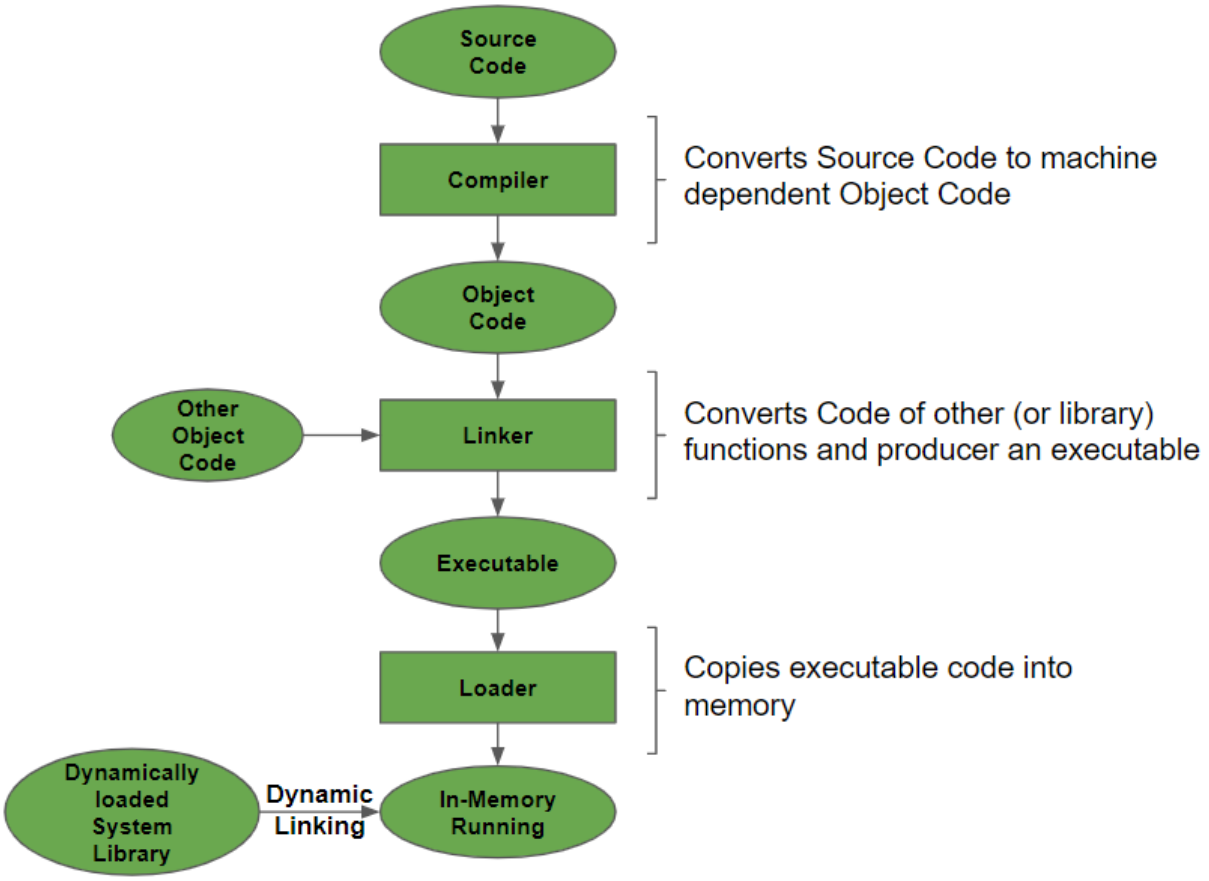
Programs Compiled and Run



Example:

```c

#include <stdio.h>

int main() {
    printf("Hello");
    return 0;
}

```

Run

Let's take this sample code to understand the various steps.
Step 1: The above code is the source code which is passed to the compiler.

Step 2: The compiler converts this source code to machine-executable object code.

Step 3: Then Linker is used to linking the functions mentioned inside the main function. Here "printf()" is a library function, whose code is attached to the main function using the linker. The linking can be done statically or dynamically.

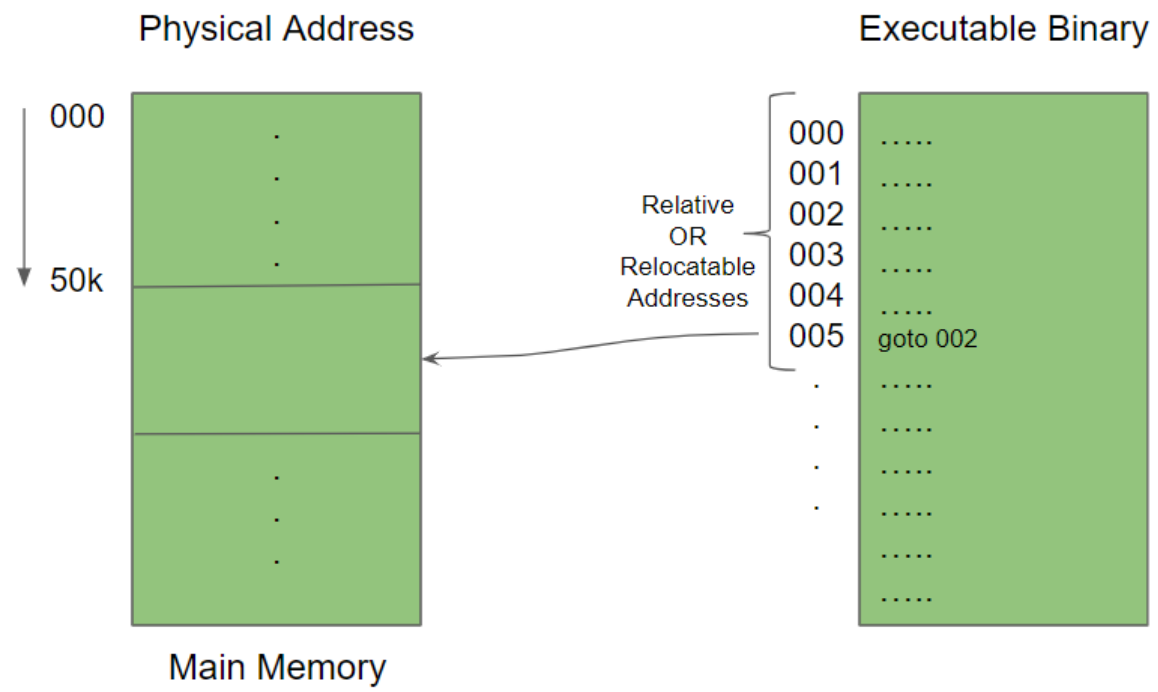
- Static Linking: In this, the code of the "printf" function is copied into the object file and then one executable file is generated and the execution is done line by line.

- Dynamic Linking: This refers to the linking that is done during load or run-time and not when the file is created. When one program is dependent on some other program rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when it's required.

Step 4: After the creation of the executable file the loader copies the executable code into the main memory.

Step 5: Finally the code is run in the main memory and while running dynamic linking can be made with other system libraries.

Address Binding



Address binding refers to the mapping of relocatable or relative address to physical addresses. The binary of code contains relative or relocatable addresses which are used to commute between various instructions. There can be various binary instructions ranging from goto, load, add etc. To execute this sort of binary files, they are needed to be loaded into the main memory. This main memory has physical addresses, and any processes stored in the main memory acquires a slot. The physical addresses of the main memory influence the binary addresses accordingly by readjusting them accordingly. For eg., when the binary file is loaded into the slot after 50k, they are readjusted to 50k, 50k+1, 50k+2, 50k+3 etc.

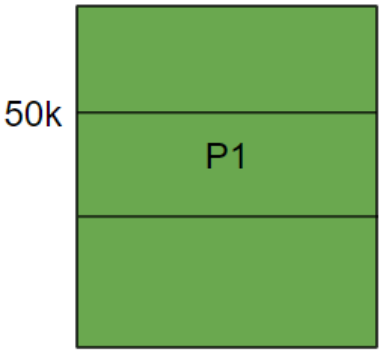
These bindings can happen at various stages of execution:

1. **Compile Time:** Its work is to generate a logical address(also known as virtual address). If you know that during compile time where the process will reside in memory then the absolute address is generated.
    - Instruction are translated into an absolute address.
    - It works with the logical address.
    - It is static.
    - Code is compiled here.

2. **Load time:** It will generate a physical address. If at the compile-time it is not known where the process will reside then relocatable address will be generated. In this, if address changes then we need to reload the user code.
    - Absolute address is converted to relocatable address
    - It works with a physical address

- It is static
- Instructions are loaded in memory

Problem with Load Time Binding is that once it is loaded into the main memory, it cannot be relocated. Let's look at this example:

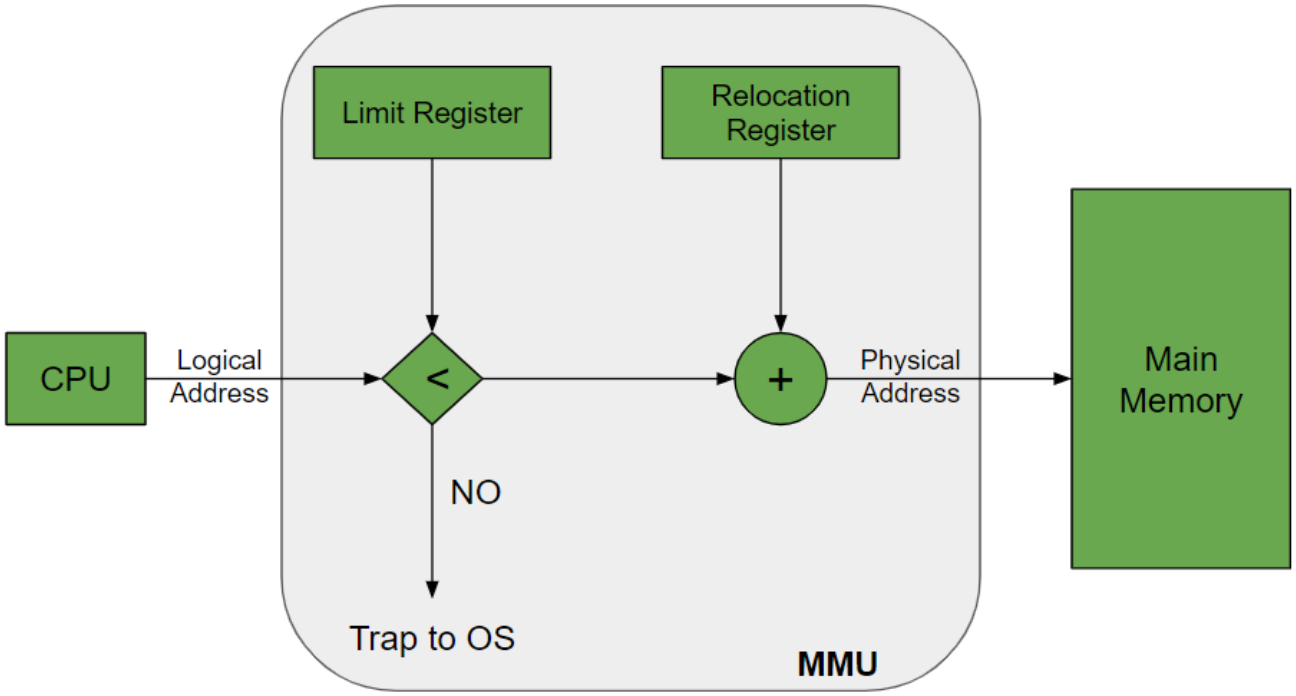


Suppose there is this process P1 which is loaded at 50k of the main memory. If it is waiting for some I/O, then the process is swapped out to make the memory available to some other processes. Now, when P1 completes its I/O work and it is swapped back into the main memory, the process is stored at some different address, suppose 150k. This cannot be implemented in Load Time binding.

3. **Run Time:** It is the most preferred binding technique in modern OS. The addresses generated by the CPU during the run time are logical addresses and not the physical addresses. This logical address is then converted into physical addresses using a memory-management unit(MMU). This is used in the process can be moved from one memory to another during execution(dynamic linking-Linking that is done during load or run time). Hardware support is required during run time-binding.
    - The dynamic absolute address is generated.
    - It works in between and helps in execution.
    - It is dynamic.
    - From memory, instructions are executed by CPU.

- Runtime Binding

Runtime Binding happens through hardware. The memory management unit in CPU has two registers namely, Limit Register and the Relocation Register. When a process is loaded into the memory, during context switching, the OS automatically loads these registers into the memory. The registers are loaded according to the slots in the memory.

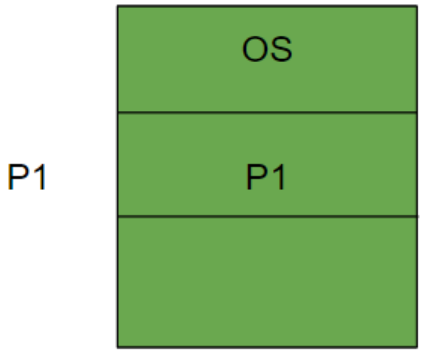


The process begins with the CPU generating the logical addresses. Then the process is sent to the memory management unit where the limit register first checks whether the generated logical address is within or beyond the memory allocation. If the address is beyond the limit, a trap is sent to the OS else the control flows to the Relocation register which relocates or adjusts the address and converts the logical address to physical address.

Why is the whole process implemented in hardware?
If a software implements this runtime binding, then during the context switching, when the logical address is generated, an OS system call will happen, which will convert the logical address to physical address. This will require Mode switching from process to OS context which will be done for every instruction, generated by the CPU. This will be a very slow process and hence the hardware is implemented to do the binding.

- Evolution of Memory Management

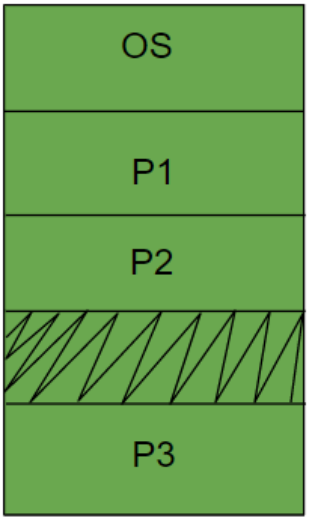
The memory management evolved in two phases:
Single Tasking Systems: This sort of system has a part loaded with OS. When one process arrives it is assigned to a slot and only after its completion can a second process be allowed a slot in the memory. Suppose in the diagram only after the execution of P1 completes can a process P2 be loaded into the memory.



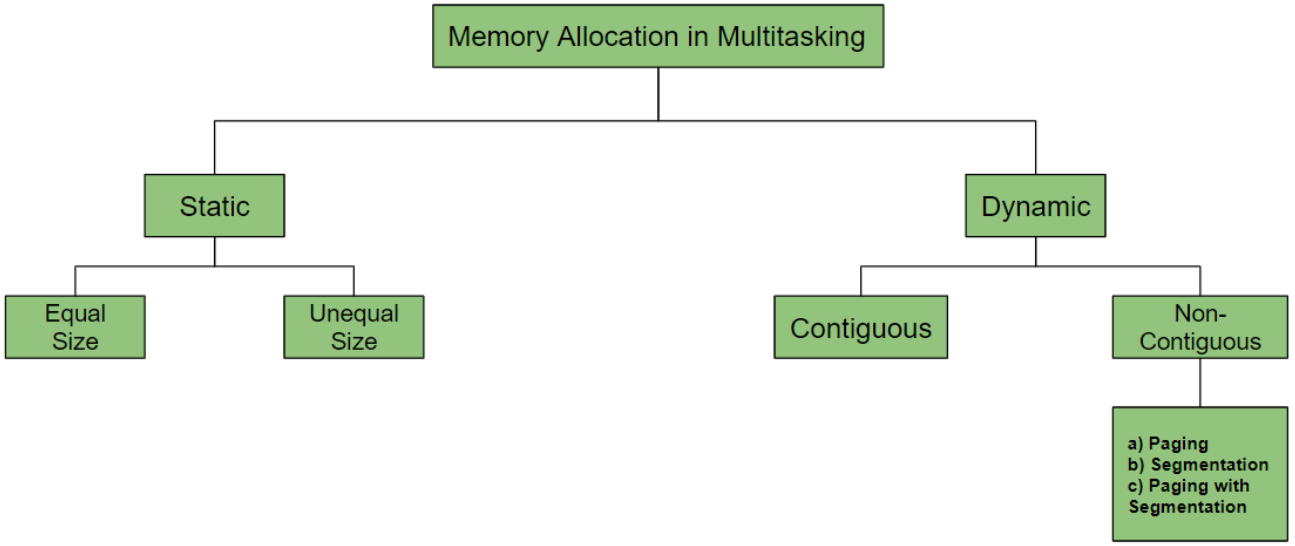
The problem with this system was that the context switching could not take place in a single-tasking system when multiple processes are sent into the memory. This results in a low degree of multiprogramming leading to the less efficient use of CPU.

Multitasking Systems: In this multiple processes can be taken by the memory at a single time and each process are run

one by one using efficient memory management techniques, leading to the higher degree of multiprogramming. In the below diagram we can see that P1, P2 and P3 are stored inside the memory at the same time and can be run simultaneously.

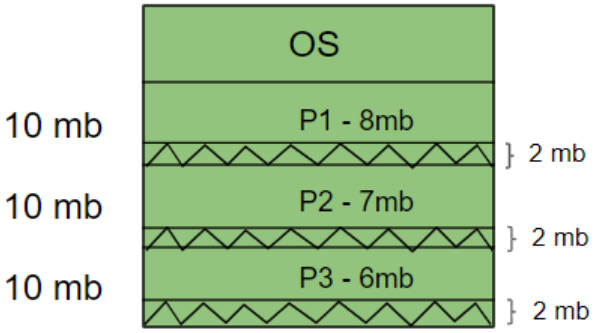


There can be various types of memory management allocation technique in Multitasking.



Static or Fixed Partitioning Technique: This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, a number of partitions (non-overlapping) in RAM is fixed but the size of each partition may or may not be same. As it is contiguous allocation, hence no spanning is allowed. Here partition is made before execution or during system configure. This can be of two types:

1. **Equal Size Memory Allocation:** In this technique, fixed memory size is allocated to each process. Let's look at the below example:



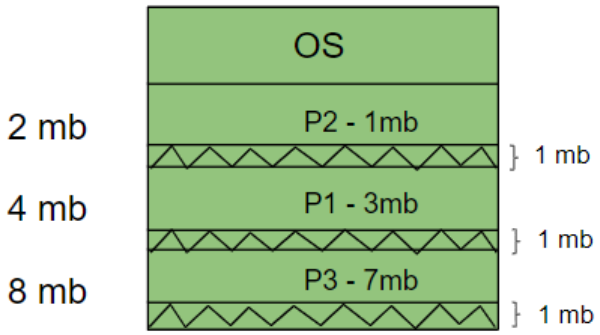
In this case, each process P1, P2 and P3 are allocated 10MB each irrelevant of how much memory each process requires. Suppose P1 had a demand of 8MB, P2 7MB and P3 6Mb. After each allocation, we can see that 2MB, 3MB, and 4Mb of memory is wasted from each slot. Although it adds up to 9 Mb, when a process P4 demands 5Mb of memory, it cannot be allocated, since they are available in different chunks. This phenomenon is referred to as external fragmentation. Let's understand fragmentation:
Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request.

- **External Fragmentation:** External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.

- **Internal Fragmentation:** Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

In the case of equal size memory allocation, internal fragmentation leads to external fragmentation. This results in high memory loss since we can observe that 9 MB of memory goes wasted.

2. **Unequal Size Memory allocation:** This technique is used to reduce memory loss in comparison to the equal size memory allocation. Let's look at the below example.



Suppose we want to allocate a process of 3 MB to the memory. Instead of allocating it to a fixed size of memory, a minimum MB slot is found and P1 is allocated to it. In our case, the 4MB slot is allocated to P1 consuming 3MB memory. If another process P2 arrives with a memory requirement of less than 2 Mb, then the first slot is allocated to it. Similarly, another process of 7Mb can take the 8Mb slot. So, we can see that although here also there is memory loss due to internal fragmentation and ultimately leading to external fragmentation, it is less in comparison to the Equal size memory allocation. Here there is a memory loss of total 3 MBs.

Advantages of Fixed Partitioning:

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

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

Disadvantages of Fixed Partitioning -

1. **Internal Fragmentation:** Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.

2. **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).

3. **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.

4. **Limitation on Degree of Multiprogramming:** Partition in Main Memory are made before execution or during system configure. Main Memory is divided into fixed number of partition. Suppose if there are $n1$ partitions in RAM and $n2$ are the number of processes, then $n2 \leq n1$ condition must be fulfilled. Number of processes greater than number of partitions in RAM is invalid in Fixed Partitioning.

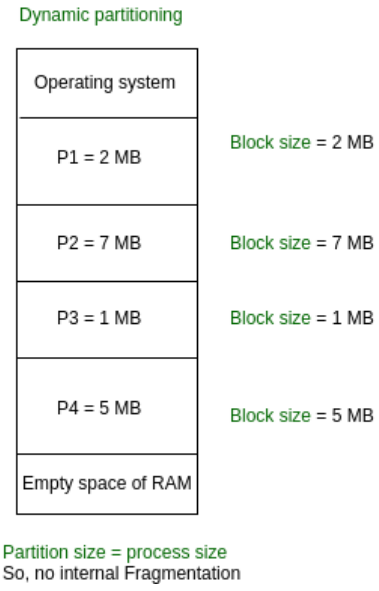
Dynamic or Variable Partitioning Technique:
It is a part of Contiguous allocation technique. It is used to alleviate the problem faced by Fixed Partitioning. In contrast with fixed partitioning, partitions are not made before the execution or during system configure. Various **features** associated with variable Partitioning-

1. Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure.

2. The size of partition will be equal to incoming process.

3. The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM.

4. Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.



There are some advantages and disadvantages of variable partitioning over fixed partitioning as given below.

Advantages of Variable Partitioning -

1. **No Internal Fragmentation:** In variable Partitioning, space in main memory is allocated strictly according to the need of process, hence there is no case of internal fragmentation. There will be no unused space left in the partition.

2. **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.

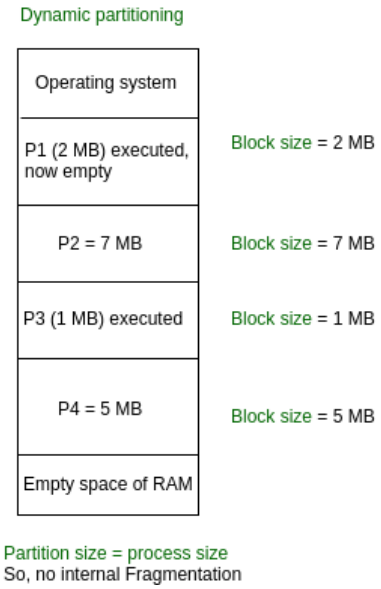
3. **No Limitation on the size of the process:** In Fixed partitioning, the process with the size greater than the size of the largest partition could not be loaded and process can not be divided as it is invalid in contiguous allocation technique. Here, In variable partitioning, the process size can't be restricted since the partition size is decided according to the process size.

Disadvantages of Variable Partitioning -

1. **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.

2. **External Fragmentation:** There will be external fragmentation inspite of absence of internal fragmentation.

   For example, suppose in above example- process P1(2MB) and process P3(1MB) completed their execution. Hence two spaces are left i.e. 2MB and 1MB. Let's suppose process P5 of size 3MB comes. The empty space in memory cannot be allocated as no spanning is allowed in contiguous allocation. The rule says that the process must be contiguously present in main memory to get executed. Hence it results in External Fragmentation. These spaces are called holes.



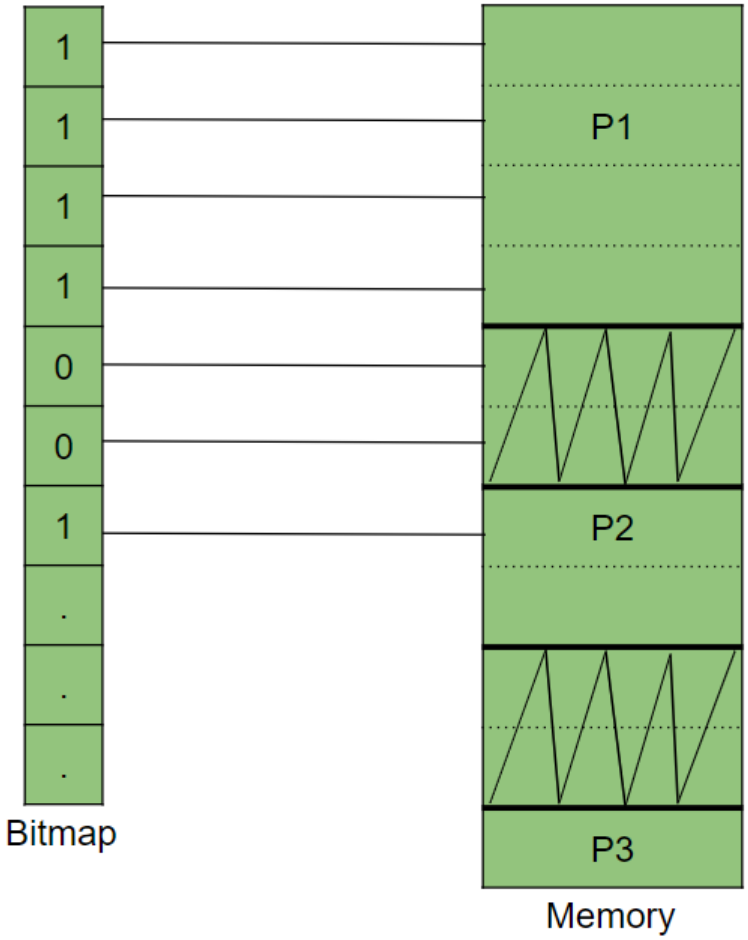
   Now P5 of size 3 MB cannot be accommodated in spite of required available space because in contiguous no spanning is allowed.

- Dynamic Partitioning

The major disadvantage of dynamic partitioning is that when a process completes its execution, it leaves the memory leaving a hole behind. Thus when multiple processes leave, similar holes are created and thus even if there is no internal fragmentation, external fragmentation is observed.

There are two methods to avoid such holes:

1. **Bitmap:** These are used to keep a track of the unit of memories that are occupied as 1 and the unallocated memories are assigned 0. So when a new process comes, it looks for the 0's to get a track of unallocated spaces and thus the problem is solved. This way memory management is made simpler and easier.

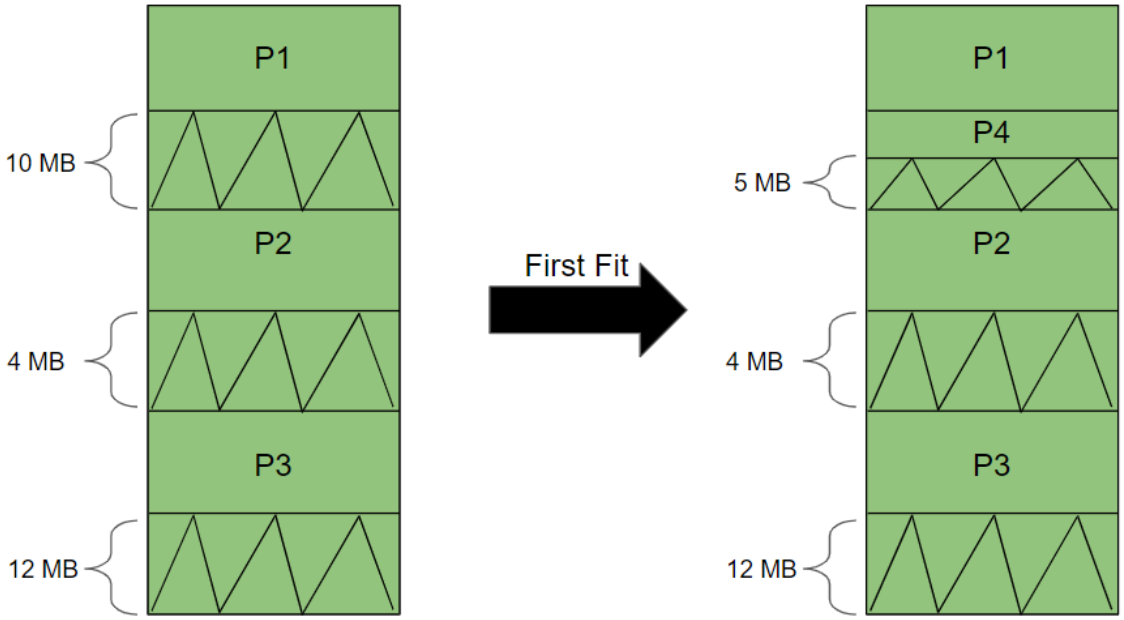


It also has a major problem. Bitmap requires a lot of space. Since for every unit of memory, a bit is needed in the bitmap to store the value. Eg., Suppose we have a memory unit of size 32 bit. So for every 32 unit, 1 bit of memory is needed to store the bit in the bitmap. Therefore for 32 bits, one bit is wasted. So 1 by 33 of the memory is wasted in the process.
If a larger size of the unit is taken, then this might lead to internal fragmentation. So both the processes are not feasible.

2. **Linked List:** In this the processes and the holes are connected using a doubly-linked list. So if a process ends in between, then the holes get connected to form a bigger hole. In the below diagram if P2 completes its execution, then both the H combines and a greater hole is generated, which can accommodate any incoming process of relevant size. But, there is a catch of how to allocate memory as in what process should get which holes. We have various strategies to do so. Let's discuss:

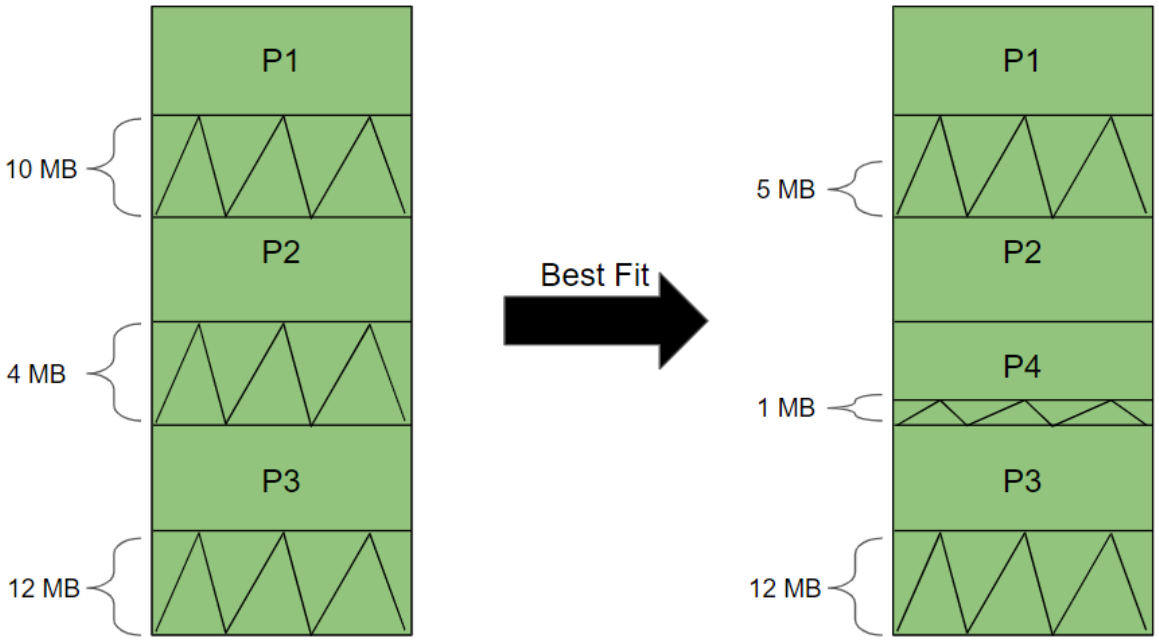
   In Partition Allocation, when there is more than one partition freely available to accommodate a process's request, a partition must be selected. To choose a particular partition, a partition allocation method is needed. A partition allocation method is considered better if it avoids internal fragmentation.

   Below are the various partition allocation schemes :
   1. **First Fit:** In the first fit, the partition is allocated which is first sufficient block from the top of Main Memory. Let's look at the example below:



Suppose in this setup a process P4 of size 5 MB makes a request. In First Fit, the linked list is traversed from the top of the memory and wherever, a size of 5 MB or more is available, the process gets allocated. In this case, the P4 gets allocated in the 10 MB hole leaving another hole of 5 MB. Now if another request of 3 MB is requested then it gets stored in this 5 MB hole. Again if a 10 MB request arrives, then the linked list is traversed from the top and since only the 12 MB hole can accommodate the request, it gets allocated here.

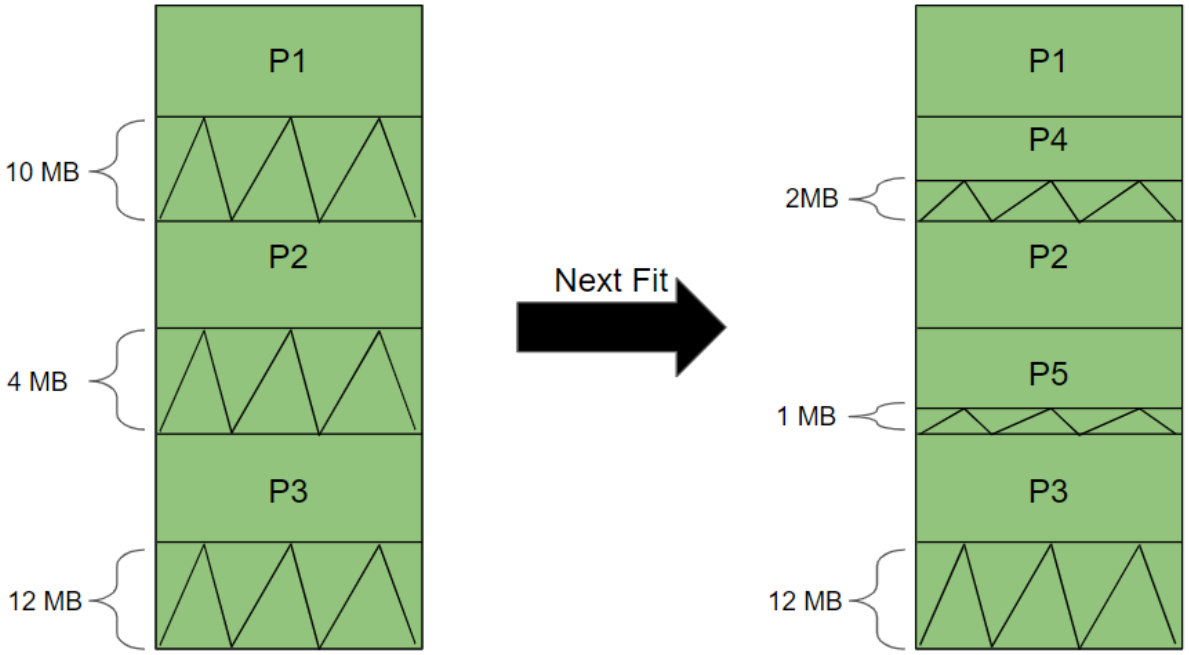
2. **Best Fit:** Allocate the process to the partition which is the first smallest sufficient partition among the free available partition. Let's look at the example below:



Suppose in this setup a process P4 of size 3 MB makes a request. In Best Fit, the linked list is traversed from the top of the memory and instead of assigning the process to wherever the sufficient memory is available, it keeps a track of the minimum block which can accommodate the process. In this case the P4 with 3 MB is assigned to the 4 MB block even if 10 MB block is first encountered. It has a few disadvantages:

- Every time a process makes a request, the whole linked list is needed to be traversed, thus this approach has greater time complexity.
- Various small memory holes are created which remains unused as this could not be assigned to any processes.

3. **Next Fit:** Next fit is similar to the first fit but it will search for the first sufficient partition from the last allocation point. Let's look at the example below:



Suppose in this setup a process P4 of size 8 MB and a process P5 of size 3 MB makes a request. At first, the linked list is traversed from the top of the memory and wherever, a size of 8 MB or more is available, the process gets allocated. In this case, the P4 gets allocated in the 10 MB hole leaving another hole of 2 MB. When the next request P5 of 3 MB arrives then, instead of traversing the whole list from the top, it is traversed from the last stop i.e., from P4 and process P5 gets allocated into the 4 MB block.

4. **Worst Fit:** Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory.

After doing some analysis, it has been found out that the First fit is the best approach since the best-fit algorithm needs to traverse the whole list every time a request is made.