EXP.NO:4	MEMORY ALLOCATION TECHNIQUES
DATE:	

AIM:

To write a program to implement memory allocation techniques using c.

ALGORITHM:

Memory allocation in operating systems is generally handled by various techniques, with some of the common approaches being:

1. First Fit: Allocates the first available memory block that is large enough to satisfy the request.
2. Best Fit: Allocates the smallest block of memory that is large enough to fulfill the request.
3. Worst Fit: Allocates the largest available memory block that can satisfy the request.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_BLOCKS 100

typedef struct {

    int size;

    int isFree;

} MemBlock;

MemBlock mem[MAX_BLOCKS];

int blkCount;

void initializeMemory(int blkSize[], int n) {

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

        mem[i].size = blkSize[i];

        mem[i].isFree = 1;

    }

    blkCount = n;

}

int firstFit(int ps) {

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

        if (mem[i].isFree && mem[i].size >= ps) {
```

```
 mem[i].isFree = 0;
            return i;
        }
    }
    return -1;
}
int bestFit(int ps) {
    int bestIndex = -1;
    for (int i = 0; i < blkCount; i++) {
        if (mem[i].isFree && mem[i].size >= ps) {
            if (bestIndex == -1 || mem[i].size < mem[bestIndex].size) {
                bestIndex = i;
            }
        }}
    if (bestIndex != -1) {
        mem[bestIndex].isFree = 0;
    }
    return bestIndex;
}
int worstFit(int ps) {
    int worstIndex = -1;
    for (int i = 0; i < blkCount; i++) {
        if (mem[i].isFree && mem[i].size >= ps) {
            if (worstIndex == -1 || mem[i].size > mem[worstIndex].size) {
                worstIndex = i;
            }}}
    if (worstIndex != -1) {
```

```
  mem[worstIndex].isFree = 0;
    }  return worstIndex;
}
void printMemory() {
    for (int i = 0; i < blkCount; i++) {
        printf("Block %d size = %d, %s\n", i + 1, mem[i].size,
              mem[i].isFree ? "Free" : "Allocated");
}}
int main() {
    int blkSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int n = sizeof(blkSize) / sizeof(blkSize[0]);
    int processCount = sizeof(processSize) / sizeof(processSize[0]);
    initializeMemory(blkSize, n);
    for (int i = 0; i < processCount; i++) {
        printf("\nAllocating process %d of size %d\n", i + 1, processSize[i];
        int allocation = firstFit(processSize[i]);
        if (allocation != -1) {
            printf("First Fit: Allocated to Block %d\n", allocation + 1);
        } else {
            printf("First Fit: No suitable block found\n");
        }
        initializeMemory(blkSize, n);
    }
    for (int i = 0; i < processCount; i++) {
        printf("\nAllocating process %d of size %d\n", i + 1, processSize[i]);
        int allocation = bestFit(processSize[i]);
```

OUTPUT:

Allocating process 1 of size 212

First Fit: Allocated to Block 2

Allocating process 2 of size 417

First Fit: Allocated to Block 2

Allocating process 3 of size 112

First Fit: Allocated to Block 2

Allocating process 1 of size 212

Best Fit: Allocated to Block 4

Allocating process 2 of size 417

Best Fit: Allocated to Block 2

Allocating process 3 of size 112

Best Fit: Allocated to Block 3

Allocating process 1 of size 212

Worst Fit: Allocated to Block 5

Allocating process 2 of size 417

Worst Fit: Allocated to Block 5

Allocating process 3 of size 112

Worst Fit: Allocated to Block 5

Block 1 size = 100, Free

Block 2 size = 500, Free

Block 3 size = 200, Free

Block 4 size = 300, Free

Block 5 size = 600, Free

```
        if (allocation != -1) {
            printf("Best Fit: Allocated to Block %d\n", allocation + 1);
        } else {
            printf("Best Fit: No suitable block found\n");
        }
        initializeMemory(blkSize, n);
    }
    for (int i = 0; i < processCount; i++) {
        printf("\nAllocating process %d of size %d\n", i + 1, processSize[i]);
        int allocation = worstFit(processSize[i]);
        if (allocation != -1) {
            printf("Worst Fit: Allocated to Block %d\n", allocation + 1);
        } else {
            printf("Worst Fit: No suitable block found\n");
        }
        initializeMemory(blkSize, n);
    }
    printMemory();
    return 0;
  }
}
```

PROGRAM &EXECUTION	
CLASS PERFORMNCE	
VIVA	
TOTAL	

RESULT:

Thus the program to implement c program Memory allocation Techniques has been verified successfully.

EXP.NO:5	DEADLOCK AVOIDANCE
DATE:	

AIM:

To write a program to implement deadlock avoidance using c.

ALGORITHM:

1) Work and Finish are two vectors, each with lengths of m and n.
Initialize: Work = Available Finish[i] is false when i=1, 2, 3, 4...n

2) Find an I such that both

a) Finish[i] = false

b) Need i <= Work if such an I does not exist. goto step (4)

3) Work = Work + Allocation[i] Finish[i] = true go to step (2)

4) If Finish[i] = true for each and every i then system is in a secure state.

PROGRAM:

```
#include<stdio.h>
int main()
{
  // P0 , P1 , P2 , P3 , P4 are the Process names here
  int n , m , i , j , k;
  n = 5; // Number of processes
  m = 3; // Number of resources
  int alloc[ 5 ] [ 3 ] = { { 0 , 1 , 0 }, // P0 // Allocation Matrix
                { 2 , 0 , 0 } , // P1
                { 3 , 0 , 2 } , // P2
                { 2 , 1 , 1 } , // P3
                { 0 , 0 , 2 } } ; // P4
  int max[ 5 ] [ 3 ] = { { 7 , 5 , 3 } , // P0 // MAX Matrix
              { 3 , 2 , 2 } , // P1
              { 9 , 0 , 2 } , // P2
              { 2 , 2 , 2 } , // P3
              { 4 , 3 , 3 } } ; // P4
  int avail[3] = { 3 , 3 , 2 } ; // Available Resources
  int f[n] , ans[n] , ind = 0 ;
  for (k = 0; k < n; k++) {
    f[k] = 0;
```

```
}
int need[n][m];
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j] ;
}
int y = 0;
for (k = 0; k < 5; k++){
    for (i = 0; i < n; i++){
        if (f[i] == 0){
            int flag = 0;
            for (j = 0; j < m; j++) {
                if(need[i][j] > avail[j]){
                    flag = 1;
                    break;
                }
            }
            if ( flag == 0 ) {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y] ;
                f[i] = 1;
            }
        }
    }
}
int flag = 1;
for(int i=0;i<n;i++)
{
if(f[i] == 0)
{
    flag = 0;
    printf(" The following system is not safe ");
```

OUTPUT:

Following is the SAFE Sequence

P1 -> P3 -> P4 -> P0 -> P2

.....

Process execute din 1.33 seconds

Press any key to continue.

```
        break;
    }
    }
    if (flag == 1)
    {
    printf(" Following is the SAFE Sequence \ n ");
    for (i = 0; i < n - 1; i++)
        printf(" P%d -> " , ans[i]);
    printf(" P%d ", ans[n - 1]);
    }
    return(0);
}
```

PROGRAM &EXECUTION	
CLASS PERFORMNCE	
VIVA	
TOTAL	

RESULT:

Thus the program to implement c program for deadlock avoidance has been verified successfully.

EXP.NO:6	DEADLOCK DETECTION ALGORITHM
DATE:	

AIM:

To write a program to implement deadlock detection algorithm using c.

ALGORITHM:

Step 1

1. Let Work(vector) length = m
2. Finish(vector) length = n
3. Initialize Work= Available.
4. if Allocation = 0 ∀ i ∈[0,N-1], then Finish[i] = true;

otherwise, Finish[i]= false.

Step 2

1. Find the index i with the conditions
2. Finish[i] == false
3. Work >= Request i

If exists no i, go to step 4.

Step 3

1. Work += Allocation i
2. Finish[i] = true

Go to Step 2.

Step 4

1. For some i in [0, N), if Finish[i]==false, the deadlock occurred. Finish [i]==false, the process Pi is deadlocked.

PROGRAM:

```
#include <stdio.h>

#define MAX_PROCESSES 10

#define MAX_RESOURCES 10


int allocation[MAX_PROCESSES][MAX_RESOURCES];

int request[MAX_PROCESSES][MAX_RESOURCES];

int available[MAX_RESOURCES];

int resources[MAX_RESOURCES];

int work[MAX_RESOURCES];

int marked[MAX_PROCESSES];
```

```
int main() {
 int num_processes, num_resources;

 printf("Enter the number of processes: ");
 scanf("%d", &num_processes);

 printf("Enter the number of resources: ");
 scanf("%d", &num_resources);

 // Input total resources
 for (int i = 0; i < num_resources; i++) {
 printf("Enter the total amount of Resource R%d: ", i + 1);
 scanf("%d", &resources[i]);
 }

 // Input request matrix
 printf("Enter the request matrix:\n");
 for (int i = 0; i < num_processes; i++) {
 for (int j = 0; j < num_resources; j++) {
 scanf("%d", &request[i][j]);
 }
 }

 // User Input allocation matrix
 printf("Enter the allocation matrix:\n");
 for (int i = 0; i < num_processes; i++) {
 for (int j = 0; j < num_resources; j++) {
 scanf("%d", &allocation[i][j]);
 }
 }

 // Initialization of the available resources
```

```
for (int j = 0; j < num_resources; j++) {
available[j] = resources[j];
for (int i = 0; i < num_processes; i++) {
available[j] -= allocation[i][j];
}
}

// Mark processes with zero allocation
for (int i = 0; i < num_processes; i++) {
int count = 0;
for (int j = 0; j < num_resources; j++) {
if (allocation[i][j] == 0) {
count++;
} else {
break;
}
}
if (count == num_resources) {
marked[i] = 1;
}
}

// Initialize work with available
for (int j = 0; j < num_resources; j++) {
work[j] = available[j];
}

// Mark processes with requests <= work
for (int i = 0; i < num_processes; i++) {
int can_be_processed = 1;
if (marked[i] != 1) {
for (int j = 0; j < num_resources; j++) {
```

```
if (request[i][j] > work[j]) {
can_be_processed = 0;
break;
}
}
if (can_be_processed) {
marked[i] = 1;
for (int j = 0; j < num_resources; j++) {
work[j] += allocation[i][j];
}
}
}
}

// Check for unmarked processes (deadlock)
int deadlock = 0;
for (int i = 0; i < num_processes; i++) {
if (marked[i] != 1) {
deadlock = 1;
break;
}
}

if (deadlock) {
printf("Deadlock detected\n");
} else {
printf("No deadlock possible\n");
}

return 0;
}
```

OUTPUT:

Enter the number of processes: 3

Enter the number of resources: 3

Enter the total amount of Resource R1: 4

Enter the total amount of Resource R2: 5

Enter the total amount of Resource R3: 7

Enter the request matrix:

1 2 3

4 5 6

7 8 9

Enter the allocation matrix:

5 6 7 1 2 3 7 8 9

Deadlock detected

PROGRAM &EXECUTION	
CLASS PERFORMNCE	
VIVA	
TOTAL	

RESULT:

Thus the program to implement c program for deadlock detection algorithm has been verified successfully