**OPERATING SYSTEMS**

LAB EXPERIMENT - 3

Submitted To: Prashant Giridhar Shambharkar Sir.

Submitted By: Manav Agrawal (2K19/CO/216)

---------------------------------------------------------------------

Aim:

Write a C program to simulate the following contiguous memory allocation Techniques a) Worst fit b) Best fit c) First fit.

Introduction:

### Contiguous Memory Allocation:

Contiguous memory allocation is basically a method in which a single contiguous section/part of memory is allocated to a process or file needing it. Because of this all the available memory space resides at the same place together, which means that the freely/unused available memory partitions are not distributed in a random fashion here and there across the whole memory space.The main memory is a combination of two main portions- one for the operating system and other for the user program. We can implement/achieve contiguous memory allocation by dividing the memory partitions into fixed size partitions.

### Non-Contiguous Memory Allocation:

Non-Contiguous memory allocation is basically a method on the contrary to a contiguous allocation method, allocating the memory space present in different locations to the process as per it’s requirements. As all the available memory space is in a distributed pattern so the freely available memory space is also scattered here and there. This technique of memory allocation helps to reduce the wastage of memory, which eventually gives rise to Internal and external fragmentation.

The following are three Contiguous Memory Allocation techniques which we will simulate in this lab experiment:

1. **Worst Fit:**

The worst fit approach is to locate the largest available free portion available in the main memory, so that the portion left will be big enough to be useful. If a large process comes at a later stage, then memory will not have space to accommodate it. It is the reverse of best fit.

1. **Best Fit:**

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to the actual process size needed.

1. **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. It finishes after finding the first suitable free partition.

Algorithms:

1. **Worst Fit:**

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.

1. **Best Fit:**

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.

1. **First Fit:**

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 the current block.

4- If size-of-process <= size-of-block if yes then

assign and check for the next process.

5- If not then keep checking the further blocks.

Implementation:

**A) Worst Fit:**

#include<bits/stdc++.h>

using namespace std;

void worstFit(int blockSize[], int m, int processSize[],int n)

{

int allocation[n];

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

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

{

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 (wstIdx != -1)

{

allocation[i] = wstIdx;

blockSize[wstIdx] -= processSize[i];

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

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

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize)/sizeof(blockSize[0]);

int n = sizeof(processSize)/sizeof(processSize[0]);

worstFit(blockSize, m, processSize, n);

return 0 ;

}

**B) Best Fit:**

#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++)

{

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 (bestIdx != -1)

{

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

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

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize)/sizeof(blockSize[0]);

int n = sizeof(processSize)/sizeof(processSize[0]);

bestFit(blockSize, m, processSize, n);

return 0 ;

}

**C) First Fit:**

#include<bits/stdc++.h>

using namespace std;

void firstFit(int blockSize[], int m,

int processSize[], int n)

{

int allocation[n];

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

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

{

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

{

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

{

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t"

<< processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

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

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

firstFit(blockSize, m, processSize, n);

return 0 ;

}

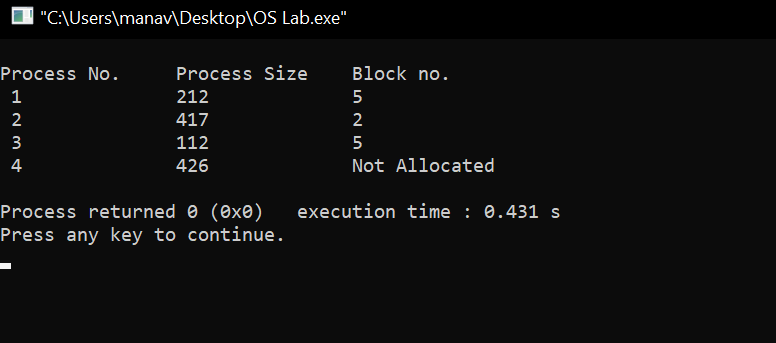
Output:

The Input given in the program for all three cases were:

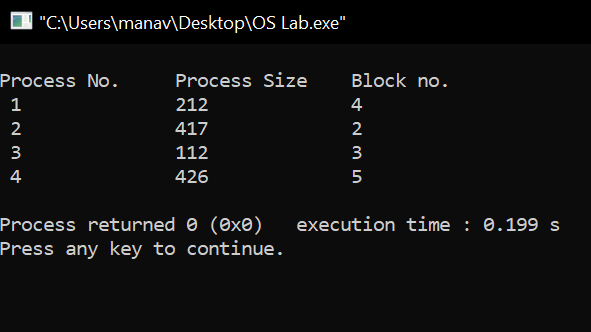
Block Size[] = {100, 500, 200, 300, 600};

Process Size[] = {212, 417, 112, 426};

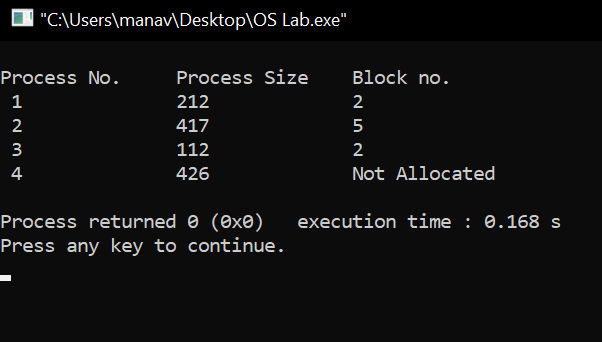
**A) Worst Fit:**

****

**B) Best Fit:**

****

**C) First Fit:**

****

Learning From The Experiment:

The algorithms are full of pros and also cons for some cases (given below) which we learned from this experiment;

Worst Fit 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 which is a great advantage of this algorithm. However, 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.

Advantage of Best fit is that memory utilization in best fit is much better than first fit as it searches the smallest free partition first available.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. Still, it has a disadvantage that it is slower and may even tend to fill up memory with tiny useless holes.

First fit is the fastest algorithm because it searches as little as possible. Although the disadvantage is that the remaining unused memory areas left after allocation become waste if it is too small. Thus requests for larger memory requirements cannot be accomplished.

***THANK YOU!***