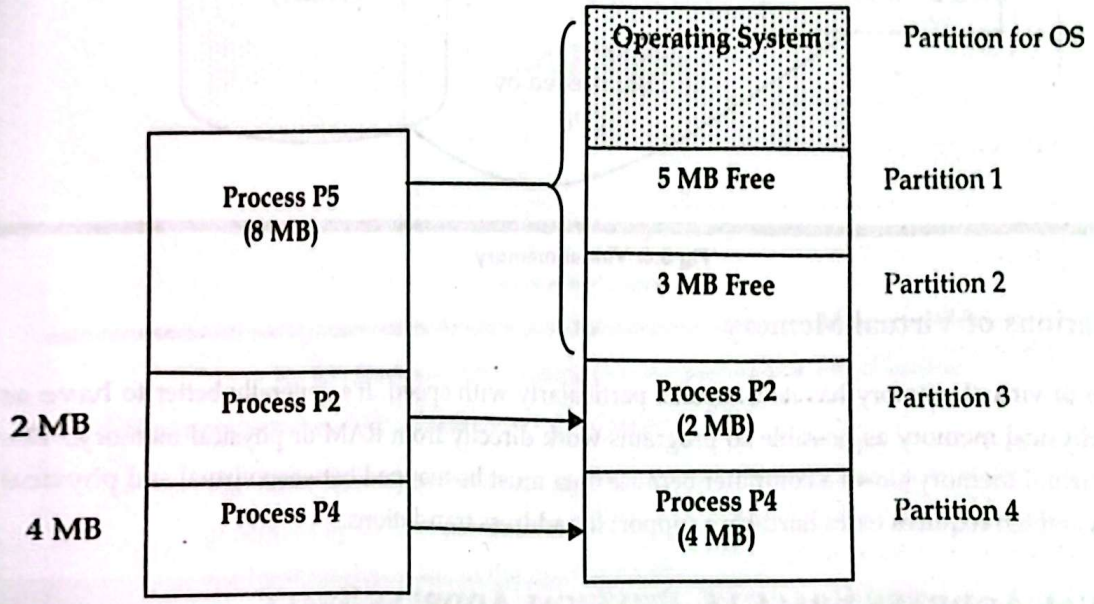
4. 426K must wait

In this example, Best-Fit turns out to be the best.

COALESCING AND COMPACTION

Coalescing is the act of merging two adjacent free blocks of memory. When an application frees memory, gaps can fall in the memory segment that the application uses. Among other techniques, coalescing is used to reduce external fragmentation, but is not totally effective. Coalescing can be done as soon as blocks are freed, or it can be deferred until sometime later (known as deferred coalescing), or it might not be done at all.Compaction is a process in which the free space is collected in a large memory chunk to make some space available for processes. In memory management, swapping creates multiple fragments in the memory because of the processes moving in and out. Compaction refers to combining all the empty spaces together and processes. It helps to solve the problem of fragmentation, but it requires too much of CPU time. It moves all the occupied areas of store to one end and leaves one large free space for incoming jobs, instead of numerous small ones. In compaction, the system also maintains relocation information and it must be performed on each new allocation of job to the memory or completion of job from memory.

In compaction, all the free partitions are made contiguous and all the loaded partitions are brought together. By applying this technique, we can store the bigger processes in the memory. The free partitions are merged which can now be allocated according to the needs of new processes. This technique is also called defragmentation.



As shown in the image above, the process P5, which could not be loaded into the memory due to the lack of contiguous space, can be loaded now in the memory since the free partitions are made contiguous.

Problem with Compaction

The efficiency of the system is decreased in the case of compaction due to the fact that all the free spaces will be transferred from several places to a single place. Huge amount of time is invested for this procedure and the CPU will remain idle for all this time. Despite of the fact that the compaction avoids external fragmentation, it makes system inefficient.

Virtual Memory

Virtual memory is a feature of an operating system that enables a computer to be able to compensate shortages of physical memory by transferring pages of data from random access memory to disk storage.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

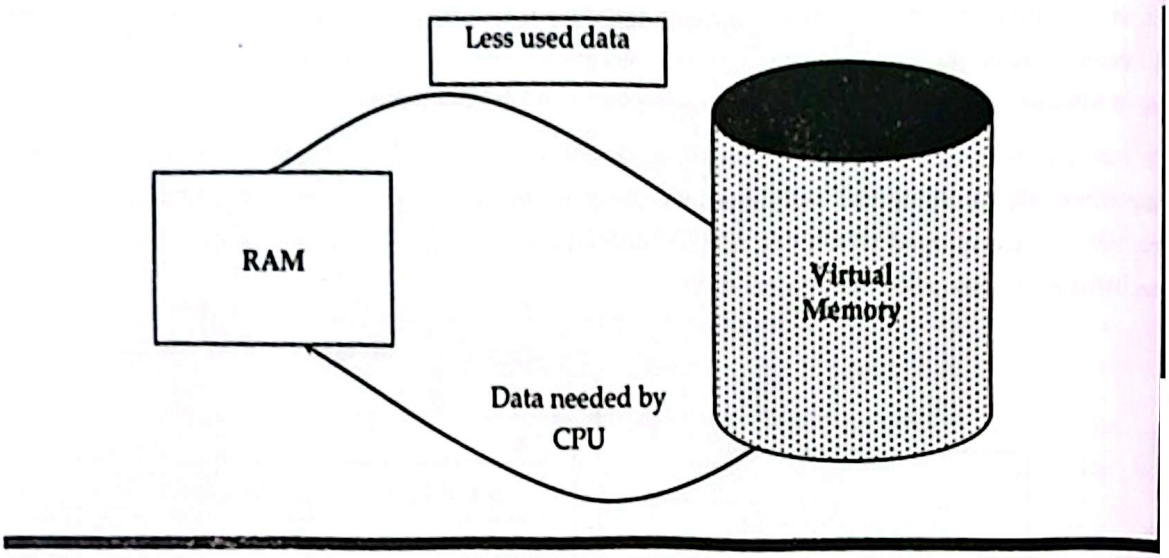


Fig 5.6: Virtual memory

Limitations of Virtual Memory

The use of virtual memory has its tradeoffs, particularly with speed. It's generally better to have as much physical memory as possible so programs work directly from RAM or physical memory. The use of virtual memory slows a computer because data must be mapped between virtual and physical memory, which requires extra hardware support for address translations.

VIRTUAL ADDRESS SPACE VS. PHYSICAL ADDRESS SPACE

Address generated by CPU while a program is running is referred as logical address. The logical address is virtual as it does not exist physically. Hence, it is also called as virtual address. This address is used as a reference to access the physical memory location. The set of all logical addresses generated by a programs perspective is called Logical Address Space. The logical address is mapped to its corresponding physical address by a hardware device called Memory-Management Unit. The address-binding methods used by MMU generate identical logical and physical address during compile time and load time. However, while run-time the address-binding methods generate different logical and physical address.

Physical address identifies a physical location in a memory. MMU (Memory-Management Unit) computes the physical address for the corresponding logical address. MMU also uses logical address computing physical address. The user never deals with the physical address. Instead, the physical address is accessed by its corresponding logical address by the user. The user program generates the logical address and thinks that the program is running in this logical address. But the program needs physical memory for its execution. Hence, the logical address must be mapped to the physical address before they are used. The logical address is mapped to the physical address using a hardware called Memory-Management Unit. The set of all physical addresses corresponding to the logical addresses in a logical address space is called physical address space.

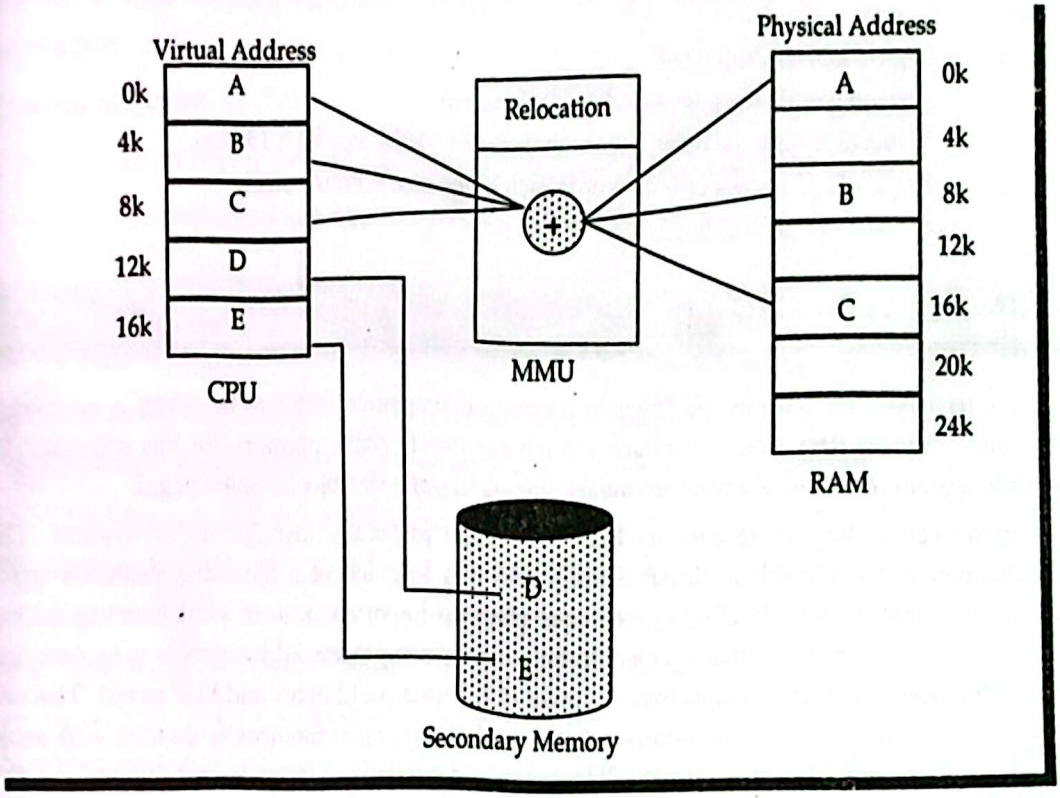


Fig 5.7: Block diagram showing physical address and virtual address

Key differences between logical and physical address in OS

- The basic difference between logical and physical address is that logical address is generated by CPU in perspective of a program. On the other hand, the physical address is a location that exists in the memory unit.
- The set of all logical addresses generated by CPU for a program is called logical address space. However, the set of all physical address mapped to corresponding logical addresses is referred as physical address space.
- The logical address is also called virtual address as the logical address does not exist physically in the memory unit. The physical address is a location in the memory unit that can be accessed physically.
- Identical logical address and physical address are generated by compile-time and load time address binding methods.

- The logical and physical address generated while run-time address binding method differs from each other.
- The logical address is generated by the CPU while program is running whereas; the physical address is computed by the MMU (Memory Management Unit).

Numerical Problem1: Consider a logical address space of 8 pages of 1024 words mapped into memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in physical address?

Answer

a. Logical address will have
3 bits to specify the page number (for 8 pages).
10 bits to specify the offset into each page (2^{10} =1024 words) = 13 bits.

b. For (2^5) 11 32 frames of 1024 words each (Page size = Frame size)
We have 5 + 10 = 15 bits.

PAGING

In computer operating systems, paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory. Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages. Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

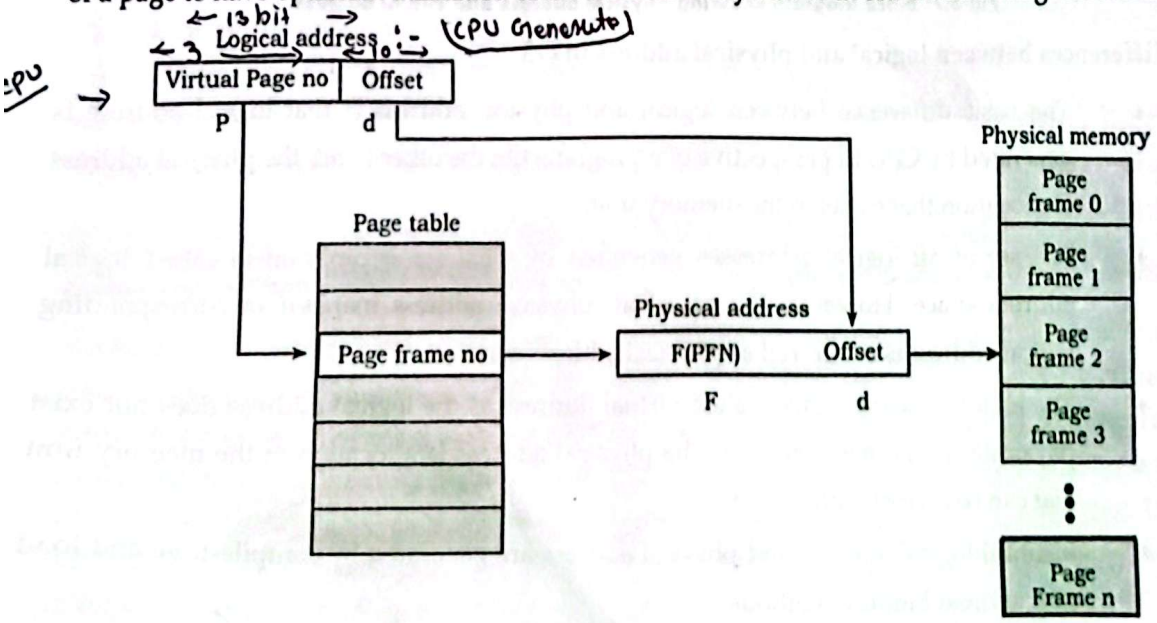


Fig 5.8: Paging Hardware

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory. Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes).

The size of the process is measured in the number of pages. Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

Drawbacks of Paging

- Size of Page table can be very big and therefore it wastes main memory.
- CPU will take more time to read a single word from the main memory.

PAGE TABLE

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses.

It is a data structure that holds information (such as page number, offset, present/absent bit, modified bit etc.) about virtual pages. Since it acts as bridge between virtual pages and page frames thus mathematically we can say that the page table is a function, with the virtual page number as argument and the physical frame number as result. Page table entry has the following information:

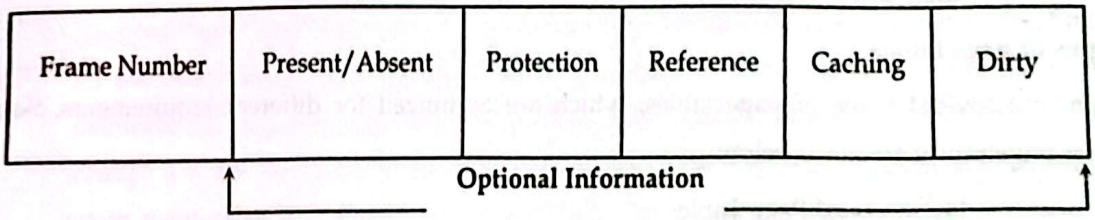


Fig5.9: Page table entry structure

- **Frame Number:** It gives the frame number in which the current page you are looking for is present. The number of bits required depends on the number of frames.

  Number of bits for frame = Size of physical memory/frame size

- **Present/Absent bit:** Present or absent bit says whether a particular page you are looking for is present or absent. In case if it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Used to control page fault by the operating system to support virtual memory. Sometimes this bit is also known as valid/invalid bits.

Protection bit: Protection bit says that what kind of protection you want on that page. So, these bit for the protection of the page frame (read, write etc).

- **Referenced bit:** Referenced bit will say whether this page has been referred in the last clock cycle or not. It is set to 1 by hardware when the page is accessed.
- **Caching enabled/disabled:** Sometimes we need the fresh data. Let us say the user is typing some information from the keyboard and your program should run according to the input given by the user. In that case, the information will come into the main memory. Therefore main memory contains the latest information which is typed by the user. Now if you try to put that page in the cache, that cache will show the old information. So whenever freshness is required, we don't want to go for caching or many levels of the memory. The information present in the closest level to the CPU and the information present in the closest level to the user might be different. So we want the information has to be consistency, which means whatever information user has given, CPU should be able to see it as first as possible. That is the reason we want to disable caching. So, this bit enables or disables caching of the page.
- **Modified bit:** Modified bit says whether the page has been modified or not. Modified means sometimes you might try to write something on to the page. If a page is modified, then whenever you should replace that page with some other page, then the modified information should be kept on the hard disk or it has to be written back or it has to be saved back. It is set to 1 by hardware on write-access to page which is used to avoid writing when swapped out. Sometimes this modified bit is also called as the dirty bit.

Types of page tables

There are several types of page tables, which are optimized for different requirements. Some of major page tables are listed below:

a. Hierarchical Page Table

b. Inverted Page Table

c. Shared page table

a. Hierarchical Page Table

It is also called multilevel page table. The multilevel page table method is to avoid keeping all the pages tables in memory all the time. The top level page table with 1024 entries corresponding to the 10 bits PTL (Page Table Length Filed). When a virtual address is presented to the memory management unit (MMU), it first extracts the PTL (Page Table Length Filed), and uses this value as an index into the top level page table.

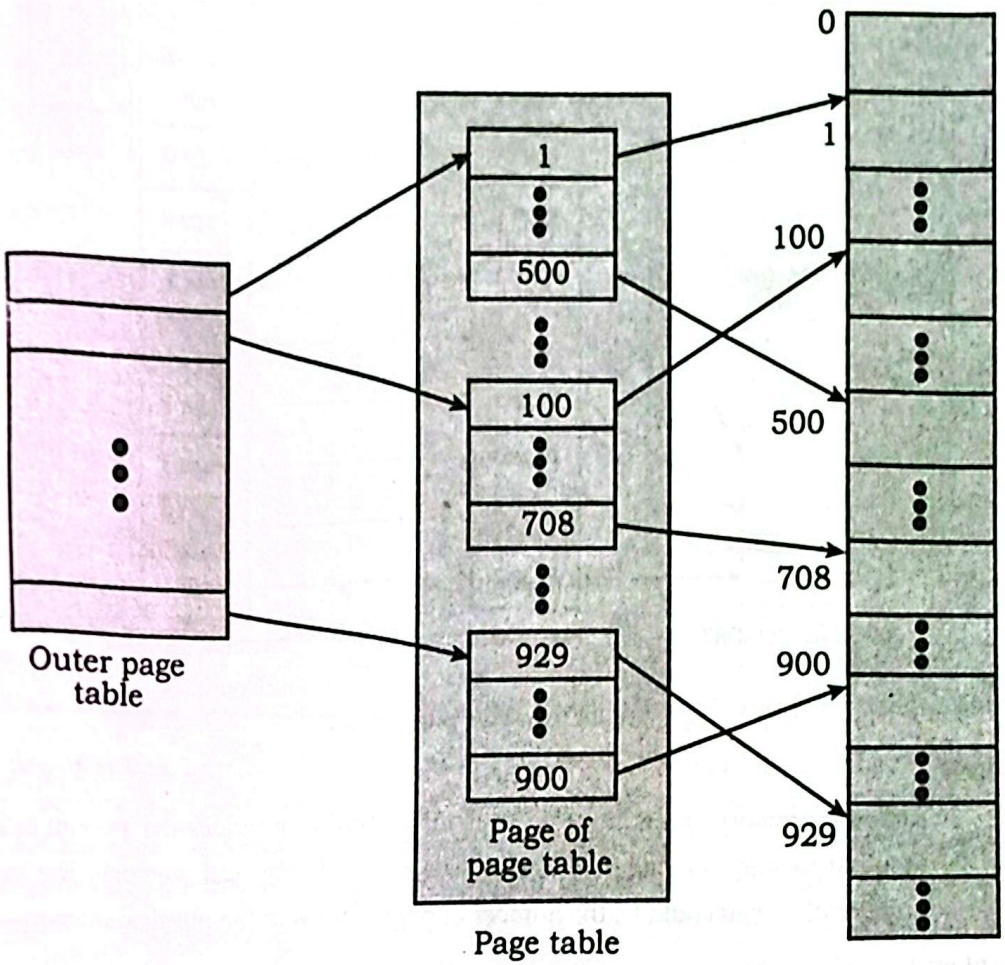


Fig 5.10: Two-level page table

Inverted Page Table

Inverted Page Table is the global page table which is maintained by the Operating System for all the processes. In inverted page table, the number of entries is equal to the number of frames in the main memory. It can be used to overcome the drawbacks of page table. There is always a space reserved for the page regardless of the fact that whether it is present in the main memory or not. However, this is simply the wastage of the memory if the page is not present. We can save this wastage by just inverting the page table. We can save the details only for the pages which are present in the main memory. Frames are the indices and the information saved inside the block will be Process ID and page number.

It has only one page table for all processes. This table has an entry for each real page (block of physical memory). Each element contains the virtual address of the page stored in that physical location, with information about the process that owns that page.

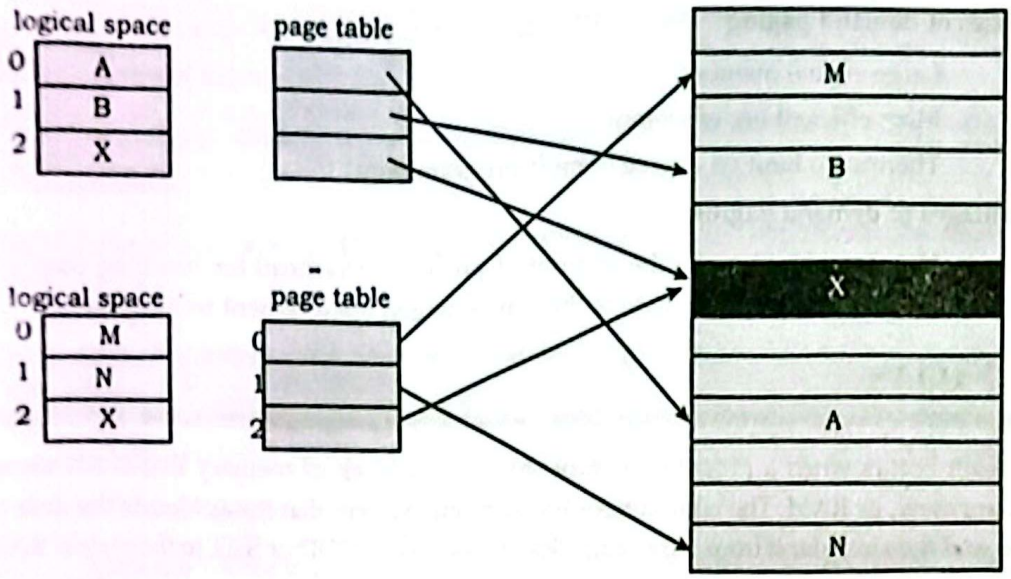


Fig 5.12: Shared page table

DEMAND PAGING

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

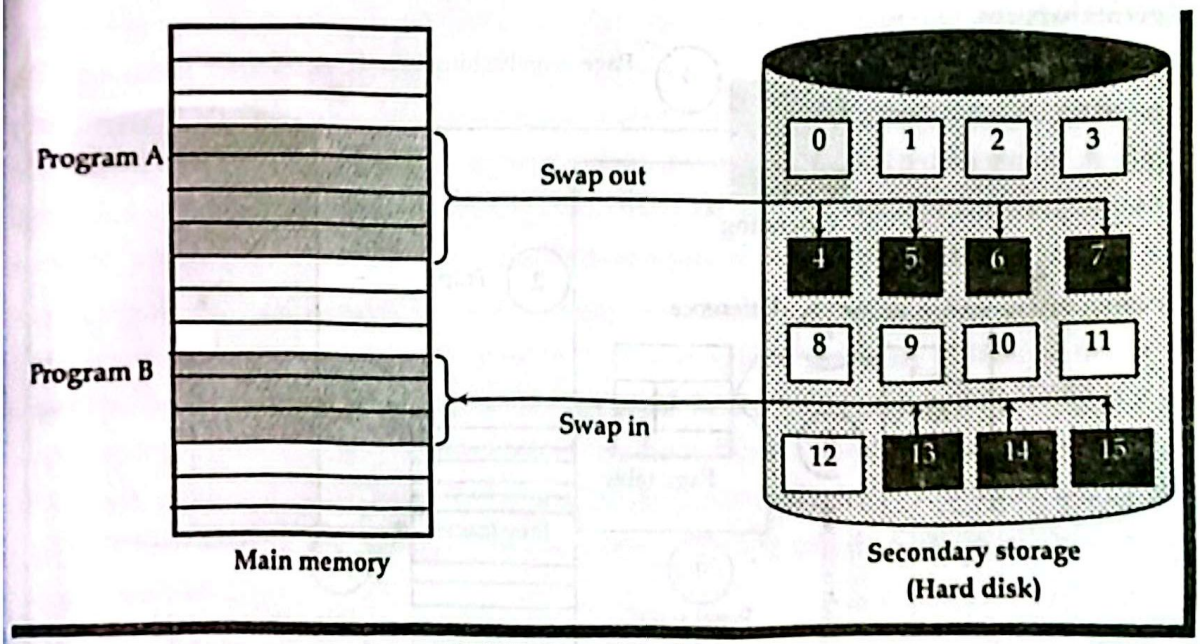


Fig 5.13: Demand Paging

While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a page fault and transfers control from the program to the operating system to demand the page back into the memory.

Advantages of demand paging

- Large virtual memory
- More efficient use of memory
- There is no limit on degree of multiprogramming

Disadvantages of demand paging

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

PAGE FAULTS

A page fault occurs when a program attempts to access a block of memory that is not stored in the physical memory, or RAM. The fault notifies the operating system that it must locate the data in virtual memory, and then transfer it from the storage device, such as an HDD or SSD, to the system RAM.

While page faults are common when working with virtual memory, each page fault requires transferring data from secondary memory to primary memory. This process may only take a few milliseconds, but that can still be several thousand times slower than accessing data directly from memory. Therefore, installing more system memory can increase your computer's performance, since it will need to access virtual memory less often.

HANDLING PAGE FAULTS

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM. So when page fault occurs then following sequence of events happens:

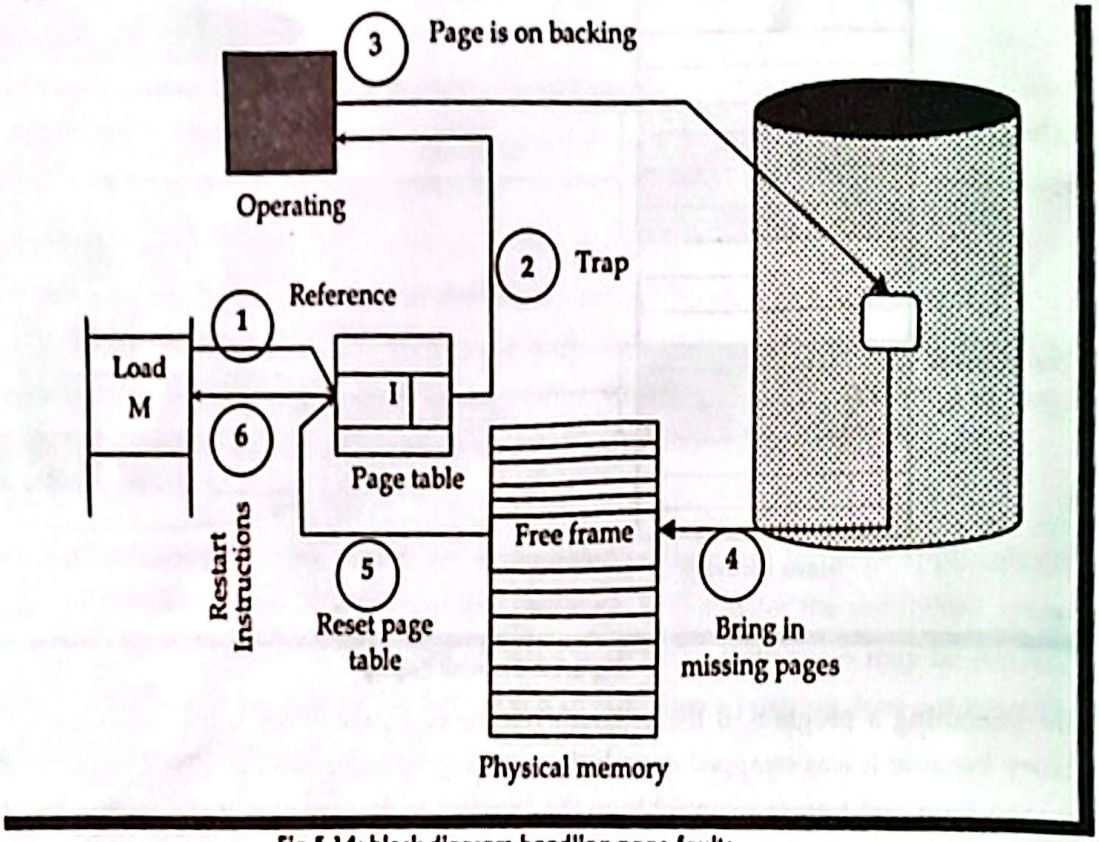


Fig 5.14: block diagram handling page faults

If a page is needed that was not originally loaded up, then a page fault trap is generated, which must be handled in a series of steps:

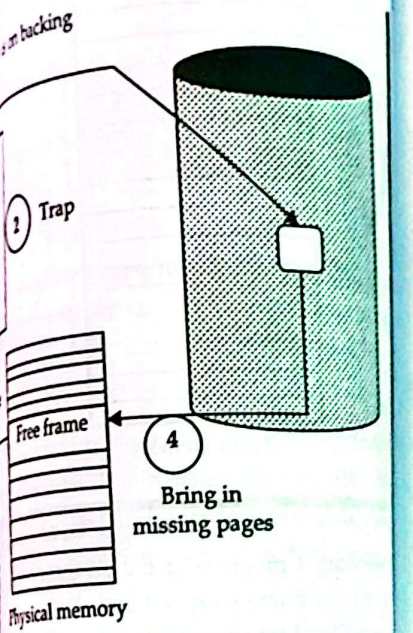
- The memory address requested is first checked, to make sure it was a valid memory request.
- If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
- A free frame is located, possibly from a free-frame list.
- A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime)
- When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
- The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU)

TRANSLATION LOOK-ASIDE BUFFER (TLB'S)

A translation look-aside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. It is a part of the chip's memory-management unit (MMU). The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache. A TLB may reside between the CPU and the CPU cache, between CPU cache and the main memory or between the different levels of the multi-level cache. The majority of desktop, laptop, and server processors include one or more TLBs in the memory-management hardware, and it is nearly always present in any processor that utilizes paged or segmented virtual memory.

Since the page tables are stored in the main memory, each memory access of a program requires at least one memory accesses to translate virtual into physical address and to try to satisfy it from the cache. On the cache miss, there will be two memory accesses. The key to improving access performance is to rely on locality of references to page table. When a translation for a virtual page is used, it will probably be needed again in the near future because the references to the words on that page have both temporal and spatial locality. Each virtual memory reference can cause two physical memory accesses:

- One to fetch the page table
- One to fetch the data



gram handling page faults

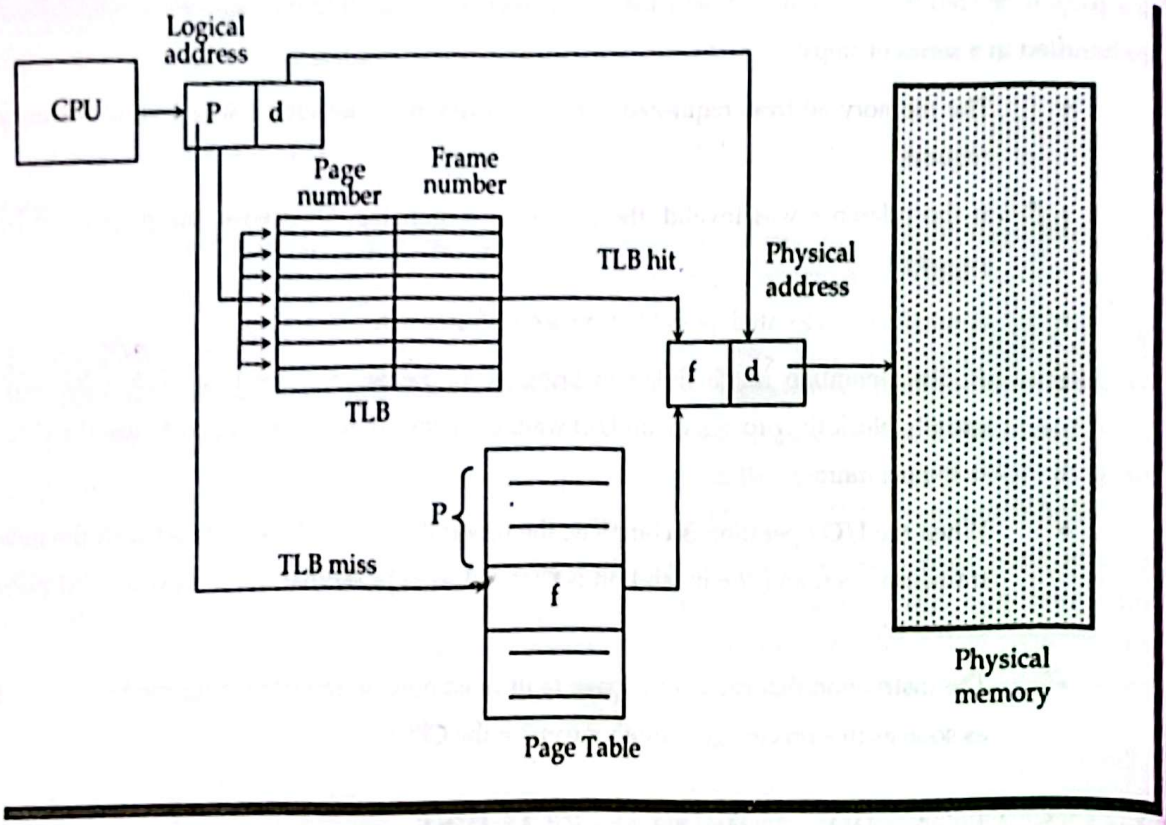


Fig 5.15: Paging hardware with TLB

To overcome this problem a high-speed cache is set up for page table entries called a Translation Look aside Buffer (TLB). Translation Look aside Buffer (TLB) is nothing but a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used. Given a virtual address, processor examines the TLB If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table. TLB first checks if page is already in main memory, if not in main memory a page fault is issued then the TLB is updated to include the new page entry.

PAGE REPLACEMENT ALGORITHMS

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

Page Fault: A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

There are a lot of page replacement algorithms some of major page replacement algorithms are listed below:

- FIFO Page Replacement Algorithms
- Optimal Page Replacement Algorithms
- LRU Page Replacement Algorithms (Least Recently use)
- Second Chance Page Replacement Algorithms
- LFU Page Replacement Algorithms

FIFO Page Replacement Algorithms

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue; oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

For example: 3 page frames which initially empty and 20 pages system the FIFO page replacement is as below,

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
Pf	Pf	Pf	Pf	X	Pf	Pf	Pf	Pf	Pf	Pf	X	X	Pf	Pf	X	X	Pf	Pf	Pf

Page fault =15

Advantages FIFO page replacement algorithm

- Easy to understand and program
- Distribute fair chance to all

Disadvantages FIFO page replacement algorithm

- FIFO is likely to replace heavily (or constantly) used pages and they are still needed for further processing
- It is not very effective
- System needs to keep track of each frame
- Bad replacement choice increases the page fault rate and slow process execution
- Sometimes it behaves abnormally. This behavior is called Belady's anomaly.

Belady'sAnomaly

In computer storage, Belady's anomaly is the phenomenon in which increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns. This phenomenon is commonly experienced when using the first-in first-out (FIFO) page replacement algorithm.

For example, if we consider reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 and number of frames allocated=3, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Reference string

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4
Pf	Pf	Pf	Pf	Pf	Pf	Pf	X	X	Pf	Pf	X

Total page faults = 9

But if we use page frame size 4 for the same reference string we get 10 page faults as below;

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3
Pf	Pf	Pf	Pf	X	X	Pf	Pf	Pf	Pf	Pf	Pf

Total number of page faults = 10

Second Chance Page Replacement Algorithms

It is the modified form of the FIFO page replacement algorithm. One way to implement is to have a circular queue. If the value is equal to 0, then we proceed to replace the page. But if the reference bit is equal to 1, then we give the page second chance. When the page get second chance, its reference bit is cleared. Second-chance page replacement algorithm gives every page a second-chance.

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is set to 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced. One way to implement the second-chance (sometimes referred to as the clock) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

Example: if we consider reference string 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2 and number of frames allocated=3, we get 7 total page faults by using this page replacement algorithm.

reference strings

2	3	2	1	5	2	4	5	3	2	5	2
2(0)	2(0)	2(1)	2(1)	2(0)	2(1)	2(0)	2(0)	3(0)	3(0)	3(0)	3(0)
	3(0)	3(0)	3(0)	5(0)	5(0)	5(0)	5(1)	5(0)	5(0)	5(1)	5(1)
			1(0)	1(0)	1(0)	4(0)	4(0)	4(0)	2(0)	2(0)	2(1)
Pf	Pf	X	Pf	Pf	X	Pf	X	Pf	Pf	X	X

al page faults = 7

dvantages

- Obvious improvement over FIFO
- Allows commonly used pages to stay in queue

isadvantages

- Still suffers from Belady's anomaly

ptimal Page Replacement

n this algorithm, pages are replaced which would not be used for the longest duration of tim ime. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithm ll never suffer from Belady's anomaly. Such an algorithm does exist, and has been called OPT. It is simply replace the page that will not be used for the longest period of time. Use age replacement algorithm guarantees the lowest possible page fault rate for a fixed nu mes. For example, on our sample reference string, the optimal page-replacement algorithm uld 9 page faults. The first three references cause faults that fill the three empty fra erence to page reference string 2 replaces page 7, because 7 will not be used until refe curs page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, ll be the last of the three pages in memory to be referenced again. With only 9 page fault placement is much better than a FIFO algorithm, which had 15 faults.

ample: if we consider reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 and mes allocated=3, we get 9 total page faults by using this page replacement algorithm.

ference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1
Pf	Pf	Pf	Pf	X	Pf	X	Pf	X	X	Pf	X	X	Pf	X	X	X

tal page faults = 9

dvantages of optimal page replacement algorithm

- It is less complex and easy to implement.
- A page is replaced with minimum fuss.
- Simple data structures are used for this purpose.

Reference strings

2	3	2	1	5	2	4	5	3	2	5	2
2(0)	2(0)	2(1)	2(1)	2(0)	2(1)	2(0)	2(0)	3(0)	3(0)	3(0)	3(0)
	3(0)	3(0)	3(0)	5(0)	5(0)	5(0)	5(1)	5(0)	5(0)	5(1)	5(1)
			1(0)	1(0)	1(0)	4(0)	4(0)	4(0)	2(0)	2(0)	2(1)
Pf	Pf	X	Pf	Pf	X	Pf	X	Pf	Pf	X	X

Total page faults = 7

Advantages

- Obvious improvement over FIFO
- Allows commonly used pages to stay in queue

Disadvantages

- Still suffers from Belady's anomaly

Optimal Page Replacement

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms, and will never suffer from Belady's anomaly. Such an algorithm does exist, and has been called OPT or MIN. It is simply replace the page that will not be used for the longest period of time. Use of this page replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield 9 page faults. The first three references cause faults that fill the three empty frames. The reference to page reference string 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only 9 page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults.

Example: if we consider reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 and number of frames allocated=3, we get 9 total page faults by using this page replacement algorithm.

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
Pf	Pf	Pf	Pf	X	Pf	X	Pf	X	X	Pf	X	X	Pf	X	X	X	Pf	X	X

Total page faults = 9

Advantages of optimal page replacement algorithm

- It is less complex and easy to implement.
- A page is replaced with minimum fuss.
- Simple data structures are used for this purpose.

- Lowest page fault rate.
- Never suffers from Belady's anomaly.
- Twice as good as FIFO Page Replacement Algorithm.

Disadvantages of optimal replacement algorithm

- Not all operating systems can implement this algorithm.
- Error detection is harder.
- Least recently used page will be replaced which may sometimes take a lot of time.

LRU Page Replacement Algorithms

In this algorithm, the page that has not been used for the longest period of time has to be replaced.

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time. This approach is the least-recently-used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page replacement algorithm looking backward in time, rather than forward.

Example: if we consider reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 and number of frames allocated=3, we get 12 total page faults by using this page replacement algorithm.

Reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
Pf	Pf	Pf	Pf	X	Pf	X	Pf	Pf	Pf	Pf	X	X	Pf	X	Pf	X	Pf	X	X

Total page faults = 12

Advantages of LRU page replacement algorithm

- It is amenable to full statistical analysis
- Never suffers from Belady's anomaly

CONCEPT OF LOCALITY OF REFERENCE

Locality of reference, also known as the principle of locality, is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed, depending on the memory access pattern. There are two basic types of reference locality:

- Temporal locality of reference and
- Spatial locality of reference

Temporal locality of reference

Temporal locality means current data or instruction that is being fetched may be r
So we should store that data or instruction in the cache memory so that we can
searching in main memory for the same data and thus saving time.

Spatial locality of reference

Spatial locality means instruction or data near to the current memory location
fetched, may be needed soon in near future.

Temporal locality refers to the reuse of specific data, and resources, within re
time duration whereas spatial locality refers to the use of data elements within r
storage locations.

ple: let's take a code segment

```
sum = 0;
for (i = 0; i<arr.length; i++)
        sum += arr[i];
return sum;
```

w looking at this example, here variable sum is being used again and a
poral locality and then the values of array arr is being accessed in order i.e. a
d so on and which shows spatial locality as arrays are contiguous memory l
rent memory location is being fetched.

THRASHING

rashing is a condition or a situation when the system is spending a ma
ecuting the page faults, but the actual processing done is very negligible.

process that is spending more time paging than executing is said to be th
ans that the process doesn't have enough frames to hold all the pag
apping pages in and out very frequently to keep executing. Sometim
quired in the near future have to be swapped out. Initially when the
ess scheduling mechanism to increase the level of multiprogramm
to the memory at the same time, allocating a limited amount of fr
mory fills up, process starts to spend a lot of time for the required
ding to low CPU utilization because most of the processes are
duler loads more processes to increase CPU utilization, as this
mplete system comes to a stop.

a. **Temporal locality of reference**

Temporal locality means current data or instruction that is being fetched may be needed soon. So we should store that data or instruction in the cache memory so that we can avoid again searching in main memory for the same data and thus saving time.

b. **Spatial locality of reference**

Spatial locality means instruction or data near to the current memory location that is being fetched, may be needed soon in near future.

Temporal locality refers to the reuse of specific data, and resources, within relatively small time duration whereas spatial locality refers to the use of data elements within relatively close storage locations.

Example: let's take a code segment

```
sum = 0;
for (i = 0; i<arr.length; i++)
        sum += arr[i];
return sum;
```

Now looking at this example, here variable sum is being used again and again which shows temporal locality and then the values of array arr is being accessed in order i.e. arr[0], arr[1], arr[2],... and so on and which shows spatial locality as arrays are contiguous memory blocks so data near to current memory location is being fetched.

THRASHING

Thrashing is a condition or a situation when the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very negligible.

A process that is spending more time paging than executing is said to be thrashing. In other words it means that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out. Initially when the CPU utilization is low, the process scheduling mechanism to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process. As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the processes are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.

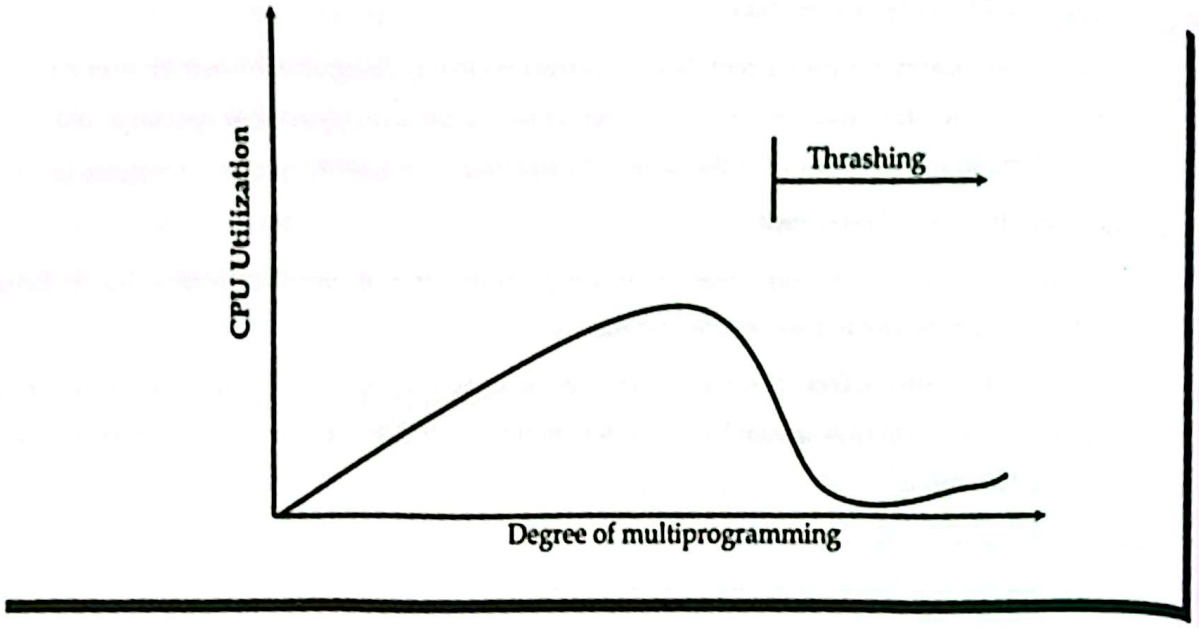


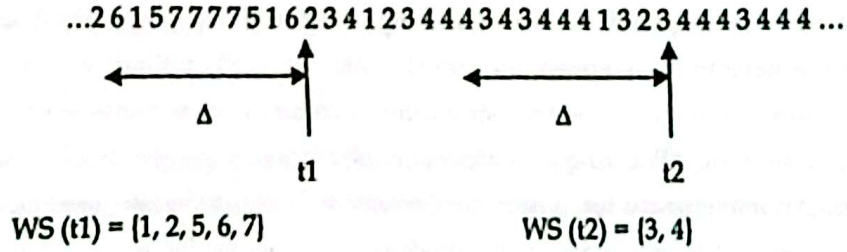
Fig 5.16: Thrashing

To prevent thrashing we must provide processes with as many frames as they really need right now and we also can use working set described as below.

WORKING SET

Working set is a concept in computer science which defines the amount of memory that a process requires in a given time interval. The set of pages that a process is currently using is called its working set.

The working set model is based on the assumption of locality. This model uses a parameter Δ to define the working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is called the working set. If a page is in active use it will be in the working set. If it no longer being used it will drop from the working set Δ time units after its last reference. Thus the working set is an approximation of the program's locality. In the following example, we use a value of Δ to be 10 and identify two localities of reference, one having five pages and the other having two pages.



We now ientify various localities in the process execution trace given in the previous section. Here are the first two and last localities are: L1={18-26, 31-34}, L2={18-23, 29-31, 34}, and last={18-20, 24-34}. Note that in that last locality, pages 18-20 are referenced right in the beginning only and are effectively out of the locality.

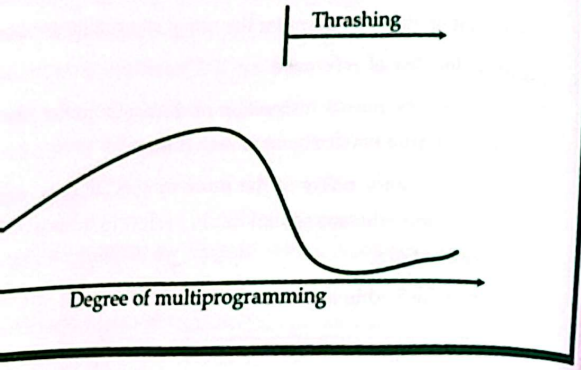


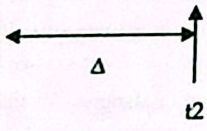
Fig 5.16: Thrashing

le processes with as many frames as they really need right now
ribed as below.

science which defines the amount of memory that a process
set of pages that a process is currently using is called its

assumption of locality. This model uses a parameter Δ to
is to examine the most recent Δ page references. The set of
s is called the working set. If a page is in active use it will be
sed it will drop from the working set Δ time units after its
approximation of the program's locality. In the following
identify two localities of reference, one having five pages

!4343444132344443444 ...



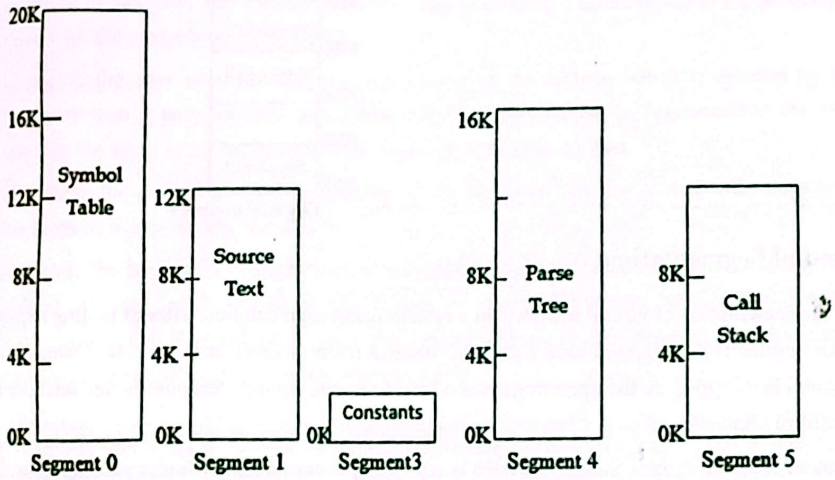
WS (t2) = {3, 4}

ss execution trace given in the previous section. Here
!6, 31-34}, L2={18-23, 29-31, 34}, and last={18-20, 24-34}
ıre referenced right in the beginning only and are

SEGMENTATION

In Operating Systems, Segmentation is a memory management technique in which, the memory is divided into the variable size parts. Each part is known as segment which can be allocated to a process. The details about each segment are stored in a table called as segment table. Segment table is stored in one (or many) of the segments. Segment table contains mainly following information about segment:

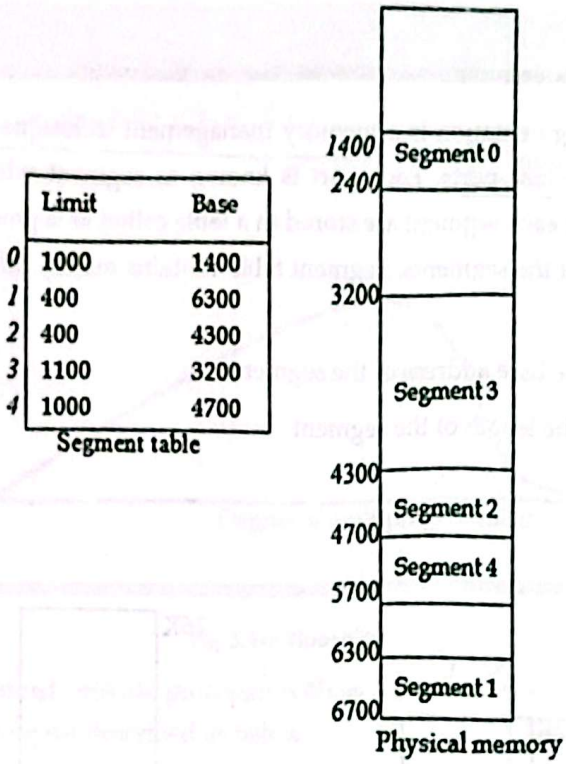
- **Base:** It is the base address of the segment
- **Limit:** It is the length of the segment



Each segment has a name and length. The addresses specify both the segment number and offset within the segment. The user therefore specifies each address by two quantities: **segment number** and an **offset**. Let us consider following example.

The segment number and the offset together combinely generates the address of the segment in the physical memory space.

Example: As an example, consider the situation shown in figure below. We have five segments numbered through 0 to 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in the physical memory (or base) and length of that segment (or limit). For example segment 2 is 400 byte long and begins at location 4300. Thus a reference to byte 53 of segment 2 is mapped onto location 4300+53=4353. A reference to segment 3, byte 852 is mapped to 3200 (the base of segment 3) + 852=4052. A reference to byte 1222 of segment 0 would result in a rap to operating system, as this segment is only 1000 bytes long.



Need of Segmentation

An implementation of virtual memory on a system using segmentation without paging requires that entire segments be swapped back and forth between main memory and secondary storage. When a segment is swapped in, the operating system has to allocate enough contiguous free memory to hold the entire segment.

Memory segmentation is the division of a computer's primary memory into segments or sections. In a computer system using segmentation, a reference to a memory location includes a value that identifies a segment and an offset (memory location) within that segment. Segments or sections are also used in object files of compiled programs when they are linked together into a program image and when the image is loaded into memory. Segments usually correspond to natural divisions of a program such as individual routines or data tables so segmentation is generally more visible to the programmer than paging alone. Different segments may be created for different program modules, or for different classes of memory usage such as code and data segments. Certain segments may be shared between programs.

Advantages of Segmentation

- No internal fragmentation
- Average Segment Size is larger than the actual page size
- Less overhead
- It is easier to relocate segments than entire address space
- The segment table is of lesser size as compare to the page table in paging

Disadvantages of Segmentation

- It can have external fragmentation
- It is difficult to allocate contiguous memory to variable sized partition
- Costly memory management algorithms

Differences between paging and segmentation

1. The basic difference between paging and segmentation is that a page is always of fixed block size whereas; a segment is of variable size.
2. Paging may lead to internal fragmentation as the page is of fixed block size, but it may happen that the process does not acquire the entire block size which will generate the internal fragment in memory. The segmentation may lead to external fragmentation as the memory is filled with the variable sized blocks.
3. In paging the user only provides a single integer as the address which is divided by the hardware into a page number and Offset. On the other hands, in segmentation the user specifies the address in two quantities i.e. segment number and offset.
4. The size of the page is decided or specified by the hardware. On the other hands, the size of the segment is specified by the user.
5. In paging, the page table maps the logical address to the physical address, and it contains base address of each page stored in the frames of physical memory space. However, in segmentation, the segment table maps the logical address to the physical address, and it contains segment number and offset (segment limit).

SEGMENTATION WITH PAGING (MULTICS)

Instead of an actual memory location the segment information includes the address of a page table for the segment. When a program references a memory location the offset is translated to a memory address using the page table. A segment can be extended simply by allocating another memory page and adding it to the segment's page table.

An implementation of virtual memory on a system using segmentation with paging usually only moves individual pages back and forth between main memory and secondary storage, similar to a paged non-segmented system. Pages of the segment can be located anywhere in main memory and need not be contiguous. This usually results in a reduced amount of input/output between primary and secondary storage and reduced memory fragmentation.

The MULTICS operating system was one of the most influential operating systems ever. Having had a major influence on topics as disparate as UNIX, the x86 memory architecture, TLBs, and cloud computing. It was started as a research project at M.I.T. and went live in 1969. The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.

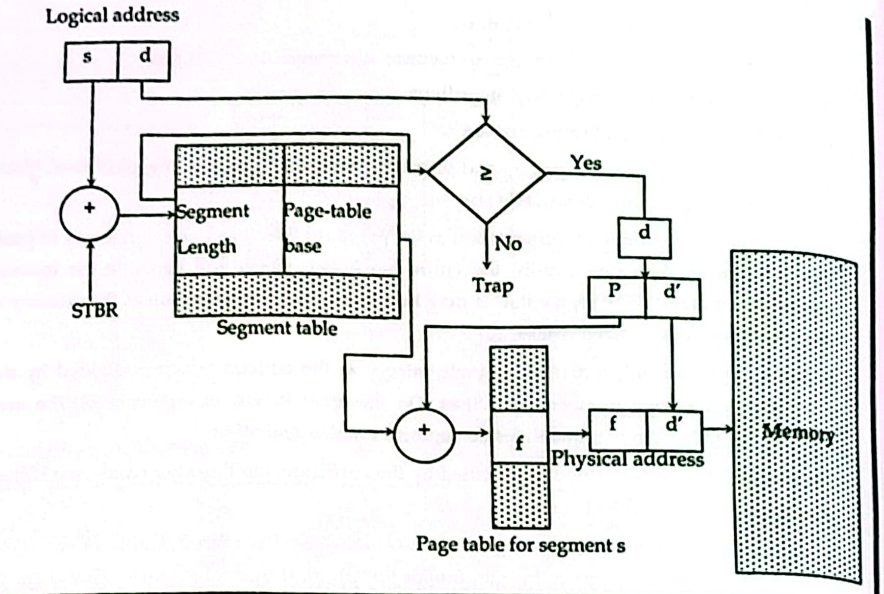


Fig 5.17: Segmentation with paging

When a memory reference occurred, the following algorithm was carried out.

- The segment number was used to find the segment descriptor.
- A check was made to see if the segment's page table was in memory. If it was, it was located. If it was not, a segment fault occurred. If there was a protection violation, a fault (trap) occurred.
- The page table entry for the requested virtual page was examined. If the page itself was not in memory, a page fault was triggered. If it was in memory, the main-memory address of the start of the page was extracted from the page table entry.
- The offset was added to the page origin to give the main memory address where the word was located.
- The read or store finally took place.

Numerical Problem 1: Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical

a. 0,430
b. 1, 10
c. 2,500
d. 3,400
e. 4,112

Answer

a. 219 + 430 = 649
b. 2300 + 10 = 2310
c. illegal reference, trap to operating system
d. 1327 + 400 = 1727
e. illegal reference, trap to operating system

Numerical Problem 2: Given memory partitions of 10 would each of the First-fit, Best fit and worst fit algor 426K in order? Which algorithm makes the most effici

Solution:

a. **First-fit**
- 212K is put in 500K partition
- 417K is put in 600K partition
- 112K is put in 288K partition (new parti
- 426K must wait

b. **Best-fit**
- 212K is put in 300K partition
- 417K is put in 500K partition
- 112K is put in 200K partition
- 212K is put in 500K partition
- 426K is put in 600K partition

c. **Worst-fit**
- 212K is put in 600K partition
- 417K is put in 500K partition
- 112K is put in 388K partition
- 426K must wait

In this example, Best-fit turns out to be the best.

What are the physical addresses for the following logical addresses?

a. 0,430

b. 1, 10

c. 2,500

d. 3,400

e. 4,112

Answer

a. 219 + 430 = 649

b. 2300 + 10 = 2310

c. illegal reference, trap to operating system

d. 1327 + 400 = 1727

e. illegal reference, trap to operating system

Numerical Problem 2: Given memory partitions of 100K, 500K, 200K, 300K and 600K in order, how would each of the First-fit, Best fit and worst fit algorithms place processes of 212K, 417K, 112K and 426K in order? Which algorithm makes the most efficient use of the memory?

Solution:

a. **First-fit**

- 212K is put in 500K partition
- 417K is put in 600K partition
- 112K is put in 288K partition (new partition 288K=500K-212K)
- 426K must wait

b. **Best-fit**

- 212K is put in 300K partition
- 417K is put in 500K partition
- 112K is put in 200K partition
- 212K is put in 500K partition
- 426K is put in 600K partition

c. **Worst-fit**

- 212K is put in 600K partition
- 417K is put in 500K partition
- 112K is put in 388K partition
- 426K must wait

In this example, Best-fit turns out to be the best.

Output:
2 -1 -1
2 3 -1
2 3 -1
2 3 1
2 5 1
2 5 1
2 5 4
2 5 4
3 5 4
3 5 2
3 5 2
3 5 2
No of page faults: 7

EXERCISE

Multiple Choice Questions

1. A memory buffer used to accommodate a speed differential is called _____
   a) Stack pointer b) cache
   c) Accumulator d) disk buffer
2. Which one of the following is the address generated by CPU?
   a) Physical address b) absolute address
   c) Logical address d) none of the mentioned
3. Memory management technique in which system stores and retrieves data from secondary storage for use in main memory is called?
   a) Fragmentation b) paging
   c) Mapping d) none of the mentioned
4. The page table contains _____
   a) Base address of each page in physical memory
   b) page offset
   c) Page size
   d) none of the mentioned
5. Operating System maintains the page table for _____
   a) Each process b) each thread
   c) Each instruction d) each address

In contiguous memory allocation _____

a) Each process is contained in a single contiguous section of memory
b) All processes are contained in a single contiguous section of memory
c) The memory space is contiguous
d) None of the mentioned

In fixed size partition, the degree of multiprogramming is bounded by _____

a) The number of partitions
b) the CPU utilization
c) The memory size
d) all of the mentioned

A solution to the problem of external fragmentation is _____

a) Compaction
b) larger memory space
c) Smaller memory space
d) none of the mentioned

_____ is generally faster than _____ and _____

a) First fit, best fit, worst fit
b) best fit, first fit, worst fit
c) Worst fit, best fit, first fit
d) none of the mentioned

External fragmentation will not occur when?

a) First fit is used
b) best fit is used
c) Worst fit is used
d) No matter which algorithm is used, it will always occur

Subjective Questions

1. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)?Which algorithm makes the most efficient use of memory?
2. Most systems allow programs to allocate more memory to its address space during execution. Data allocated in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?
   a) contiguous-memory allocation
   b) pure segmentation
   c) pure paging
3. On a system with paging, a process cannot access memory that it does not own; why? How could the operating system allow access to other memory? Why should it or should it not?
4. Compare paging with segmentation with respect to the amount of memory required by the address translation structures in order to convert virtual addresses to physical addresses.
5. Why are segmentation and paging sometimes combined into one scheme?
6. Explain why it is easier to share a reentrant module using segmentation than it is to do so when pure paging is used.

7. Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

8. What is the purpose of paging the page tables?
9. Consider the hierarchical paging scheme used by the VAX architecture. How many memory operations are performed when a user program executes a memory load operation?
10. Compare the segmented paging scheme with the hashed page table's scheme for handling large address spaces. Under what circumstances is one scheme preferable over the other?
11. Consider the following page reference string: 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 FIFO, LRU and optimal page replacement algorithm? Assuming four frames, remember all frames are initially empty.
12. What is the copy-on-write feature and under what circumstances is it beneficial to use this feature? What is the hardware support required to implement this feature?
13. What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?
14. Is it possible for a process to have two working sets? One representing data and another representing code? Explain.
15. Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discarding that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?
16. Discuss situations under which the least frequently used page-replacement algorithm generates fewer page faults than the least recently used page replacement algorithm. Also discuss under what circumstance does the opposite holds.
17. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)?Which algorithm makes the most efficient use of memory?
18. Explain the difference between logical and physical addresses.
19. Explain the difference between internal and external fragmentation.
20. Why is it that, on a system with paging, a process cannot access memory it does not own? How could the operating system allow access to other memory? Why should it or should it not?

ANSWERS KEY

1. (b)	2. (c)	3. (b)	4. (a)	5. (a)	6. (a)	7. (a)	8. (a)	9. (a)	10. (b)

□□□