The operating system is also refereed as the resource manager which allocates the different resources to the processes. One of such processes is the operating system memory which is allocated based on the demand of the processes. The operating system uses different schemes for this purpose, the most common among them are, first fit, best fit, and worst fit.

**How first fit works?**

Whenever a process (p1) comes with memory allocation request the following happens –

OS sequentially searches available memory blocks from the first index

Assigns the first memory block large enough to accommodate process

Whenever a new process P2 comes, it does the same thing. Search from the first index again.

Methods discussed

We will look at two different methods –

Method 1 – Blocks allowed to keep just one single process

Method 2 – Blocks allowed to keep multiple processes, if partitioned fragmentation big enough for new processes

Method 1 Code

class Main

{

public static void main (String[]args)

{

int blockSize[] = {100, 50, 30, 120, 35};

int processSize[] = {20, 60, 70, 40};

int m = blockSize.length;

int n = processSize.length;

implimentFirstFit(blockSize, m, processSize, n);

}

static void implimentFirstFit(int blockSize[], int blocks, int processSize[], int processes) {

// This will store the block id of the allocated block to a process

int allocate[] = new int[processes];

int occupied[] = new int [blocks];

// initially assigning -1 to all allocation indexes

// means nothing is allocated currently

for (int i = 0; i < allocate.length; i++)

allocate[i] = -1;

for(int i = 0; i < blocks; i++){

occupied[i] = 0;

}

// take each process one by one and find

// first block that can accomodate it

for (int i = 0; i < processes; i++)

{

for (int j = 0; j < blocks; j++)

{

if (!(occupied[j] > 0) && blockSize[j] >= processSize[i])

{

// allocate block j to p[i] process

allocate[i] = j;

occupied[j] = 1;

break;

}

}

}

System.out.println("\nProcess No.\tProcess Size\tBlock no.\n");

for (int i = 0; i < processes; i++)

{

System.out.print(i + 1 + "\t\t\t" + processSize[i] + "\t\t\t");

if (allocate[i] != -1)

System.out.println(allocate[i] + 1);

else

System.out.println("Not Allocated");

}

}

}

Output:

Process No. Process Size Block no.

1 20 1

2 60 4

3 70 Not Allocated

4 40 2

Method 2

This method allows multiple processes to share same block if size is enough

Run

class Main

{

public static void main (String[]args)

{

int blockSize[] = {100, 50, 30, 120, 35};

int processSize[] = {20, 60, 70, 40};

int m = blockSize.length;

int n = processSize.length;

implimentFirstFit(blockSize, m, processSize, n);

}

static void implimentFirstFit(int blockSize[], int blocks, int processSize[], int processes) {

// This will store the block id of the allocated block to a process

int allocate[] = new int[processes];

int occupied[] = new int [blocks];

// initially assigning -1 to all allocation indexes

// means nothing is allocated currently

for (int i = 0; i < allocate.length; i++)

allocate[i] = -1;

// take each process one by one and find

// first block that can accommodate it

for (int i = 0; i < processes; i++)

{

for (int j = 0; j < blocks; j++) {

if (blockSize[j] >= processSize[i])

{

// allocate block j to p[i] process

allocate[i] = j;

// Reduce size of block j as it has accommodated p[i]

blockSize[j] -= processSize[i];

break;

}

}

}

System.out.println("\nProcess No.\tProcess Size\tBlock no.\n");

for (int i = 0; i < processes; i++)

{

System.out.print(i + 1 + "\t\t\t" + processSize[i] + "\t\t\t");

if (allocate[i] != -1)

System.out.println(allocate[i] + 1);

else

System.out.println("Not Allocated");

}

}

}

Output:

Process No. Process Size Block no.

1 20 1

2 60 1

3 70 4

4 40 2

**Best Fit:**

In the case of the Best Fit, the operating system searches the empty memory slots sequentially and allocates the memory segment that best suits the demand.

The operating system allocates the memory with minimum wastage. The approach is said to be the best memory allocation scheme for this reason. However, searching for the best fit memory can be time taking.

Best Fit

import java.io.\*;

class PrepInsta

{

public static void main(String args[])throws IOException

{

//initialization and declaration of variables

int flag[]=new int[10];

int m\_segments[]=new int[10];

int i,sr,memory;

int sp, loc=0;

int cntrl=1000;

DataInputStream in=new DataInputStream(System.in);

// input memory segments

System.out.println(“Enter the no of memory segments\n”);

memory=Integer.parseInt(in.readLine());

//input size of memory segments

//Scanning memory segments for best fit allocation

System.out.println(“Enter the size of memory segments\n”);

for(i=0;i<memory;i++)

sm[i]=Integer.parseInt(in.readLine());

for(i=0;i<memory;i++)

flag[i]=0;

System.out.println(“Before best fit allocation\n”);

System.out.println(“\nIndex\t\tMemory Segments\n”);

for(i=0;i<memory;i++)

System.out.println((i+1)+”\t\t”+m\_segments[i]);

//Input space for the process for memory allocation

System.out.println(“\nEnter the space requirement for new process\n”);

sr=Integer.parseInt(in.readLine());

//searching memory for best fit allocation

for(i=0;i<memory;i++)

{

if(flag[i]==0)

{

sp=m\_segments[i];

if(sr<=sp) { if(cntrl>sp)

{

cntrl=sp;

loc=i;

}

}

}

}

if(cntrl==0)

System.out.println(“\n Space not available”);

else

{

m\_segments[loc]=sr;

flag[loc]=1;

}

// Best fit memory allocation

System.out.println(“\nAfter Bestfit Allocation\n”);

System.out.println(“Index \t\t Memory Segment\n”);

for(i=0;i<memory;i++)

{

System.out.println((i+1)+”\t\t”+m\_segments[i]);

}

System.out.println(“\n The process allocated to the memory segments “+(loc+1));

}

}

Output

Enter the no of memory segments

2

Enter the size of memory segments

10

12

Before best fit allocation

Index Memory Segments

1 10

2 12

Enter the space requirement for new process

12

After Bestfit Allocation

Index Memory Segment

1 30

2 12

The process allocated to the memory segments 2s

**Worst Fit:**

In the case of worst fit memory allocation scheme, the processor assigns any random free memory block which is greater than the memory demanded by the process. In this case, there is a probability of maximum memory wastage as the processor assigns any free memory block randomly to the demanding process. Therefore, this memory allocation scheme is also known as the worst fit.

Program code for Worst Fit Memory management Scheme

public class PrepInsta

{

//Function to allocate memory to empty blocks based on the worst fit algorithm.

static void WorstFit(int b\_size[], int m, int p\_size[], int n)

{

//stores block id of the block which is allocated to a process

Int allocate[] = new int[n];

// no block is assigned to a process initially

for ( int I =0; i< allocate.length++)

allocate[i] =-1;

//select each process and find suitable blocks as per its size

//block assignment is also done here

for (int i=0; i<n; i++)

{

// determine the first fit block for the current process

Int wstIdx = -1;

for (int j=0; j<m; j++) { if (b\_Size[j] >= p\_Size[i])

{

if (wstIdx == -1)

wstIdx = j;

else if (b\_Size[wstIdx] < b\_Size[j])

wstIdx = j;

}

}

// searching an empty memory block for the current process

if (wstIdx != -1)

{

// assigning memory block j to the process p[i]

allocate[i] = wstIdx;

// Reduce available memory in this block.

b\_Size[wstIdx] -= p\_Size[i];

}

}

System.out.println

("\nProcess Number \tProcess Size\tBlock Number ");

for (int i = 0; i < n; i++)

{

System.out.print

(" " + (i+1) + "\t\t" + processSize[i] + "\t\t");

if (allocation[i] != -1)

System.out.print(allocation[i] + 1);

else

System.out.print("Not Allocated");

System.out.println();

}

}

// Driver code

public static void main(String[] args)

{

int b\_Size[] = {100, 500, 200, 300, 600};

int p\_Size[] = {212, 417, 112, 426};

int m = b\_Size.length;

int n = p\_Size.length;

worstFit(b\_Size, m, p\_Size, n);

}

}

Output

Process Number Process Size Block Number

1 212 5

2 417 2

3 112 5

4 426 Not Allocated

**Next Fit:**

The Next Fit algorithm can also be said as the modified version of First Fit algorithm. The memory search method starts as a first fit to find a free memory space, but when the method is called the second time, it searches the memory from where it left off, not from the beginning. The concept uses a moving pointer which moves along the memory blocks to search for an empty space for the next fit. The overall concept helps to avoid using the memory always from the starting point or the head of the free block chain.

The Next Fir Algorithm can also be said as the changed version of the First Fit algorithm as its basic concept is similar to the first fit method of searching memory. Also, this method does not allocate memory from the beginning of memory allocation. Therefore, the operating system uses scheduling algorithms for the memory allocation method.

Algorithm for memory allocation using Next Fit

Step 1. Start

Step 2. Enter the number of memory blocks.

Step 3. Enter the size of each memory block.

Step 4. Enter the number of processes with their sizes.

Step 5. Start by selecting each process to check if it can be assigned to the current memory block.

Step 6. If the condition in step 4 is true, then allocate the process with the required memory and check for the next process from the memory block where the searching was halted, not from the starting.

Step 7. If the current memory size is smaller, then continue to check the next blocks.

Step 8. Stop

Java Program for Next Fit Memory Management

// Java code for next fit

import java.util.Arrays;

public class GFG {

// Method to allocate memory to the blocks following the Next fit algorithm

static void Next\_Fit(int block\_size[], int m, int process\_size[], int n)

{

// The code will store the block id for a block which is assigned to a process

int allocate[] = new int[n], j = 0;

// No block is assigned to any process at the beginning

Arrays.fill(allocate, -1);

// Find a suitable block for each process as per its size and assign memory to it

for(int i = 0; i < n; i++) {

// Not starting from the beginning

while (j < m) {

if (block\_size[j] >= process\_size[i]) {

// block j is allocated to p[i] process

allocate[i] = j;

// Reduce available memory in this block.

Block\_size[j] -= process\_size[i];

break;

}

// mod m will traverse the blocks from the starting when the pointer reaches at the end.

j = (j + 1) % m;

}

}

System.out.print("\nProcess No.\tProcess Size\tBlock no.\n");

for (int i = 0; i < n; i++) {

System.out.print( i + 1 + "\t\t" + process\_size[i]

+ "\t\t");

if (allocate[i] != -1) {

System.out.print(allocate[i] + 1);

} else {

System.out.print("Not Allocated");

}

System.out.println("");

}

}

// Driver program

static public void main(String[] args) {

int block\_size[] = {5, 10, 20};

int process\_size[] = {10, 20, 5};

int m = block\_size.length;

int n = process\_size.length;

NextFit(block\_size, m, process\_size, n);

}

}

Output

Process No. Process Size Block No.

1 10 2

2 20 3

3 5 1