**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

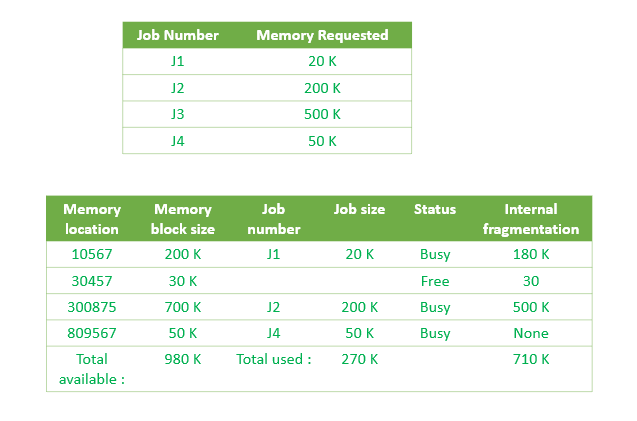
**2.** Best-Fit Memory Allocation

**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.