Ragu.v

191921031

                                                         Operating System Lab

1)

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define NUM\_FRAMES 3

#define NUM\_PAGES 10

int frames[NUM\_FRAMES];

int pages[NUM\_PAGES];

int page\_faults = 0;

int next\_replace = 0;

void initializeFrames() {

    for (int i = 0; i < NUM\_FRAMES; i++) {

        frames[i] = -1; // Initialize frames to -1 (indicating an empty frame)

    }

}

bool isPageInFrames(int page) {

    for (int i = 0; i < NUM\_FRAMES; i++) {

        if (frames[i] == page) {

            return true;

        }

    }

    return false;

}

int findNextReference(int start) {

    for (int i = start; i < NUM\_PAGES; i++) {

        if (!isPageInFrames(pages[i])) {

            return i;

        }

    }

    return INT\_MAX;

}

void replacePage(int page) {

    frames[next\_replace] = page;

    next\_replace = (next\_replace + 1) % NUM\_FRAMES;

}

void simulateOptimal() {

    for (int i = 0; i < NUM\_PAGES; i++) {

        printf("Accessing Page %d: ", pages[i]);

        if (!isPageInFrames(pages[i])) {

            page\_faults++;

            int max\_future\_reference = -1;

            int page\_to\_replace = -1;

            for (int j = 0; j < NUM\_FRAMES; j++) {

                int next\_reference = findNextReference(i + 1);

                if (next\_reference > max\_future\_reference) {

                    max\_future\_reference = next\_reference;

                    page\_to\_replace = j;

                }

            }

            replacePage(pages[i]);

        }

        for (int j = 0; j < NUM\_FRAMES; j++) {

            if (frames[j] == -1) {

                printf("- ");

            } else {

                printf("%d ", frames[j]);

            }

        }

        printf("\n");

    }

}

int main() {

    initializeFrames();

    printf("Enter the sequence of page references (0-9):\n");

    for (int i = 0; i < NUM\_PAGES; i++) {

        scanf("%d", &pages[i]);

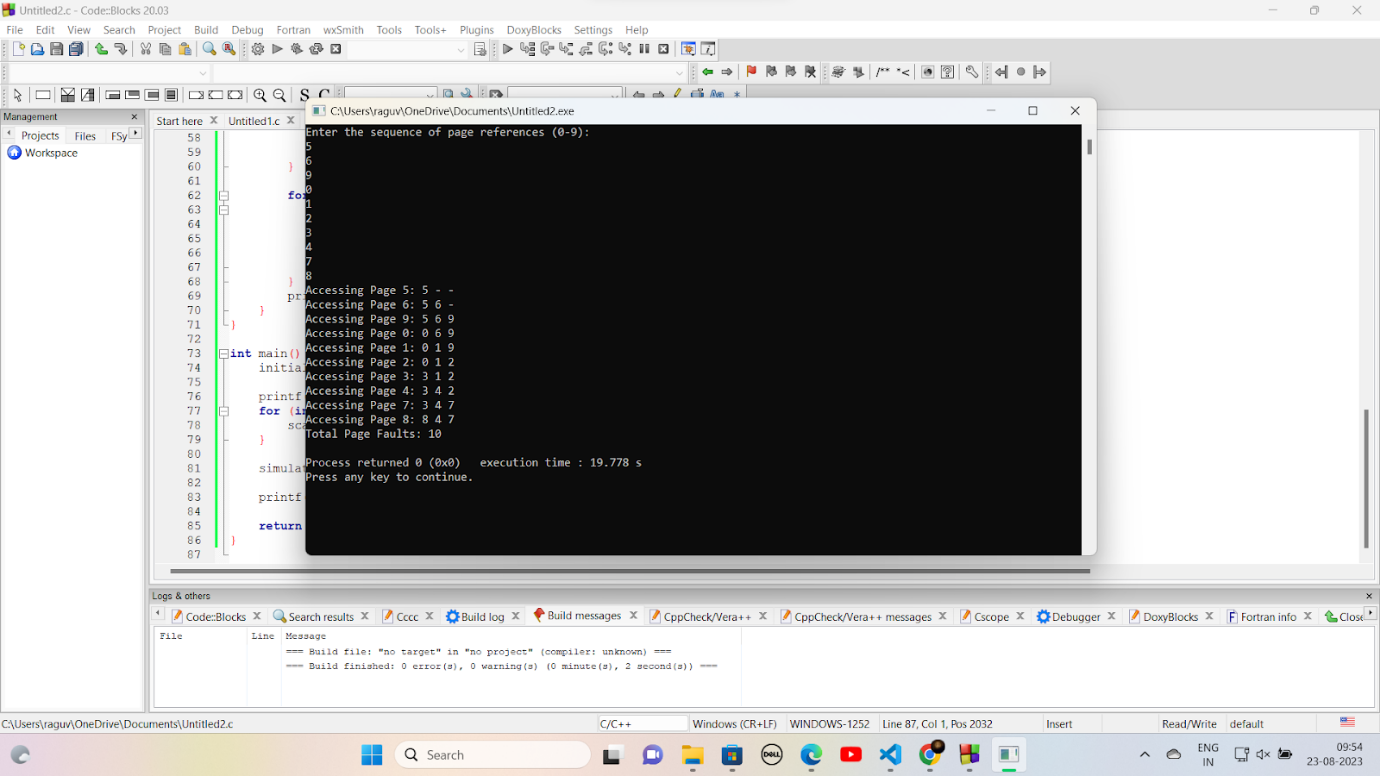
    }

    simulateOptimal();

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

    return 0;

}



EX.9 Write C programs to Implement the various File Organization Techniques

a. Sequential File Organization

Program

#include <stdio.h>

#include <string.h>

struct Student {

    int roll\_number;

    char name[50];

};

int main() {

    struct Student student;

    FILE \*file;

    file = fopen("students.dat", "wb");

    if (file == NULL) {

        printf("Error opening file.\n");

        return 1;

    }

    for (int i = 0; i < 3; i++) {

        printf("Enter details for Student %d:\n", i + 1);

        printf("Roll Number: ");

        scanf("%d", &student.roll\_number);

        printf("Name: ");

        scanf("%s", student.name);

        fwrite(&student, sizeof(struct Student), 1, file);

    }

    fclose(file);

    file = fopen("students.dat", "rb");

    if (file == NULL) {

        printf("Error opening file.\n");

        return 1;

    }

    printf("\nStudent details:\n");

    while (fread(&student, sizeof(struct Student), 1, file)) {

        printf("Roll Number: %d, Name: %s\n", student.roll\_number, student.name);

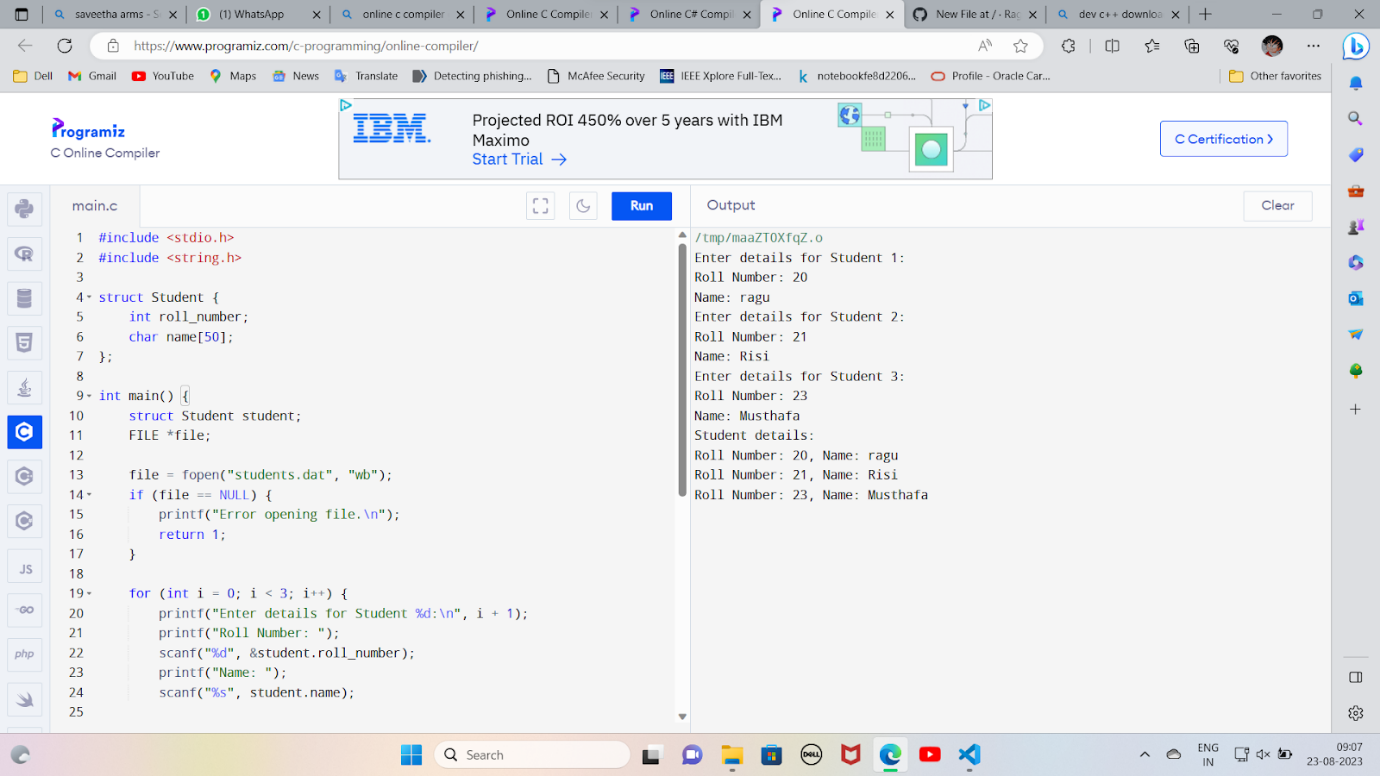
    }

    fclose(file);

    return 0;

}

Output:



b. Indexed File Organization

#include <stdio.h>

#include <string.h>

struct Student {

    int roll\_number;

    char name[50];

};

struct Index {

    int roll\_number;

    long offset;

};

int main() {

    struct Student student;

    struct Index index;

    FILE \*data\_file, \*index\_file;

    data\_file = fopen("students.dat", "wb");

    if (data\_file == NULL) {

        printf("Error opening data file.\n");

        return 1;

    }

    index\_file = fopen("index.dat", "wb");

    if (index\_file == NULL) {

        printf("Error opening index file.\n");

        return 1;

    }

    for (int i = 0; i < 3; i++) {

        printf("Enter details for Student %d:\n", i + 1);

        printf("Roll Number: ");

        scanf("%d", &student.roll\_number);

        printf("Name: ");

        scanf("%s", student.name);

        fwrite(&student, sizeof(struct Student), 1, data\_file);

        index.roll\_number = student.roll\_number;

        index.offset = ftell(data\_file) - sizeof(struct Student);

        fwrite(&index, sizeof(struct Index), 1, index\_file);

    }

    fclose(data\_file);

    fclose(index\_file);

    int roll;

    printf("\nEnter Roll Number to retrieve details: ");

    scanf("%d", &roll);

    index\_file = fopen("index.dat", "rb");

    if (index\_file == NULL) {

        printf("Error opening index file.\n");

        return 1;

    }

    int found = 0;

    while (fread(&index, sizeof(struct Index), 1, index\_file)) {

        if (index.roll\_number == roll) {

            found = 1;

            break;

        }

    }

    if (found) {

        data\_file = fopen("students.dat", "rb");

        if (data\_file == NULL) {

            printf("Error opening data file.\n");

            return 1;

        }

        fseek(data\_file, index.offset, SEEK\_SET);

        fread(&student, sizeof(struct Student), 1, data\_file);

        printf("\nStudent details:\n");

        printf("Roll Number: %d, Name: %s\n", student.roll\_number, student.name);

        fclose(data\_file);

    } else {

        printf("Student not found.\n");

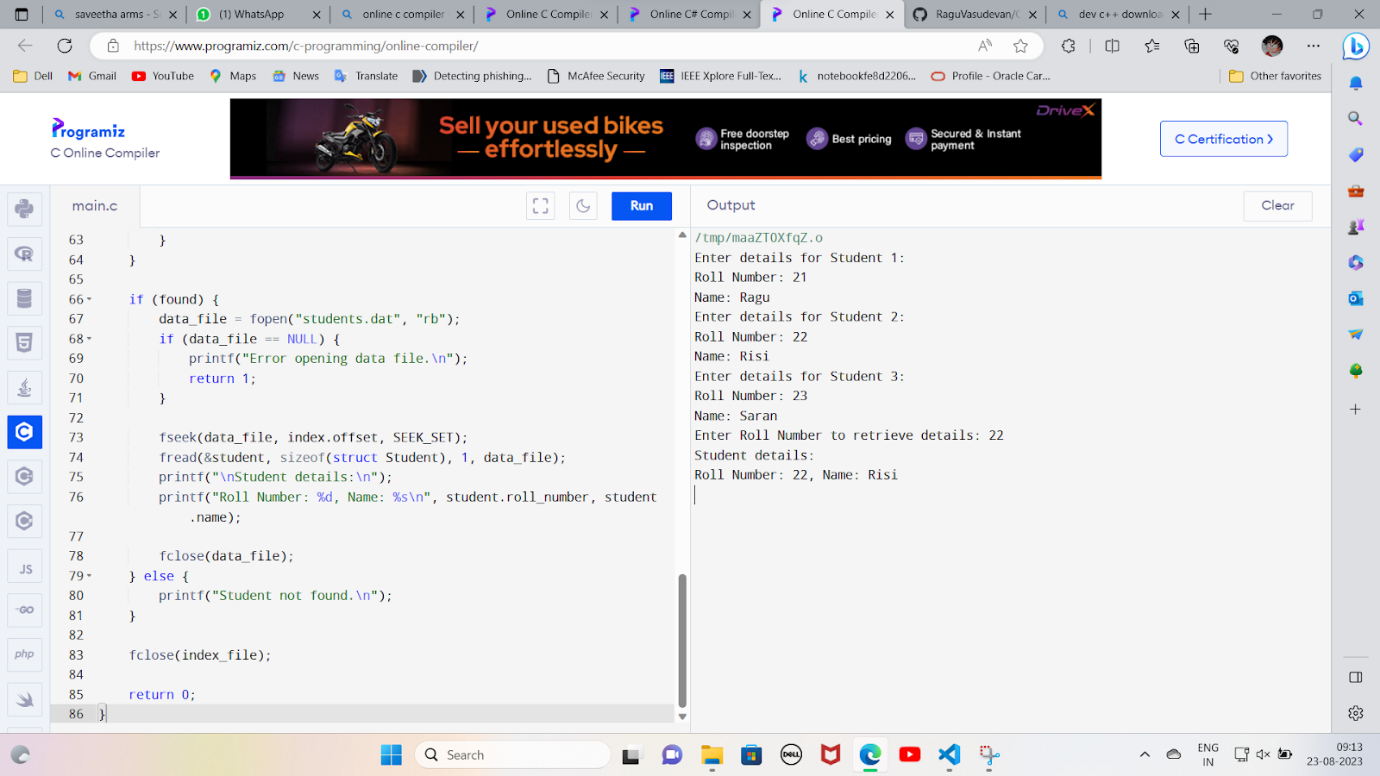
    }

    fclose(index\_file);

    return 0;

}

Output:



C. Indexed

Program

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct File {

    char name[50];

    int index\_block;

};

struct IndexBlock {

    int blocks[10]; // Assuming each index block can point to 10 data blocks

};

int main() {

    struct File files[10];

    struct IndexBlock index\_blocks[10];

    int data\_blocks[100] = {0}; // Initialize all data blocks as free

    for (int i = 0; i < 10; i++) {

        printf("Enter name of File %d: ", i + 1);

        scanf("%s", files[i].name);

        printf("Enter number of blocks needed for File %d: ", i + 1);

        int num\_blocks;

        scanf("%d", &num\_blocks);

        int allocated\_blocks = 0;

        for (int j = 0; j < num\_blocks; j++) {

            for (int k = 0; k < 100; k++) {

                if (data\_blocks[k] == 0) {

                    data\_blocks[k] = 1; // Mark data block as allocated

                    index\_blocks[i].blocks[j] = k; // Store data block index in index block

                    allocated\_blocks++;

                    break;

                }

            }

        }

        index\_blocks[i].blocks[num\_blocks] = -1; // Mark end of index block

        files[i].index\_block = i; // Set index block number for the file

        printf("File %s allocated %d blocks.\n", files[i].name, allocated\_blocks);

    }

    printf("\nFile Allocation Table:\n");

    for (int i = 0; i < 10; i++) {

        printf("File: %s, Index Block: %d\n", files[i].name, files[i].index\_block);

        printf("Index Block: ");

        for (int j = 0; index\_blocks[i].blocks[j] != -1; j++) {

            printf("%d ", index\_blocks[i].blocks[j]);

        }

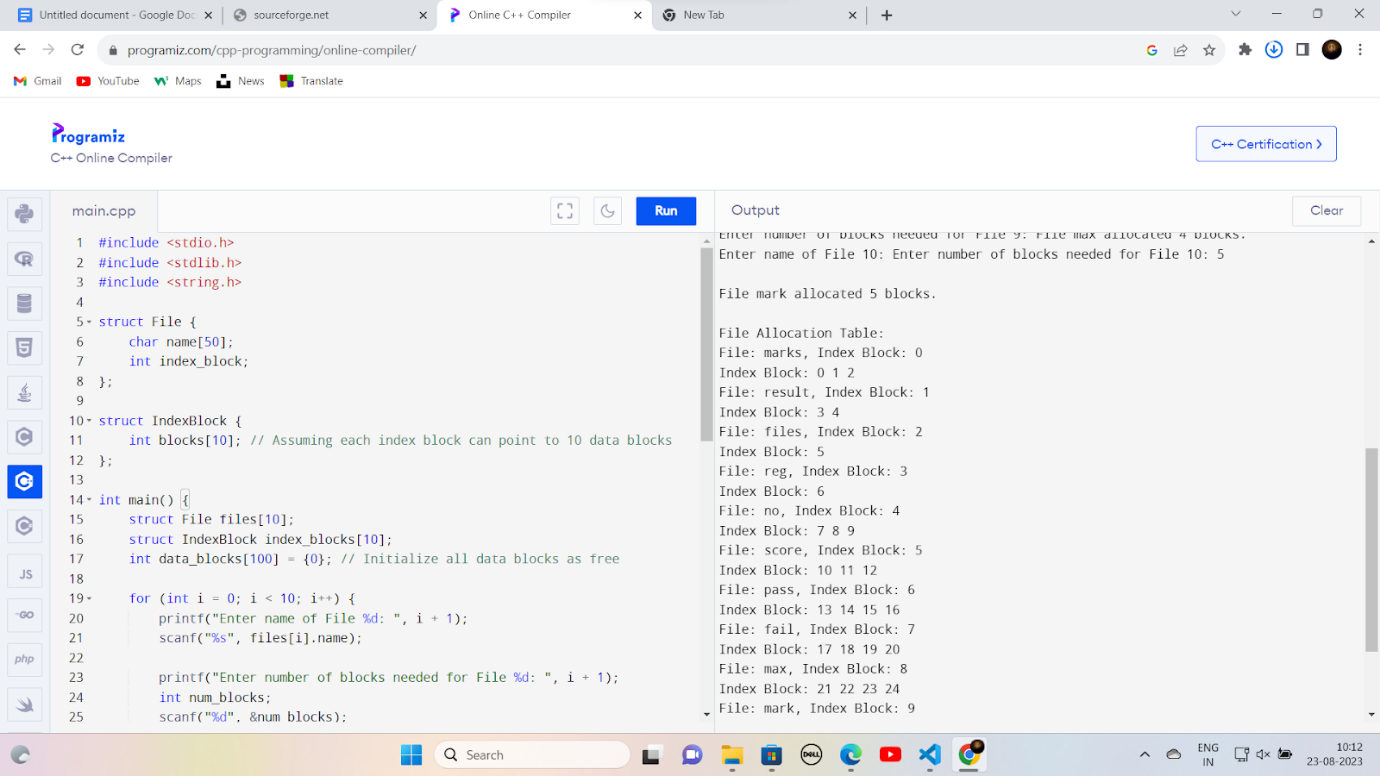
        printf("\n");

    }

    return 0;

}

OUTPUT



Program for DeadLock:

#include <bits/stdc++.h>

using namespace std;

int arrmax[100][100];

int alloc[100][100];

int need[100][100];

int avail[100];

int n, r;

void input()

{

    int i, j;

    cout << "Enter the no of Processes\t";

    cin >> n;

    cout << "Enter the no of resource instances\t";

    cin >> r;

    cout << "Enter the Max Matrix\n";

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < r; j++)

        {

            cin >> arrmax[i][j];

        }

    }

    cout << "Enter the Allocation Matrix\n";

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < r; j++)

        {

            cin >> alloc[i][j];

        }

    }

    cout << "Enter the available Resources\n";

    for (j = 0; j < r; j++)

    {

        cin >> avail[j];

    }

}

void show()

{

    int i, j;

    cout << "Process\t Allocation\t Max\t Available\t";

    for (i = 0; i < n; i++)

    {

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

        for (j = 0; j < r; j++)

        {

            cout << alloc[i][j] << " ";

        }

        cout << "\t\t";

        for (j = 0; j < r; j++)

        {

            cout << arrmax[i][j] << " ";

        }

        cout << "\t ";

        if (i == 0)

        {

            for (j = 0; j < r; j++)

                cout << avail[j] << " ";

        }

    }

}

void cal()

{

    int finish[100], temp, need[100][100], flag = 1, k, c1 = 0;

    int dead[100];

    int safe[100];

    int i, j;

    for (i = 0; i < n; i++)

    {

        finish[i] = 0;

    }

    //find need matrix

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < r; j++)

        {

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

        }

    }

    while (flag)

    {

        flag = 0;

        for (i = 0; i < n; i++)

        {

            int c = 0;

            for (j = 0; j < r; j++)

            {

                if ((finish[i] == 0) && (need[i][j] <= avail[j]))

                {

                    c++;

                    if (c == r)

                    {

                        for (k = 0; k < r; k++)

                        {

                            avail[k] += alloc[i][j];

                            finish[i] = 1;

                            flag = 1;

                        }

                        //cout<<"\nP%d",i;

                        if (finish[i] == 1)

                        {

                            i = n;

                        }

                    }

                }

            }

        }

    }

    j = 0;

    flag = 0;

    for (i = 0; i < n; i++)

    {

        if (finish[i] == 0)

        {

            dead[j] = i;

            j++;

            flag = 1;

        }

    }

    if (flag == 1)

    {

        cout << "\n\nSystem is in Deadlock and the Deadlock process are\n";

        for (i = 0; i < n; i++)

        {

            cout << "P" << dead[i] << "\t";

        }

    }

    else

    {

        cout << "\nNo Deadlock Occur";

    }

}

int main()

{

    int i, j;

    cout << "\*\*\*\* Deadlock Detection Algorithm \*\*\*\*\n";

    input();

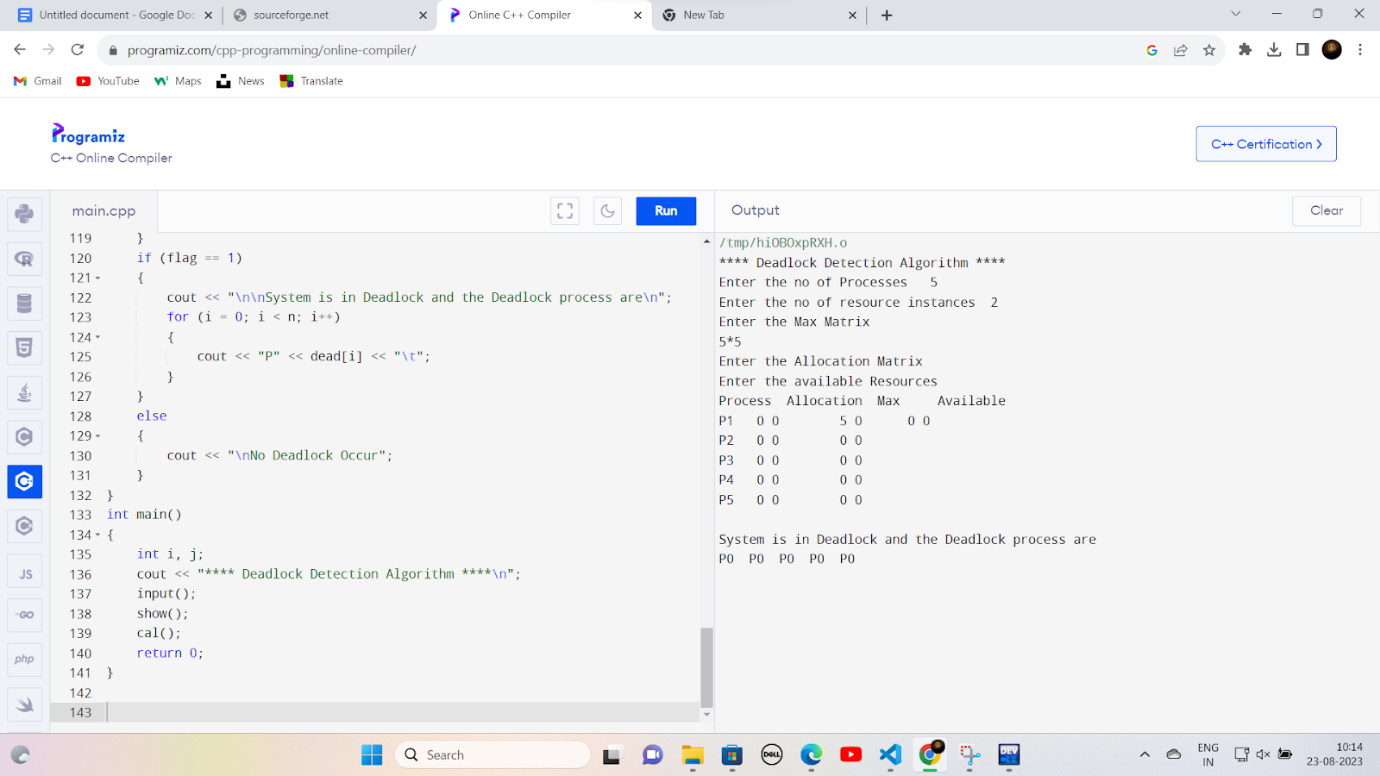
    show();

    cal();

    return 0;

}

Output



 EX/10Process Management using System Calls : Fork, Exit, Getpid, Wait, Close

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main() {

    int child\_pid;

    // Fork a child process

    child\_pid = fork();

    if (child\_pid == -1) {

        perror("Fork failed");

        return 1;

    }

    if (child\_pid == 0) {

        // This is the child process

        printf("Child Process:\n");

        printf("PID: %d\n", getpid());

        printf("Parent PID: %d\n", getppid());

        printf("Child process exiting...\n");

        exit(0);

    } else {

        // This is the parent process

        printf("Parent Process:\n");

        printf("PID: %d\n", getpid());

        int status;

        wait(&status); // Wait for the child process to exit

        printf("Child process has exited.\n");

        printf("Parent process closing...\n");

    }

    return 0;

}

Output:

Thread

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

int g = 0;

void \*myThreadFun(void \*vargp)

{

         int \*myid = (int \*)vargp;

         static int s = 0;

         ++s; ++g;

         printf("Thread ID: %d, Static: %d, Global: %d\n", \*myid, ++s, ++g);

}

int main()

{

         int i;

         pthread\_t tid;

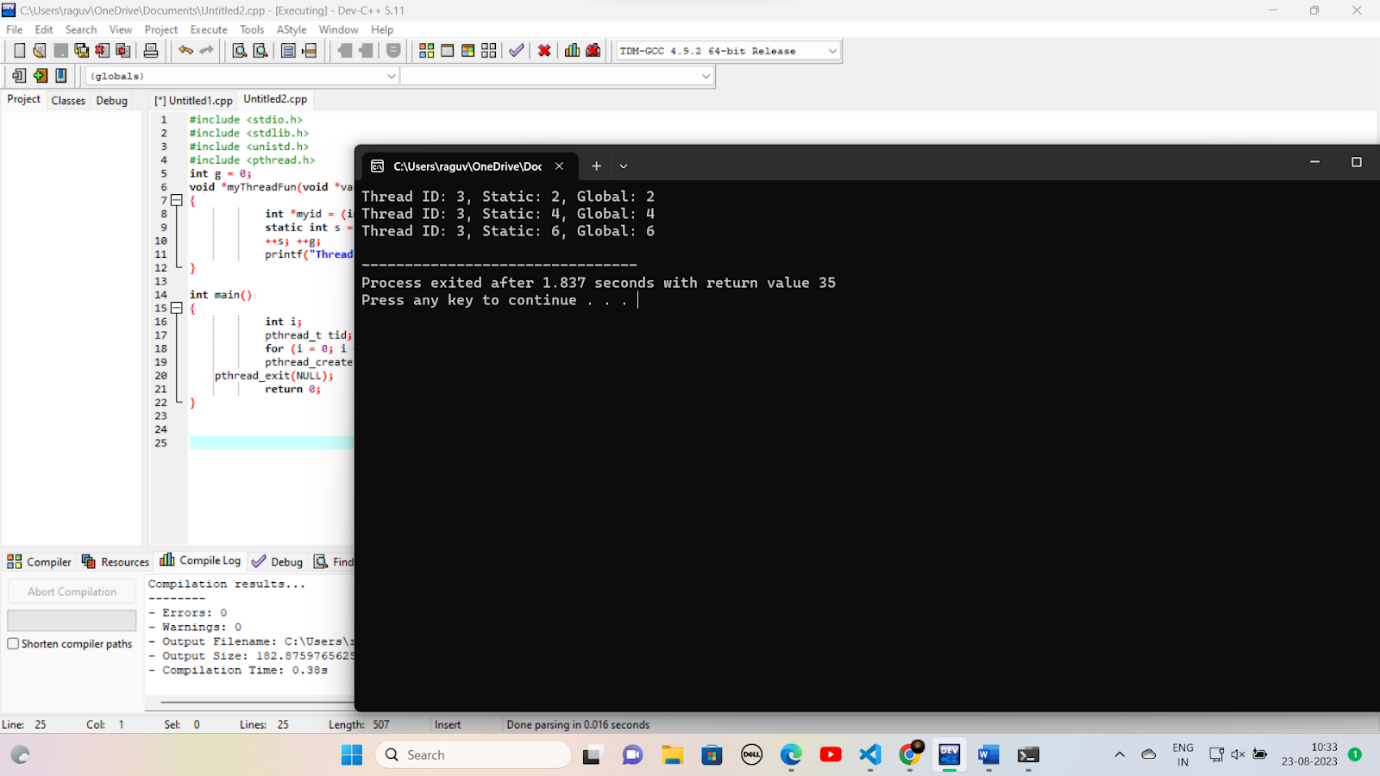
         for (i = 0; i < 3; i++)

         pthread\_create(&tid, NULL, myThreadFun, (void \*)&tid);

  pthread\_exit(NULL);

         return 0;

}



FIFO

#include <stdio.h>

#define NUM\_FRAMES 3

#define NUM\_PAGES 10

int frames[NUM\_FRAMES];

int pages[NUM\_PAGES];

int page\_faults = 0;

void initializeFrames() {

for (int i = 0; i < NUM\_FRAMES; i++) {

     frames[i] = -1; // Initialize frames to -1 (indicating an empty frame)

}

}

int findLRUIndex() {

int min = frames[0], index = 0;

for (int i = 0; i < NUM\_FRAMES; i++) {

     if (frames[i] < min) {

         min = frames[i];

         index = i;

     }

}

return index;

}

int isPageInFrames(int page) {

for (int i = 0; i < NUM\_FRAMES; i++) {

     if (frames[i] == page) {

         return 1;

     }

}

return 0;

}

void printFrames() {

for (int i = 0; i < NUM\_FRAMES; i++) {

     if (frames[i] == -1) {

         printf("- ");

     } else {

         printf("%d ", frames[i]);

     }

}

printf("\n");

}

void simulatePaging() {

for (int i = 0; i < NUM\_PAGES; i++) {

     printf("Accessing Page %d: ", pages[i]);

     if (!isPageInFrames(pages[i])) {

         page\_faults++;

         int emptyIndex = -1;

         for (int j = 0; j < NUM\_FRAMES; j++) {

             if (frames[j] == -1) {

                 emptyIndex = j;

                 break;

             }

         }

         if (emptyIndex != -1) {

             frames[emptyIndex] = pages[i];

         } else {

             int replaceIndex = findLRUIndex();

             frames[replaceIndex] = pages[i];

         }

     }

     printFrames();

}

}

int main()

{

printf("Enter the sequence of page references (0-9):\n");

for (int i = 0; i < NUM\_PAGES; i++) {

     scanf("%d", &pages[i]);

}

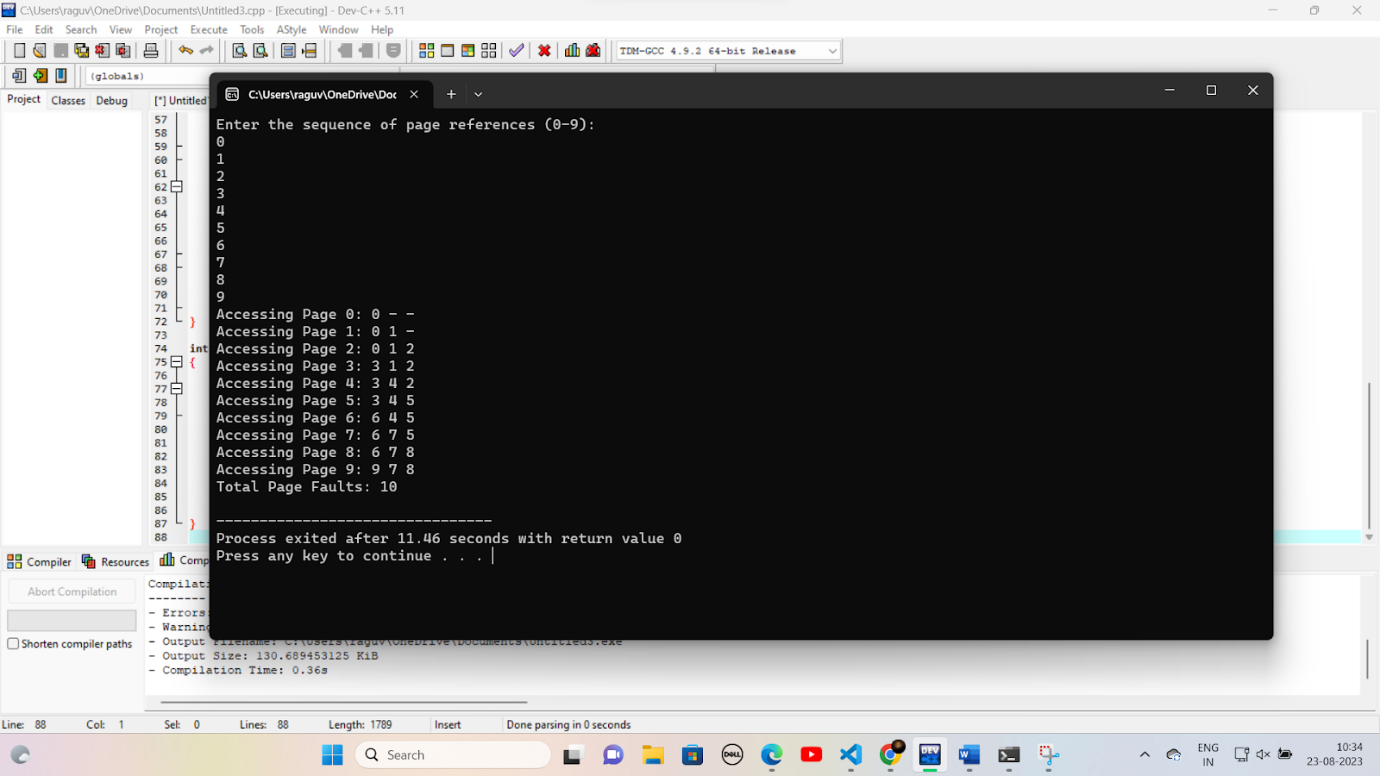
initializeFrames();

simulatePaging();

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

return 0;

}



First fit

#include <stdio.h>

#define MEMORY\_SIZE 100

#define MAX\_BLOCKS 10

int memory[MEMORY\_SIZE];

int blocks[MAX\_BLOCKS];

int num\_blocks;

void initializeMemory() {

for (int i = 0; i < MEMORY\_SIZE; i++) {

     memory[i] = -1; // Initialize memory to -1 (indicating free space)

}

}

void firstFit(int process\_id, int size) {

for (int i = 0; i < MEMORY\_SIZE; i++) {

     if (memory[i] == -1) { // Check for free space

         int j, count = 0;

         for (j = i; j < MEMORY\_SIZE && memory[j] == -1; j++) {

             count++;

             if (count == size) {

                 break;

             }

         }

         if (count == size) {

             for (int k = i; k < j; k++) {

                 memory[k] = process\_id;

             }

             printf("Process %d allocated at memory location %d\n", process\_id, i);

             return;

         }

     }

}

printf("Insufficient memory to allocate Process %d\n", process\_id);

}

int main() {

initializeMemory();

printf("Enter number of memory blocks: ");

scanf("%d", &num\_blocks);

printf("Enter size of each block:\n");

for (int i = 0; i < num\_blocks; i++) {

     scanf("%d", &blocks[i]);

}

for (int i = 0; i < num\_blocks; i++) {

     int process\_id, size;

     printf("Enter process ID and size for Block %d: ", i + 1);

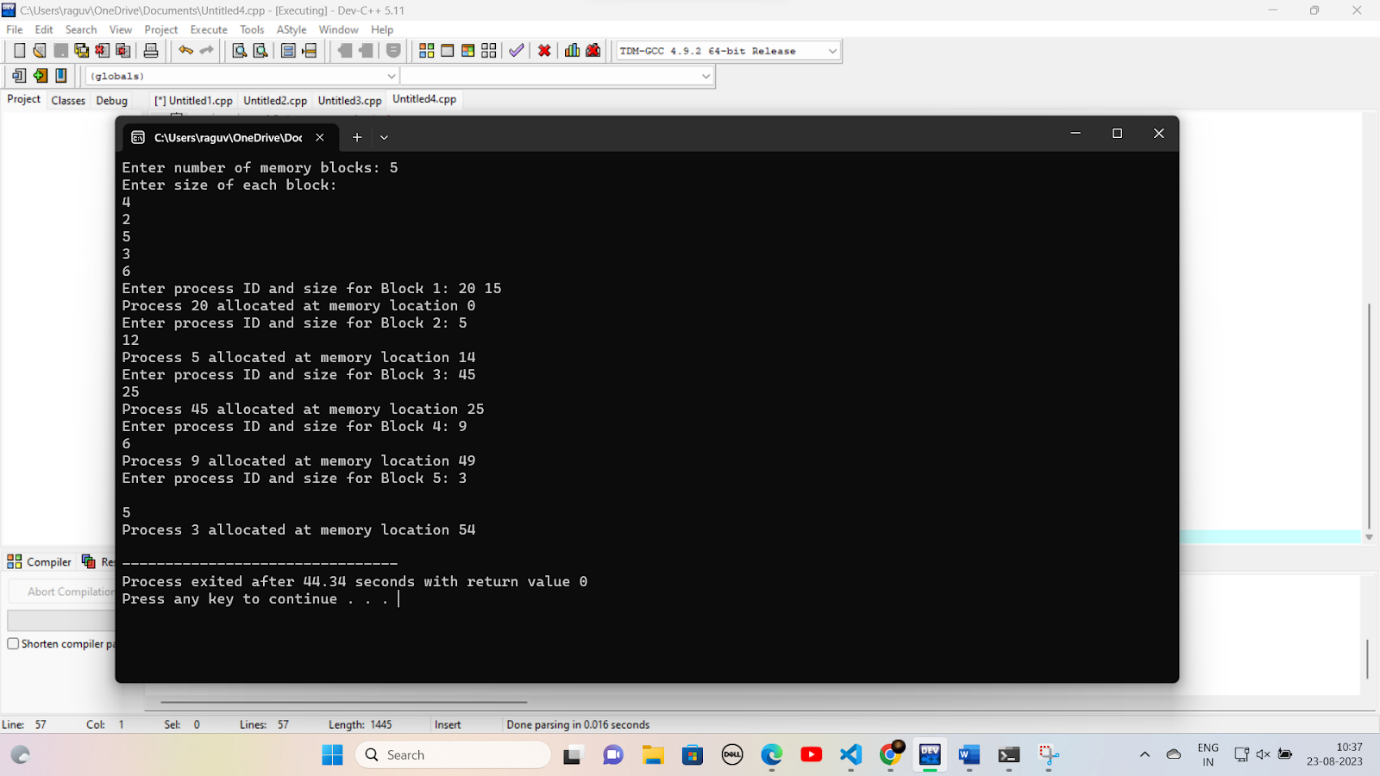
     scanf("%d %d", &process\_id, &size);

     firstFit(process\_id, size);

}

return 0;

}



Worst fit

#include <stdio.h>

#define MEMORY\_SIZE 100

#define MAX\_BLOCKS 10

int memory[MEMORY\_SIZE];

int blocks[MAX\_BLOCKS];

int num\_blocks;

void initializeMemory() {

for (int i = 0; i < MEMORY\_SIZE; i++) {

     memory[i] = -1; // Initialize memory to -1 (indicating free space)

}

}

void worstFit(int process\_id, int size) {

int max\_size = 0, max\_index = -1;

for (int i = 0; i < MEMORY\_SIZE; i++) {

     if (memory[i] == -1) { // Check for free space

         int j, count = 0;

         for (j = i; j < MEMORY\_SIZE && memory[j] == -1; j++) {

             count++;

         }

         if (count >= size && count > max\_size) {

             max\_size = count;

             max\_index = i;

         }

     }

}

if (max\_index != -1) {

     for (int k = max\_index; k < max\_index + size; k++) {

         memory[k] = process\_id;

     }

     printf("Process %d allocated at memory location %d\n", process\_id, max\_index);

} else {

     printf("Insufficient memory to allocate Process %d\n", process\_id);

}

}

int main() {

initializeMemory();

printf("Enter number of memory blocks: ");

scanf("%d", &num\_blocks);

printf("Enter size of each block:\n");

for (int i = 0; i < num\_blocks; i++) {

     scanf("%d", &blocks[i]);

}

for (int i = 0; i < num\_blocks; i++) {

     int process\_id, size;

     printf("Enter process ID and size for Block %d: ", i + 1);

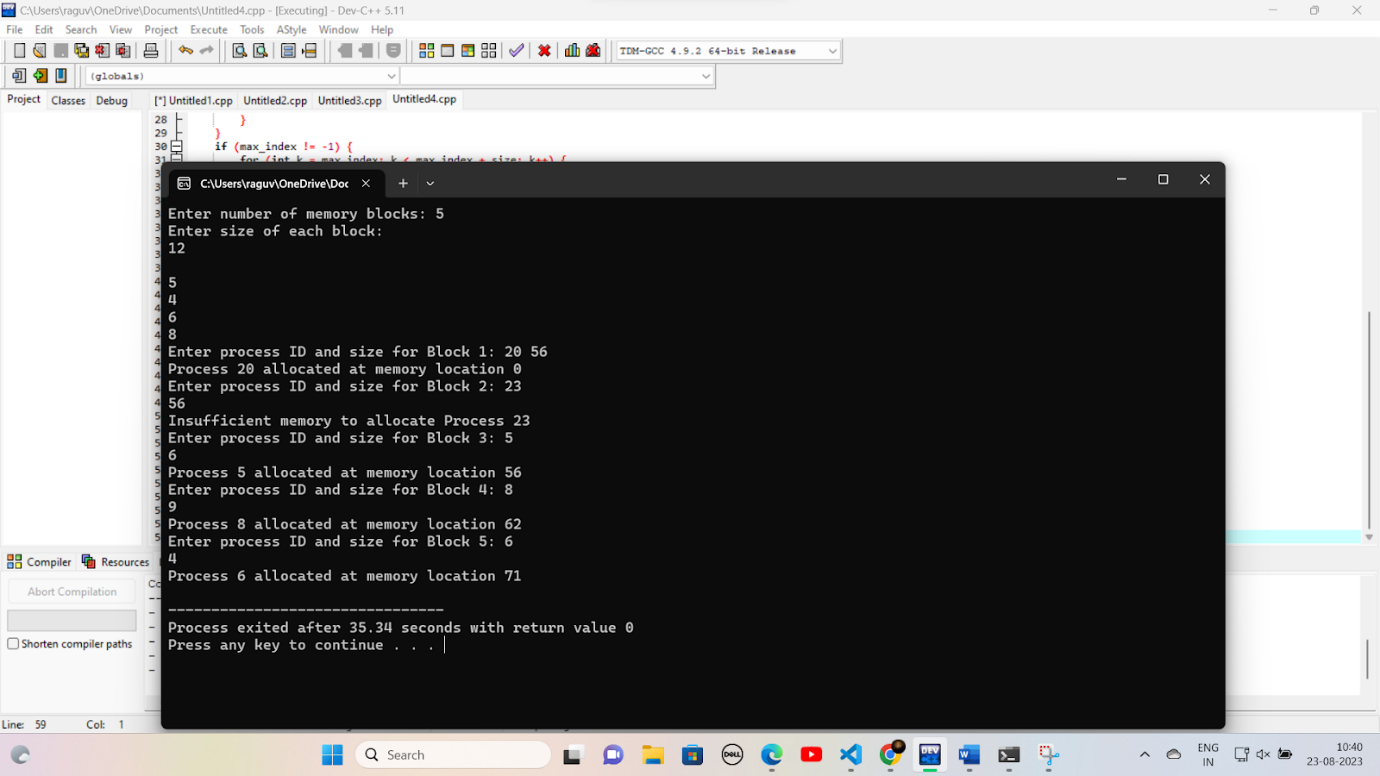
     scanf("%d %d", &process\_id, &size);

     worstFit(process\_id, size);

}

return 0;

}



Best fit:

#include <stdio.h>

#define MEMORY\_SIZE 100

#define MAX\_BLOCKS 10

int memory[MEMORY\_SIZE];

int blocks[MAX\_BLOCKS];

int num\_blocks;

void initializeMemory() {

for (int i = 0; i < MEMORY\_SIZE; i++) {

     memory[i] = -1; // Initialize memory to -1 (indicating free space)

}

}

void bestFit(int process\_id, int size) {

int min\_size = MEMORY\_SIZE, min\_index = -1;

for (int i = 0; i < MEMORY\_SIZE; i++) {

     if (memory[i] == -1) { // Check for free space

         int j, count = 0;

         for (j = i; j < MEMORY\_SIZE && memory[j] == -1; j++) {

             count++;

         }

         if (count >= size && count < min\_size) {

             min\_size = count;

             min\_index = i;

         }

     }

}

if (min\_index != -1) {

     for (int k = min\_index; k < min\_index + size; k++) {

         memory[k] = process\_id;

     }

     printf("Process %d allocated at memory location %d\n", process\_id, min\_index);

} else {

     printf("Insufficient memory to allocate Process %d\n", process\_id);

}

}

int main() {

initializeMemory();

printf("Enter number of memory blocks: ");

scanf("%d", &num\_blocks);

printf("Enter size of each block:\n");

for (int i = 0; i < num\_blocks; i++) {

     scanf("%d", &blocks[i]);

}

for (int i = 0; i < num\_blocks; i++) {

     int process\_id, size;

     printf("Enter process ID and size for Block %d: ", i + 1);

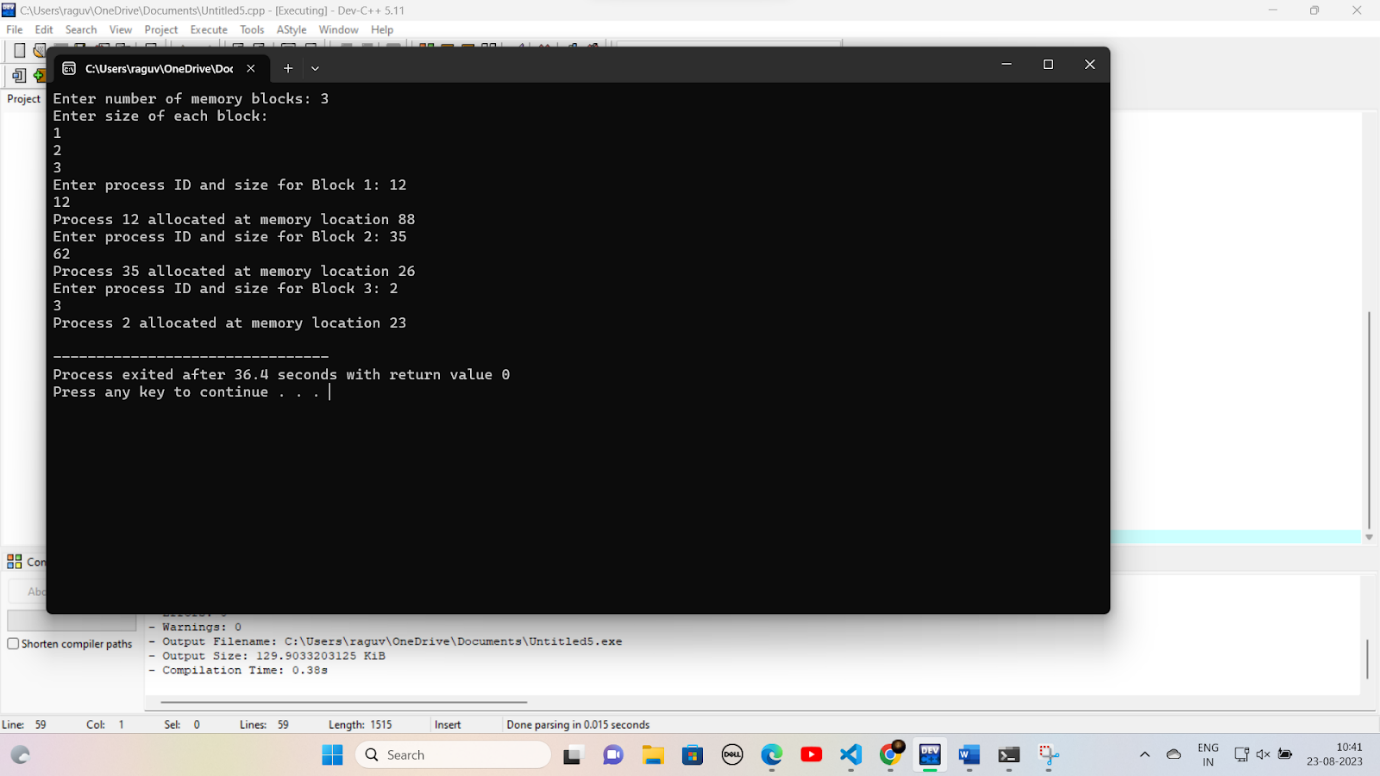
     scanf("%d %d", &process\_id, &size);

     bestFit(process\_id, size);

}

return 0;

}



LRU

#include <stdio.h>

#include <stdbool.h>

#define NUM\_FRAMES 3

#define NUM\_PAGES 10

int frames[NUM\_FRAMES];

int pages[NUM\_PAGES];

int page\_faults = 0;

int access\_order[NUM\_PAGES];

int next\_replace = 0;

void initializeFrames() {

for (int i = 0; i < NUM\_FRAMES; i++) {

     frames[i] = -1; // Initialize frames to -1 (indicating an empty frame)

}

}

bool isPageInFrames(int page) {

for (int i = 0; i < NUM\_FRAMES; i++) {

     if (frames[i] == page) {

         return true;

     }

}

return false;

}

void updateAccessOrder(int page) {

for (int i = 0; i < NUM\_PAGES; i++) {

     if (access\_order[i] == page) {

         for (int j = i; j > 0; j--) {

             access\_order[j] = access\_order[j - 1];

         }

         access\_order[0] = page;

         break;

     }

}

}

void replacePage(int page) {

frames[next\_replace] = page;

updateAccessOrder(page);

next\_replace = (next\_replace + 1) % NUM\_FRAMES;

}

void simulateLRU() {

for (int i = 0; i < NUM\_PAGES; i++) {

     printf("Accessing Page %d: ", pages[i]);

     if (!isPageInFrames(pages[i])) {

         page\_faults++;

         replacePage(pages[i]);

     } else {

         updateAccessOrder(pages[i]);

     }

     for (int j = 0; j < NUM\_FRAMES; j++) {

         if (frames[j] == -1) {

             printf("- ");

         } else {

             printf("%d ", frames[j]);

         }

     }

     printf("\n");

}

}

int main() {

initializeFrames();

printf("Enter the sequence of page references (0-9):\n");

for (int i = 0; i < NUM\_PAGES; i++) {

     scanf("%d", &pages[i]);

     access\_order[i] = -1;

}

simulateLRU();

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

return 0;

}

