**Peterson’s Problem**

[Operating System](https://www.tutorialspoint.com/questions/category/Operating-System)[Windows](https://www.tutorialspoint.com/questions/category/windows)[MCA](https://www.tutorialspoint.com/questions/category/MCA)

Peterson’s solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

do {

   flag[i] = true;

   turn = j;

   while (flag[j] && turn == j);

   /\* critical section \*/

   flag[i] = false;

   /\* remainder section \*/

}

while (true);

The structure of process Pi in Peterson’s solution. This solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1. We use Pj for convenience to denote the other process when Pi is present; that is, j equals 1 − I, Peterson’s solution requires the two processes to share two data items −

int turn;

boolean flag[2];

The variable turn denotes whose turn it is to enter its critical section. I.e., if turn == i, then process Pi is allowed to execute in its critical section. If a process is ready to enter its critical section, the flag array is used to indicate that. For E.g., if flag[i] is true, this value indicates that Pi is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in above. To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. Turn will be set to both i and j at roughly the same time, if both processes try to enter at the same time. Only one of these assignments will occur ultimately; the other will occur but will be overwritten immediately. The final value of turn determines which of the two processes is allowed to enter its critical section first. We now prove that this solution is correct. We need to show that −

* Mutual exclusion is preserved.
* The progress requirement is satisfied.
* The bounded-waiting requirement is met.

To prove 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations indicate that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes — say, Pj — must have successfully executed the while statement, whereas Pi had to execute at least one additional statement (“turn == j”). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as Pj is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that if a process is stuck in the while loop with the condition flag[j] == true and turn == j, process Pi can be prevented from entering the critical section only; this loop is the only one possible. flag[j] will be == false, and Pi can enter its critical section if Pj is not ready to enter the critical section. If Pj has set, flag[j] = true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, Pi will enter the critical section then. Pj will enter the critical section, If turn == j. Although once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. Pj must also set turn to i, if Pj resets flag[j] to true. Hence, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

**Disadvantage**

* Peterson’s solution works for two processes, but this solution is best scheme in user mode for critical section.
* This solution is also a busy waiting solution so CPU time is wasted. So that **“SPIN LOCK”** problem can come. And this problem can come in any of the busy waiting solution.

Semaphores in Process Synchronization

Last Updated: 20-11-2019

Prerequisite: [process-synchronization](https://www.geeksforgeeks.org/process-synchronization-set-1/), [Mutex vs Semaphore](https://www.geeksforgeeks.org/mutex-vs-semaphore/)

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Semaphores are of two types:

1. **Binary Semaphore –** This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Now let us see how it do so.

First, look at two operations which can be used to access and change the value of the semaphore variable.

**Some point regarding P and V operation**

1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization.See below image.critical section of Process P is in between P and V operation.

Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details which is Binary semaphore.

**Implementation of binary semaphores:**

filter\_none

brightness\_4

|  |
| --- |
| struct semaphore {      enum value(0, 1);        // q contains all Process Control Blocks (PCBs)      // corresponding to processes got blocked      // while performing down operation.      Queue<process> q;    } P(semaphore s)  {      if (s.value == 1) {          s.value = 0;      }      else {          // add the process to the waiting queue          q.push(P)              sleep();      }  }  V(Semaphore s)  {      if (s.q is empty) {          s.value = 1;      }      else {            // select a process from waiting queue          q.pop();          wakeup();      }  } |

The description above is for binary semaphore which can take only two values 0 and 1 and ensure the mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one.

Now suppose there is a resource whose number of instance is 4. Now we initialize S = 4 and rest is same as for binary semaphore. Whenever process wants that resource it calls P or wait function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 process P1, P2, P3, P4 and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls signal function and value of semaphore becomes positive.

**Limitations**

1. One of the biggest limitation of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process which is not in sleep state.Therefore a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and to signal the semaphore.

**Problem in this implementation of semaphore**

Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle.

**Problem Statement –** We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

**Initialization of semaphores –**  
mutex = 1  
Full = 0 // Initially, all slots are empty. Thus full slots are 0  
Empty = n // All slots are empty initially

**Solution for Producer –**

do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer

signal(mutex);

signal(full);

}while(true)

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.

**Solution for Consumer –**

do{

wait(full);

wait(mutex);

// remove item from buffer

signal(mutex);

signal(empty);

// consumes item

}while(true)

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

What is Memory?

Computer memory can be defined as a collection of some data represented in the binary format. On the basis of various functions, memory can be classified into various categories. We will discuss each one of them later in detail.

A computer device that is capable to store any information or data temporally or permanently, is called storage device.

Need for Multi programming

However, The CPU can directly access the main memory, Registers and cache of the system. The program always executes in main memory. The size of main memory affects degree of Multi programming to most of the extant. If the size of the main memory is larger than CPU can load more processes in the main memory at the same time and therefore will increase degree of Multi programming as well as CPU utilization.

1. Let's consider,
2. Process Size = 4 MB
3. Main memory size = 4 MB
4. The process can only reside in the main memory at any time.
5. If the time for which the process does IO is P,
7. Then,
9. CPU utilization = (1-P)
10. let's say,
11. P = 70%
12. CPU utilization = 30 %
13. Now, increase the memory size, Let's say it is 8 MB.
14. Process Size = 4 MB
15. Two processes can reside in the main memory at the same time.
16. Let's say the time for which, one process does its IO is P,
18. Then
20. CPU utilization = (1-P^2)
21. let's say P = 70 %
22. CPU utilization = (1-0.49) =0.51 = 51 %

Therefore, we can state that the CPU utilization will be increased if the memory size gets increased.

# 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.

**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.

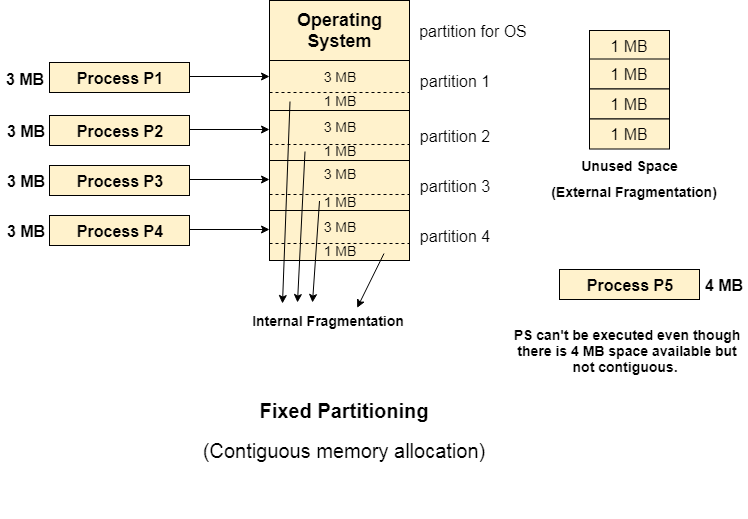
As shown in the image below, the remaining 1 MB space of each partition cannot be used as a unit to store a 4 MB process. Despite of the fact that the sufficient space is available to load the process, process will not be loaded.

**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**

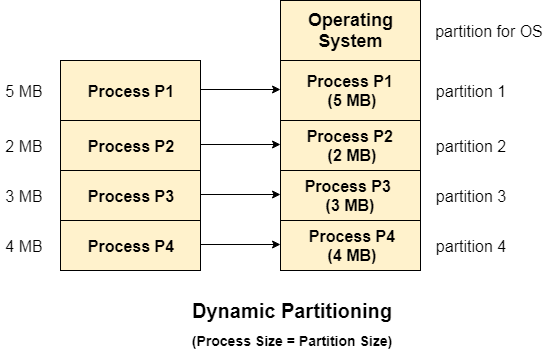
By Degree of multi programming, we simply mean the maximum number of processes that can be loaded into the memory at the same time. In fixed partitioning, the degree of multiprogramming is fixed and very less due to the fact that the size of the partition cannot be varied according to the size of processes.



# 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.



## 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

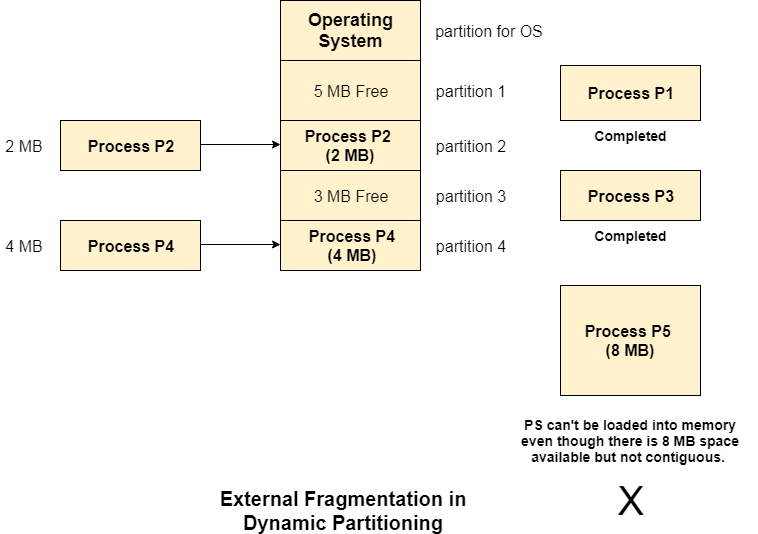
### External Fragmentation

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

Let's consider three processes P1 (1 MB) and P2 (3 MB) and P3 (1 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.



### Complex Memory Allocation

In Fixed partitioning, the list of partitions is made once and will never change but in dynamic partitioning, the allocation and deallocation is very complex since the partition size will be varied every time when it is assigned to a new process. OS has to keep track of all the partitions.

Due to the fact that the allocation and deallocation are done very frequently in dynamic memory allocation and the partition size will be changed at each time, it is going to be very difficult for OS to manage everything.

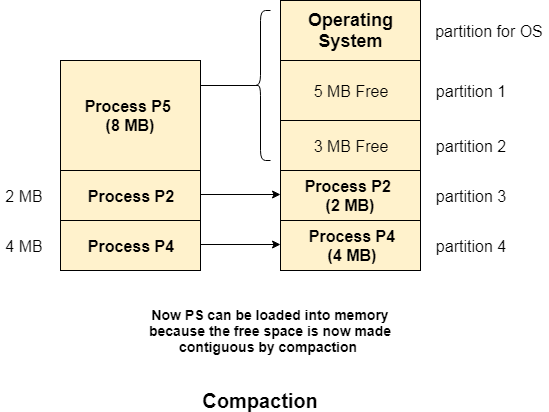
# Compaction

We got to know that the dynamic partitioning suffers from external fragmentation. However, this can cause some serious problems.

To avoid compaction, we need to change the rule which says that the process can't be stored in the different places in the memory.

We can also use compaction to minimize the probability of external fragmentation. In compaction, all the free partitions are made contiguous and all the loaded partitions are brought together.

By applying this technique, we can store the bigger processes in the memory. The free partitions are merged which can now be allocated according to the needs of new processes. This technique is also called defragmentation.



As shown in the image above, the process P5, which could not be loaded into the memory due to the lack of contiguous space, can be loaded now in the memory since the free partitions are made contiguous.

### Problem with Compaction

The efficiency of the system is decreased in the case of compaction due to the fact that all the free spaces will be transferred from several places to a single place.

Huge amount of time is invested for this procedure and the CPU will remain idle for all this time. Despite of the fact that the compaction avoids external fragmentation, it makes system inefficient.

Let us consider that OS needs 6 NS to copy 1 byte from one place to another.

1. 1 B transfer needs 6 NS
2. 256 MB transfer needs 256 X 2^20 X 6 X 10 ^ -9 secs

hence, it is proved to some extent that the larger size memory transfer needs some huge amount of time that is in seconds.

[**next →**](https://www.javatpoint.com/os-partitioning-algorithms)[**← prev**](https://www.javatpoint.com/os-bit-map-for-dynamic-partitioning)

# Linked List for Dynamic Partitioning

The better and the most popular approach to keep track the free or filled partitions is using Linked List.

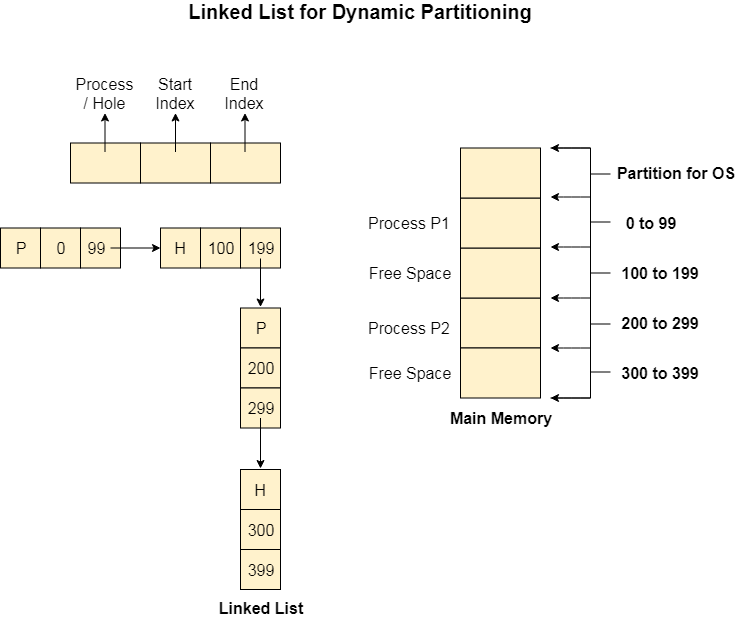
In this approach, the Operating system maintains a linked list where each node represents each partition. Every node has three fields.

1. First field of the node stores a flag bit which shows whether the partition is a hole or some process is inside.
2. Second field stores the starting index of the partition.
3. Third filed stores the end index of the partition.

If a partition is freed at some point of time then that partition will be merged with its adjacent free partition without doing any extra effort.

There are some points which need to be focused while using this approach.

1. The OS must be very clear about the location of the new node which is to be added in the linked list. However, adding the node according to the increasing order of starting index is suggestible.
2. Using a doubly linked list will make some positive effects on the performance due to the fact that a node in the doubly link list can also keep track of its previous node.



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.

When it is time to load a process into main memory and if there is more than one free block of memory of sufficient size then the OS decide which free block to allocate.

There are different Placement Algorithm:

A. First Fit

B. Best Fit

C. Worst Fit

D. Next Fit

**1. First Fit Algorithm**

First Fit algorithm scans the linked list and whenever it finds the first big enough hole to store a process, it stops scanning and load the process into that hole. This procedure produces two partitions. Out of them, one partition will be a hole while the other partition will store the process.

First Fit algorithm maintains the linked list according to the increasing order of starting index. This is the simplest to implement among all the algorithms and produces bigger holes as compare to the other algorithms.

**2. Next Fit Algorithm**

Next Fit algorithm is similar to First Fit algorithm except the fact that, Next fit scans the linked list from the node where it previously allocated a hole.

Next fit doesn't scan the whole list, it starts scanning the list from the next node. The idea behind the next fit is the fact that the list has been scanned once therefore the probability of finding the hole is larger in the remaining part of the list.

Experiments over the algorithm have shown that the next fit is not better then the first fit. So it is not being used these days in most of the cases.

**3. Best Fit Algorithm**

The Best Fit algorithm tries to find out the smallest hole possible in the list that can accommodate the size requirement of the process.

Using Best Fit has some disadvantages.

1. 1. It is slower because it scans the entire list every time and tries to find out the smallest hole which can satisfy the requirement the process.
2. Due to the fact that the difference between the whole size and the process size is very small, the holes produced will be as small as it cannot be used to load any process and therefore it remains useless.  
   Despite of the fact that the name of the algorithm is best fit, It is not the best algorithm among all.

**4. Worst Fit Algorithm**

The worst fit algorithm scans the entire list every time and tries to find out the biggest hole in the list which can fulfill the requirement of the process.

Despite of the fact that this algorithm produces the larger holes to load the other processes, this is not the better approach due to the fact that it is slower because it searches the entire list every time again and again.