Project Title : Library of Memory Allocation and Page Replacement Algorithms

Team Members: Swaranjana Nayak, 19BCE0977

Faculty in charge: Dr. R Manjula, SCOPE, VIT

Project Abstract :

The project introduces a library containing memory allocation schemes - first fit, best fit, worst fit and page replacement algorithms - First In First Out, Least Recently Used and Optimal Page Replacement algorithms. The .h and .c files are separate. The project implements concepts from Data Structures and Algorithms with Advanced C Programming concepts such as pointers, structures, dynamic memory allocation preprocessor directives, file handling, macros.

Project Description:

The project contains .c file, .h file, input text file, input generation program.

acp_library.c has following functions

For memory allocation algorithms:-

1. `memory *initial_memory(memory *m, int n);` - Function to initialize the memory block
2. `process *initial_process(process *p, int m);` - Function to initialize the process block
3. `int find_first_fit(memory *m, int pb, int n, int pno);` - Function to find first fit
4. `void first_fit(memory *m, process *p, int n, int g);` - Function to print first fit
5. `Int find_best_fit(memory *m, int pb, int n, int pno);` - Function to find best fit
6. `void best_fit(memory *m, process *p, int n, int g);` - Function to print best fit
7. `int find_worst_fit(memory *m, int pb, int n, int pno);` - Function to find worst fit
8. `void worst_fit(memory *m, process *p, int n, int g);` - Function to print worst fit

For page replacement algorithms:-

1. `page_frames_fifo *initial_pf(page_frames_fifo *p, int n);` - Function to initialize page frames for FIFO Page Replacement
2. `void FIFO(page_frames_fifo *p, Queue *q, int *ps, int s, int frame);` - Function to carry out FIFO page replacement
3. `page_frames_opt *initial_pf2(page_frames_opt *p, int n);` - Function to initialize page frames for Optimal Page Replacement
4. `void future_ref_opt(int *cur, int *fut, int *ps, int s, int frame);`
5. `int find_rep_opt(int *fut, int frame);`
6. `void OPT(page_frames_opt *p, int *ps, int s, int frame);` - Function to carry out Optimal page replacement
7. `page_frames_lru *initial_pf3(page_frames_lru *p, int n);` - Function to initialize page frames for LRU Page Replacement
8. `void min_used_index(int *cur, int *past, int *ps, int s, int frame);`
9. `int find_rep_lru(int *past, int frame);`

10. `void LRU(page_frames_lru *p, int *ps, int s, int frame);` - Function to carry out LRU page replacement

Standard queue functions:-

```
1. Queue *initial_q(Queue *cq, int q);
2. int Is_empty(struct qu *a);
3. int Is_full(struct qu *a, int max);
4. void Enqueue(struct qu *a, int val, int frame);
5. int Dequeue(struct qu *a);
6. void Display(struct qu *a, int frame);
```

acp_library.h prototypes the functions as external functions under __ACP_LIBRARY__ macro. It defines the following structures

1. Structure for Memory blocks
2. Structure for process blocks
3. Structure for page frames of FIFO Page Replacement
4. Structure to store information of queue
5. Structure for page frames of Optimal Page Replacement
6. Structure for page frames of FIFO Page Replacement

Input text files - 2 types of inputs, one for memory allocation and other as page stream for page replacement algorithms

Algorithm test program - Programs to test the functioning of the library

acp_library.c

Code:

```
/*************************************************************************************************
 *                        LIBRARY OF MEMORY ALLOCATION AND PAGE REPLACEMENT ALGORITHMS                *
 *************************************************************************************************/


# include <stdio.h>
# include <stdlib.h>
# include "acp_library.h"
# define MAX 10
# define M 5

// // Structure of memory
// typedef struct mem
// {
//     int *mem_blocks;
//     int *occupied;
//     int *wastage;
// }memory;

// // Structure of process
// typedef struct proc
// {
//     int *proc_blocks;
//     int *allocated;
// }process;

// // Main memory
// typedef struct pf
// {
//     int *frames;
//     int capacity;
//     int flag;
//     int page_fault;
// }page_frames_fifo;

// // Stucture of Queue
// typedef struct qu
// {
//     int count;
//     int front;
//     int rear;
//     int *Q;
// }Queue;

// // Main memory
// typedef struct pf2
// {
//     int *frames;
//     int *future;
//     int *current;
//     int capacity;
//     int flag;
//     int page_fault;
// }page_frames_opt;

// // Main memory
// typedef struct pf3
// {
//     int *frames;
//     int *past;
//     int *current;
```

```
//     int capacity;
//     int flag;
//     int page_fault;
// }page_frames_lru;


// Function to initialize memory block information
memory *initial_memory(memory *m, int n)
{
    int i;

    m = (memory *)malloc(sizeof(memory));

    m->mem_blocks = (int *)malloc(n * sizeof(int));
    m->occupied   = (int *)malloc(n * sizeof(int));
    m->wastage    = (int *)malloc(n * sizeof(int));

    for(i = 0; i < n; i++)
    {
        *((m->mem_blocks) + i)  = 0;
        *((m->occupied) + i)    = -1;
        *((m->wastage) + i)     = -1;
    }

    return m;
}

// Function to initialize process block information
process *initial_process(process *p, int m)
{
    int i;

    p = (process *)malloc(sizeof(process));

    p->proc_blocks = (int *)malloc(m * sizeof(int));
    p->allocated   = (int *)malloc(m * sizeof(int));

    for(i = 0; i < m; i++)
    {
        *((p->proc_blocks) + i)  = 0;
        *((p->allocated) + i)    = -1;
    }

    return p;
}

/*****************************************************************************
 *                                FIRST FIT ALGORITHM                        *
 *****************************************************************************/


// Function to find first fit for a memory block
int find_first_fit(memory *m, int pb, int n, int pno)
{
    int min = 0, i, flag = -1;

    for(i = 0; i < n; i++)
    {
        if(m->mem_blocks[i] >= pb && m->occupied[i] == -1)
        {
            flag = i;
            min  = m->mem_blocks[i] - pb;
            break;
        }
    }

    if (flag != -1)
    {
        m->occupied[flag] = pno;
```

```
        m->wastage[flag]  = min;
    }

    return (flag + 1);
}

// Function to execute best fit algorithm for all processes
void first_fit(memory *m, process *p, int n, int g)
{
    int i, internal_frag = 0, external_frag = 0;

    for(i = 0; i < g; i++)
    {
        p->allocated[i] = find_first_fit(m, p->proc_blocks[i], n, i+1);
    }

    // Printing the final configuration for memory
    printf("\n\n------------------------------ MEMORY ------------------------------\n\n");
    printf("Block no.\t Block size\t Process no.\t Internal Fragmentation\n");
    printf("--------------------------------------------------------------------\n");

    for(i = 0; i < n; i++)
    {
        if(m->occupied[i] != -1)
        {
            printf(" \t%d\t %d kb\t\t Process %d\t %d kb\n", i + 1, m->mem_blocks[i], m->occupied[i], m->
wastage[i]);
        }
        else
        {
            printf(" \t%d\t %d kb\t\t No Process\t Free\n", i + 1, m->mem_blocks[i]);
        }
    }

    // Printing the final configuration for process
    printf("\n\n----------------  PROCESS ----------------\n\n");
    printf("Process no.\t Process size\t Block no.\n");
    printf("-----------------------------------------\n");

    for(i = 0; i < g; i++)
    {
        if(p->allocated[i] != 0)
        {
            printf("  \t%d\t  %d kb\t  Block %d\n", i + 1, p->proc_blocks[i], p->allocated[i]);
        }
        else
        {
            printf("  \t%d\t  %d kb\t  Not Allocated\n", i + 1, p->proc_blocks[i]);
        }

    }

    // Calculating total Internal and External Fragmentation

    for(i = 0; i < n; i++)
    {
        if(m->wastage[i] != -1)
        {
            internal_frag += m->wastage[i];
        }
        else
        {
            external_frag += m->mem_blocks[i];
        }
    }

    printf("\nTotal Internal Fragmentation: %d KB", internal_frag);
    printf("\nTotal External Fragmentation: %d KB", external_frag);
    printf("\n\n");
```

```
}


/*********************************************************************************
 *                                BEST FIT ALGORITHM                             *
 *********************************************************************************/


// Function to find best fit for a memory block
int find_best_fit(memory *m, int pb, int n, int pno)
{
    int min = 0, i, flag = -1, j;

    for(i = 0; i < n; i++)
    {
        if(m->mem_blocks[i] >= pb && m->occupied[i] == -1)
        {
            flag = i;
            min  = m->mem_blocks[i] - pb;
            break;
        }
    }

    for(j = i + 1; j < n; j++)
    {
        if(m->mem_blocks[j] >= pb && m->occupied[j] == -1)
        {
            if(min > m->mem_blocks[j] - pb)
            {
               flag = j;
               min  = m->mem_blocks[j] - pb;
            }
        }
    }

    if (flag != -1)
    {
        m->occupied[flag] = pno;
        m->wastage[flag]  = min;
    }

    return (flag + 1);
}

// Function to execute best fit algorithm for all processes
void best_fit(memory *m, process *p, int n, int g)
{
    int i, internal_frag = 0, external_frag = 0;

    for(i = 0; i < g; i++)
    {
        p->allocated[i] = find_best_fit(m, p->proc_blocks[i], n, i+1);
    }

    // Printing the final configuration for memory
    printf("\n\n------------------------------ MEMORY -----------------------------\n\n");
    printf("Block no.\t Block size\t Process no.\t Internal Fragmentation\n");
    printf("-------------------------------------------------------------------\n");

    for(i = 0; i < n; i++)
    {
        if(m->occupied[i] != -1)
        {
            printf(" \t%d\t %d kb\t\t Process %d\t %d kb\n", i + 1, m->mem_blocks[i], m->occupied[i], m->
wastage[i]);
        }
        else
        {
            printf(" \t%d\t %d kb\t\t No Process\t Free\n", i + 1, m->mem_blocks[i]);
        }
```

```
    }

    // Printing the final configuration for process
    printf("\n\n----------------  PROCESS ----------------\n\n");
    printf("Process no.\t Process size\t Block no.\n");
    printf("-----------------------------------------\n");

    for(i = 0; i < g; i++)
    {
        if(p->allocated[i] != -1)
        {
            printf("  \t%d\t  %d kb\t  Block %d\n", i + 1, p->proc_blocks[i], p->allocated[i]);
        }
        else
        {
            printf("  \t%d\t  %d kb\t  Not Allocated\n", i + 1, p->proc_blocks[i]);
        }

    }

    // Calculating total Internal and External Fragmentation

    for(i = 0; i < n; i++)
    {
        if(m->wastage[i] != -1)
        {
            internal_frag += m->wastage[i];
        }
        else
        {
            external_frag += m->mem_blocks[i];
        }
    }

    printf("\nTotal Internal Fragmentation: %d KB", internal_frag);
    printf("\nTotal External Fragmentation: %d KB", external_frag);
    printf("\n\n");

}


/*****************************************************************************
 *                                WORST FIT ALGORITHM                        *
 *****************************************************************************/

// Function to find worst fit for a memory block
int find_worst_fit(memory *m, int pb, int n, int pno)
{
    int max = 0, i, flag = -1, j;

    for(i = 0; i < n; i++)
    {
        if(m->mem_blocks[i] >= pb && m->occupied[i] == -1)
        {
            flag = i;
            max  = m->mem_blocks[i] - pb;
            break;
        }
    }

    for(j = i + 1; j < n; j++)
    {
        if(m->mem_blocks[j] >= pb && m->occupied[j] == -1)
        {
            if(max < m->mem_blocks[j] - pb)
            {
               flag = j;
               max  = m->mem_blocks[j] - pb;
            }
        }
```

```
    }

    if (flag != -1)
    {
        m->occupied[flag] = pno;
        m->wastage[flag]  = max;
    }

    return (flag + 1);
}

// Function to execute worst fit algorithm for all processes
void worst_fit(memory *m, process *p, int n, int g)
{
    int i, internal_frag = 0, external_frag = 0;

    for(i = 0; i < g; i++)
    {
        p->allocated[i] = find_worst_fit(m, p->proc_blocks[i], n, i+1);
    }

    // Printing the final configuration for memory
    printf("\n\n------------------------------ MEMORY ------------------------------\n\n");
    printf("Block no.\t Block size\t Process no.\t Internal Fragmentation\n");
    printf("--------------------------------------------------------------------\n");

    for(i = 0; i < n; i++)
    {
        if(m->occupied[i] != -1)
        {
            printf(" \t%d\t %d kb\t\t Process %d\t %d kb\n", i + 1, m->mem_blocks[i], m->occupied[i], m->
wastage[i]);
        }
        else
        {
            printf(" \t%d\t %d kb\t\t No Process\t Free\n", i + 1, m->mem_blocks[i]);
        }
    }

    // Printing the final configuration for process
    printf("\n\n----------------  PROCESS ----------------\n\n");
    printf("Process no.\t Process size\t Block no.\n");
    printf("-----------------------------------------\n");

    for(i = 0; i < g; i++)
    {
        if(p->allocated[i] != 0)
        {
            printf("  \t%d\t  %d kb\t  Block %d\n", i + 1, p->proc_blocks[i], p->allocated[i]);
        }
        else
        {
            printf("  \t%d\t  %d kb\t  Not Allocated\n", i + 1, p->proc_blocks[i]);
        }

    }

    // Calculating total Internal and External Fragmentation

    for(i = 0; i < n; i++)
    {
        if(m->wastage[i] != -1)
        {
            internal_frag += m->wastage[i];
        }
        else
        {
            external_frag += m->mem_blocks[i];
        }
    }
```

```
    printf("\nTotal Internal Fragmentation: %d KB", internal_frag);
    printf("\nTotal External Fragmentation: %d KB", external_frag);
    printf("\n\n");

}


/*********************************************************************************
 *                          FIFO PAGE REPLACEMENT ALGORITHM                      *
 *********************************************************************************/


// Function to initialize page frames data structure
// n is the number of page frame in the main memory
page_frames_fifo *initial_pf(page_frames_fifo *p, int n)
{
    p = (page_frames_fifo *)malloc(sizeof(page_frames_fifo));

    p->frames     = (int *)malloc(n * sizeof(int));
    p->capacity   = n;
    p->flag       = 0;
    p->page_fault = 0;

    return p;
}

// Function to perform FIFO Page Replacement
// p = Page Frame data structure
// q is queue for FIFO principle, keeps track of pages in FIFO order
// ps is the Page Stream to be checked
// s is the length of page stream
// frame is the number of page frames in main memory
void FIFO(page_frames_fifo *p, Queue *q, int *ps, int s, int frame)
{
    int i, j, repl;

    printf("\nPAGE FRAMES \t PAGE FAULT\n");
    printf("-----------------------------\n");

    for(i = 0; i < s; i++)
    {
        if(p->capacity != 0)
        {
            p->flag = 0;
            //(p->page_fault)++;

            Enqueue(q, *(ps + i), frame);
            p->frames[frame - p->capacity] = ps[i];

            (p->capacity)--;

            for(j = 0; j < (frame - p->capacity); j++)
            {
                printf("| %d ", p->frames[j]);
            }
            printf("|");
            for(j = 0; j <= p->capacity; j++)
            {
                printf("    ");
            }
            printf("\t%d\n", p->flag);

        }
        else
        {
            p->flag = 0;

            // Searching for the page
            for(j = 0; j < frame; j++)
```

```
            {
                if(p->frames[j] == *(ps + i))
                {
                    p->flag = 1;
                    break;
                }
            }

            // If there's a page fault, then we'll replace
            if(p->flag == 0)
            {
                repl = Dequeue(q);
                // Display(q, frame);
                // printf("repl = %d\n", repl);
                Enqueue(q, ps[i], frame);
                // Display(q, frame);
                for(j = 0; j < frame; j++)
                {
                    if(p->frames[j] == repl)
                    {
                        p->frames[j] = *(ps + i);
                    }
                }
            }

            for(j = 0; j < frame; j++)
            {
                printf("| %d ", p->frames[j]);
            }
            printf("| \t%d \n", p->flag);
        }

        if(p->flag == 0)
            (p->page_fault)++;

        //Display(q);

    }

    printf("\nTotal no. of page faults : %d\n", p->page_fault);
    printf("Total no. of hits: %d\n", s - p->page_fault);
    printf("Hit ratio : %0.2f\n\n", ((float)(s - p->page_fault))/((float)s));
}

// Function to initialize the queue data structure
Queue *initial_q(Queue *cq, int q)
{
    cq = (Queue *)malloc(sizeof(Queue));
    cq->count = 0;
    cq->front = -1;
    cq->rear = -1;
    cq->Q    = (int *)malloc(q * sizeof(int));

    return cq;
}

// Function to check if the queue is empty
int Is_empty(struct qu *a)
{
    if(a->count == 0)
        return 1;

    return 0;
}

// Function to check if the queue is full
int Is_full(struct qu *a, int max)
{
    if(a->count == max)
        return 1;
```

```c
    return 0;
}

// Function to enqueue an element
void Enqueue(struct qu *a, int val, int frame)
{
    if(Is_full(a, frame))
    {
        printf("\nQueue Overflow!\n\n");
        return;
    }
    else
    {
        a->rear = (a->rear + 1) % frame;
        a->Q[a->rear] = val;

        if(a->front == -1)
        {
            a->front = a->front + 1;
        }
        a->count = a->count + 1;
    }

}

// Function to dequeue an element
int Dequeue(struct qu *a)
{
    int ret;

    if(Is_empty(a))
    {
        printf("\nQueue Underflow! No elements to dequeue.\n\n");
        return -1;
    }
    else
    {
        ret = a->Q[a->front];

        a->Q[a->front] = 28;
        if(a->front == a->rear)
        {
            a->front = -1;
            a->rear = -1;
        }
        else
        {
            a->front = a->front + 1;
        }
        a->count = a->count - 1;
    }

    return ret;
}

// Function to display the queue
void Display(struct qu *a, int frame)
{
    int i, j;
    if(Is_empty(a))
    {
        printf("\nQueue Underflow! No elements to display.\n\n");
        return;
    }
    else
    {
        printf("\n------ QUEUE ------\n\n");
        printf("|");
        for(i = a->front, j = 0; j < a->count ; i = (i+1) % frame, j++)
        {
            printf(" %d |", a->Q[i]);
```

```
        }
        printf("\n\n");
    }
}

/*************************************************************************
 *                      OPTIMAL PAGE REPLACEMENT ALGORITHM                *
 *************************************************************************/


// Function to initialize page frames data structure
// n is the number of page frame in the main memory
page_frames_opt *initial_pf2(page_frames_opt *p, int n)
{
    int i;

    p = (page_frames_opt *)malloc(sizeof(page_frames_opt));

    p->frames     = (int *)malloc(n * sizeof(int));
    p->future     = (int *)malloc(n * sizeof(int));
    p->current    = (int *)malloc(n * sizeof(int));
    p->capacity   = n;
    p->flag       = 0;
    p->page_fault = 0;

    for(i = 0; i < n; i++)
    {
        p->future[i]  = -1;
        p->current[i] = -1;
    }

    return p;
}

// Find future reference
// Fut is the array that will store future index; ps is the page stream
// cur is the array that will store current indices
// s is length of the page stream, frame is the no of frames in main memory
void future_ref_opt(int *cur, int *fut, int *ps, int s, int frame)
{
    int i, j, flag;

    for(i = 0; i < frame; i++)
    {
        flag = 0;
        for(j = cur[i] + 1; j < s; j++)
        {
            if(ps[j] == ps[cur[i]])
            {
                fut[i] = j;
                flag = 1;
                break;
            }
        }
        if(flag == 0)
        {
            fut[i] = -1;
        }
    }
}

// Function to find index of page to replace
int find_rep_opt(int *fut, int frame)
{
    int ind, max;

    max = 0;

    for(ind = 0; ind < frame; ind++)
    {
```

```
        if(fut[ind] == -1)
            return ind;
        else if(fut[max] < fut[ind])
        {
            max = ind;
        }
    }

    return max;
}

// Function to simulate optimal page replacement algorithm
// p = Page Frame data structure
// ps is the Page Stream to be checked
// s is the length of page stream
// frame is the number of page frames in main memory
void OPT(page_frames_opt *p, int *ps, int s, int frame)
{
    int i, j, repl;

    printf("\nPAGE FRAMES \t PAGE FAULT\n");
    printf("------------------------------\n");

    for(i = 0; i < s; i++)
    {
        if(p->capacity != 0)
        {
            p->flag = 0;
            //(p->page_fault)++;

            p->frames[frame - p->capacity] = ps[i];
            p->current[frame - p->capacity] = i;

            (p->capacity)--;

            for(j = 0; j < (frame - p->capacity); j++)
            {
                printf("| %d ", p->frames[j]);
            }
            printf("|");
            for(j = 0; j <= p->capacity; j++)
            {
                printf("    ");
            }
            printf("\t%d\n", p->flag);

        }
        else
        {
            p->flag = 0;

            // Searching for the page
            for(j = 0; j < frame; j++)
            {
                if(p->frames[j] == *(ps + i))
                {
                    p->flag = 1;
                    p->current[j] = i;
                    break;
                }
            }

            // If there's a page fault, then we'll replace
            if(p->flag == 0)
            {
                future_ref_opt(p->current, p->future, ps, s, frame);
                repl = find_rep_opt(p->future, frame);

                p->frames[repl]  = *(ps + i);
                p->current[repl] = i;
            }
```

```
            for(j = 0; j < frame; j++)
            {
                printf("| %d ", p->frames[j]);
            }
            printf("| \t%d \n", p->flag);
        }

        if(p->flag == 0)
            (p->page_fault)++;

    }

    printf("\nTotal no. of page faults : %d\n", p->page_fault);
    printf("Total no. of hits: %d\n", s - p->page_fault);
    printf("Hit ratio : %0.2f\n\n", ((float)(s - p->page_fault))/((float)s));
}


/*****************************************************************************
 *                          LRU PAGE REPLACEMENT ALGORITHM                   *
 *****************************************************************************/


// Function to initialize page frames data structure
// n is the number of page frame in the main memory
page_frames_lru *initial_pf3(page_frames_lru *p, int n)
{
    int i;

    p = (page_frames_lru *)malloc(sizeof(page_frames_lru));

    p->frames     = (int *)malloc(n * sizeof(int));
    p->past       = (int *)malloc(n * sizeof(int));
    p->current    = (int *)malloc(n * sizeof(int));
    p->capacity   = n;
    p->flag       = 0;
    p->page_fault = 0;

    for(i = 0; i < n; i++)
    {
        p->past[i]  = -1;
        p->current[i] = -1;
    }

    return p;
}

// Find past reference
// past is the array that will store past index; ps is the page stream
// cur is the array that will store current indices
// s is length of the page stream, frame is the no of frames in main memory
void min_used_index(int *cur, int *past, int *ps, int s, int frame)
{
    int i, j, flag;

    for(i = 0; i < frame; i++)
    {
        flag = 0;
        for(j = cur[i]; j >= 0; j--)
        {
            if(ps[j] == ps[cur[i]])
            {
                past[i] = j;
                flag = 1;
                break;
            }
        }
        if(flag == 0)
        {
```

```
            past[i] = -1;
        }
    }
}

// Function to find index of page to replace
int find_rep_lru(int *past, int frame)
{
    int ind, min;

    min = 0;

    for(ind = 0; ind < frame; ind++)
    {
        if(past[ind] == -1)
            return ind;
        else if(past[min] >= past[ind])
        {
            min = ind;
        }
    }

    return min;
}

// Function to simulate LRU page replacement algorithm
// p = Page Frame data structure
// ps is the Page Stream to be checked
// s is the length of page stream
// frame is the number of page frames in main memory
void LRU(page_frames_lru *p, int *ps, int s, int frame)
{
    int i, j, repl;

    printf("\nPAGE FRAMES \t PAGE FAULT\n");
    printf("-----------------------------\n");

    for(i = 0; i < s; i++)
    {
        if(p->capacity != 0)
        {
            p->flag = 0;
            //(p->page_fault)++;

            p->frames[frame - p->capacity] = ps[i];
            p->current[frame - p->capacity] = i;

            (p->capacity)--;

            for(j = 0; j < (frame - p->capacity); j++)
            {
                printf("| %d ", p->frames[j]);
            }
            printf("|");
            for(j = 0; j <= p->capacity; j++)
            {
                printf("    ");
            }
            printf("\t%d\n", p->flag);

        }
        else
        {
            p->flag = 0;

            // Searching for the page
            for(j = 0; j < frame; j++)
            {
                if(p->frames[j] == *(ps + i))
                {
```

```
                    p->flag = 1;
                    p->current[j] = i;
                    break;
                }
            }

            // If there's a page fault, then we'll replace
            if(p->flag == 0)
            {
                min_used_index(p->current, p->past, ps, s, frame);

                // for(int e = 0; e < frame; e++)
                //     printf("%d   ", p->past[e]);

                // printf("\n");

                repl = find_rep_lru(p->past, frame);

                p->frames[repl]  = *(ps + i);
                p->current[repl] = i;
            }

            for(j = 0; j < frame; j++)
            {
                printf("| %d ", p->frames[j]);
            }
            printf("| \t%d \n", p->flag);
        }

        if(p->flag == 0)
            (p->page_fault)++;

    }

    printf("\nTotal no. of page faults : %d\n", p->page_fault);
    printf("Total no. of hits: %d\n", s - p->page_fault);
    printf("Hit ratio : %0.2f\n\n", ((float)(s - p->page_fault))/((float)s));
}
```

acp_library.h

Code:

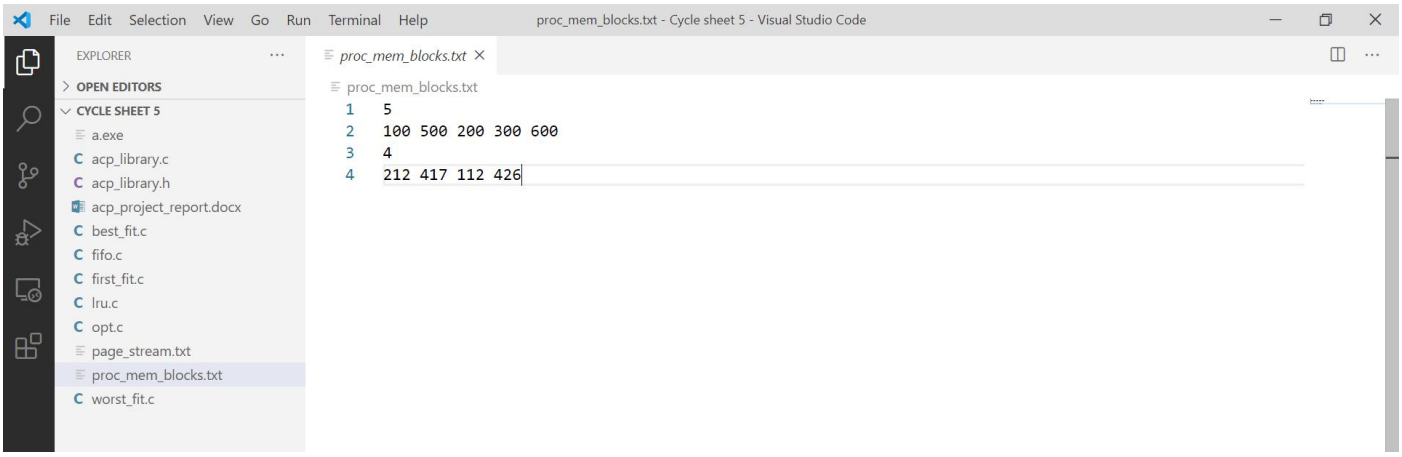
```
# ifndef __ACP_LIBRARY__
# define __ACP_LIBRARY__


// Structure of memory
typedef struct mem
{
    int *mem_blocks;
    int *occupied;
    int *wastage;
}memory;


// Structure of process
typedef struct proc
{
    int *proc_blocks;
    int *allocated;
}process;


// Main memory
typedef struct pf
{
    int *frames;
    int capacity;
    int flag;
    int page_fault;

}page_frames_fifo;


// Stucture of Queue
typedef struct qu
{
    int count;
    int front;
    int rear;
    int *Q;
}Queue;


// Main memory
typedef struct pf2
{
    int *frames;
    int *future;
    int *current;
    int capacity;
    int flag;
    int page_fault;
}page_frames_opt;
```

```
// Main memory
typedef struct pf3
{

int *frames;
    int *past;
    int *current;
    int capacity;
    int flag;
    int page_fault;

}page_frames_lru;


extern memory *initial_memory(memory *m, int n);
extern process *initial_process(process *p, int m);
extern int find_first_fit(memory *m, int pb, int n, int pno);
extern void first_fit(memory *m, process *p, int n, int g);
extern int find_best_fit(memory *m, int pb, int n, int pno);
extern void best_fit(memory *m, process *p, int n, int g);
extern int find_worst_fit(memory *m, int pb, int n, int pno);
extern void worst_fit(memory *m, process *p, int n, int g);



extern page_frames_fifo *initial_pf(page_frames_fifo *p, int n);
extern void FIFO(page_frames_fifo *p, Queue *q, int *ps, int s, int frame);

extern Queue *initial_q(Queue *cq, int q);
extern int Is_empty(struct qu *a);
extern int Is_full(struct qu *a, int max);
extern void Enqueue(struct qu *a, int val, int frame);
extern int Dequeue(struct qu *a);
void Display(struct qu *a, int frame);


extern page_frames_opt *initial_pf2(page_frames_opt *p, int n);
extern void future_ref_opt(int *cur, int *fut, int *ps, int s, int frame);
extern int find_rep_opt(int *fut, int frame);
extern void OPT(page_frames_opt *p, int *ps, int s, int frame);


extern page_frames_lru *initial_pf3(page_frames_lru *p, int n);
extern void min_used_index(int *cur, int *past, int *ps, int s, int frame);
extern int find_rep_lru(int *past, int frame);
extern void LRU(page_frames_lru *p, int *ps, int s, int frame);


# endif
```

Input text files

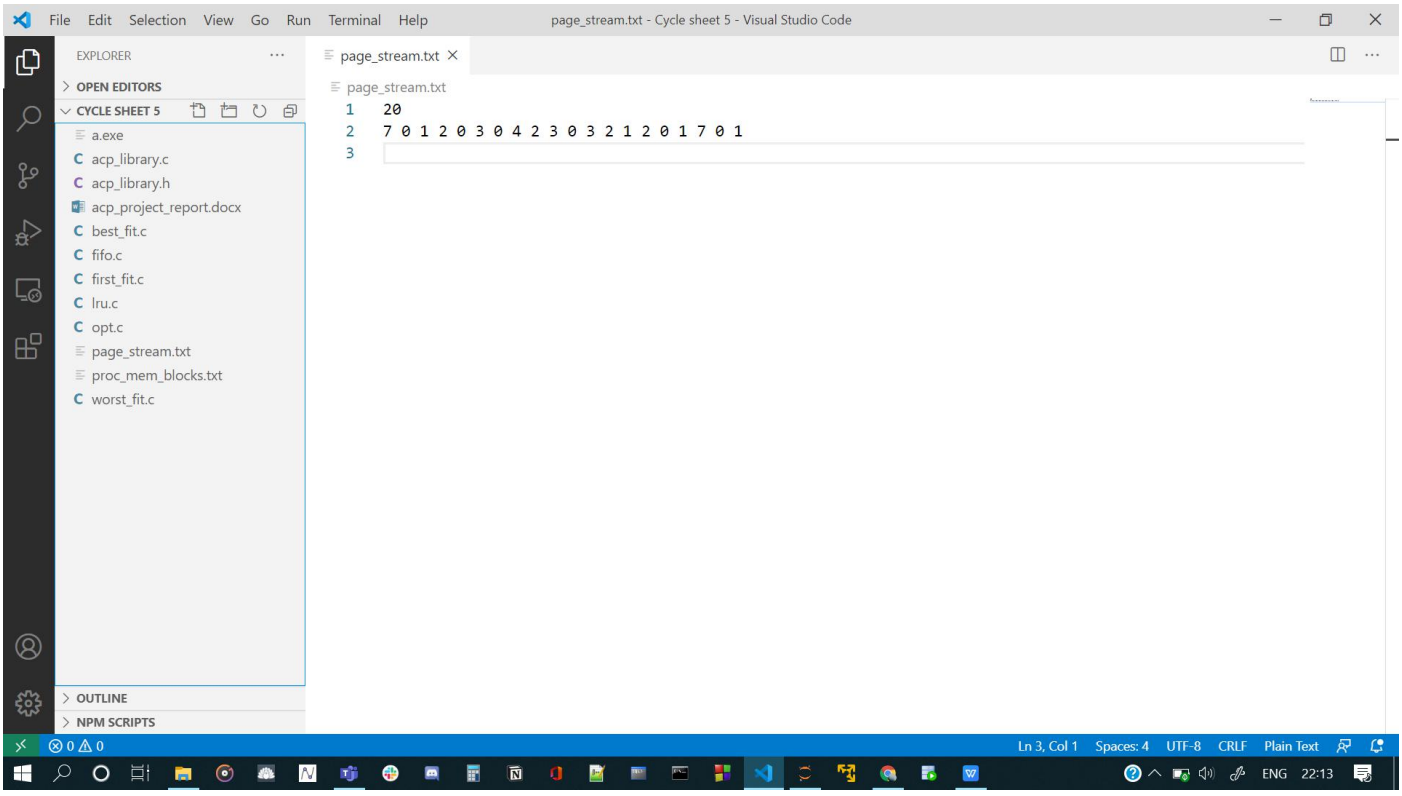
1. Input for memory allocation algorithms

    Line 1 -> Number of memory blocks
    Line 2 -> Size of each memory block separated by a space
    Line 3 -> Number of process blocks
    Line 4 -> Size of each process block separated by a space



2. Input for page replacement algorithms

    Line 1 -> Number of pages in the page stream
    Line 2 -> Page numbers in the page stream separated by a space



RESULT ANANLYSIS AND DISCUSSION

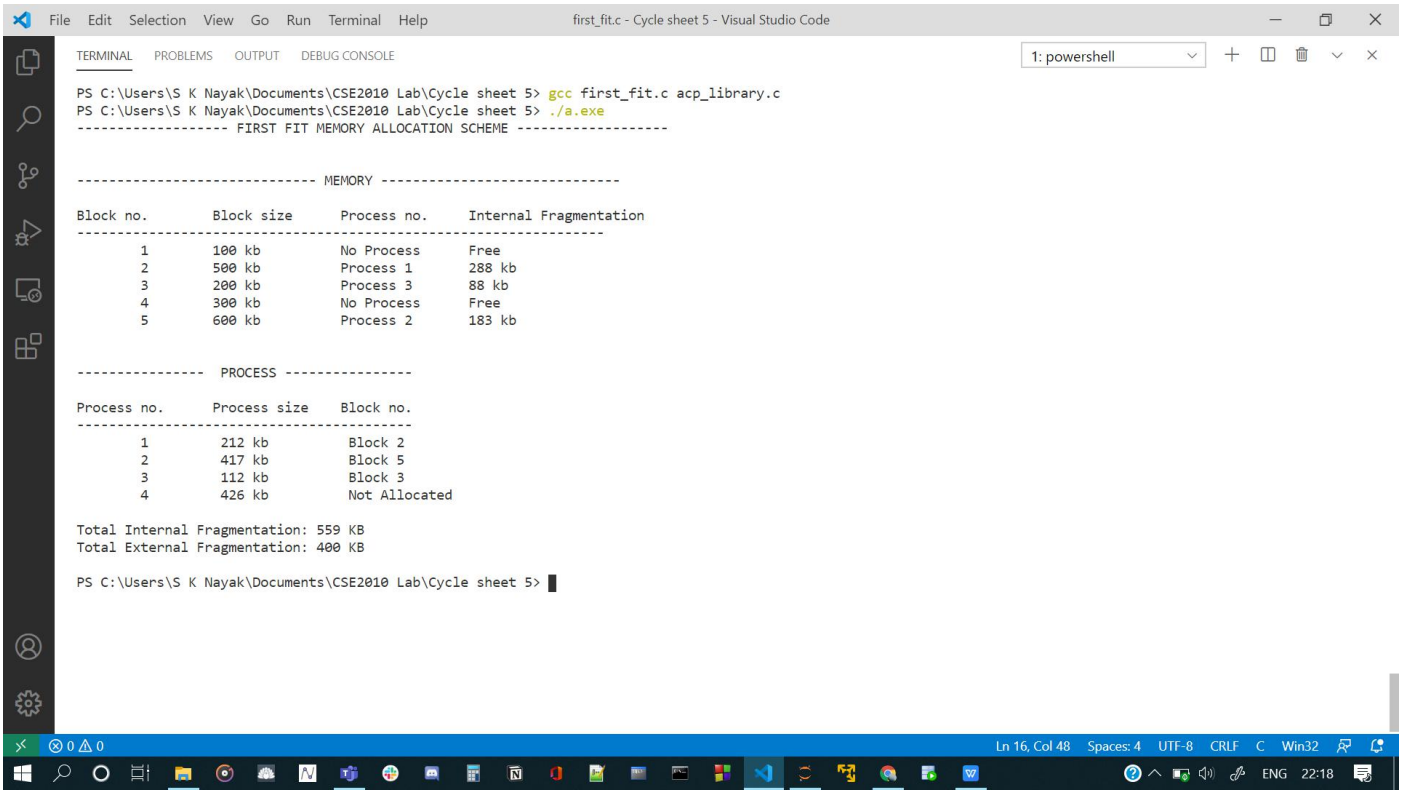
(WITH TEST PROGRAM CODES AND OUTPUTS)

1. First Fit memory allocation algorithm (Test program)

Code:

```c
// C program to simulate first fit algorithm


# include <stdio.h>
# include <stdlib.h>
# include "acp_library.h"
# define MAX 10


// MAIN FUNCTION
int main()
{
    process *procb;
    memory  *memb;
    int i, n, m, status;

    printf("------------------- FIRST FIT MEMORY ALLOCATION SCHEME -------------------\n");

    FILE *fptr;
    fptr = fopen("proc_mem_blocks.txt", "r");

    fscanf(fptr, "%d", &n);
    memb = initial_memory(memb, n);

    for(i = 0; i < n; i++)
    {
        fscanf(fptr, "%d", (memb->mem_blocks) + i);
    }

    fscanf(fptr, "%d", &m);
    procb = initial_process(procb, m);

    for(i = 0; i < m; i++)
    {
        fscanf(fptr, "%d", (procb->proc_blocks) + i);
    }

    first_fit(memb, procb, n, m);

    return 0;

}
```

Output:



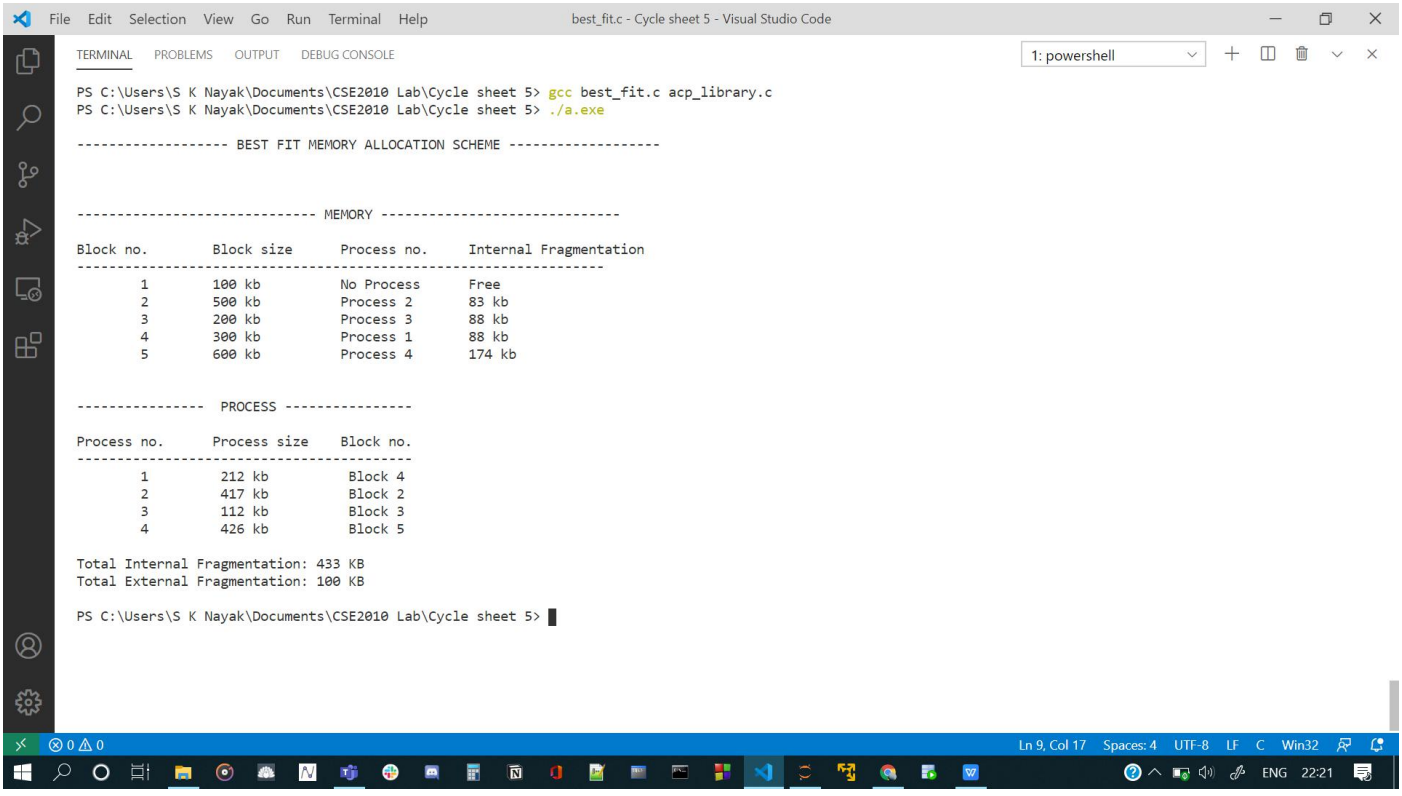
2. Best Fit memory allocation algorithm (Test program)

Code:

```c
// C program to simulate best fit algorithm

# include <stdio.h>
# include <stdlib.h>
# include "acp_library.h"
# define MAX 10



// MAIN FUNCTION
int main()
{

    process *procb;
    memory  *memb;
    int i, n, m, status;

    printf("\n------------------ BEST FIT MEMORY ALLOCATION SCHEME ----------------\n\n");

    FILE *fptr;
    fptr = fopen("proc_mem_blocks.txt", "r");

    fscanf(fptr, "%d", &n);
    memb = initial_memory(memb, n);
```

```
    for(i = 0; i < n; i++)
    {
        fscanf(fptr, "%d", (memb->mem_blocks) + i);
    }

    fscanf(fptr, "%d", &m);
    procb = initial_process(procb, m);


    for(i = 0; i < m; i++)
    {
        fscanf(fptr, "%d", (procb->proc_blocks) + i);
    }

    best_fit(memb, procb, n, m);

    return 0;

}
```

Output:

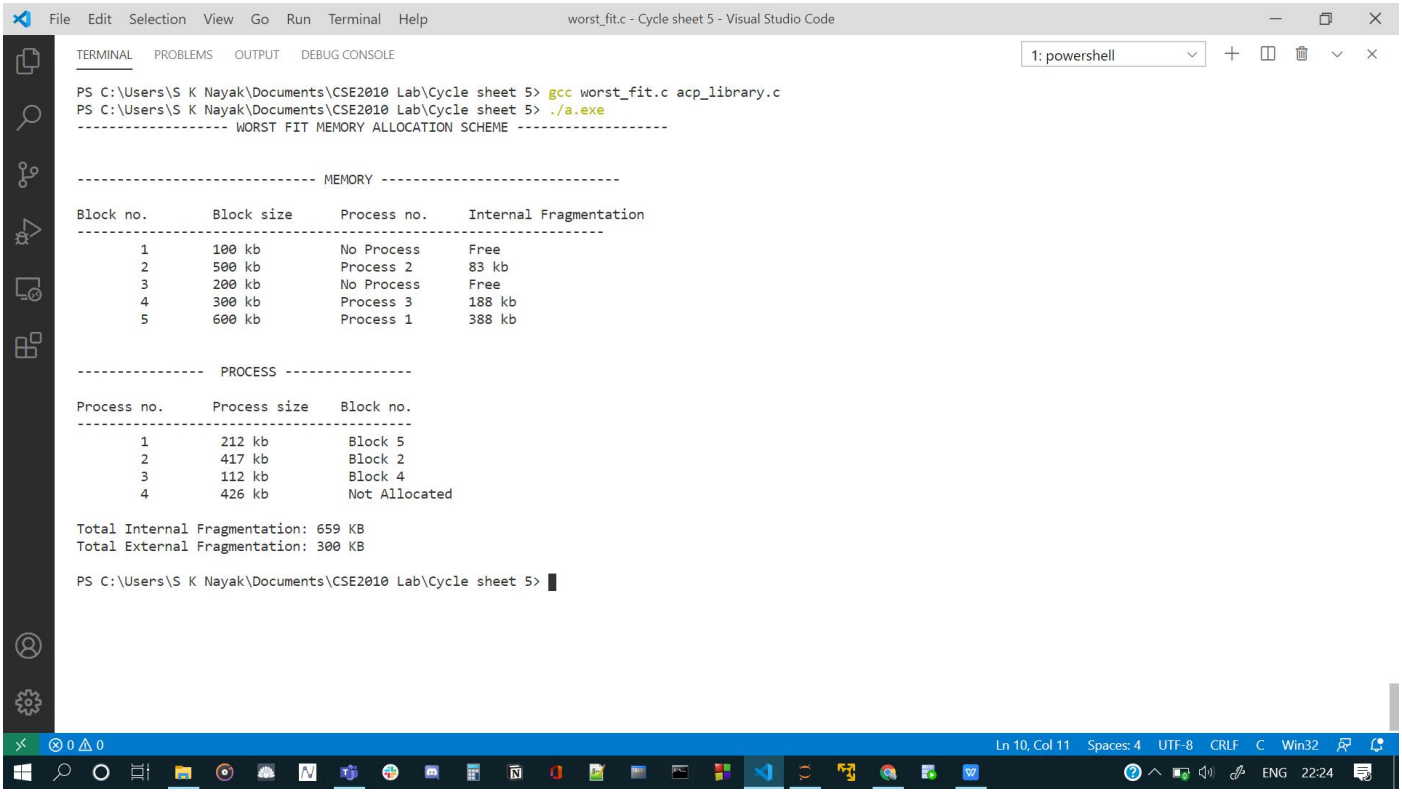


3. Worst Fit memory allocation algorithm (Test program)

Code:

```c
// C program to simulate best fit algorithm


# include <stdio.h>
# include <stdlib.h>
# include "acp_library.h"
# define MAX 10


// MAIN FUNCTION
int main()
{

    process *procb;
    memory  *memb;
    int i, n, m, status;

    printf("------------------- WORST FIT MEMORY ALLOCATION SCHEME -------------------\
n");


    FILE *fptr;
    fptr = fopen("proc_mem_blocks.txt", "r");


    fscanf(fptr, "%d", &n);
    memb = initial_memory(memb, n);


    for(i = 0; i < n; i++)
    {
        fscanf(fptr, "%d", (memb->mem_blocks) + i);
    }


   fscanf(fptr, "%d", &m);
    procb = initial_process(procb, m);

    for(i = 0; i < m; i++)
    {
        fscanf(fptr, "%d", (procb->proc_blocks) + i);
    }

    worst_fit(memb, procb, n, m);

    return 0;

}
```

Output:



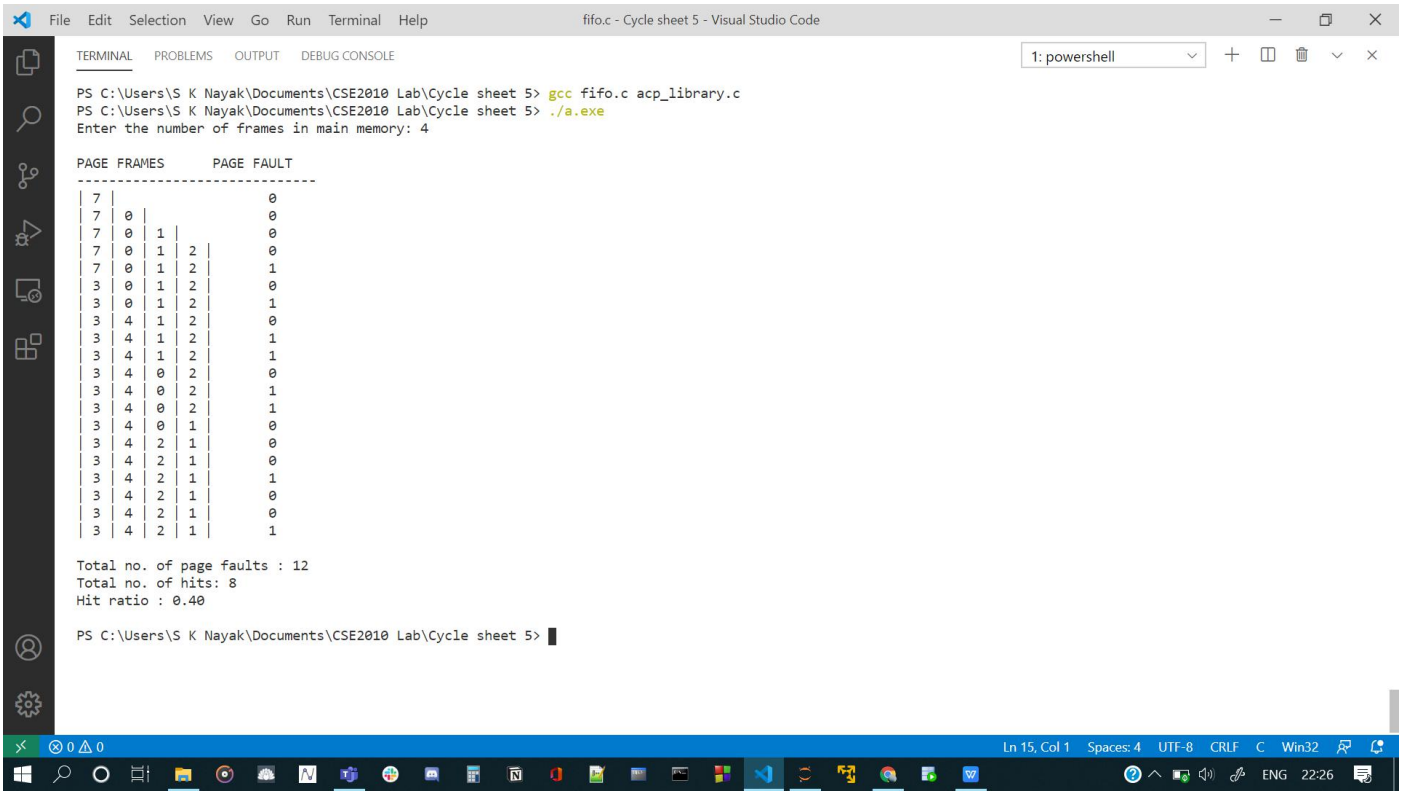
4. FIFO Page Replacement algorithm (Test program)

Code:

```c
// C program to simulate FIFO page replacement scheme

# include <stdio.h>
# include <stdlib.h>
# include "acp_library.h"
# define MAX 5

// Main Driver Function
int main()
{
    int status, mm_frame, *page_stream, s, i;

    page_frames_fifo *fifo_pf = NULL;
    Queue *queue = NULL;

    printf("Enter the number of frames in main memory: ");
    status = scanf("%d", &mm_frame);

    // Input validation
    while (status == 0 || mm_frame <= 0 || mm_frame > MAX)
    {
        printf("Invalid Input!\n");
        printf("Enter the number of frames in main memory: ");
        status = scanf("%d", &mm_frame);
        fflush(stdin);
    }

    FILE *fptr;
```

```c
    fptr = fopen("page_stream.txt", "r");

    fscanf(fptr, "%d", &s);

    page_stream = (int *)malloc(s * sizeof(int));

    for(i = 0; i < s; i++)
    {
        fscanf(fptr, "%d", page_stream + i);
    }

    fclose(fptr);

    // Initializing the data structures
    fifo_pf = initial_pf(fifo_pf, mm_frame);
    queue   = initial_q(queue, mm_frame);

    // FIFO page replacement simulation
    FIFO(fifo_pf, queue, page_stream, s, mm_frame);

    return 0;
}
```

Output:

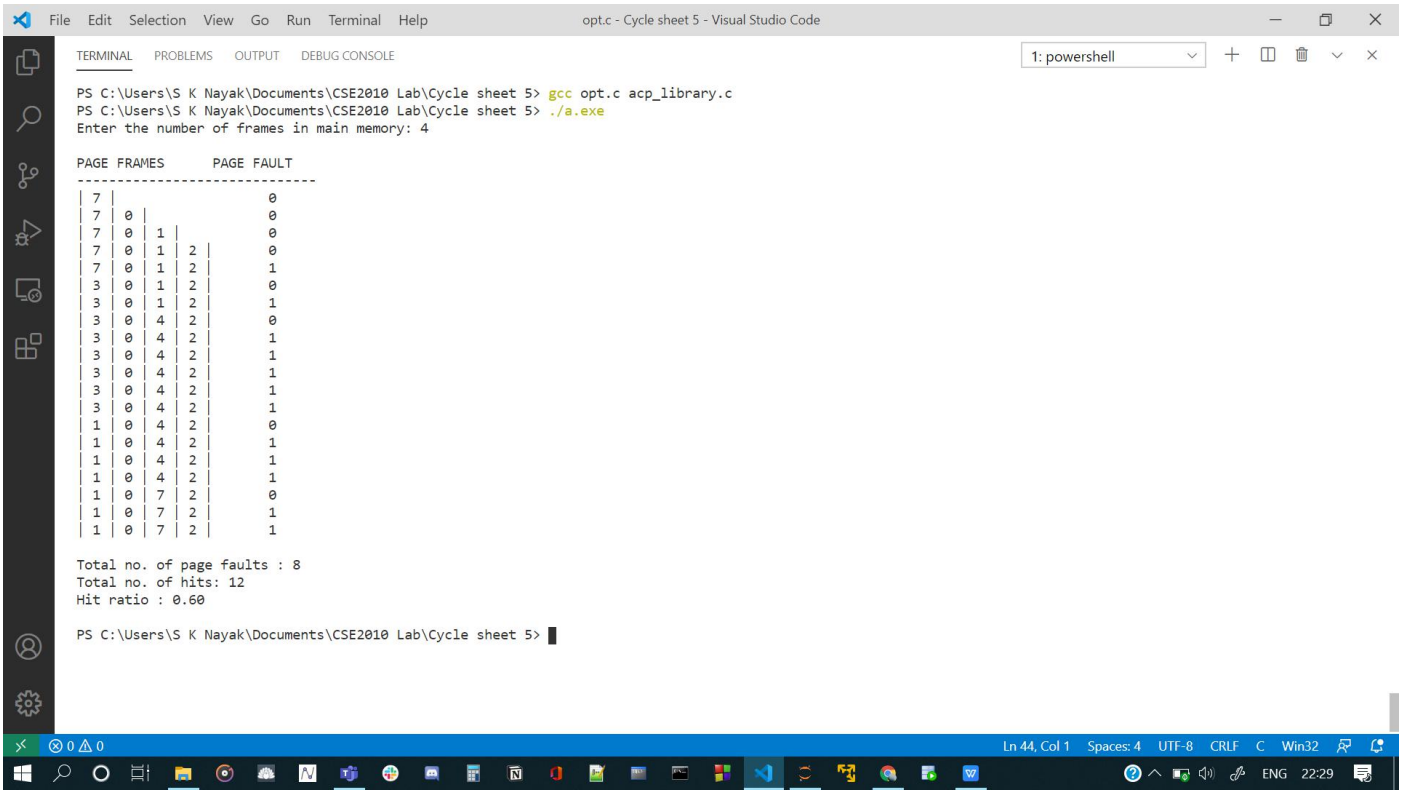


5. Optimal Page Replacement algorithm (Test program)

Code:

```c
// C program to simulate optimal page replacement scheme
//  OS replaces the page that will not be used for the longest period of time in future


# include <stdio.h>
# include <stdlib.h>
# include "acp_library.h"
# define MAX 5


int main()
{
    int status, mm_frame, *page_stream, s, i;

    page_frames_opt *opt_pf2 = NULL;

    printf("Enter the number of frames in main memory: ");
    status = scanf("%d", &mm_frame);

    // Input validation
    while (status == 0 || mm_frame <= 0 || mm_frame > MAX)
    {
        printf("Invalid Input!\n");
        printf("Enter the number of frames in main memory: ");
        status = scanf("%d", &mm_frame);
        fflush(stdin);
    }

    FILE *fptr;
    fptr = fopen("page_stream.txt", "r");

    fscanf(fptr, "%d", &s);

    page_stream = (int *)malloc(s * sizeof(int));

    for(i = 0; i < s; i++)
    {
        fscanf(fptr, "%d", page_stream + i);
    }

    fclose(fptr);

    // Initializing the data structures
    opt_pf2 = initial_pf2(opt_pf2, mm_frame);

    OPT(opt_pf2, page_stream, s, mm_frame);

    return 0;
}
```

Output:



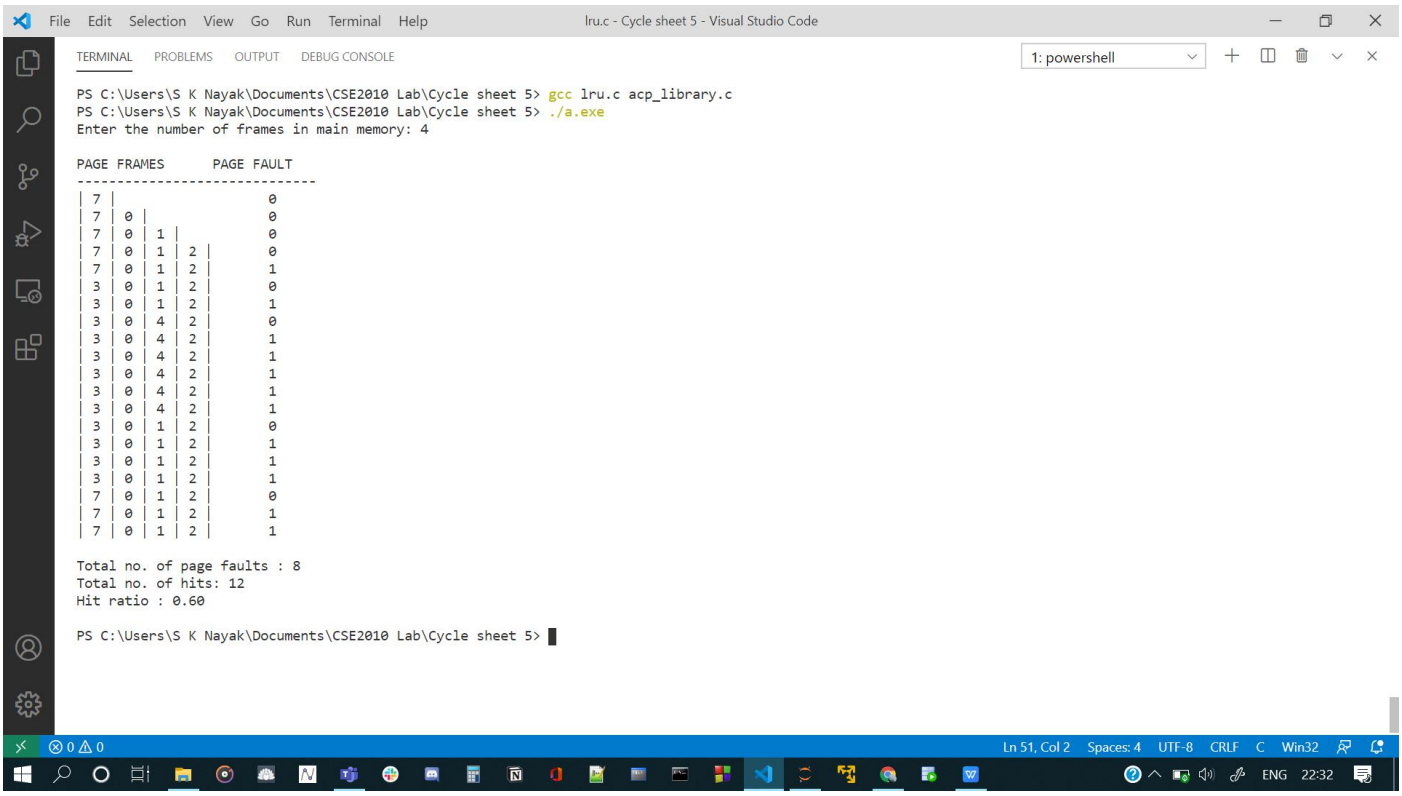
6. LRU Page Replacement algorithm (Test program)

Code:

```
    // C Program to simulate LRU Page replacement scheme
    // In Least Recently Used (LRU) algorithm is a Greedy algorithm
    // where the page to be replaced is least recently used
    // Given memory capacity (as number of pages it can hold) and a string representing pages t
o be referred
    // write a function to find number of page faults

    # include <stdio.h>
    # include <stdlib.h>
    # include "acp_library.h"
    # define M 5


    int main()
    {
        int status, mm_frame, *page_stream, s, i;

        page_frames_lru *lru_pf3 = NULL;

        FILE *fptr;
        fptr = fopen("page_stream.txt", "r");

        fscanf(fptr, "%d", &s);

        page_stream = (int *)malloc(s * sizeof(int));

        for(i = 0; i < s; i++)
        {
```

```
        fscanf(fptr, "%d", page_stream + i);
    }

    fclose(fptr);

    printf("Enter the number of frames in main memory: ");
    status = scanf("%d", &mm_frame);

    // Input validation
    while (status == 0 || mm_frame <= 0 || mm_frame > M)
    {
        printf("Invalid Input!\n");
        printf("Enter the number of frames in main memory: ");
        status = scanf("%d", &mm_frame);
        fflush(stdin);
    }

    // Initializing the data structures
    lru_pf3 = initial_pf3(lru_pf3, mm_frame);

    LRU(lru_pf3, page_stream, s, mm_frame);

    return 0;
}
```

Output:



We can see that, all the above programs are functioning perfectly. Hence the library is working just as it was written to. The header file is giving all the correct abstractions. For the particular inputs taken, we can see that best fit works best as a memory allocation scheme and optimal page replacement algorithm works best as a page replacement scheme in operating system.

CONCLUSION

The project has provided a C library with various functions that cater to the simulation of First Fit memory allocation algorithm, Best Fit memory allocation algorithm, Worst Fit memory allocation algorithm and FIFO Page Replacement Algorithm, Optimal Page Replacement Algorithm, LRU page replacement algorithm.