Lab # 10

Implementation of Memory Management using (i) First Fit (ii) Best Fit and (iii) Worst Fit

The term memory can be defined as a collection of data in a specific format. It is used to store instructions and process data. The memory comprises a large array or group of words or bytes, each with its own location. The primary purpose of a computer system is to execute programs. These programs, along with the information they access, should be in the main memory during execution. The CPU fetches instructions from memory according to the value of the program counter.

Main Memory

The main memory is central to the operation of a Modern Computer. Main Memory is a large of bytes, ranging in size from hundreds of thousands to billions. Main memory is a repository of rapidly available information shared by the CPU and I/O devices. Main memory is the place where programs and information are kept when the processor is effectively utilizing them. Main memory is associated with the processor, so moving instructions and information into and out of the processor is extremely fast. Main memory is also known as RAM (Random Access Memory). This memory is volatile. RAM loses its data when a power interruption occurs.

To achieve a degree of multiprogramming and proper utilization of memory, memory management is important

Memory Management

In a multiprogramming computer, the Operating System resides in a part of memory, and the rest is used by multiple processes. The task of subdividing the memory among different processes is called Memory Management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

Static and Dynamic Loading

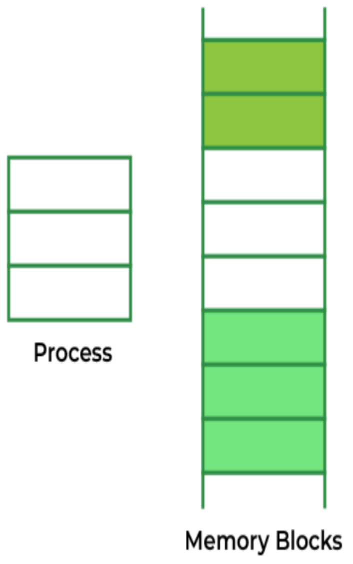
Loading a process into the main memory is done by a loader. There are two different types of loading:

- **Static Loading:** Static Loading is basically loading the entire program into a fixed address. It requires more memory space.
- **Dynamic Loading:** The entire program and all data of a process must be in physical memory for the process to execute. So, the size of a process is limited to the size

of physical memory. To gain proper memory utilization, dynamic loading is used. In dynamic loading, a routine is not loaded until it is called. All routines are residing on disk in a relocatable load format.

Contiguous Memory Allocation

The main memory should accommodate both the operating system and the different client processes. Therefore, the allocation of memory becomes an important task in the operating system. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. Normally several user processes are resided in memory simultaneously. Therefore, there is a need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In adjacent memory allotment, each process is contained in a single contiguous segment of memory.



Memory Allocation

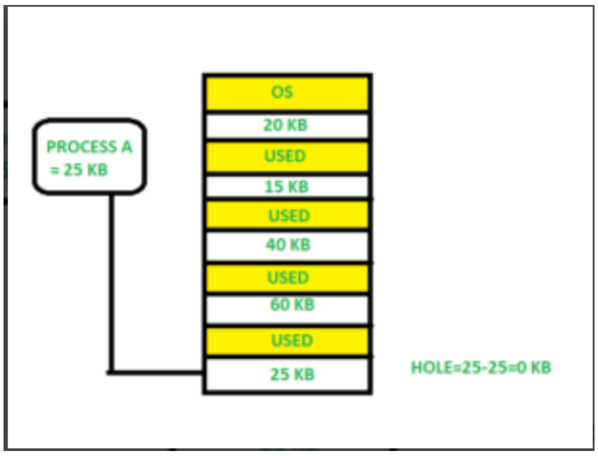
To gain proper memory utilization, memory allocation must be allocated efficient manner. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions and each partition contains exactly one process. Thus, the degree of multiprogramming is obtained by the number of partitions.

- **Multiple partition allocation:** In this method, a process is selected from the input queue and loaded into the free partition. When the process terminates, the partition becomes available for other processes.
- **Fixed partition allocation:** In this method, the operating system maintains a table that indicates which parts of memory are available and which are occupied by processes. Initially, all memory is available for user processes and is considered one large block of available memory. This available memory is known as a “Hole”. When the process arrives and needs memory, we search for a hole that is large enough to store this process. If the requirement is fulfilled then we allocate memory to process, otherwise keeping the rest available to satisfy future requests. While allocating a memory

sometimes dynamic storage allocation problems occur, which concerns how to satisfy a request of size n from a list of free holes. There are some solutions to this problem:

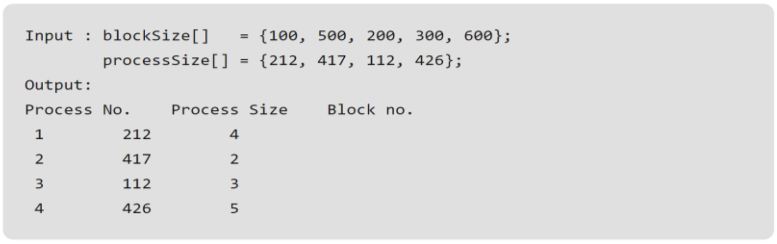
A. Best Fit

In the Best Fit, allocate the smallest hole that is big enough to process requirements. For this, we search the entire list, unless the list is ordered by size.



Implementation:

1. Input memory blocks and processes with sizes.
2. Initialize all memory blocks as free.
3. Start by picking each process and find the minimum block size that can be assigned to current process i.e., find min(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
4. If not then leave that process and keep checking the further processes.



```
Input : blockSize[]   = {100, 500, 200, 300, 600};
        processSize[] = {212, 417, 112, 426};
Output:
Process No.    Process Size    Block no.
 1        212         4
 2        417         2
 3        112         3
 4        426         5
```

// C# implementation of Best - Fit algorithm
using System;

public class GFG {

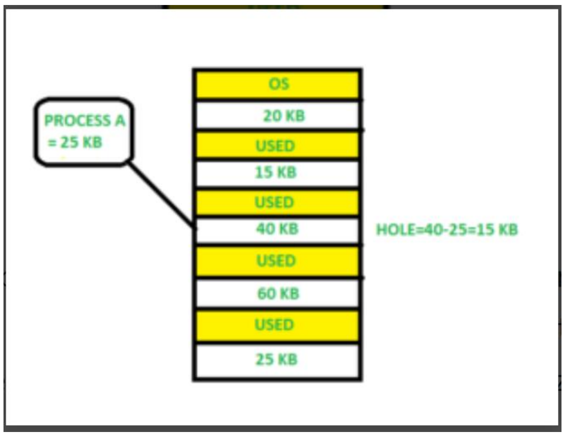
// Method to allocate memory to blocks
// as per Best fit
// algorithm
static void bestFit(int []blockSize, int m,
                                    int []processSize, int n)

```
{

        // Stores block id of the block
        // allocated to a process
        int []allocation = new int[n];

        // Initially no block is assigned to
        // any process
        for (int i = 0; i < allocation.Length; i++)
                allocation[i] = -1;

        // pick each process and find suitable
        // blocks according to its size ad
        // assign to it
        for (int i = 0; i < n; i++)
        {

                // Find the best fit block for
                // current process
                int bestIdx = -1;
                for (int j = 0; j < m; j++)
                {
                        if (blockSize[j] >= processSize[i])
                        {
                                if (bestIdx == -1)
                                        bestIdx = j;
                                else if (blockSize[bestIdx]
                                                > blockSize[j])
                                        bestIdx = j;
                        }
                }

                // If we could find a block for
                // current process
                if (bestIdx != -1)
                {

                        // allocate block j to p[i]
                        // process
                        allocation[i] = bestIdx;

                        // Reduce available memory in
                        // this block.
                        blockSize[bestIdx] -= processSize[i];
                }
        }

        Console.WriteLine("\nProcess No.\tProcess"
                                        + " Size\tBlock no.");
        for (int i = 0; i < n; i++)
        {
                Console.Write(" " + (i+1) + "\t\t"
                                            + processSize[i] + "\t\t");

                if (allocation[i] != -1)
                        Console.Write(allocation[i] + 1);
                else
                        Console.Write("Not Allocated");

                Console.WriteLine();
        }
}

// Driver Method
public static void Main()
{
        int []blockSize = {100, 500, 200, 300, 600};
        int []processSize = {212, 417, 112, 426};
        int m = blockSize.Length;
        int n = processSize.Length;

        bestFit(blockSize, m, processSize, n);
}
}
```

```
Input : blockSize[]   = {100, 500, 200, 300, 600};
        processSize[] = {212, 417, 112, 426};
Output:
Process No.    Process Size    Block no.
 1         212         4
 2         417         2
 3         112         3
 4         426         5
```

B. First Fit

In the first fit, the partition is allocated which is the first sufficient block from the top of Main Memory. It scans memory from the beginning and chooses the first available block that is large enough. Thus it allocates the first hole that is large enough.



Implementation:

1. Input memory blocks and processes with sizes.
2. Initialize all memory blocks as free.
3. Start by picking each process and check if it can be assigned to current block.
4. If size-of-process <= size-of-block if yes then assign and check for next process.
5. If not keep checking the further blocks.

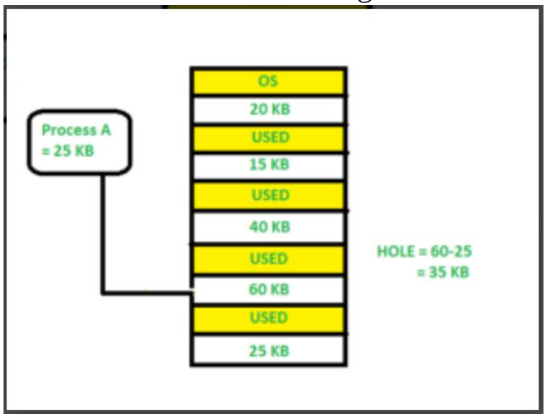
```
// C# implementation of First - Fit algorithm
using System;

class GFG
{
// Method to allocate memory to
// blocks as per First fit algorithm
static void firstFit(int []blockSize, int m,
                                        int []processSize, int n)
{
       // Stores block id of the block
       // allocated to a process
       int []allocation = new int[n];

       // Initially no block is assigned to any process
       for (int i = 0; i < allocation.Length; i++)
              allocation[i] = -1;

       // pick each process and find suitable blocks
       // according to its size ad assign to it
       for (int i = 0; i < n; i++)
       {
              for (int j = 0; j < m; j++)
              {
                     if (blockSize[j] >= processSize[i])
                     {
                            // allocate block j to p[i] process
                            allocation[i] = j;

                            // Reduce available memory in this block.
                            blockSize[j] -= processSize[i];

                            break;
                     }
              }
       }

Console.WriteLine("\nProcess No.\tProcess Size\tBlock no.");
       for (int i = 0; i < n; i++)
       {
```

```
                Console.Write(" " + (i+1) + "\t\t" +
                                    processSize[i] + "\t\t");
                if (allocation[i] != -1)
                Console.Write(allocation[i] + 1);
                else
                        Console.Write("Not Allocated");
                Console.WriteLine();
        }
}

// Driver Code
public static void Main()
{
        int []blockSize = {100, 500, 200, 300, 600};
        int []processSize = {212, 417, 112, 426};
        int m = blockSize.Length;
        int n = processSize.Length;

        firstFit(blockSize, m, processSize, n);
}
}
```

```
Input : blockSize[]   = {100, 500, 200, 300, 600};
        processSize[] = {212, 417, 112, 426};
Output:
Process No.    Process Size    Block no.
   1              212             2
   2              417             5
   3              112             2
   4              426          Not Allocated
```

C. Worst Fit

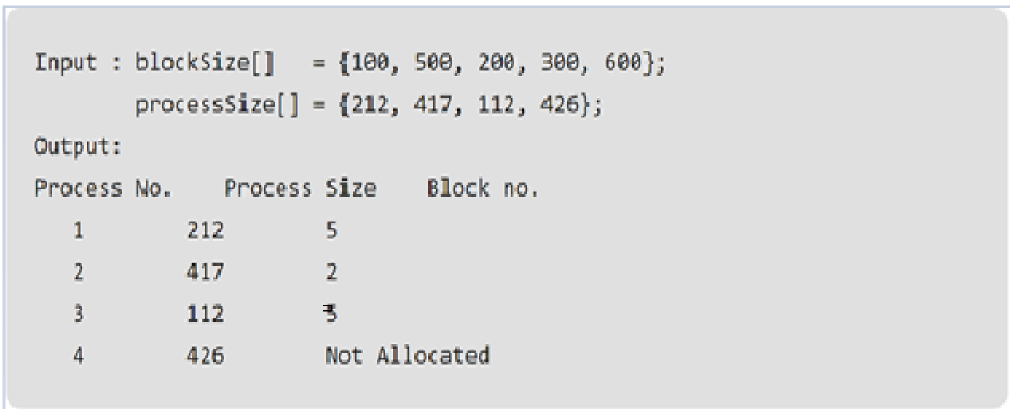
Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory. It is opposite to the best-fit algorithm. It searches the entire list of holes to find the largest hole and allocate it to process.



<u>Implementation:</u>

1. Input memory blocks and processes with sizes.
2. Initialize all memory blocks as free.
3. Start by picking each process and find the maximum block size that can be assigned to current process i.e., find max(bockSize[1], b blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
4. If not keep checking the further blocks.

```
// C# implementation of worst - Fit algorithm
using System;

class GFG
{
// Method to allocate memory to blocks
// as per worst fit algorithm
static void worstFit(int []blockSize, int m,
                                    int []processSize, int n)
{
        // Stores block id of the block allocated to a
        // process
        int []allocation = new int[n];

        // Initially no block is assigned to any process
        for (int i = 0; i < allocation.Length; i++)
                allocation[i] = -1;

        // pick each process and find suitable blocks
        // according to its size ad assign to it
        for (int i = 0; i < n; i++)
        {
                // Find the best fit block for current process
                int wstIdx = -1;
                for (int j = 0; j < m; j++)
                {
                        if (blockSize[j] >= processSize[i])
                        {
                                if (wstIdx == -1)
                                        wstIdx = j;
                                else if (blockSize[wstIdx] < blockSize[j])
                                        wstIdx = j;
                        }
                }

                // If we could find a block for current process
                if (wstIdx != -1)
                {
                        // allocate block j to p[i] process
                        allocation[i] = wstIdx;

                        // Reduce available memory in this block.
                        blockSize[wstIdx] -= processSize[i];
                }
        }

        Console.WriteLine("\nProcess No.\tProcess Size\tBlock no.");
        for (int i = 0; i < n; i++)
        {
                Console.Write(" " + (i+1) + "\t\t\t" + processSize[i] + "\t\t\t");
                if (allocation[i] != -1)
                        Console.Write(allocation[i] + 1);
                else
                        Console.Write("Not Allocated");
                Console.WriteLine();
        }
}

// Driver code
public static void Main(String[] args)
{
        int []blockSize = {100, 500, 200, 300, 600};
        int []processSize = {212, 417, 112, 426};
        int m = blockSize.Length;
        int n = processSize.Length;

        worstFit(blockSize, m, processSize, n);
}
}
```



```
Input : blockSize[]   = {100, 500, 200, 300, 600};
        processSize[] = {212, 417, 112, 426};
Output:
Process No.    Process Size    Block no.
   1        212        5
   2        417        2
   3        112        5
   4        426        Not Allocated
```

Lab Tasks

1. For below processes, and block sizes, provide results using algorithms for below
   i) First Fit
   ii) Best First and
   iii) Worst Fit

   Block Size [] = {200,400,600,500,300,250};

   Process Size [] = {357,210,468,491};

2. For below processes, and block sizes, provide results using algorithms for below
   iv) First Fit
   v) Best First and
   vi) Worst Fit

   Block Size [] = {50,150,300,350,600};

   Process Size [] = {300, 25, 125, 50};