Write a program to simulate Memory placement strategies – best fit, first fit, next fit and worst fit.

Computer Engineering (Savitribai Phule Pune University)

Assignment No 5

TITLE : Write a program to simulate Memory placement strategies – best fit, first fit, next fit and worst fit.

OBJECTIVES :

- To understand different memory placement strategies
- To implement different memory placement strategies
- To study different memory placement strategies

PROBLEM STATEMENT : Write a program to simulate Memory placement strategies – best fit, first fit, next fit and worst fit.

OUTCOMES:

After completion of this assignment students will be able to:

- Knowledge of memory placement algorithms
- Compare different memory placement strategies.

SOFTWARE REQUIREMENTS:

JDK/Eclipse

HARDWARE REQUIREMENT:

- M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

THEORY:

A. First Fit Memory Allocation

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.

Example :

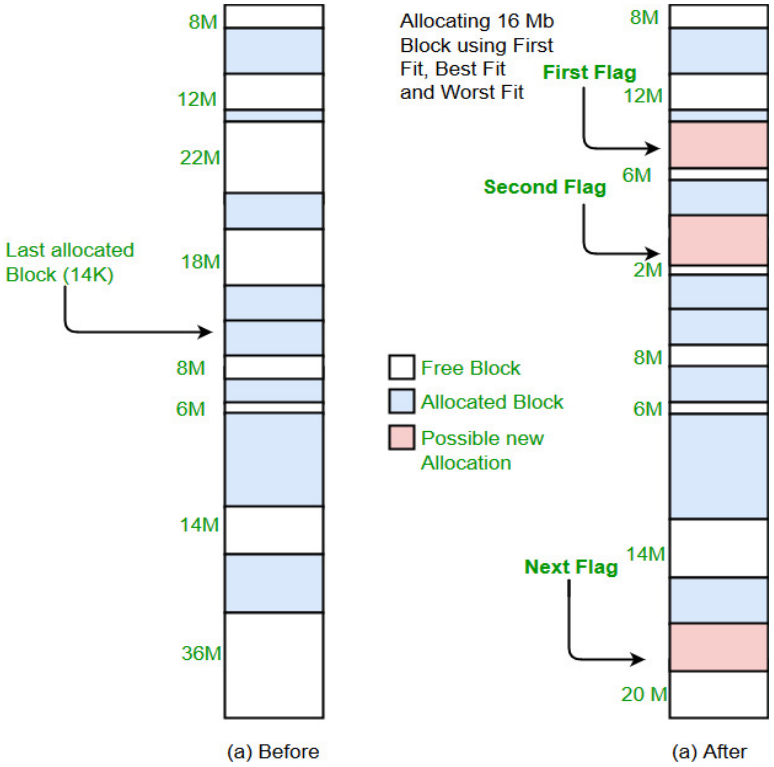
Input : blockSize[] = {100, 500, 200, 300, 600};

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

Implementation:
1- Input memory blocks with size and processes with size.
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 then keep checking the further blocks.



Advantages:
It is fast in processing. As the processor allocates the nearest available memory partition to the job, it is very fast in execution.

Disadvantages:
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.

B. Next Fit Memory Allocation

Next fit is a modified version of 'first fit'. It begins as the first fit to find a free partition but when called next time it starts searching from where it left off, not from the beginning. This policy makes use of a roving pointer. The pointer moves along the memory chain to search for a next fit. This helps in, to avoid the usage of memory always from the head (beginning) of the free block
chain.

Example:

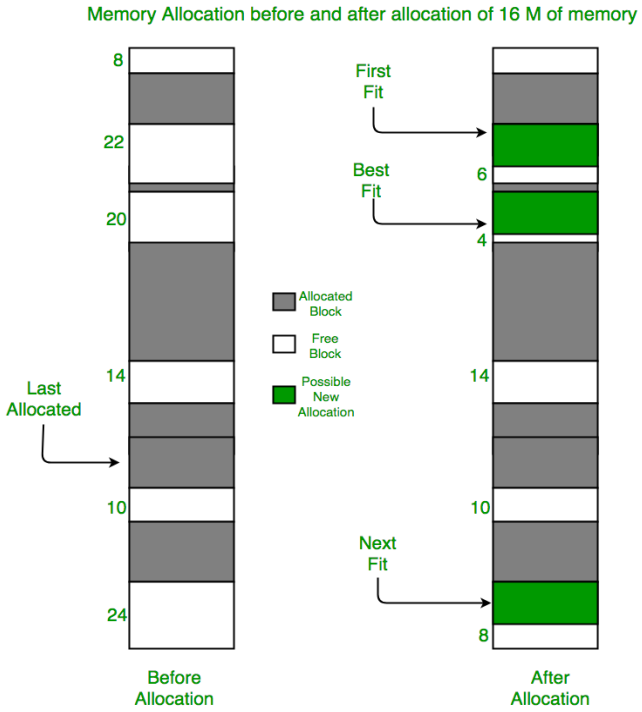
Input : blockSize[] = {5, 10, 20};
processSize[] = {10, 20, 30};

Output:

Process No.	Process Size	Block no.
1	10	2
2	20	3
3	30	Not Allocated

Algorithm:

1. Input the number of memory blocks and their sizes and initializes all the blocks as free.
2. Input the number of processes and their sizes.
3. Start by picking each process and check if it can be assigned to the current block, if yes, allocate it the required memory and check for next process but from the block where we left not from starting.
4. If the current block size is smaller then keep checking the further blocks.



Advantage over First Fit

- First fit is a straight and fast algorithm, but tends to cut large portion of free parts into small pieces due to which, processes that need a large portion of memory block would not get anything even if the
  sum of all small pieces is greater than it required which is so-called external fragmentation problem.

- Another problem of the first fit is that it tends to allocate memory parts at the beginning of the memory, which may lead to more internal fragments at the beginning. Next fit tries to address this problem by starting the search for the free portion of parts not from the start of the memory, but from where it ends last time.
- Next fit is a very fast searching algorithm and is also comparatively faster than First Fit and Best Fit Memory Management Algorithms.

C. **Best-Fit Memory Allocation:**

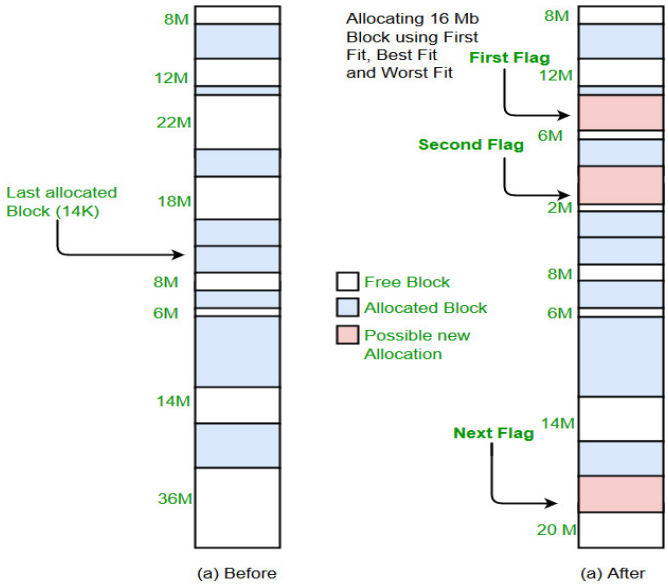
This method keeps the free/busy list in order by size – smallest to largest. In this method, the operating
system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently. Here the jobs are in the order from smallest job to largest job.

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



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.
5- If not then leave that process and keep checking the further processes.

Advantages:

Memory Efficient. The operating system allocates the job minimum possible space in the memory, making memory management very efficient. To save memory from getting wasted, it is the best method.

Disadvantages:

It is a Slow Process. Checking the whole memory for each job makes the working of the operating system very slow. It takes a lot of time to complete the work.

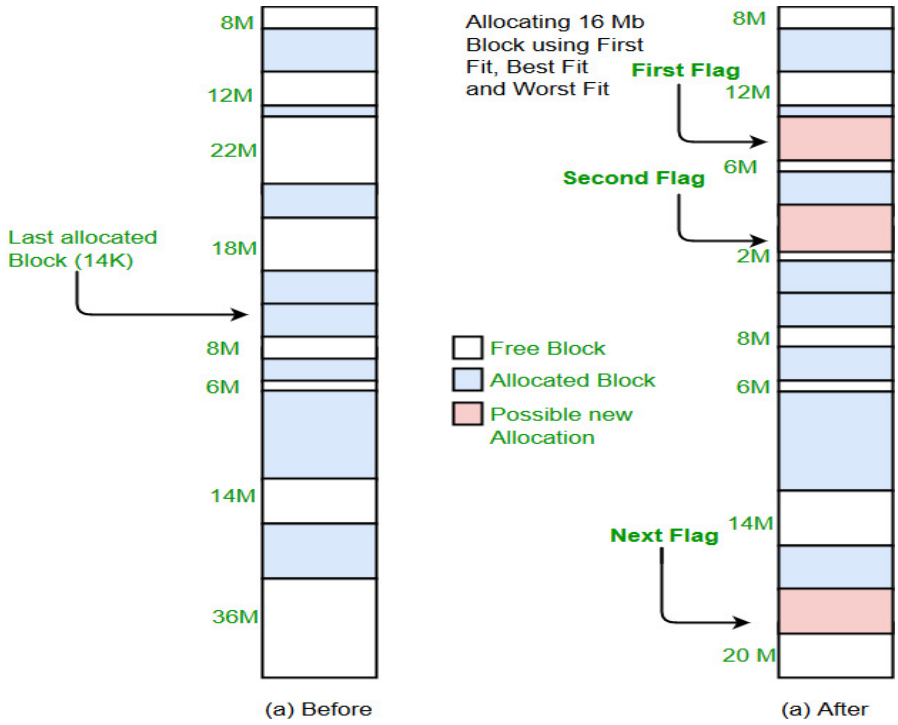
C. Worst Fit Memory Allocation

In this allocation technique, the process traverses the whole memory and always search for the largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search the largest hole.

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



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 maximum block size that can be assigned to current process i.e., find max(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
5- If not then leave that process and keep checking the further processes.

Advantages:

Since this process chooses the largest hole/partition, therefore there will be large internal fragmentation.
Now, this internal fragmentation will be quite big so that other small processes can also be placed in that leftover partition.

Disadvantages:

It is a slow process because it traverses all the partitions in the memory and then selects the largest partition among all the partitions, which is a time-consuming process.

PROGRAM:

```
    #include <bits/stdc++.h>
using namespace std;

void bestFit(int blockSize[], int m, int processSize[], int n)
{

    int allocation[n];

    memset(allocation, -1, sizeof(allocation));
    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];
        }
    }

    cout << "No.\tSize\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {
        cout << i + 1 << "\t" << processSize[i] << "\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

void firstFit(int blockSize[], int m,
```

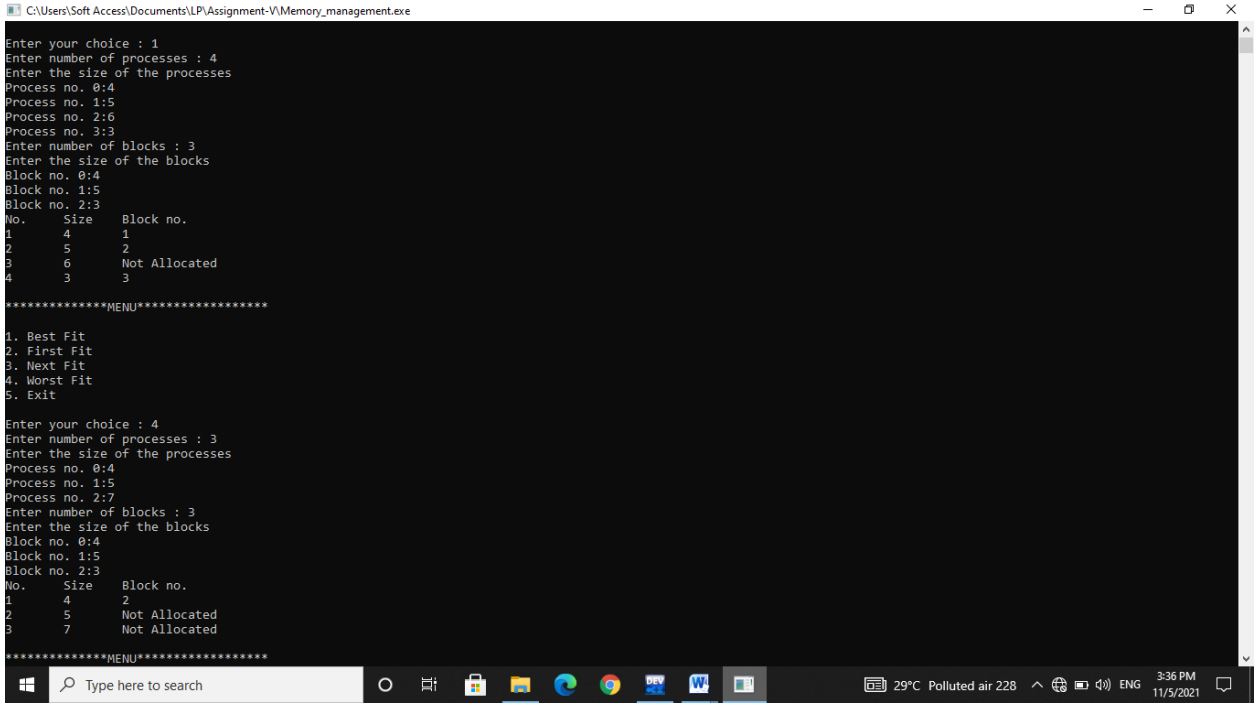
```
              int processSize[], int n)
{
  // Stores block id of the
  // block allocated to a process
  int allocation[n];

  // Initially no block is assigned to any process
  memset(allocation, -1, sizeof(allocation));

  // 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;
      }
    }
  }

  cout << "No.\tSize\tBlock no.\n";
  for (int i = 0; i < n; i++)
  {
    cout << i + 1 << '\t' << processSize[i] << '\t';
    if (allocation[i] != -1)
      cout << allocation[i] + 1;
    else
      cout << "Not Allocated";
    cout << endl;
  }
}

void NextFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n], j = 0;

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
```

```
    // according to its size ad assign to it
    for (int i = 0; i < n; i++) {

        // Do not start from beginning
        while (j < m) {

            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;
            }

            // mod m will help in traversing the blocks from
            // starting block after we reach the end.
            j = (j + 1) % m;
        }
    }

    cout << "No.\tSize\tBlock no.\n";
    for (int i = 0; i < n; i++) {
        cout << i + 1 << "\t" << processSize[i] << "\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

void worstFit(int blockSize[], int m, int processSize[],
                                                int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // 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];
        }
    }

    cout << "No.\tSize\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {
        cout << i+1 << "\t" << processSize[i] << "\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

// Driver code
int main()
{
    int n, m, processSize[20], blockSize[20], choice;

    while (1)
    {
        cout << "\n**************MENU*****************\n\n";
        cout << "1. Best Fit\n2. First Fit\n3. Next Fit\n4. Worst Fit\n5. Exit\n\n";
        cout << "Enter your choice : ";
        cin >> choice;
        if(choice == 5){
            exit(0);
        }

        cout << "Enter number of processes : ";
```

```
        cin >> n;
        cout << "Enter the size of the processes \n";
        for (int i = 0; i < n; i++)
        {
            cout << "Process no. " << i << ":";
            cin >> processSize[i];
        }

        cout << "Enter number of blocks : ";
        cin >> m;
        cout << "Enter the size of the blocks  \n";
        for (int i = 0; i < m; i++)
        {
            cout << "Block no. " << i << ":";
            cin >> blockSize[i];
        }

        switch (choice)
        {
        case 1:
        {
            bestFit(blockSize, m, processSize, n);
            break;

        }
        case 2:
        {
            firstFit(blockSize, m, processSize, n);
            break;
        }
        case 3:
        {
            NextFit(blockSize, m, processSize, n);
            break;
        }
        case 4:
        {
            worstFit(blockSize, m, processSize, n);
            break;
        }
        default:
            break;
        }

    }
    return 0;
}
```

OUTPUT:



CONCLUSION:

Thus Successfully Implemented Memory Allocation algorithms .