Table of Contents

Memory Allocation Prerequisites

Free Space Management:

The primary role of the memory management system is to satisfy requests for memory allocation. Sometimes this is implicit, as when a new process is created. At other times, processes explicitly request memory. Either way, the system must locate enough unallocated memory and assign it to the process.

Before we can allocate memory, we must locate the free memory. Naturally, we want to represent the free memory blocks in a way that makes the search efficient.

1.1 Free Bitmaps

If we are operating in an environment with fixed-sized pages, then the search becomes easy. We don't care which page, because they're all the same size. It's quite common in this case to simply store one bit per page frame, which is set to one if the page frame is free, and zero if it is allocated. With this representation, we can mark a page as either free or allocated in constant time by just indexing into this free bitmap. Finding a free page is simply a matter of locating the first nonzero bit in the map.

To make this search easier, we often keep track of the first available page. When we allocate it, we search from that point on to find the next available one. The memory overhead for a free bitmap representation is quite small. For example, if we have pages that are 4096 bytes each, the bitmap uses 1 bit for each 32,768 bits of memory, a 0.003% overhead.

Generally, when we allocate in an environment that uses paging address translation, we don't care which page frame we give a process, and the process never needs to have control over the physical relationship among pages. However, there are exceptions. One exception is a case where we do allocate memory in fixed sized units, but where there is no address translation. Another is where not all page frames are created equally. In both cases, we might need to request a number of physically contiguous page frames. When we allocate multiple contiguous page frames, we look not for the first available page, but for a run of available pages at least as large as the allocation request.

1.2 Free Lists

We can also represent the set of free memory blocks by keeping them in a linked list. When dealing with fixed-sized pages, allocation is again quite easy. We just grab the first page off the list. When pages are returned to the free set, we simply add them to the list. Both of these are constant time operations. If we are allocating memory in variable-sized units, then we need to search the list to find a suitable block. In general, this process can take an amount of time proportional to the number of free memory blocks.

Depending on whether we choose to keep the list sorted, adding a new memory block to the free list can also take O(n) time (proportional to the number of free blocks). To speed the search for particular sized blocks, we often use more complex data structures. Standard data structures such as binary search trees and hash tables are among the more commonly used ones.

Using the usual linked list representation, we have a structure that contains the starting address, the size, and a pointer to the next element in the list. In a typical 32-bit system, this structure takes 12 bytes. So if the average size of a block is 4096 bytes, the free list would take about 0.3% of the available free space.

1.3 Fragmentation

When allocating memory, we can end up with some wasted space. This happens in two ways. First, if we allocate memory in such a way that we actually allocate more than is requested, some of the allocated block will go unused. This type of waste is called **internal fragmentation.**

The other type of waste is unused memory outside of any allocated unit. This can happen if there are available free blocks that are too small to satisfy any request. Wasted memory that lies outside allocation units is called **external fragmentation**.

1.4 Partitioning

Fixed Partitioning:

The earliest and one of the simplest technique which can be used to load more than one processes into the main memory is Fixed partitioning or Contiguous memory allocation.

In this technique, the main memory is divided into partitions of equal or different sizes. The operating system always resides in the first partition while the other partitions can be used to store user processes. The memory is assigned to the processes in contiguous way.

In fixed partitioning,

1. The partitions cannot overlap.
2. A process must be contiguously present in a partition for the execution.

There are various cons of using this technique. Tutorial

1. Internal Fragmentation

If the size of the process is lesser then the total size of the partition then some size of the partition get wasted and remain unused. This is wastage of the memory and called internal fragmentation. As shown in the image below, the 4 MB partition is used to load only 3 MB process and the remaining 1 MB got wasted.

2. External Fragmentation

The total unused space of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form.

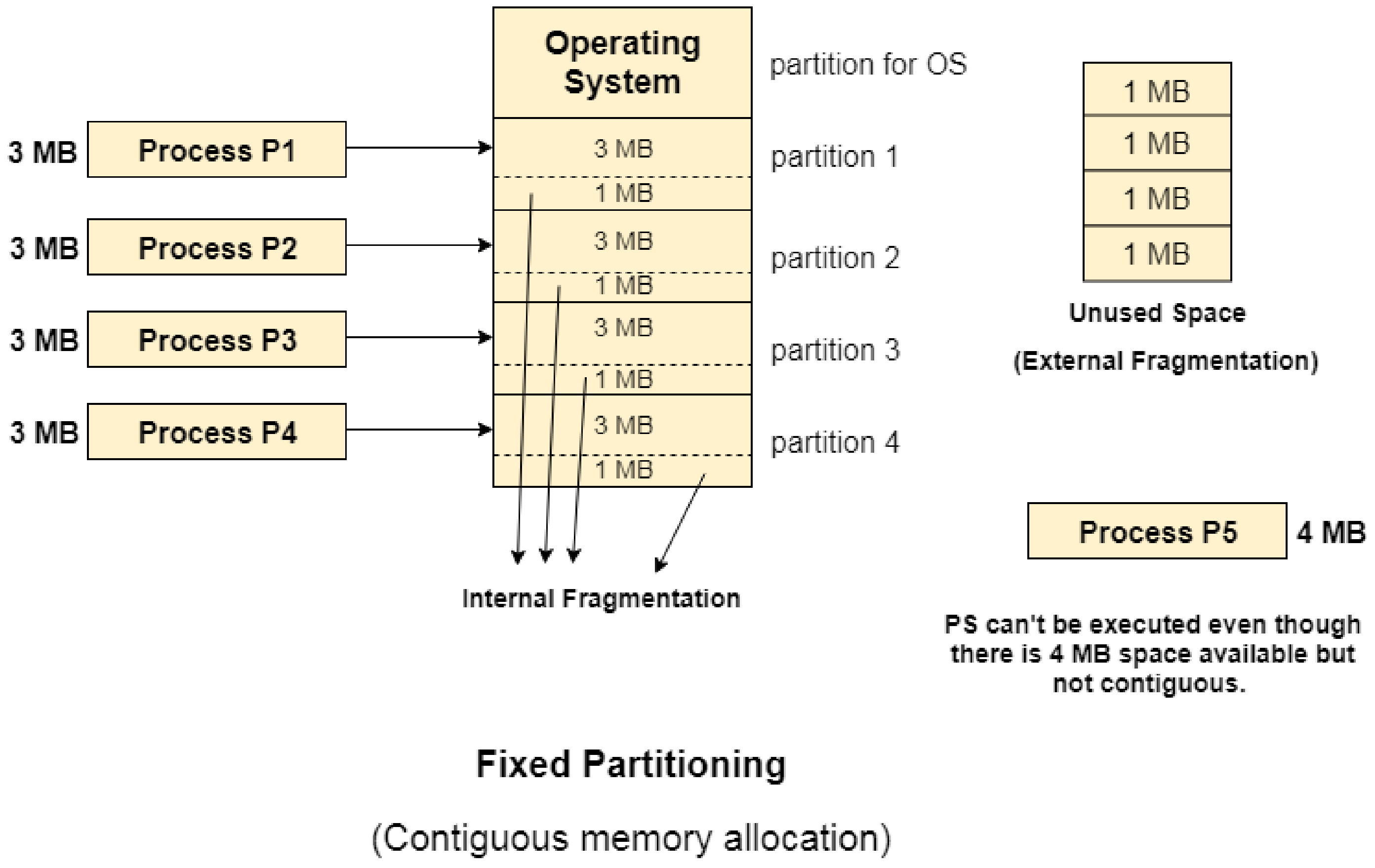
As shown in the image below, the remaining 1 MB space of each partition cannot be used as a unit to store a 4 MB process. Despite of the fact that the sufficient space is available to load the process, process will not be loaded.

3. Limitation on the size of the process

If the process size is larger than the size of maximum sized partition then that process cannot be loaded into the memory. Therefore, a limitation can be imposed on the process size that is it cannot be larger than the size of the largest partition.

4. Degree of multiprogramming is less

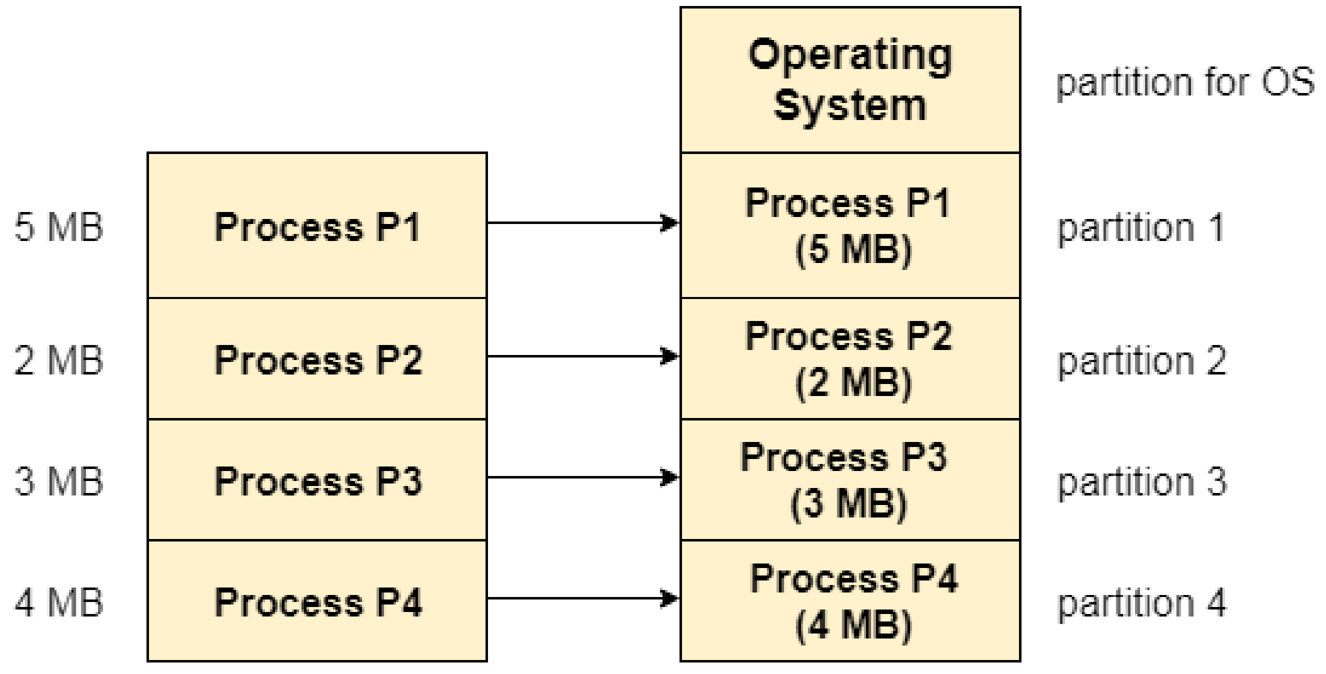
By Degree of multi programming, we simply mean the maximum number of processes that can be loaded into the memory at the same time. In fixed partitioning, the degree of multiprogramming is fixed and very less due to the fact that the size of the partition cannot be varied according to the size of processes.



Dynamic Partitioning:

Dynamic partitioning tries to overcome the problems caused by fixed partitioning. In this technique, the partition size is not declared initially. It is declared at the time of process loading.

The first partition is reserved for the operating system. The remaining space is divided into parts. The size of each partition will be equal to the size of the process. The partition size varies according to the need of the process so that the internal fragmentation can be avoided.



Dynamic Partitioning

(Process Size = Partition Size)

Advantages of Dynamic Partitioning over fixed partitioning

1. No Internal Fragmentation

Given the fact that the partitions in dynamic partitioning are created according to the need of the process, It is clear that there will not be any internal fragmentation because there will not be any unused remaining space in the partition.

2. 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 executed due to the lack of sufficient contiguous memory. Here, In Dynamic partitioning, the process size can't be restricted since the partition size is decided according to the process size.

3. Degree of multiprogramming is dynamic

Due to the absence of internal fragmentation, there will not be any unused space in the partition hence more processes can be loaded in the memory at the same time.

Disadvantages of dynamic partitioning

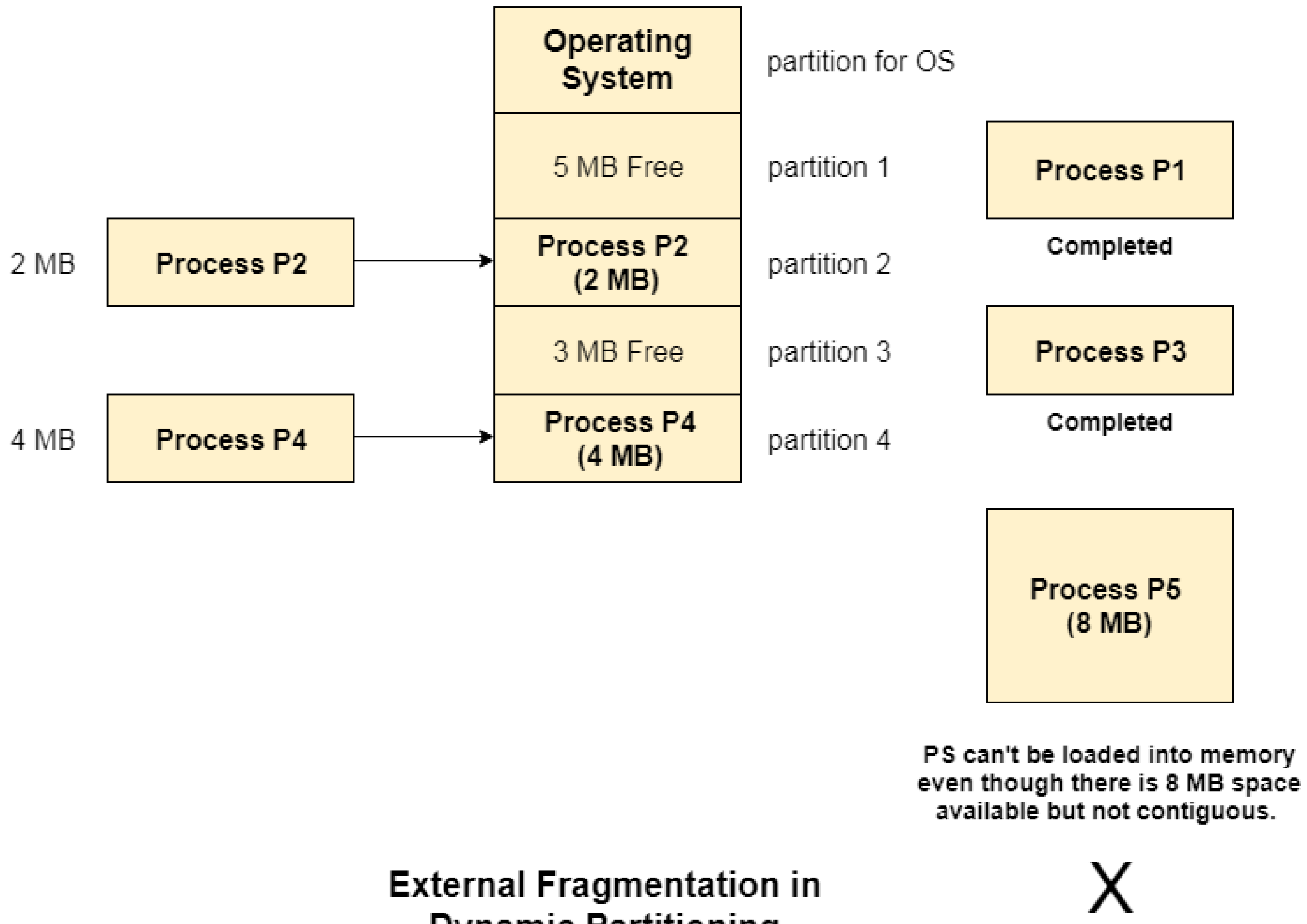
- **External Fragmentation**

Absence of internal fragmentation doesn't mean that there will not be external fragmentation.

Let's consider three processes P1 (5 MB) and P2 (2 MB) and P3 (3 MB) are being loaded in the respective partitions of the main memory.

After some time P1 and P3 got completed and their assigned space is freed. Now there are two unused partitions (1 MB and 1 MB) available in the main memory but they cannot be used to load a 2 MB process in the memory since they are not contiguously located.

The rule says that the process must be contiguously present in the main memory to get executed. We need to change this rule to avoid external fragmentation.



External Fragmentation in Dynamic Partitioning

Memory Allocation Techniques

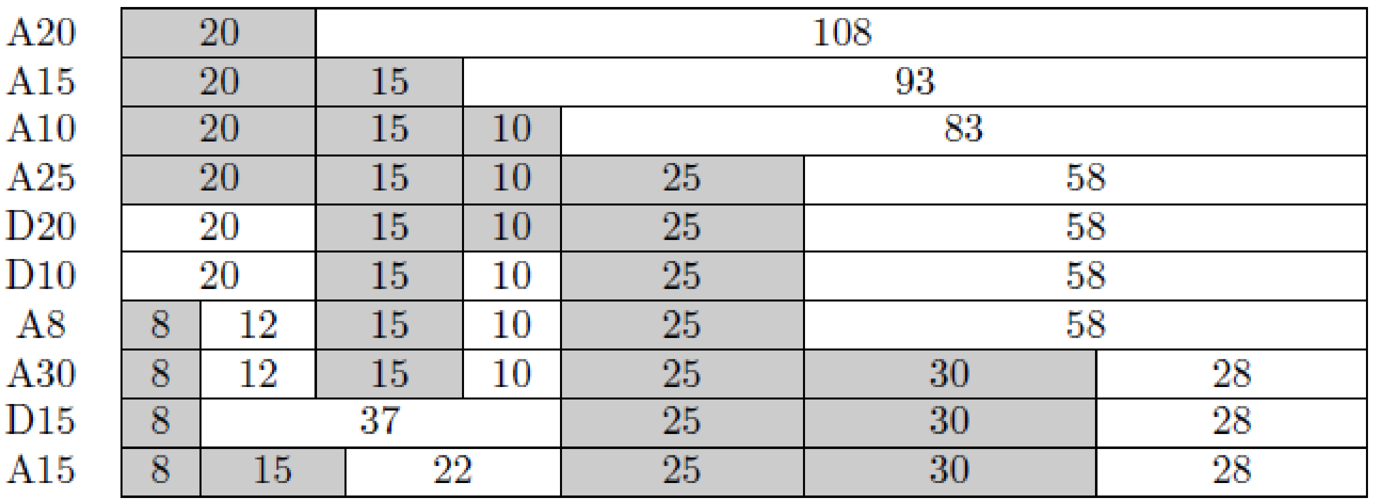
If more than one free block can satisfy a request, then which one should we pick? There are several schemes that are frequently studied and are commonly used.

First Fit

The first of these is called first fit. The basic idea with first fit allocation is that we begin searching the list and take the first block whose size is greater than or equal to the request size, as illustrated in the figure, If we reach the end of the list without finding a suitable block, then the request fails

To illustrate the behavior of first fit allocation, as well as the other allocation policies later, we trace their behavior on a set of allocation and deallocation requests. We denote this sequence as A20, A15, A10, A25, D20, D10, A8, A30, D15, A15, where An denotes an allocation request for n KB and Dn denotes a deallocation request for the allocated block of size n KB. (For simplicity of notation, we have only one block of a given size allocated at a time. None of the policies depend on this property; it is used here merely for clarity.)

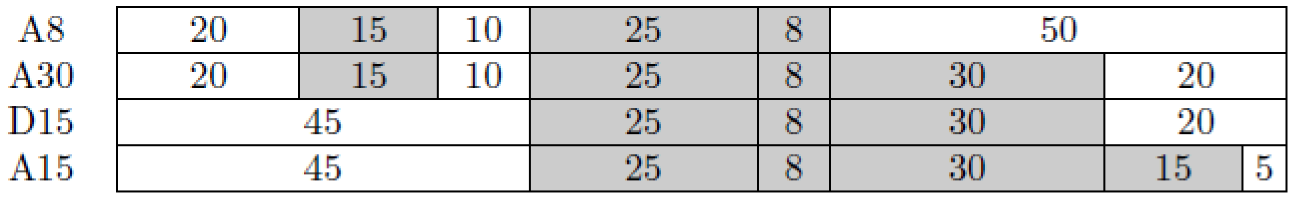
In these examples, the memory space from which we serve requests is 128 KB. Each row of table shows the state of memory after the operation labeling it on the left. Shaded blocks are allocated and unshaded blocks are free. The size of each block is shown in the corresponding box in the figure. In this, and other allocation figures in this chapter, time moves downward in the figure. In other words, each operation happens prior to the one below it.



A20	20	108					
A15	20	15	93				
A10	20	15	10	83			
A25	20	15	10	25	58		
D20	20	15	10	25	58		
D10	20	15	10	25	58		
A8	8	12	15	10	25	58	
A30	8	12	15	10	25	30	28
D15	8	37	25	30	28		
A15	8	15	22	25	30	28	

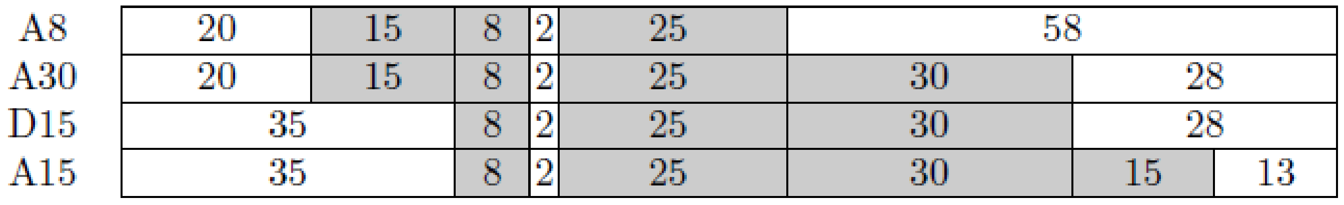
Next Fit

If we want to spread the allocations out more evenly across the memory space, we often use a policy called next fit. This scheme is very similar to the first fit approach, except for the place where the search starts. In next fit, we begin the search with the free block that was next on the list after the last allocation. During the search, we treat the list as a circular one. If we come back to the place where we started without finding a suitable block, then the search fails.



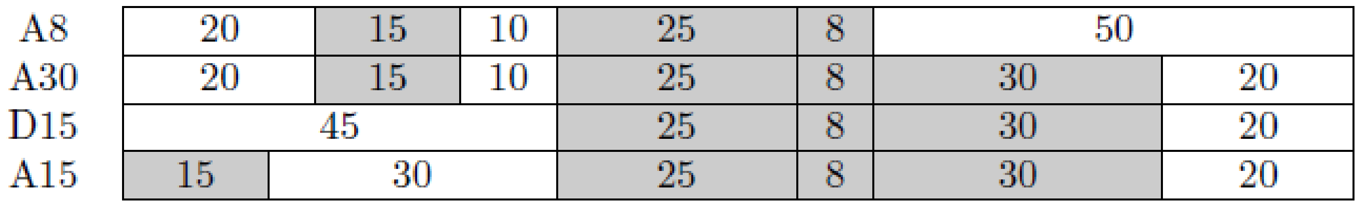
Best Fit

In many ways, the most natural approach is to allocate the free block that is closest in size to the request. This technique is called best fit. In best fit, we search the list for the block that is smallest but greater than or equal to the request size. This is illustrated in Example, Like first fit, best fit tends to create significant external fragmentation, but keeps large blocks available for potential large allocation requests.



Worst Fit

If best fit allocates the smallest block that satisfies the request, then worst fit allocates the largest block for every request. Although the name would suggest that we would never use the worst fit policy, it does have one advantage: If most of the requests are of similar size, a worst fit policy tends to minimize external fragmentation. We illustrate this technique in Example.



A8 20 15 10 25 8 50
A30 20 15 10 25 8 30 20
D15 45 25 8 30 20
A15 15 30 25 8 30 20