**Dynamic Memory Management**

First-Fit Allocation in Operating Systems

For both [fixed and dynamic memory allocation schemes](https://www.geeksforgeeks.org/partition-allocation-methods-in-memory-management/), the operating system must keep list of each memory location noting which are free and which are busy. Then as new jobs come into the system, the free partitions must be allocated.

These partitions may be allocated by 4 ways:

**1.** First-Fit Memory Allocation

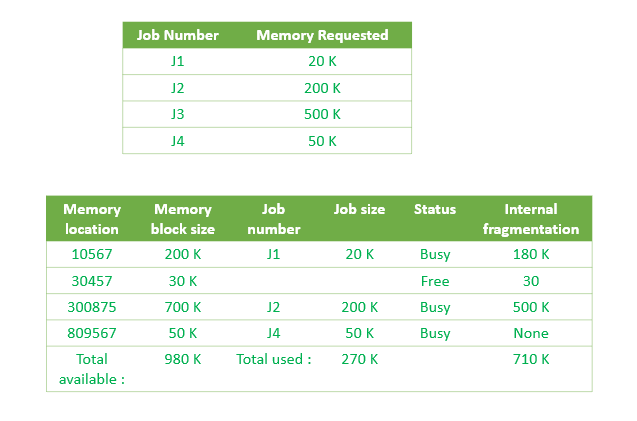
**2.** Best-Fit Memory Allocation

**3.** Worst-Fit Memory Allocation

**4.** Next-Fit Memory Allocation

These are **Contiguous** memory allocation techniques.

[**First-Fit Memory Allocation**](https://www.geeksforgeeks.org/program-first-fit-algorithm-memory-management/)**:**  
This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. In this method, first job claims the first available memory with space more than or equal to it’s size. The operating system doesn’t search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.



As illustrated above, the system assigns J1 the nearest partition in the memory. As a result, there is no partition with sufficient space is available for J3 and it is placed in the waiting list.

**Advantages of First-Fit Memory Allocation:**  
It is fast in processing. As the processor allocates the nearest available memory partition to the job, it is very fast in execution.

**Disadvantages of Fist-Fit Memory Allocation :**  
It wastes a lot of memory. The processor ignores if the size of partition allocated to the job is very large as compared to the size of job or not. It just allocates the memory. As a result, a lot of memory is wasted and many jobs may not get space in the memory, and would have to wait for another job to complete.

**Compaction and Garbage collection**

What do you do when you run out of memory? Any of these methods can fail because all the memory is allocated, or because there is too much fragmentation. Malloc, which is being used to allocate the data segment of a Unix process, just gives up and calls the (expensive) OS call to expand the data segment. A memory manager allocating real physical memory doesn't have that luxury. The allocation attempt simply fails. There are two ways of delaying this catastrophe, compaction and garbage collection.

Compaction attacks the problem of fragmentation by moving all the allocated blocks to one end of memory, thus combining all the holes. Aside from the obvious cost of all that copying, there is an important limitation to compaction: Any pointers to a block need to be updated when the block is moved. Unless it is possible to find all such pointers, compaction is not possible. Pointers can stored in the allocated blocks themselves as well as other places in the client of the memory manager. In some situations, pointers can point not only to the start of blocks but also into their bodies. For example, if a block contains executable code, a branch instruction might be a pointer to another location in the same block. Compaction is performed in three phases. First, the new location of each block is calculated to determine the distance the block will be moved. Then each pointer is updated by adding to it the amount that the block it is pointing (in)to will be moved. Finally, the data is actually moved. There are various clever tricks possible to combine these operations.

Garbage collection finds blocks of memory that are inaccessible and returns them to the free list. As with compaction, garbage collection normally assumes we find all pointers to blocks, both within the blocks themselves and “from the outside.” If that is not possible, we can still do “conservative” garbage collection in which every word in memory that contains a value that appears to be a pointer is treated as a pointer. The conservative approach may fail to collect blocks that are garbage, but it will never mistakenly collect accessible blocks. There are three main approaches to garbage collection: reference counting, mark-and-sweep, and generational algorithms.

Reference counting keeps in each block a count of the number of pointers to the block. When the count drops to zero, the block may be freed. This approach is only practical in situations where there is some “higher level” software to keep track of the counts (it's much too hard to do by hand), and even then, it will not detect cyclic structures of garbage: Consider a cycle of blocks, each of which is only pointed to by its predecessor in the cycle. Each block has a reference count of 1, but the entire cycle is garbage.

# Buddy System – Memory allocation technique

One way of dealing with internal fragmentation is to allow a variety of block sizes. Blocks of each size can be allocated and deallocated by the use of a fixed size block allocate and deallocate mechanism and if a block of one size is not available, a larger block can be allocated and a block of the desired split off of it. When this is done, all blocks resulting from splitting a particular block are called buddies, and the block from which they were split is called their parent. The resulting storage allocation mechanism is said to use a *buddy system.* All buddy systems maintain an array of lists of free blocks, where all blocks in each list are the same size, and the array is indexed by a value computed from the size.

The oldest buddy system, the *binary buddy system* has block sizes that are powers of two. Therefore, when a block is split, it is split exactly in half, and when blocks are combined, two equal size blocks are combined to make a block twice as big. With the binary buddy system, we arrange things so that blocks of size 2*n* always begin at memory addresses where the *n* least significant bits are zero. Thus, blocks of size 1 (20) may begin at any address, but blocks of size 2 (21) may only begin at even addresses, and blocks of size 4 (22) only begin at addresses with the least significant 2 bits equal to zero.

The constraints on the block addresses in the binary buddy system have an interesting consequence. When a block of size 2*n*+1 is split into two blocks of size 2*n*, the addresses of these two blocks will differ in exactly one bit, bit *n*, using the counting scheme that numbers bits starting with 0 at the least significant end. Thus, given a block of size 2*n* at address *a*, we can compute the address of its buddy, the other half of the block from which it was split, by exclusive-oring *a* with 2*n*.

**Static partition** schemes suffer from the **limitation** of having the fixed number of active processes and the usage of space may also not be optimal. The **buddy system** is a memory allocation and management algorithm that manages memory in **power of two increments**. Assume the memory size is 2U, suppose a size of S is required.

* **If 2U-1<S<=2U:** Allocate the whole block
* **Else:** Recursively divide the block equally and test the condition at each time, when it satisfies, allocate the block and get out the loop.

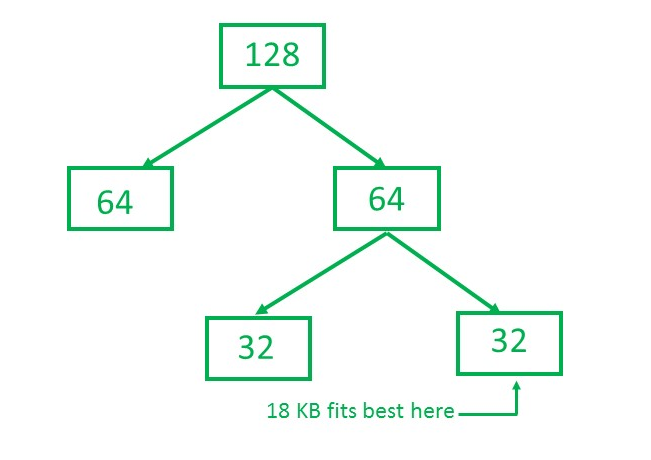
System also keep the record of all the unallocated blocks each and can merge these different size blocks to make one big chunk.  
**Advantage –**

* Easy to implement a buddy system
* Allocates block of correct size
* It is easy to merge adjacent holes
* Fast to allocate memory and de-allocating memory

**Disadvantage –**

* It requires all allocation unit to be powers of two
* It leads to internal fragmentation

**Example –**  
Consider a system having buddy system with physical address space 128 KB.Calculate the size of partition for 18 KB process.  
**Solution –**



So, size of partition for 18 KB process = 32 KB. It divides by 2, till possible to get minimum block to fit 18 KB.

**Boundary Tag**

Neither of the two buddy systems presented above completely eliminates internal fragmentation, and both have external fragmentation problems which were not present when fixed size blocks were used. The alternative is to try to allocate exactly the amount of storage requested, thus eliminating internal fragmentation. As with buddy systems, this will clearly involve splitting free blocks to make blocks of the desired size on allocation, and it will involve merging deallocated blocks with free neighbors to make larger blocks. Unlike the buddy system, however, there are a potentially unlimited number of different block sizes, so it is impractical to use different freelists to store each different size of free block. Furthermore, since the neighbors of a block with which it may be merged may be of any size, there is no way to determine the addresses of the neighbors from the address of the block itself.

As a result of these limitations, new data structures must be introduced to allow allocation of arbitrary sized blocks. These structures must include a way to identify the neighbors of any block given only a pointer to that block, and they must include enough information to determine the size and status of any block given a pointer to it. Two common ways of conveying this information are shown in Figure 14.9.

a) Tags at each end

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| size | status | data | size | status |

|\_\_\_\_\_\_|\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_|\_\_\_\_\_\_\_\_|

^

| user's pointer to block

b) Tag at one end

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| previous | next | status | data |

|\_\_\_\_o\_\_\_\_\_|\_\_\_\_o\_\_\_\_\_|\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

| | ^

<--- -----> | user's pointer to block

Figure 14.9. Descriptions of a block of memory.

In approach a illustrated in Figure 14.9, each block is bracketed with size and status of that block, thus allowing one end of any block to be found from the other, and allowing the status of a block to be inspected from either end. In approach b, each block is prefixed with its status and with the address of each of its neighbors.

Both descriptions schemes given in Figure 14.9 are really equivalent, since a size field may be considered as a relative pointer. In effect, both schemes place two pointers and status information in between each pair of consecutive blocks on the heap. In both cases, the effect is to establish a doubly linked list of blocks, with status information associated with each block. In general, descriptive information associated with a block of data is called a *tag*, and because these tag are stored in the boundaries between adjacent blocks in the heap, heap management algorithms based on these approaches are called *boundary tag* algorithms.

When boundary tags are used, a separate free space list is not needed, since a search of the list of all blocks in the heap will always include all free blocks. However, when the storage utilization is high, only a small fraction of the blocks on the main list will be available, and the introduction of a separate free space list can considerably speed up the allocation process. In boundary tag systems, the structure of this free space list is complicated by the fact that deletions must frequently be made from the middle of the list, for example when a block is merged with its free neighbor. In the buddy system, the free neighbor was always found by searching the freelist, so this deletion was not difficult. With boundary tags, however, the neighbor is found directly from the main list, so the free list must frequently be doubly linked to allow deletion!

There are two basic approaches to allocation in boundary tag systems, *first-fit* and *best-fit*. In the buddy system, best fit was always used; that is, the smallest block large enough to satisfy the current request was always the one allocated. This was easy because of the multiple free space lists. When there is only one free space list, it is more natural to use the first large enough block encountered in searching the free list, independently of whether or not some other block would have given a better fit. Best fit allocation may sometimes be better than first fit, but it is significantly harder to implement.

The choice between first fit and best fit allocation depends on the particular distribution of request sizes being served by the allocation mechanism. Since this distribution is subject to change on many systems, a rational choice between the two approaches is not always possible until after the actual distribution of request sizes has been measured. In general, best fit allocation is ideal when an exact fit is common; this happens when most of the allocations and deallocations deal with only a few popular block sizes. Best fit allocation is particularly bad if exact fits are uncommon because it tends to minimize the size of the free fragments produced, and this, in turn, minimizes the likelyhood that these fragments will be of any future use. Thus, when requests are distributed over a broad range of different block sizes, first fit is the method of choice.