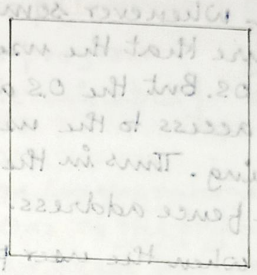
Memory Management:

· When a computer is build to perform a dedicated task, we can use a memory model which is called a bare machine.

Bare Machine: In bare machine it is assumed that the system is a dedicated system that is the kind of task perform on this system is also dedicated (i.e application specific). Since in such cases as we don't have any prespecified operating system, whatever task will be handled by that system, all those task will be decide by the designer itself. This means the memory model that will be used in such bare system is a blank memory.

The entire memory is now available to the designer. The designer has to decide where to put the monitor program, and which part will be used for storage of data or which part will be used as scratch pad etc. There is no limitation put on the designer. This is the particular type of configuration which is used mainly for building the dedicated system.

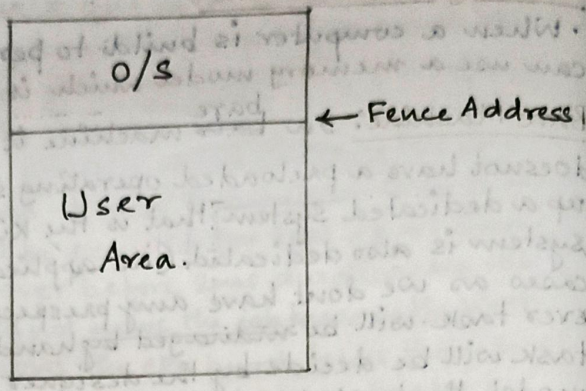


Blank Memory: No preloaded O.S.

The computer system that we use, those are not the part of bare machine. The computer system that we use, the memory model in that are called Resident Monitor system. This means a part of the operating system always resides in memory. The memory we are talking about is the main memory and not the secondary memory as CPU cannot access secondary memory. In case of resident monitor, a part of the memory is always occupied by the operating system which is known as the monitor part of the operating system. Every part of the operating system need not be in the main memory always. For eg. a device driver: whenever we want to take the output of a file, it is the device driver for that driver which is active. As the device driver is needed only when some print-out need to be taken, we can load the device driver only when it is needed. During other time, when the printer is not used, we don't have to put the device driver in the main memory. Because this will unnecessarily consume main memory. Thus, a core part of the O.S., which is used for process management, memory management, are the part which is frequently used, this part of the O.S are put in the main memory. This is called the monitor part of the operating system.

In case of the resident monitor system, the main memory is divided into two parts. One part is always used by the O.S. or the monitor part of the operating system. This portion of the memory is never available to the user program. The rest part of the memory is given to the user for loading user program or user data. This part of the memory is called user area.

When the OS is always loaded in the main memory, the user dangers that when an user program is being executed, the user program may try to access something in the O.S area or try to write something in the O.S. area, ~~then the~~ when the O.S. itself will be corrupted. This may lead to machine crash. To avoid that problem, we need some protection. Whenever some user program is executed, we must ensure that the user program is not permitted to access the OS. But the OS as ~~no~~ it monitors the entire computer, can have access to the user area. This means we need to have some fencing. Thus in the boundary of two area we need to give some fence address.

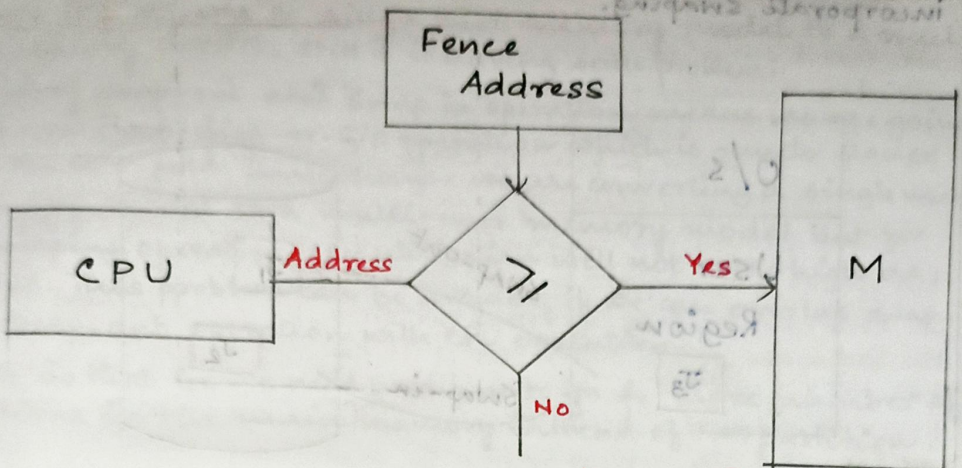


Thus, when the user program tries to access the memory, the address to be generated is checked with the fence address. if the address is less than the fence address, then the user program is trying to access the monitor area of the memory which should not be permitted. When the address generated by the CPU is less than the fence address, then there will be an interrupt saying the user is trying to access the area which is not permitted. Following the interrupt the user program has to be terminated. If the address is greater than or equal to the fence address, then it is a valid ~~program~~ address and then only the user program can continue. So always we have to ~~had~~ have a check with the fence address.

Now, the question arises how to put this fence address. One option is that the fence address can be placed in hardware. ~~when~~ever But there is one problem, whenever the user tries to access any memory, the memory address is compared with the fence address. There are two ways by which this comparison can be made.

- One is software. So we can write a program that whenever the user program tries to access a memory there will be a part of the software which will compare this address as the fence address. But in this case the process will be very slow because the comparison has to be made by the software.

- The other option is, we can go for a hardware comparator. So we can have a scheme like the flowchart as stated.

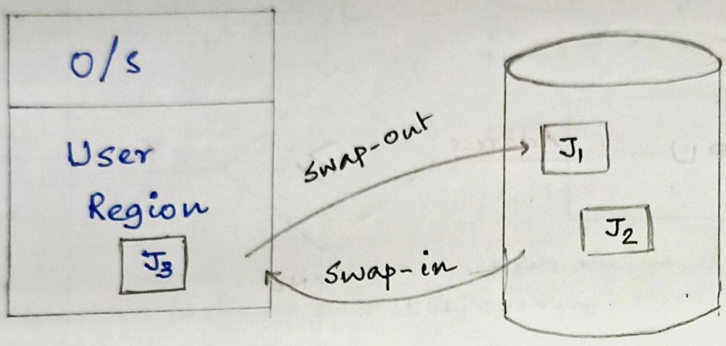


· Thus, we can have such a hardware comparator and fence address can also build in hardware. One of the inputs of these comparators can be tied to the fence address permanently. But there may be a problem. If the fence address is build in a hardware, the fence address is fixed. So, if we are having fixed fence address, there may be a ~~problem~~ danger if we want to upgrade the operating system it may be ~~error~~ cause a problem. Suppose if we are having ~~a fixed fence address~~, an OS taking 40KB of memory. So the fence address will be set to 40. But in next day if we want to upgrade the OS, say the area taken by the OS is 60 KB. But if the fence address is inbuild in hard ware which is set at 40KB, we will find that 20KB of OS are no more protected. In the reverse if the memory taken by the new OS is less than the previous version of the OS, in that case a part of the memory will be fenced unnecessarily, though it can be given to the user. Thus, both problem exist.

To solve this problem, let us not fixed the fence address in hardware, let us have something called fence ~~register~~ register. The content of the fence register will say what is the fence address, and the comparator part still be in hard ware. Thus, the fence address will be stored in a fence register and whenever we upgrade the OS, depending upon the memory requirement of the upgraded OS, we have to ~~reload~~ reload this fence register.

Thus, having this fence register and hardware comparator we can make this comparison process fast at the same time it is flexible that is whenever we try to update the OS, accordingly we can set the value of the fence register. Since, the memory is partitioned into two partition, one OS and the other is user area, whichever way we specify this fence address, this is ~~mostly~~ mostly suitable for a single user system. This is not much suitable for a multi-user system. This model can be converted into a multi-user system if we

incorporate swaping.



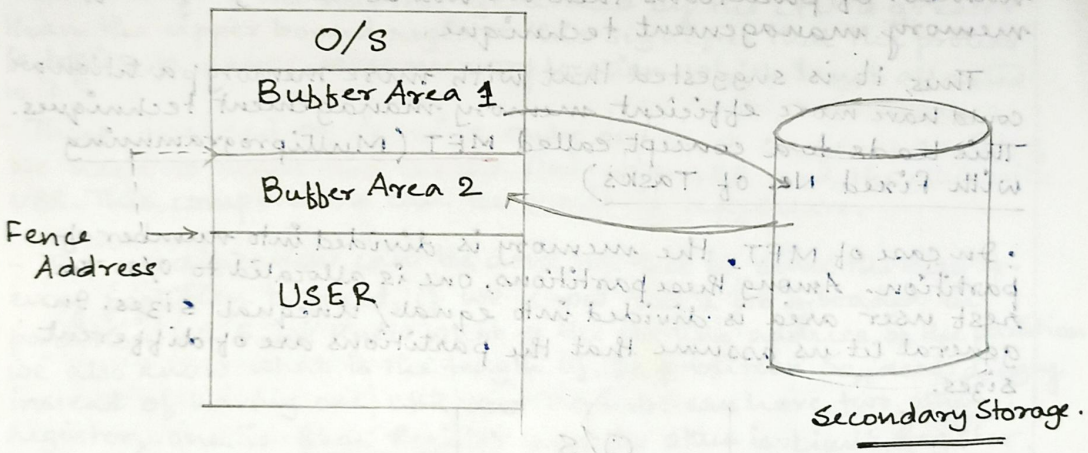
Secondary Storage.

- Let the secondary storage contains many user programs say J_1, J_2 etc. At a particular time, J_3, a user program which is loaded in the user area of main memory. This means Job 3 (J_3) is being executed by the CPU. This is a single user configuration. Now, we want to convert it in a multi-user configuration.
  - Whenever the user program Job 3 is complete, we need to swap out this into secondary storage and the next job which is to be executed by the CPU (Say J_2) will be swapped in to the user area of main memory from the secondary storage.
  - Which ever program is to be terminated or suspended, because of various reason, at the point CPU has to be given to some other program.
  - The second can be that the program terminates itself. This means it has completed all its operations and the program is terminated. Then also we need to remove the job from the user area and need to bring a new job in the user area of the main memory.
  - The third one can be an abnormal termination. A situation when a trap occurs because the user program is trying to access the operating system area of the main memory. There also, the job needs to be terminated and once a job is terminated, we need to bring the next job from the secondary storage into the main memory for execution.
- When a program is to be moved out of the main memory, ~~we need to~~ we can have a situation where the program is modified in the main memory. In this case when ~~we move~~ the program is swapped out of the main memory, the program need to be written back in to the second storage. If it is not modified, this ~~means~~ means a copy of that is already residing in the main memory. So we neednot write it back in the secondary storage. We simply overwrite the old program with the new program that comes from the secondary storage.

• Though it converts a single user memory model to a multi user memory model, still it is having some problem:

- Every swap-out and swap in operation means we are going to perform some disk or I/O operation which is much slower than the CPU speed. Thus, though we are converting a single user memory model to a multi-user memory model but for this I/O operation, CPU utilization will not be as high as expected. This problem can be avoided, if we can overlap swap-in and swap-out operation with CPU execution.

- To do that we need to go for more number of partitions in the main memory instead of two partition.



• A partition of main memory is always given to the O/S.

• The user area is divided into three partitions.

- One of the partition is called Buffer Area 1.
- The other partition is called Buffer Area 2.
- The last partition is called the actual user area which contains the program which is currently active. This is program that is being executed by the CPU.
- Then we have the secondary storage.

• With this more number of partitions what can be done is, the user program or active program in this fourth partition can be executed on the CPU at the same time, the previous program can be swapped out from Buffer Area 1 and the next program to be executed can be swapped-in in Buffer Area 2.

• Thus, the program that has already been executed will be pushed in buffer area 1. From buffer area 1 it will be swapped out. Transferring a block from user area to buffer area 1 is done very fast as it is a main-memory to main-memory transfer.

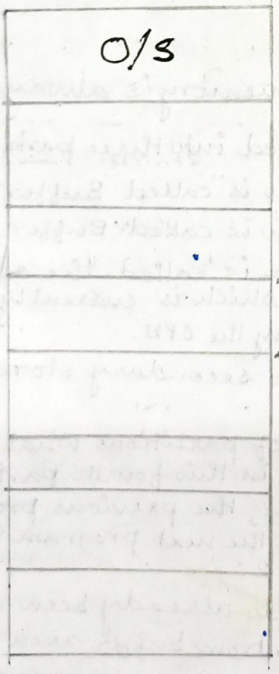
• A job which is loaded in buffer area 2 is transferred from buffer area 2 to user area when it is active and is ready to execute. All these processes can run con-currently.

• Again here as the program is executed in the user area so we need to give a fence address. Because while execution, we donot want the program in user area should access the other areas (neither Buffer area 1 nor buffer area 2 nor O/S). So we need to have a fence address which will protect the user area from other memory area.

The process can be made much faster. As we know that the next program to be executed is residing buffer area 2. So instead of transferring the data, we could simply shift the fence address and let the next program to be executed from the location itself. Thus we donot need to transfer block from buffer area 2 to user area 1, so this saves time and makes the process faster. So if we have more and more number of partitions then we will be having more efficient memory management technique.

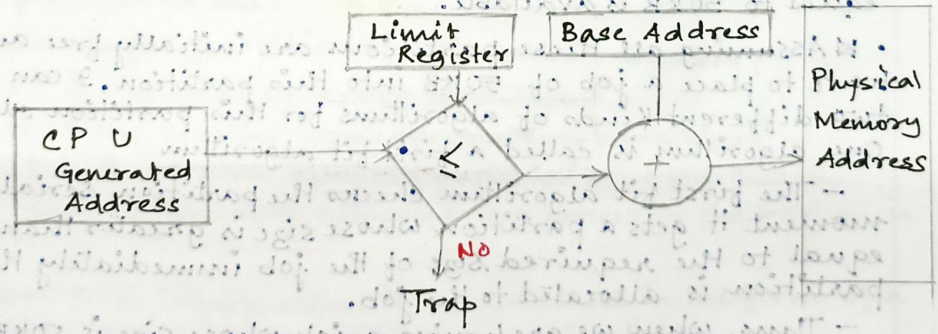
Thus, it is suggested that with more memory partition we could have more efficient memory management techniques. This leads to a concept called MFT (Multiprogramming with Fixed No. of Tasks)

• In case of MFT, the memory is divided into number of partition. Among these partitions, one is allocated to O/S. The rest user area is divided into equal/unequal sizes. In general let us assume that the partitions are of different sizes.



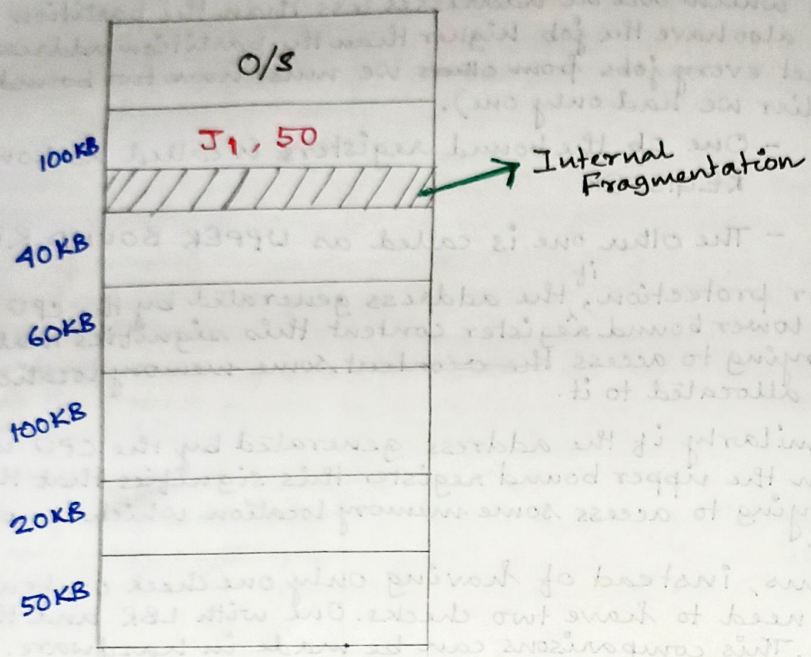
• Each of these partitions can contain each of the program. As we are having a number of partitions and different partitions can have different user program so the degree of multiprogramming is decided by no. of partitions. So in this case switching over from one job to another is simply changing the instruction pointer content.

- As we are having multiple no. of ~~jobs, so~~ partitions, so we have jobs which are at addresses less than the partition address. We can also have the job higher than the partition address. Thus, to protect every jobs from others we must have two bound registers. (Earlier we had only one).
    - One of the bound registers is called as LOWER BOUND REGISTER
    - The other one is called as UPPER BOUND REGISTER.
- For protection, if the address generated by the CPU is less than the lower bound register content this signifies that the process is trying to access ~~the o content~~ some memory location which is not allocated to it.
- Similarly if the address generated by the CPU is greater than the upper bound register this signifies that the process is trying to access some memory location which is not allocated to it.
- Thus, instead of having only one check on fence register, we need to have two checks. One with LBR and the other with UBR. This comparisons can be made in hardware.
- The modification can be done on this is, since the size of every partition is fixed, if we know that a job is loaded in partition no. 5, we know what is the starting address of the partition we also know what is the length of the partition. So, accordingly instead of having one LBR and UBR we can have two other register, one is Basic Register and the other is Limit Register. So the comparison can be given in the below form:



- If the CPU generated address is less than or equal to the Limit register, in that case CPU generated address is added with the base address to get the physical memory address which is to be accessed by the process. The address generated by the CPU is the logical address. Physical address tells which particular location of the main memory needs to be accessed.

As we said that the user area is divided into number of partitions and different partitions can be of different size so, naturally a question comes than whenever a new job is put into the main memory (i.e the new job has to be moved into the ready queue) which of these partitions should be allocated to the new job. So, we can have a situation like this:



• Once we decide ~~if~~ the respective partition sizes, the partition size is fixed. Thus whenever a new job is to be put in the main memory, we have to find out a partition which can take that job. So, if we are having a job whose memory requirement is 120KB, then in the above partition the job cannot be allocated. If we are having a job whose size is 50KB then the job can be loaded in the memory provided a partition of size higher or equal to 50KB is available.

• ⊠ Assuming all these partitions are initially free and we have to place a job of 50KB into this partition, 9 can go for two different kinds of algorithms for this partition allocation. One algorithm is called a first fit algorithm

– The first fit algorithm checks the partition serially. The moment it gets a partition whose size is greater than or equal to the required size of the job immediately that partition is allocated to the job.

– Thus, when we are having a job whose size is 50KB, it first finds a free partition whose size is 100KB. So this 100KB partition ~~is~~ immediately will be allocated to job J_1 whose size requirement is 50KB. Out of this 100KB and because this is fixed partition, part of these partition cannot be allocated to any other job. Either a partition has to be allocated fully to a job or don't allocate at all. Thus, a 50KB job is allocated to a partition whose size is 100KB, remaining 50KB is ~~wastage~~ wasted. This cannot be used by any other process/job. This partition which is actually allocated but not used and cannot be used by any other ~~partition~~ process is called a fragmentation. As in this case this fragmented area is a part of this allocated partition, so it is called internal fragmentation.

• We can have another algorithm which is known as Best Fit Algorithm. Best Fit algorithm will check all the free partitions and will allocate that free partition to this job which will lead to minimum internal fragmentation. Since the requirement of the job is 50KB and it checks all the partitions which are free and it finds any partitions which is free and the size is more than 50KB and nearest to 50 KB will be allocated to the particular job. In this case we have a partition which is of 50KB. Thus if we allocate this job J_1 in that case there will be no wastage of memory. So, in this case the internal fragmentation will be zero. This, is a special case and cannot be guranteed all the time. But in case of MFT Technique we will have some amount of internal fragmentation. The problem with best fit algorithm is that the complexity is more compare to first fit algorithm.

• ~~We can~~ Now, we can have a situation when we will be having a job say of 120KB, though all the partitions are free, we cannot allocate the a job. This fragmentation is called external fragmentation. The fragmentation which is not because of non-utilisation portion of a partition, which is allocated to a process, is known as an external fragmentation. In case of MFT Technique we can lead to both internal and external fragmentation.

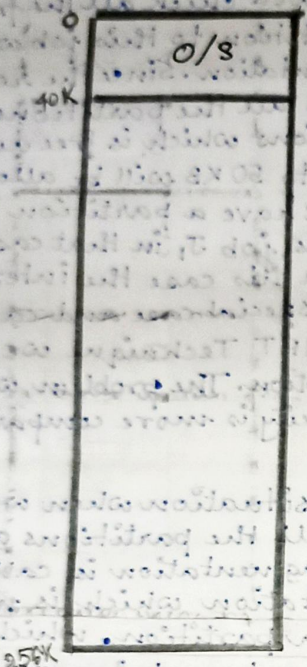
• To implement this MFT ~~to~~ technique, there are some information which is to be maintained. This information is used by memory management module for allocations of the partitions. Those informations are:

- What are the partitions in the main memory?
- What is the size of every partition?
- What is the starting location of every partition?
- What is the status of every partition? (i.e whether the partition is already allocated to a process or the partion is free)

~~This infor~~ These information is maintained in the form of a table which is known as partition allocation table.

There is another kind of memory management which is called Multiprogramming with variable number of tasks (MVT). Unlike in the previous case, in case of MFT where the no. of partitions were fixed, the maximum no. of partitions which can be contained in the main memory is also fixed. It is the no. of task which are there in the main memory that decides what is the length of the ready queue.

In case of MVT we donot have any such predecided partition. Rather the partitions are created of appropriate size when they are needed. The no. of partition may varried depending on size requirement of different jobs and depending upon the no. of jobs that we have to execute on the machine.



• In case of MFT or MVT, the memory will be primarily divided into two partitions.

- One is Operating System area (The monitor part of O/S).
- The other is user area.

Initially when there is no job loaded in the main memory we will be having a single user partition.

- Let us assume our memory size is 256K KB. Out of which 40KB is given to the Operating System. Suppose we have a number of jobs. Jobs with required memory is given below.

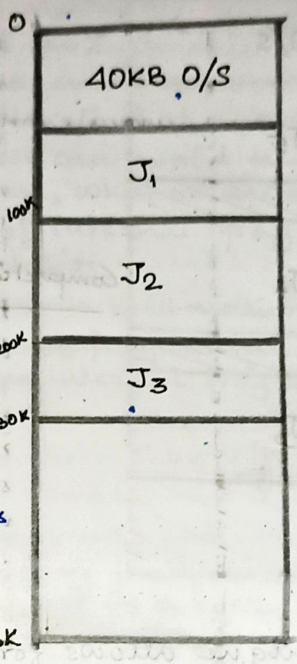
Job	Memory	Execution Time
J_1	60KB	10 TU
J_2	100KB	5 TU
J_3	30KB	20 TU
J_4	70KB	8 TU
J_5	50KB	15 TU
J_6	60KB	9 TU

- Let us use FCFS type of scheduling algorithm.
- The process which have arrived first will be loaded first in the main memory.

Let us see using MVT technique how these job will be allocated to the main memory.

- Let us assume we are having 256K of main memory. Out of which 40K is given to the OS. So we are having a remaining of 216K main memory as a single user space.

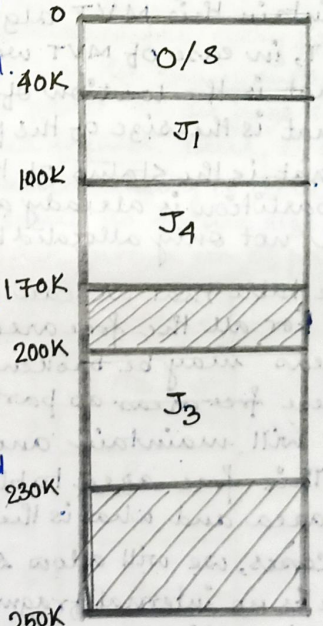
• First job J_1 will be placed in the main memory as we are following FCFS algorithm. Thus, in remaining 216K, we make one partition of 60K and put job J_1 in this partition. Here, the boundary will be (40+60)K = 100K ~~Now, we ask~~



• Now, we are having a free partition of 156K. Next J_2 will be placed in the memory. J_2 has a requirement of 100K memory. So, the remaining free partition is again further divided into two partitions.

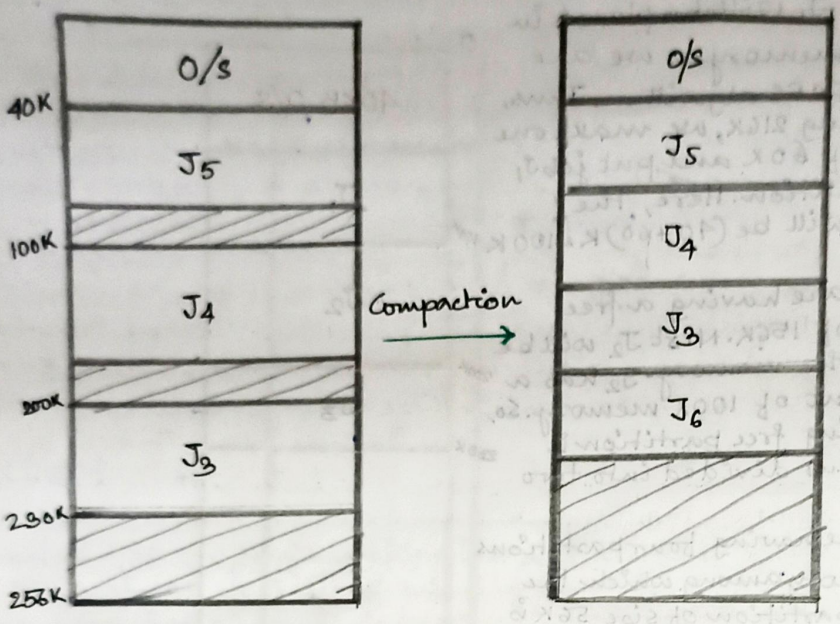
• Now, we are having four partitions in the memory among which the ~~4th~~ fourth partition of size 56K is still free. ~~Agai~~ Next J_3 will be placed and again this 56K will be divided into two partitions. Now, we are having five partitions among which a partition of 26K is still free. ~~But none of the rest jobs~~ Coming into the job queue, we are having three jobs J_4, J_5 and J_6. But none of these jobs can fit into the residual free partition. This 26K at least at this moment will remain as free partition. This is a fragmentation and this is an external fragmentation.

• After 5 TU job J_2 will complete its CPU burst. So, after 5 TU the situation is given below:



• After 5 TU J_2 will be completed. So the corresponding memory will be free. ~~Next job~~ So we will be having another 100K of free partition. Next job to be executed is J_4 which require 70K memory. So it will be ~~place~~ placed in place of J_2 with an internal fragmentation of 30K.

• After 5 TU J_1 will complete its execution. So, J_1 will be freed from the main memory and there will be a new free partition of 60K.



• MVT technique allows for memory compaction where we can put all allocated partitions at one end and all free partitions at another end and merge all those free partitions to get a larger free partition. If we go for such compaction then we will be having a free memory of 66K which can be divided into partitions one of 60K where we can place J_6 and we will be having a ~~remaining~~ remaining of 6K partition.

Now we have to find that what are the information we need to maintain this MVT algorithm.

• Like MFT, in case of MVT we need to know
  - What is the location of the partition?
  - What is the size of the partition?
  - What is the status of the partition? (This defines when the partition is already allocated. In that case the status will contain not only allocated but which job it is allocated).

This is one table that we can maintain for all the job which is allocated. For all the free areas, the partition is not yet decided. The free areas may be broken into several partitions. Thus, instead of calling these ~~free areas~~ as partitions, we will call these as free areas and will maintain another table which is known as free area table. This free area table will contain what is the location of the free area and what is the size of the free area.

In some cases, we will allow some internal fragmentation. Ideally there is no internal fragmentation as we have always created the partition depending upon the size that is required. So there is ideally no internal fragmentation. But when a partition size becomes very small (may be 10 bytes or 15 bytes or 100 bytes), if we maintain that small partition as a free area in that case we have to have an entry in the free area table and the no. of

bytes needed to maintain the information is more than the free area itself. More over the size of the free area is so small that no job can be fitted into it. In such cases it is better that instead of maintaining such a small amount of free area as a separate free area, we allocate that with one of the allocated partitions. By this we can avoid maintaining one entry in the free area table. Thus, what we are doing is we are allowing small amount of internal fragmentation along with the external fragmentation.

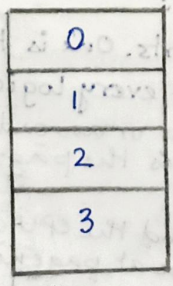
Thus, external fragmentation ~~is~~ will naturally arise in case of MVT technique. Internal fragmentation is ideally not present but we allowed some internal fragmentation to avoid the costing problem.

In the previous example it has been shown that J_6 is allocated by using compaction technique. But compaction is a costly technique. The moment you start compaction, until and unless the compaction is over, the CPU cannot perform any fruitful task. Though compaction is a possible solution, but its a costly solution.

As it is a costly solution, so we will try to see if there is any other alternative so that whenever the memories are available, sum of free memories is larger than the total requirement of the job so that the job can be fitted irrespective of the entire memory is available as a single partition or the entire memory is available as the multiple partition. This gives rise to the concept of paging technique.

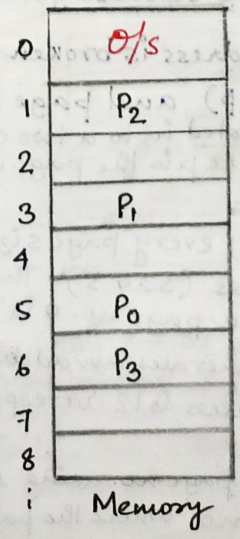
Paged Memory Management

In case of paged memory management, every process is divided into number of pages. Similarly, the main memory is divided into partitions, whose size is same as the page size. The partitions in the main memory are called frames. Thus, every job is divided into no. of pages, and the main memory is divided into no. of frames where the page size is same as frame size. Now the question arises that how do we break a task into a no. of pages.



Job

[Logical Address Space of a process]



Memory