#include<stdio.h> //first-fit 8

// Function to allocate memory to

// blocks as per First fit algorithm

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

{

    int i, j;

    // Stores block id of the

    // block allocated to a process

    int allocation[n];

    // Initially no block is assigned to any process

    for(i = 0; i < n; i++){

        allocation[i] = -1;

    }

    // pick each process and find suitable blocks

    // according to its size ad assign to it

    for (i = 0; i < n; i++)  //here, n -> number of processes

    {

        for (j = 0; j < m; j++)  //here, m -> number of blocks

        {

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

                // allocating block j to the ith process

                allocation[i] = j;

                // Reduce available memory in this block.

                blockSize[j] -= processSize[i];

                break; //go to the next process in the queue

            }

        }

    }

    printf("\nProcess No.\tProcess Size\tBlock no.\n");

    for (int i = 0; i < n; i++)

    {

        printf(" %i\t\t\t", i+1);

        printf("%i\t\t\t\t", processSize[i]);

        if (allocation[i] != -1)

            printf("%i", allocation[i] + 1);

        else

            printf("Not Allocated");

        printf("\n");

    }

}

int main(){

    int m; //number of blocks in the memory

    int n; //number of processes in the input queue

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

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

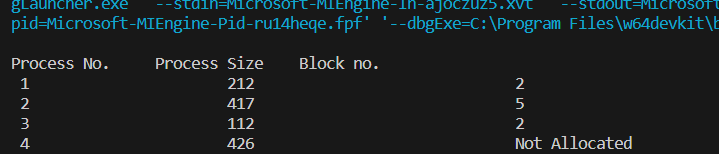
    m = sizeof(blockSize) / sizeof(blockSize[0]);

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

    firstFit(blockSize, m, processSize, n);

    return 0 ;

}



// best fit 8

#include <stdio.h>

int main()

{

    int i, j, alloc[100], avail[100], min;

    int blockSize[4] = {100, 200, 150, 300}, nb = 4;

    int jobSize[5] = {100, 90, 200, 50, 400}, n = 5;

    for (i = 0; i < n; i++){

        alloc[i] = -1;

    }

    for (i = 0; i < nb; i++){

        avail[i] = 9999;

    }

    // Check for each process the blocks available

    for (i = 0; i < n; i++){

        for (j = 0; j < nb; j++){

            if (blockSize[j] > jobSize[i])

            {

                avail[j] = blockSize[j] - jobSize[i];

            }

        }

        min = 0;

        for (j = 0; j < nb; j++){

            if (avail[min] > avail[j]) {

                min = j;

            }

        }

        alloc[i] = min;

        if (avail[min] >= 9999) {

            alloc[i] = -1;

        }

        blockSize[min] = -1;

        //  Initialize avail to 99999

        for (j = 0; j < n; j++){

            avail[j] = 9999;

        }

    }

    // Print the results

    printf("Process P of {size} is allocated to block \n");

    for (i = 0; i < n; i++){

        if (alloc[i] != -1)

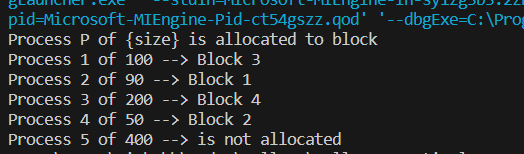
            printf("Process %d of %d --> Block %d\n", i + 1, jobSize[i], alloc[i] + 1);

        else

            printf("Process %d of %d --> is not allocated \n", i + 1, jobSize[i]);

    }

}



#include <stdio.h> //9

#define MAXTRACE 20 // Expt. 9 Program for FIFO page replacement algorithm int

int main()

{

    int page\_faults = 0, m, n, s, pages, frames, pf, k = 0;

    pages = 10;

    int reference\_string[10] = {20, 10, 30, 50, 40, 70, 90, 34, 89, 93};

    frames = 4;

    int temp[frames];

    for (m = 0; m < frames; m++)

    {

        temp[m] = -1;

    }

    for (pf = 0; pf < frames; pf++)

        printf("PF#%-2d\t", pf + 1);

    for (m = 0; m < pages; m++){

        s = 0;

        for (n = 0; n < frames;

             n++)

        {

            if (reference\_string[m] == temp[n])

            {

                s++;

                page\_faults--;

            }

        }

        page\_faults++;

        if ((page\_faults <= frames) && (s == 0))

        {

            temp[k++] = reference\_string[m];

        }

        else if (s == 0)

        {

            temp[(page\_faults -

                  1) %

                 frames] = reference\_string[m];

        }

        printf("\n");

        for (n = 0; n < frames; n++)

        {

            printf("%d\t",

                   temp[n]);

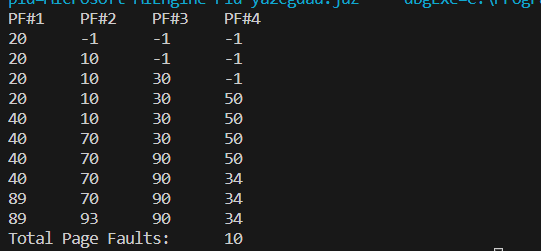
        }

    }

    printf("\nTotal Page Faults:\t%d\n", page\_faults);

    return 0;

}



// Expt. no. 10

// Banker's  safety algorithm.

#include <stdio.h>

#define n 5 // number of processess

#define m 3 // number of resource types

#define true 1

#define false 0

typedef int boolean;

int main(){

    int i, j, k, l;

    boolean temp;

    int safe\_seq[n] = {-1, -1, -1, -1, -1};

    boolean finish[5] = {false, false, false, false, false}; // initially no process is finished

    int total[m] = {10, 5, 7};                               // total number of system resource types are three

    int max[n][m] = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};

    int alloc[n][m] = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};

    int available[m];

    int need[n][m];

    int work[m];

    int nf = 0; // number of finished processes

    int s[3];   // summation of allocation

    s[0] = s[1] = s[2] = 0;

    // calculations for available vector

    for (j = 0; j < m; j++)

    {

        for (i = 0; i < n; i++)

            s[j] = s[j] + alloc[i][j];

        available[j] = total[j] - s[j];

    }

    // print available vector

    printf("\nThe available vector is :");

    for (j = 0; j < m; j++)

    {

        printf("%d\t", available[j]);

    }

    // calculations for need matrix

    for (i = 0; i < n; i++) // first find a process that can finish

    {

        for (j = 0; j < m; j++)

            need[i][j] = max[i][j] - alloc[i][j];

    }

    // initialise work vector with available vector

    for (j = 0; j < m; j++)

        work[j] = available[j];

    boolean statechange = false;

    do

    {

        // Apply the Banker's Avoidance core logic

        for (i = 0; i < n; i++) // first find a process that can finish

        {

            temp = true; // assume that process can finish

            if (finish[i] == false)

            {

                for (j = 0; j < m; j++) // for each resource type

                {

                    if (need[i][j] > work[j])

                    {

                        statechange = false;

                        temp = false; // process can not finish

                        break;

                    }

                }

                if (temp) // if this process can finish

                {

                    statechange = true;

                    finish[i] = true;

                    printf("\nProcess finished =P%d", i);

                    safe\_seq[nf++] = i;

                    for (l = 0; l < m; l++)

                        work[l] += alloc[i][l];

                }

            }

        } // end of core logic

        if (nf == n)

            statechange = false;

    } while (statechange); // if there is no process finished in the last cyle, stop

    // if all processes are finished, print the safe sequence safe\_seq

    if (nf < n)

    {

        printf("\nNo process can finish safely further...");

        printf("\nNo safe sequence exist, hence the system is in unsafe state");

    }

    else // print the safe sequence

    {

        printf("\n\n The safe sequence is:\n");

        for (i = 0; i < n; i++)

            printf("\tP%d ", safe\_seq[i]);

    }

}

