Memory Management

Main Memory is a volatile array of words or bytes, each with its own unique address. Memory is shared by the CPU and I/O devices. The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be at least partially in main memory during execution. Program must be brought into memory and placed within a process for it to be executed. The CPU fetches instructions from memory according to the value of the Program Counter (PC). 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 (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). You may already know how a program generates a memory address and role of Memory Address Register and Memory Buffer/Data Register. Here we will explain how symbolic memory addresses are mapped to actual physical addresses, and how and when logical addresses are translated into physical addresses. Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.

The part of Operating System which manages memory activities is Memory Management Module.

- It Keeps track of which parts of memory are currently being used and by whom.
- Decides which processes to load when memory space becomes available.
- Allocate and de-allocate memory space to processes as required.

Memory Management module is responsible for relocation, protection, sharing, logical organization, and physical organization of memory. For all these activities different approaches and algorithms are used. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system.

In a single programming system, main memory is divided into two parts: one part for the operating system and other part for the program currently being executed. Normally operating system resides at lower addresses (0 – N-1) and the program resides from N address (memory location). In a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes to have a separate address space for each process. For different processes separate memory spaces, must be protected to ensure that the process can access only their legal addresses. This protection can be provided by using two registers, usually a base and a limit register.
The base register holds the smallest legal physical memory address; the limit register specifies the size/range of program. Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users’ memory results in a trap to the operating system, which treats the attempt as a fatal

error. To make sure that programs are not accessing operating system area, in most of the computer systems operating system is loaded at lower address.

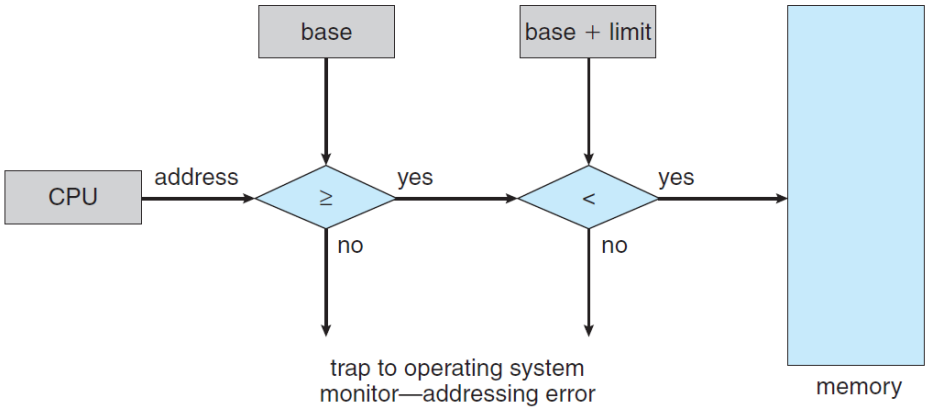


Figure-1.

Address Binding and Address Mapping/Translation

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available. Most systems allow a user process to reside in any part of the physical memory available at load time.

In most cases, a user program goes through several steps—some of which may be optional before being executed. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically binds these symbolic addresses to relocatable addresses (such as "16 bytes from the beginning of this module"). The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 58016). Each binding is a mapping from one address space to another. The binding of instructions and data to memory addresses can be done at any step along the way:

Compile time: If it is known at compile time where the process will reside in memory, then absolute code can be generated. For example, if it is known that a user process will reside starting at location N, then the generated compiler code will start at N and extend up from there. The MS-DOS .COM-format programs are bound at compile time. Only disadvantage is that if, at some later time, the starting location changes, then it will be necessary to recompile this code.

Load time: At compile time it is not known where the process will reside in memory, then the compiler must generate relocatable code. Final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a *logical address*, whereas an address seen by the memory unit (loaded into the Memory-Address Register of the memory) is commonly referred to as a *physical address*. The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this situation, at sometimes logical address is referred as a virtual address. The set of all logical addresses generated by a program is a logical address space. The set of all physical addresses corresponding to these logical addresses is a physical address space. The run-time mapping from virtual to physical addresses is done by a hardware device called the Memory Management Unit (MMU). There are different methods to accomplish such mapping (will discuss later) and simplest mapping is a generalization of the base and limit register scheme. The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory. The user program never sees the real physical addresses. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range B + 0 to B + max for a base value B). The user program generates only logical addresses and thinks that the process runs in locations 0 to max. However, these logical addresses must be mapped to physical addresses before they are used.

Program Size, Address Range, Address Bits

Before we discuss, memory allocation schemes, better to clarify address space range, and address bits required for a given and available size of memory. You may recall address bits being used and passed to Memory Address Register to fetch a memory location. Memory Size, Address Range, and Address Bits are related and using a simple manipulation you can know one from the other. Following table explains the relationships.

Memory Size	**2^N**	**Address bits**	**Address Range**
16B	2^4	4	0 – (2^4 - 1) 0 - 15
64B	2^6	6	0 – (2^6 - 1) 0 – 31
1KB	2^{10}	10	0 – (2^{10} - 1) 0 - 1023
64KB	?	?	0 – ($2^?$ - 1)
IMB	2^{20}	20	0 – (2^{20} - 1)
16MB	?	?	?
IGB	2^{30}	30	0 – (2^{30} - 1)
64GB	?	?	?

Table-1: Memory size, address bits and address range.

For memory of 16 bytes, which is 2^4, we require 4 bits to address 0 – 15 bytes. Similarly for memory of 1 KB, which is 2^{10}, we require 10 bits to address 0 – 1023 bytes.

Memory Addressing

Address bits being represented and manipulated internally in binary form, we can for understanding and simple manipulation can use Hexadecimal representation for addresses. We can use simple table for 0 – 15 locations represented in Decimal, Binary and Hexadecimal format.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table-2: Decimal number 0 -15 in binary and Hexadecimal

By using the Table-1 and Table-2 we can easily convert a memory address into Hexadecimal format by making groups of 4 bits as explained below.
For a memory of 1KB, which is 2^{10}, we require 10 bits to address 0 – 1023 bytes.
Maximum memory location is 1023, where all bits are '1' '1111111111'
Manually manipulating and converting this string of binary values is not easy, and there is a chance of missing bit values. We can easily convert this binary string into Hexadecimal which is more readable. We start from least significant bits and make groups of 4 bits as listed below.

'1111111111' → 11- 1111-1111

There may not be enough bits to make the last group of 4 bits, in this case we can add '0' as most significant bits to make a group of 4 bits as listed below.

'1111111111' → 0011- 1111-1111

These groups of 4 binary values can easily be converted into Hexadecimal values with the help of Table-2 (which you must have remembered by now to manually manipulate memory address). We can simply write hexadecimal values, which will be memory address in Hexadecimal. For memory location '1023' address in Hexadecimal will be '3FF'.

We will be using this address manipulation for different memory allocation schemes to compute physical address from logical address. So better to clearly understand this process and do some exercises for address computation.
Let us compute memory address for memory location '512'. In computer, memory addressing always start from '0'. Memory bits required to address '512' ?
Have a look at Table-1, for memory size of '512' bytes, we need 9 bits, and address range of that memory will be 0 – 511(means all 9 bits on →'111111111' → '1FF'). We have to compute memory address of location '512', which is one location more than location '511'. We can compute it by adding '1' to memory address ('111111111') which is '1000000000' in binary and when making groups of 4 for Hexadecimal it is '200'
Carefully look the pattern of binary string, when we add one binary value, where all least significant bits are '1', and most significant bit is '0', then most significant bit becomes '1' and least significant bits become '0'

'0011' add '1' and it becomes '0100'

Similarly for Hexadecimal where maximum value is 'F', if add '1' when least significant values are 'F', then after addition, most significant value is added by '1' and all least significant values become '0'

'1FF' add '1" and it becomes '200'

You can easily compute memory address in Binary and Hexadecimal by factorizing the value to nearest power of 2, and then by adding or subtracting small number in binary or Hexadecimal form.

Let us do a simple exercise, to compute memory address and range for 48B.
Not a power of 2. (However 48 = 32 + 16).

32 → 2^5 → 5 bits → 0 - 2^5 -1 → '0' - '11111' → '00' - '1F' (0 – 31)
16 → 2^4 → 4 bits → 0 – 2^4 -1 → '0' - '1111' → '0' - 'F' (0 – 15)

To address 64bytes, we need 6 bits (all '1" for maximum memory location).
To address 48th byte, we will need 6 bits, however not all bits will be '1'.
We can compute binary addresses by performing addition on binary or Hexadecimal values of addresses. We already computed, 31st byte address as '11111' and '1F' in binary and hexadecimal respectively. By adding '1' location we can have address of 32nd location which is '100000' and '20' in binary and hexadecimal respectively. To compute address of 48th location, we add 16 to 32 for decimal computation. Similarly, we can add binary and hexadecimal equivalent of 16 to binary and hexadecimal equivalent of 32.

'100000' + '10000' → '110000' (Binary)
'20' + '10' → '30' (hexadecimal)

Similarly, we can compute address of 40th location by adding 8 to 32 for decimal computation and by adding binary and hexadecimal equivalent of 8 to binary and hexadecimal equivalent of 32.

‘100000’ + ‘1000’ → ‘101000’ (Binary)
‘20’ + ‘08’ → ‘28’ (hexadecimal)

You can verify by factorizing and adding values to compute address for memory location 56th byte and 6KB.
6KB not a power of 2. (However 6KB = 4KB + 2KB)

4KB → 2^{12} → 12 bits → 0 - 2^{12} -1 → ‘111111111111’ → ‘FFF’
2KB → 2^{11} → 11 bits → 0 - 2^{11} -1 → ‘11111111111’ → ‘7FF’

To address 8KB, we need 13 bits (all ‘1” for maximum memory location).
To compute maximum memory location of 6KB, we can add maximum value of 2KB to maximum value of 4KB. Since memory address is starting from ‘0’ location we have to add ‘1’ while adding additional addresses to get the base address of the next location.

‘FFF’ + ‘1’ + ‘7FF’ = ‘17FF’

Compute address of 6KB in binary format

Address Translation and Memory Protection

Before we discuss memory allocation schemes, we explain memory allocation, memory protection, and logical to physical address translation, by using a simple memory size of 64KB, where lower 4KB is being used by the operating system. Memory is divided in two partitions. 4KB (0 to 4KB-1) for operating system and 60KB (4KB to 64KB-1) for user processes as shown in Figure-2.

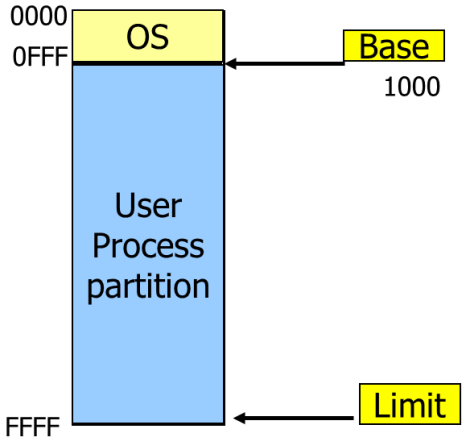


Figure-2: Physical memory addresses.

Address values are in Hexadecimal. We can use Base and Limit Registers to compute Physical address for a user process and can also see how operating system can protect itself and restricts a user process. We can use memory system in Figure-2 to place a user process of memory size 16KB. After placement of that program the memory system of Figure-2 will look like as shown in Figure-3.

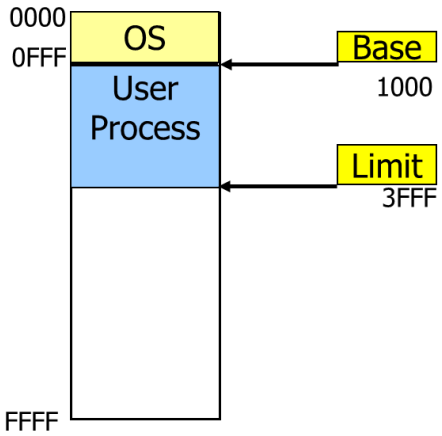


Figure-3: Physical address space of OS and user process.

From Figure-3 we can see that operating system can protect itself by comparing logical address of the process with the base register, it should be greater than or equal to base register value (in this case 1000). This basically is done at address translation time as shown in Figure-1. In the example of Figure-2, we have a process of memory size 16KB. We can easily convert a logical address less than 16KB by using base and limit register as shown in Figure-1. If we have a logical address '2000' which is 8KB location, it will be converted into physical address by adding this address to Base value ('1000' + '2000' = '3000'). If the process generates a logical address beyond its limit of 16KB, a trap will be generated as depicted in Figure-1.

Single Partition: Contiguous Memory Allocation

First, we discuss simple two partitions (sometimes called single partition, since a single program is memory resident at any given time) contiguous memory allocation, where memory is divided into two parts, one for the operating system and other for user processes. This allocation scheme is very simple, operating system keeps track of the memory status, free or allocated. This can be done by using a flag or register, along with the memory size available for user processes. Placement is simple, if memory required by a program is within limits of available memory and is free, memory is allocated, and base and limit registers are updated with the information. When the process is completed, memory status as of 'free' is maintained and memory is allocated to the next program in queue. At any given time only one user process is memory resident and processes having a memory requirement less than or equal to available memory for user processes can be allocated memory. This was first memory management scheme used in batch operating system.

Resource's utilization in this scheme is not good. At any given time only one process is memory resident, and a program with small memory requirement is allocated all the memory available. It means for all those programs with small memory requirements memory is underutilized. For a compute bound process, IO devices are underutilized during the lifecycle of that process. For an IO bound process, CPU is underutilized during the lifecycle of that process. No special hardware required, memory is being protected by using base and limit registers along with address translation.

Multiple Partitions: Contiguous Memory Allocation

In single partition, we noticed that resources are underutilized. To overcome resources underutilization, multiple partitions are created to place more than one programs in memory to improve resources utilization.

Fixed (Static) Partitions

In this scheme, memory is divided into fixed size partitions (all partitions are of the same size or different partitions of different sizes). It is fixed in a sense that partitions are created at operating system generation time. It means once created partitions (number and size) cannot be changed, so also called static. Operating system keeps track of status of all partitions (Free/Allocated, Size, and Base Address). Operating system can maintain a list of partitions (base address, size, status: Free/Allocated). When all the partitions are of the same size, placement scheme is simple. Only restriction is that the program to be placed in memory must have memory requirement less than or equal to partitions size. Memory management module will search the list to find the first free partition, allocate it, update the list with partition status being changed to Allocated, and update base and limit register information in the Process Control Block. By using memory of 64KB of Figure-4, where operating system uses 4KB, rest of the 60KB is divided into equal size partitions of 4KB each. Memory partitions along with base addresses of a few partitions is shown in Figure-4.

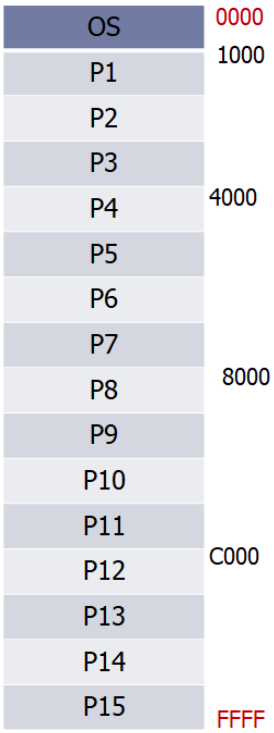


Figure 4: Fixed Partitions of 4KB each

Assume at a given time, free partitions are (P4, P8, and P12). The memory management module will place a program (Pa, memory requirement 2KB) by searching the list and finding first free partition, which in this case is P4. So, program Pa is allocated memory of partition P4 (Base address: 4000, and Limit: 2KB). When Pa is allocated CPU, Base and Limit Registers values will be updated at context switch time and process status from Ready to Running state. Pa's logical address will be generated by using base and limit

registers. If Pa's logical address generated at any given time by CPU is 1KB (0400) location, it will be translated into physical address (4000 + 0400 = 4400).
Though fixed partitions are of 4KB size, Pa required 2KB, a whole partition of 4KB is allocated. Unutilized 2KB created a memory fragment which will never be used during the lifecycle of Pa. The unutilized memory is internal fragmentation, because it creates a fragment of memory internal to fixed partition. As for CPU and IO utilization, at any given time more than one programs (as per number of partitions) are memory resident, it has better utilization of CPU and IO devices due to multiprogramming. Fixed partitions memory allocation was used in early-stage multiprogramming operating systems. Fixed partitions of equal size restrict program size to be a fixed size. Different application programs are of different sizes, so fixed partitions memory allocation scheme with different partitions sizes was implemented in different multiprogramming operating systems.

In this scheme, memory is divided into different size partitions. By using memory of 64KB of Figure-2, where operating system uses 4KB, rest of the 60KB is divided into different size partitions (2 partitions P1-P2 of 4KB each, 2 partitions P3-P4 of 8KB each, and 3 partitions P5-P7 of 12KB each, Memory partitions along with base addresses of a few partitions is shown in Figure-5.

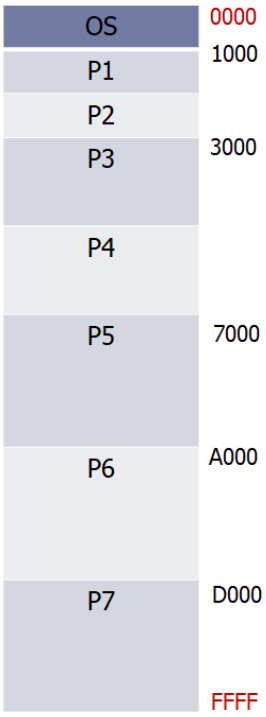


Figure-5: Fixed Partitions of different sizes

When fixed partitions are of different size, placement strategy must be designed so that to allocate a partition which will have minimum internal fragmentation. Two schemes were explored, first to have a single queue of programs for all partitions, another scheme is to have a separate queue for each partition size, where programs of almost same size will compete. Both schemes have merits and demerit. Placement schemes which were used in different multiprogramming systems are First Fit or Best Fit approach. In first fit, memory management module searches first free partition, which is large enough to accommodate the program. In this scheme free partitions list is maintained on base addresses. In best fit, management module searches smallest free partition, which is large enough to

accommodate the program. In this scheme free partitions list is maintained on partition sizes (sorted in ascending order). First fit is fast, however has large internal fragmentation, whereas best fit has sorting overheads, it has smallest internal fragmentation.

We can use Figure-5 example, to explain first fit and best fit approach allocation. Program to be placed is Pb with memory requirements of 7KB. Assume at a given time, free partitions are (P3:8KB, P5:12KB, and P7:12KB). For first fit, free partitions list is managed on higher addresses as shown in Table-3.

Base Address	Partition	Size
D000	P7	12KB
7000	P5	12KB
3000	P3	8KB

Table-3: Free partitions list.

In first fit approach list/table is searched from the top of the list, and first free partition which is large enough to accommodate the program Pb is allocated. In this case P7 is the partition, which is allocated to program Pb, which required 7KB with 5KB of internal fragmentation.

For best fit, free partitions list is sorted in ascending order on free partitions size as shown in Table-4.

Size	Partition	Base Address
8KB	P3	3000
12KB	P5	7000
12KB	P7	D000

Table-4: Free partitions list sorted by size.

In best fit approach list/table is searched from the top of the list, and first free partition which is large enough to accommodate the program Pb is allocated. In this case P3 is the partition, which is allocated to program Pb(7KB) with internal fragmentation of 1KB.
In fixed partitions scheme of variable sizes, program size restriction is that of largest partition. Program size varies and a program larger than largest partition cannot be placed in memory. Main drawback of this scheme is internal fragmentation, which runs into poor utilization of memory. To overcome this internal fragmentation, variable partitions scheme was designed.

Variable (Dynamic) Partitions

Initially one large partition of memory, and memory is allocated as per programs memory requirements. After allocation and de-allocation of memory to different programs, there will be variable number of partitions of variable sizes. In case of fixed partitions, the list size of partitions is fixed, whereas in variable partitions, the list size is dynamic, initially one free partition, one program allocated, list size is two (one partition allocated, one free) and as programs are allocated memory, partitions number dynamically increased. As for deallocation after programs completion, adjacent free partitions are merged to create a large

partition. After deallocation of all partitions, there will be one large partition as of at allocation starting time. Since at any given time, number of partitions and their sizes will be variable, different allocation schemes are implemented in different operating systems. In variable partitions approach, memory deallocation is not that simple since adjacent free partitions must be merged to create a large free partition.
Following Figures show status of memory partitions during different times.

Initially one free partition.

After allocation of P1 to P6 partitions of different sizes as per memory requirements of different programs, and one free partition.

After P4, P5, and P1, P2 are deallocated.

After P6 is deallocated.

Now when P3 is deallocated, adjacent free partitions are merged, and we will have one free partition just like when we started this example.

Placement Strategies/Approaches
Just like in fixed partitions of different sizes, we describe first fit and best fit approach for memory allocation. In variable partitions, multiple placement approaches were explored and implemented. Commonly used were first fit and best fit. Variations of first fit called next fit and of best fit called next fit were explored.
First fit: Allocate first partition that is big enough to accommodate the program, starting from the top of the list of free partitions.
Best fit: Allocate smallest size partition that is big enough to accommodate the program; must search entire list, unless list of free partitions is ordered by size in ascending order. Produces the smallest leftover partition, which may not be used, thus creating external fragmentation.
Next fit: A variation of first fit. Allocate first partition that is big enough to accommodate the program, starting from where the last allocation was made.
Worst fit: A variation of first fit. Allocate the largest size partition; must also search entire list, unless list of free partitions is ordered by size in descending order. Produces the large size leftover partition which may be large enough to accommodate a program.

We can explore working of these approaches by using an example, where memory is 256MB, OS uses 40MB, remaining 216MB is used to allocate and deallocate following job sequence. J1(60MB), J2(16MB), J3(30MB), J4(40MB), J5(70MB), J1 terminated, J4 terminated, J6(40MB), J7(50MB).

Initial state of memory

OS	216MB

After allocation of memory to J1- J5.

OS 40MB	J1 60MB	J2 16MB	J3 30MB	J4 40MB	J5 70MB

After termination of J1 and J4.

OS 40MB	60MB	J2 16MB	J3 30MB	40MB	J5 70MB

First Fit: In this approach, free memory partitions list is searched and first free partition which is large enough is allocated. List of free partitions is printed below.

Base Address	**Size**
XXXX	60MB
YYYY	40MB

First free partition is of 60MB, so 40MB is allocated to J6, and another partition of 20MB is created and new list of free partitions is:

Base Address	**Size**
XXYY	20MB
YYYY	40MB

After allocation of memory to P6(40MB)

OS 40MB	J6 40MB	20MB	J2 16MB	J3 30MB	40MB	J5 70MB

Now J7(50MB) cannot be accommodated, though total free memory is 60MB, however fragmented in two partitions of 20MB and 40MB.

In case of our example being discussed, for all schemes, memory partitions status is the same as of after termination ofJ1 and J4. So, for all other schemes we will describe allocation of J6 and J7.

OS 40MB	60MB	J2 16MB	J3 30MB	40MB	J5 70MB

We can apply **Best Fit** approach to allocate memory to J6(40MB). In this approach free memory partitions list (sorted in ascending order of partition size) is searched and first free partition which is large enough is allocated. List of free partitions is printed below.

Base Address	Size
YYYY	40MB
XXXX	60MB

First free partition is of 40MB, so 40MB is allocated to P6, and new list of free partitions is:

Base Address	Size
XXXX	60MB

After allocation of memory to J6(40MB) memory status is printed below.

OS 40MB	60MB	J2 16MB	J3 30MB	J6 40MB	J5 70MB

Now we see placement of J7(50MB). Using Best fit approach, it is allocated 50MB from the 60MB partition and after allocation memory status will be:

OS 40MB	J7 50MB	10MB	J2 16MB	J3 30MB	J6 40MB	J5 70MB

You can see that for a given memory size, for a specific job sequence using First fit, all jobs cannot be allocated memory due to fragmentation. While using Best fit, all jobs are allocated memory.

Now we describe Next fit and Worst fit, which are variants of First fit and Best fit respectively.
In Next fit, free partitions list is not searched from the top of the list, rather a pointer is maintained to the address where the last allocated is made. The list is searched from that pointer in a circular manner (when end of list is reached, search continues from the top of the list to the pointer location. Last allocation was for J5 (at ZZYY location). List of free partitions after last allocation is printed below, and pointer points to ZZYY location.

Base Address	Size
XXXX	60MB
YYYY	40MB
ZZYY	-

First free partition is of 60MB, so 40MB is allocated to J6, and another partition of 20MB is created and new list of free partitions is printed below, where pointer point to location XXYY.

Base Address	Size
XXYY	20MB
YYYY	40MB

After allocation of memory to J6(40MB).

OS 40MB	J6 40MB	20MB	J2 16MB	J3 30MB	40MB	J5 70MB

Now J7(50MB) cannot be accommodated, though total free memory is 60MB, however fragmented in two partitions of 20MB and 40MB.

In worst fit, free memory partitions list (sorted in descending order of partition size) is searched and first free partition which is large enough is allocated. List of free partitions is printed below.

Base Address	Size
XXXX	60MB
YYYY	40MB

First free partition is of 60MB, so 40MB is allocated to J6, and another partition of 20MB is created and new list of free partitions is printed below.

Base Address	Size
YYYY	40MB
XXYY	20MB

Memory status after allocation of memory to P6(40MB).

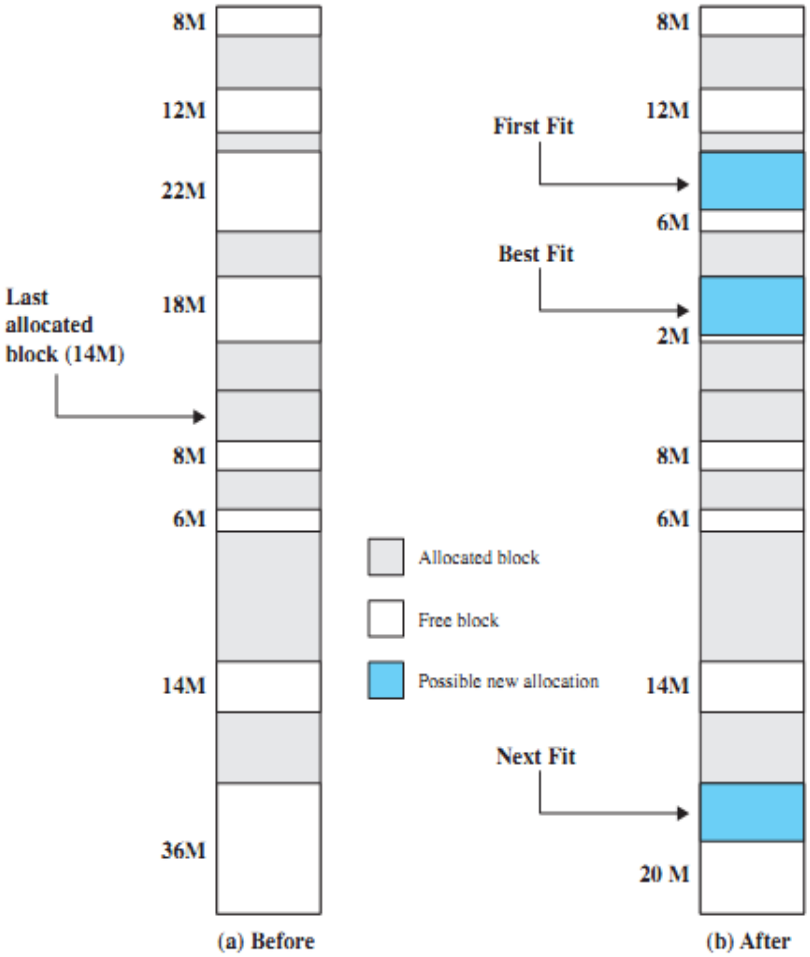
OS 40MB	J6 40MB	20MB	J2 16MB	J3 30MB	40MB	J5 70MB

Now J7(50MB) cannot be accommodated, though total free memory is 60MB, however fragmented in two partitions of 20MB and 40MB.

You can do following exercise to see working of First/Next fit and Best/Worst fit for the job sequence with a user partitions memory of 256MB.
Job Sequence: Job1 (140MB), Job2 (16MB), Job3(80MB),
Job1 terminate, Job3 terminate, Job4 (40MB), Job5 (128MB)

In variable partitions, memory is allocated as per process memory requirement. If the partition selected for process placement is slightly larger than the memory required, memory is allocated and for the remaining memory a partition of small fragment is created. This small fragment is not part of any other allocated partition and is external to partitions. In this scheme at any given time, there may be large number of external fragments of memory, which can be used for program placement. Main drawback of variable partitions approach is external fragmentation.

To summarize First/Next/Best/Worst fit working, following example is self-explanatory. You can hide, right side column of memory partitions and try to allocate memory of 16MB using First/Next/Best/Worst fit approach.

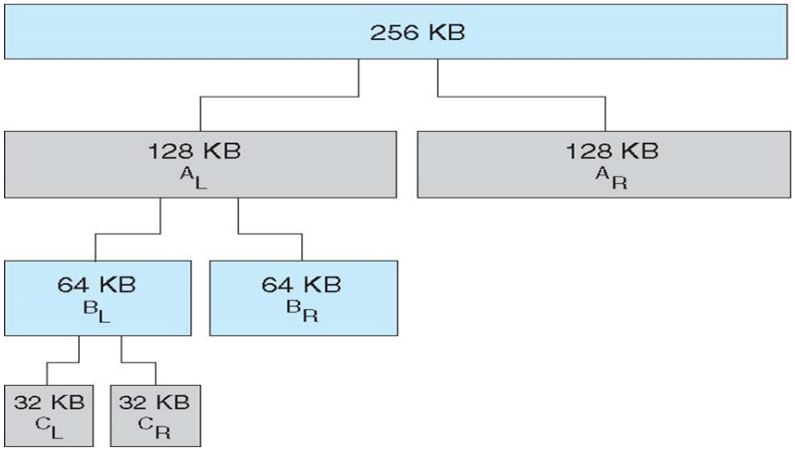


Buddy System

A fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes. A variable partitioning scheme is more complex to maintain and includes the overhead of compaction. An interesting compromise is the buddy system. In early operating systems the approach was explored, under the assumption that most of the programs memory requirement is in power of 2. If memory requirement is near to power of 2, then to create multiple partitions of power of 2 is fast and addressing of these partitions is simple.

In a buddy system, memory blocks are available of size 2^K bytes, $L \leq K \leq U$, where
2^L = smallest size block that is allocated.
2^U = largest size block that is allocated; generally, 2^U is the size of the entire memory available for allocation.
Memory allocated using power-of-2 allocation; Satisfies requests in units sized as power of 2.
To begin, the entire space available for allocation is treated as a single block of size 2^U. If a request of size **s** such that $2^{U-1} < \mathbf{s} \leq 2^U$ is made, then the entire block is allocated. Otherwise, the block is split into two equal buddies of size 2^{U-1}. If $2^{U-2} < \mathbf{s} \leq 2^{U-1}$, then the request is allocated to one of the two buddies. This process is repeated until the smallest block greater or equal to **s** is generated. Two buddies are coalesced whenever both of them become unallocated.

Following figure illustrate, how left and right buddies are created incrementally, until the required size partition is created. In this example available memory is of 256KB, and a process's memory requirement is 30KB (i.e. we need a partition of 32KB).



Working of buddy system memory allocation in partitions of power of 2 and merging of buddies is illustrated in Figure-9. Initially available memory is 1MB and partitions are created to place a memory request.

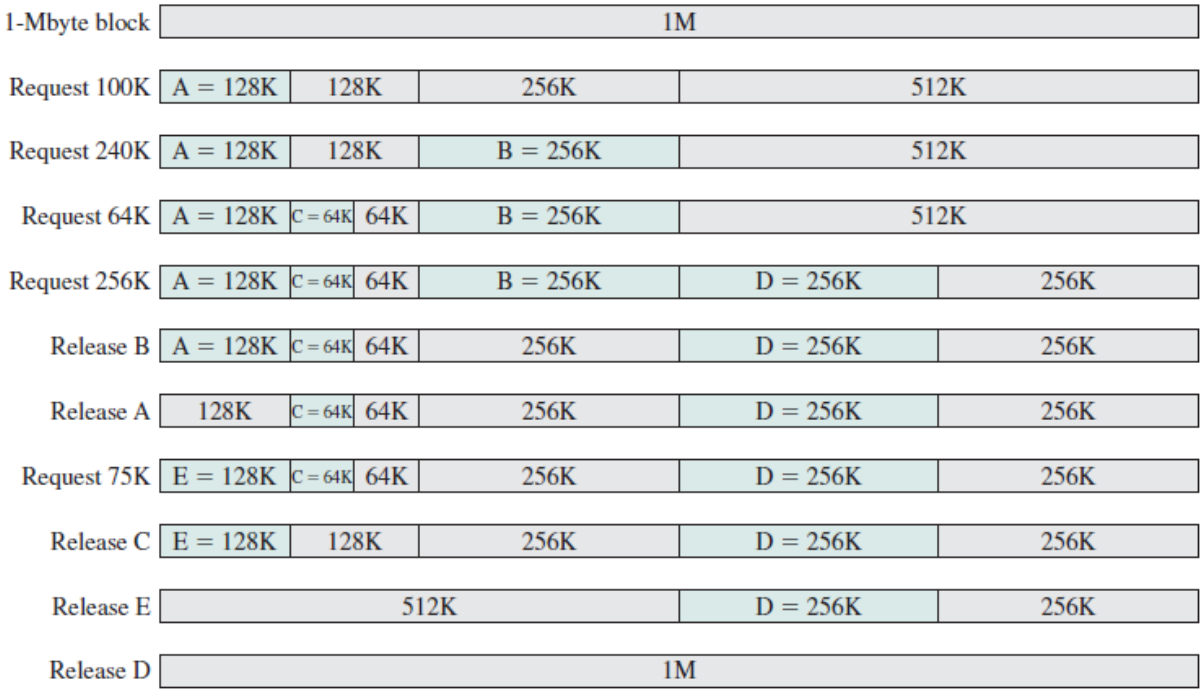


Figure-9: Buddy System for 1MB memory to dynamically create partitions.

Though buddy system is variable partitions scheme, it has internal fragmentation. As partitions are created in power of 2, any process having memory requirement less than power of 2 in size, is allocated memory in power of 2 size.

De-allocation

In variable partitions, deallocation of memory partitions is complex, since memory management module keeps track of the status of adjacent partitions and adjacent free partitions are merged to create a large free partition. All the activities of allocation and deallocation are dynamic. In Figure-8, four cases of memory deallocation are depicted.

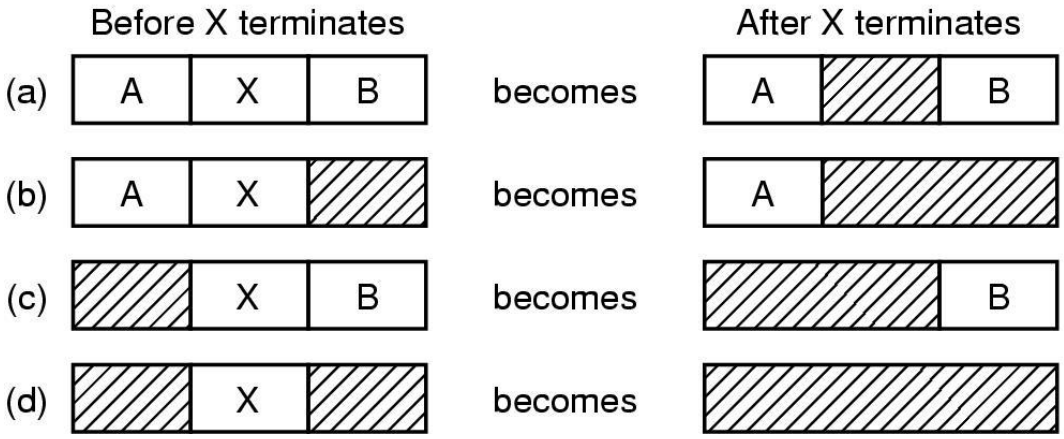


Figure-8

Fragmentation

Internal Fragmentation: Unused memory within partitions. Internal fragmentation occurs in variable partitions scheme.

External Fragmentation: Unused memory external to partitions. Total memory space exists to satisfy a memory request, but it is not contiguous. External fragmentation occurs in variable partitions scheme. Variable partitions scheme may have internal fragmentation, when after allocation of memory, the remaining memory is of a very small size, in this case rather than to keep a partition of that small size in the list better to allocate by keeping the partition of available size. The small amount memory not being used will create an internal fragment to that partition.

Compaction

To overcome external fragmentation issue, concept of compaction was explored, where memory contents are moved towards lower or higher addresses to create a large partition of all free fragments of memory. Compaction is possible only if relocation is dynamic and is done at execution time. Question arises, when to do compaction? if done at every allocation or deallocation of partition time, it has overheads, because when system is doing compaction, system is not doing any useful activity apart from moving data within memory. Compaction could be performed at allocation time, when total available memory in different fragments is large enough to accommodate a process. How to do compaction is more complicated, since aim is to create a large contiguous memory by moving least amount of memory contents.

Another solution for external fragmentation problem is to allocate more than one fragments, i.e., allocate non-contiguous memory partitions. Theoretically it is possible to place a program in non- contiguous memory. The problem is that of dynamic address translation. You can recall, in all memory allocations schemes discussed so far, memory management module uses base and limit register to generate physical address. If a process is placed in more than one non-contiguous partition, memory management module need multiple base and limit registers, one pair for each partition allocated. Since at that no hardware support was available, address translation to generate physical address has to be done through software to compute offsets for different partitions. This ad-hoc solution using multiple base and limit registers, explored that a program is executed while placed in non-contiguous memory.