**Ex. No:7 a IMPLEMENTATION OF PROCESS SCEDULING MECHANISM – FCFS, SJF, PRIORITY QUEUE**

**Program:**

```
#include <stdio.h>

#include <stdlib.h>


// Structure to represent a process

typedef struct {

    int pid;        // Process ID

    int burstTime; // Burst time of the process

} Process;


// Function to implement FCFS scheduling

void fcfs(Process processes[], int n) {

    int totalTime = 0;  // Current total time elapsed

    float avgWaitingTime = 0, avgTurnaroundTime = 0;


    // Calculate completion time, waiting time, and turnaround time for each process

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

        if (i == 0) {

            processes[i].completionTime = processes[i].burstTime;

        } else {

            processes[i].completionTime = processes[i-1].completionTime + processes[i].burstTime;

        }


        processes[i].waitingTime = processes[i].completionTime - processes[i].burstTime;

        processes[i].turnaroundTime = processes[i].completionTime;


       avgWaitingTime += processes[i].waitingTime;
```

```c
        avgTurnaroundTime += processes[i].turnaroundTime;
    }

    // Calculate average waiting time and average turnaround time
    avgWaitingTime /= n;
    avgTurnaroundTime /= n;

    // Display the process details
    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\tCompletion Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burstTime,
               processes[i].waitingTime, processes[i].turnaroundTime, processes[i].completionTime);
    }

    // Display average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", avgWaitingTime);
    printf("\nAverage Turnaround Time: %.2f\n", avgTurnaroundTime);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    Process processes[n];

    // Input burst times for each process
    printf("Enter burst time for each process:\n");
    for (int i = 0; i < n; i++) {
```

```
        processes[i].pid = i + 1;

        printf("Process %d: ", i + 1);

        scanf("%d", &processes[i].burstTime);

    }


    // Perform FCFS scheduling

    fcfs(processes, n);


    return 0;

}
```

**7b SJF Program:**

```
#include <stdio.h>

#include <stdlib.h>


// Structure to represent a process

typedef struct {

    int pid;       // Process ID

    int burstTime; // Burst time of the process

    int waitingTime; // Waiting time of the process

    int turnaroundTime; // Turnaround time of the process

    int completionTime; // Completion time of the process

} Process;


// Function to implement SJF scheduling

void sjf(Process processes[], int n) {

    int totalTime = 0;  // Current total time elapsed

    float avgWaitingTime = 0, avgTurnaroundTime = 0;


    // Sort processes by burst time (SJF - Non-preemptive)
```

```
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
        if (processes[j].burstTime > processes[j + 1].burstTime) {
            // Swap processes[j] and processes[j+1]
            Process temp = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = temp;
        }
    }
}

// Calculate completion time, waiting time, and turnaround time for each process
for (int i = 0; i < n; i++) {
    if (i == 0) {
        processes[i].completionTime = processes[i].burstTime;
    } else {
        processes[i].completionTime = processes[i-1].completionTime + processes[i].burstTime;
    }

    processes[i].waitingTime = processes[i].completionTime - processes[i].burstTime;
    processes[i].turnaroundTime = processes[i].completionTime;

    avgWaitingTime += processes[i].waitingTime;
    avgTurnaroundTime += processes[i].turnaroundTime;
}

// Calculate average waiting time and average turnaround time
avgWaitingTime /= n;
avgTurnaroundTime /= n;
```

```
    // Display the process details
    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\tCompletion Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burstTime,
               processes[i].waitingTime, processes[i].turnaroundTime, processes[i].completionTime);
    }

    // Display average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", avgWaitingTime);
    printf("\nAverage Turnaround Time: %.2f\n", avgTurnaroundTime);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    Process processes[n];

    // Input burst times for each process
    printf("Enter burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
    }

    // Perform SJF scheduling
```

```
    sjf(processes, n);


    return 0;
}
```

**7c Priority Scheduling: Program:**

```
#include <stdio.h>
#include <stdlib.h>

// Structure to represent a process
typedef struct {
    int pid;            // Process ID
    int burstTime;      // Burst time of the process
    int priority;       // Priority of the process (lower value means higher priority)
    int waitingTime;    // Waiting time of the process
    int turnaroundTime; // Turnaround time of the process
    int completionTime; // Completion time of the process
} Process;

// Function to implement Priority Queue scheduling
void priority_queue(Process processes[], int n) {
    float avgWaitingTime = 0, avgTurnaroundTime = 0;

    // Sort processes by priority (lower number for higher priority)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                // Swap processes[j] and processes[j+1]
                Process temp = processes[j];
                processes[j] = processes[j + 1];
```

```
                processes[j + 1] = temp;
            }
        }
    }

    // Calculate completion time, waiting time, and turnaround time for each process
    processes[0].completionTime = processes[0].burstTime;
    processes[0].turnaroundTime = processes[0].completionTime;
    processes[0].waitingTime = 0;

    for (int i = 1; i < n; i++) {
        processes[i].completionTime = processes[i - 1].completionTime + processes[i].burstTime;
        processes[i].turnaroundTime = processes[i].completionTime;
        processes[i].waitingTime = processes[i].turnaroundTime - processes[i].burstTime;

        avgWaitingTime += processes[i].waitingTime;
        avgTurnaroundTime += processes[i].turnaroundTime;
    }

    // Calculate average waiting time and average turnaround time
    avgWaitingTime /= n;
    avgTurnaroundTime /= n;

    // Display the process details
    printf("Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\tCompletion Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burstTime,
                              processes[i].priority,  processes[i].waitingTime,  processes[i].turnaroundTime,
processes[i].completionTime);
```

```
    }

    // Display average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", avgWaitingTime);
    printf("\nAverage Turnaround Time: %.2f\n", avgTurnaroundTime);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    Process processes[n];

    // Input burst times and priorities for each process
    printf("Enter burst time and priority for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Process %d:\n", i + 1);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burstTime);
        printf("Priority: ");
        scanf("%d", &processes[i].priority);
    }

    // Perform Priority Queue scheduling
    priority_queue(processes, n);

    return 0;
```

```
}
```

**Ex. No:8** PRODUCER CONSUMER PROBLEM USING SEMAPHORES

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5  // Size of the shared buffer
#define N 10           // Number of items to be produced/consumed

int buffer[BUFFER_SIZE];  // Shared buffer
int in = 0;  // Index where the next item will be inserted
int out = 0; // Index from where the next item will be removed

sem_t empty, full, mutex;  // Semaphores

void *producer(void *arg) {
    int item;
    for (int i = 0; i < N; i++) {
        item = rand() % 100;  // Produce a random item
        sem_wait(&empty);     // Wait if buffer is full
        sem_wait(&mutex);     // Begin critical section
        buffer[in] = item;    // Insert item into buffer
        printf("Produced: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;  // Move to next available slot
        sem_post(&mutex);     // End critical section
        sem_post(&full);      // Signal that buffer is no longer empty
```

```
        sleep(rand() % 3);    // Sleep for a random time
    }
    return NULL;
}

void *consumer(void *arg) {
    int item;
    for (int i = 0; i < N; i++) {
        sem_wait(&full);      // Wait if buffer is empty
        sem_wait(&mutex);     // Begin critical section
        item = buffer[out];   // Remove item from buffer
        printf("Consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;  // Move to next available slot
        sem_post(&mutex);     // End critical section
        sem_post(&empty);     // Signal that buffer is no longer full
        sleep(rand() % 3);    // Sleep for a random time
    }
    return NULL;
}

int main() {
    pthread_t prod_tid, cons_tid;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    // Create producer and consumer threads
```

```c
    pthread_create(&prod_tid, NULL, producer, NULL);
    pthread_create(&cons_tid, NULL, consumer, NULL);

    // Join threads
    pthread_join(prod_tid, NULL);
    pthread_join(cons_tid, NULL);

    // Destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

**Ex. No:9  READERS AND WRITERS PROBLEM**

**Program:**

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

sem_t rw_mutex; // Semaphore for controlling access to the resource
sem_t mutex;    // Semaphore for controlling access to the read count
int read_count = 0; // Number of readers currently reading

void* reader(void* arg) {
    int reader_id = *((int*)arg);

    // Entry section
    sem_wait(&mutex);
    read_count++;
    if (read_count == 1) {
        sem_wait(&rw_mutex);
    }
    sem_post(&mutex);

    // Critical section
```

```c
    printf("Reader %d: reading\n", reader_id);

    // Exit section
    sem_wait(&mutex);
    read_count--;
    if (read_count == 0) {
        sem_post(&rw_mutex);
    }
    sem_post(&mutex);

    return NULL;
}

void* writer(void* arg) {
    int writer_id = *((int*)arg);

    // Entry section
    sem_wait(&rw_mutex);

    // Critical section
    printf("Writer %d: writing\n", writer_id);

    // Exit section
    sem_post(&rw_mutex);

    return NULL;
}

int main() {
    pthread_t readers[5], writers[5];
    int reader_ids[5], writer_ids[5];

    // Initialize semaphores
    sem_init(&rw_mutex, 0, 1);
    sem_init(&mutex, 0, 1);

    // Create reader and writer threads
    for (int i = 0; i < 5; i++) {
        reader_ids[i] = i + 1;
        writer_ids[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
        pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < 5; i++) {
        pthread_join(readers[i], NULL);
```

```
        pthread_join(writers[i], NULL);
    }

    // Destroy semaphores
    sem_destroy(&rw_mutex);
    sem_destroy(&mutex);

    return 0;
}
```

**Ex. No:10  DINER'S PHILOSOPHER PROBLEM**

**Program:**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define N 5 // Number of philosophers

pthread_mutex_t forks[N];
pthread_t philosophers[N];
int philosopher_ids[N];

void* philosopher(void* arg) {
    int id = *((int*)arg);

    while (1) {
        // Thinking
        printf("Philosopher %d is thinking.\n", id);
        sleep(rand() % 3 + 1); // Random thinking time

        // Hungry
        printf("Philosopher %d is hungry.\n", id);

        // Pick up left fork
        pthread_mutex_lock(&forks[id]);
        printf("Philosopher %d picked up left fork %d.\n", id, id);

        // Pick up right fork
        pthread_mutex_lock(&forks[(id + 1) % N]);
        printf("Philosopher %d picked up right fork %d.\n", id, (id + 1) % N);

        // Eating
        printf("Philosopher %d is eating.\n", id);
        sleep(rand() % 3 + 1); // Random eating time
```

```c
        // Put down right fork
        pthread_mutex_unlock(&forks[(id + 1) % N]);
        printf("Philosopher %d put down right fork %d.\n", id, (id + 1) % N);

        // Put down left fork
        pthread_mutex_unlock(&forks[id]);
        printf("Philosopher %d put down left fork %d.\n", id, id);
    }

    return NULL;
}

int main() {
    // Initialize mutexes
    for (int i = 0; i < N; i++) {
        pthread_mutex_init(&forks[i], NULL);
    }

    // Create philosopher threads
    for (int i = 0; i < N; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_ids[i]);
    }

    // Wait for all philosopher threads to finish (they won't in this example)
    for (int i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Destroy mutexes
    for (int i = 0; i < N; i++) {
        pthread_mutex_destroy(&forks[i]);
    }

    return 0;
}
```

**Ex. No:11 FIRST FIT, WORST FIT, BEST FIT ALLOCATION STRATEGY**

**Program:**

```c
#include <stdio.h>



#define MAX 25



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

```c
    int allocation[n];

    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

    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;
            }
        }
    }

    printf("\nFirst Fit Allocation:\n");
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
```

```
for (int i = 0; i < n; i++) {
    allocation[i] = -1;
}

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 || blockSize[bestIdx] > blockSize[j]) {
                bestIdx = j;
            }
        }
    }

    if (bestIdx != -1) {
        allocation[i] = bestIdx;
        blockSize[bestIdx] -= processSize[i];
    }
}

printf("\nBest Fit Allocation:\n");
printf("Process No.\tProcess Size\tBlock No.\n");
for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t", i + 1, processSize[i]);
    if (allocation[i] != -1)
        printf("%d\n", allocation[i] + 1);
    else
        printf("Not Allocated\n");
```

```
    }
}

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

    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < n; i++) {
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {
                    worstIdx = j;
                }
            }
        }

        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }

    printf("\nWorst Fit Allocation:\n");
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main() {
    int blockSize[MAX], processSize[MAX];
    int m, n;

    printf("Enter the number of memory blocks: ");
    scanf("%d", &m);
    printf("Enter the size of each block:\n");
    for (int i = 0; i < m; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }

    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processSize[i]);
    }

    firstFit(blockSize, m, processSize, n);
```

```
    // Reset block sizes for the next strategy
    for (int i = 0; i < m; i++) {
        printf("Enter the size of block %d again: ", i + 1);
        scanf("%d", &blockSize[i]);
    }

    bestFit(blockSize, m, processSize, n);

    // Reset block sizes for the next strategy
    for (int i = 0; i < m; i++) {
        printf("Enter the size of block %d again: ", i + 1);
        scanf("%d", &blockSize[i]);
    }

    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

**Ex. No:12 BANKER'S ALGORITHM**

**Program:**

```
#include <stdio.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int allocation[MAX_PROCESSES][MAX_RESOURCES];
int maximum[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
```

```
int available[MAX_RESOURCES];
int processes, resources;

// Function to calculate the Need matrix
void calculateNeed() {
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            need[i][j] = maximum[i][j] - allocation[i][j];
        }
    }
}

// Function to check if the system is in a safe state
int isSafe() {
    int work[MAX_RESOURCES], finish[MAX_PROCESSES];
    for (int i = 0; i < resources; i++) {
        work[i] = available[i];
    }
    for (int i = 0; i < processes; i++) {
        finish[i] = 0;
    }

    int safeSequence[MAX_PROCESSES], count = 0;

    while (count < processes) {
        int found = 0;
        for (int p = 0; p < processes; p++) {
            if (finish[p] == 0) {
                int j;
```

```
            for (j = 0; j < resources; j++) {
                if (need[p][j] > work[j])
                    break;
            }
            if (j == resources) {
                for (int k = 0; k < resources; k++) {
                    work[k] += allocation[p][k];
                }
                safeSequence[count++] = p;
                finish[p] = 1;
                found = 1;
            }
        }
    }
    if (found == 0) {
        printf("The system is not in a safe state.\n");
        return 0;
    }
}

printf("The system is in a safe state.\nSafe sequence is: ");
for (int i = 0; i < processes; i++) {
    printf("%d ", safeSequence[i]);
}
printf("\n");
return 1;
}

int main() {
```

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

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

printf("Enter the maximum matrix:\n");
for (int i = 0; i < processes; i++) {
    for (int j = 0; j < resources; j++) {
        scanf("%d", &maximum[i][j]);
    }
}

printf("Enter the available resources:\n");
for (int i = 0; i < resources; i++) {
    scanf("%d", &available[i]);
}

calculateNeed();

if (isSafe()) {
    printf("The system is safe.\n");
} else {
```

```
        printf("The system is not safe.\n");
    }

    return 0;
}
```

**Ex. No:13** SIMULATE PAGING TECHNIQUE OF MEMORY MANAGEMENT

**Program:**

```
#include <stdio.h>
#include <stdlib.h>

#define PAGE_SIZE 4  // Define the page size (number of words per page)
#define MEMORY_SIZE 16  // Define the total size of the physical memory (number of words)

// Function to simulate logical to physical address translation using paging
void simulatePaging(int *pageTable, int logicalAddress) {
    int pageNumber = logicalAddress / PAGE_SIZE;
    int offset = logicalAddress % PAGE_SIZE;

    if (pageNumber >= MEMORY_SIZE / PAGE_SIZE) {
        printf("Error: Page number %d is out of bounds.\n", pageNumber);
        return;
    }

    int frameNumber = pageTable[pageNumber];
    if (frameNumber == -1) {
        printf("Error: Page %d is not loaded in memory.\n", pageNumber);
        return;
    }
```

```
    int physicalAddress = frameNumber * PAGE_SIZE + offset;
    printf("Logical Address %d -> Physical Address %d\n", logicalAddress, physicalAddress);
}

int main() {
    int pageTable[MEMORY_SIZE / PAGE_SIZE];
    int numberOfPages = MEMORY_SIZE / PAGE_SIZE;

    // Initialize page table (for simplicity, we assume pages are loaded sequentially in frames)
    for (int i = 0; i < numberOfPages; i++) {
        pageTable[i] = i;
    }

    // Set some pages as not loaded in memory (-1)
    pageTable[2] = -1;  // Simulate page 2 is not in memory
    pageTable[3] = -1;  // Simulate page 3 is not in memory

    int logicalAddress;

    // Input logical addresses to be translated
    while (1) {
        printf("Enter a logical address (negative number to quit): ");
        scanf("%d", &logicalAddress);

        if (logicalAddress < 0) break;

        simulatePaging(pageTable, logicalAddress);
    }
```

```
    return 0;

}
```

**Ex. No:14 SIMULATE PAGE REPLACEMENT ALGORITHM**

**Program:**

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_FRAMES 10
#define MAX_REF_STR_LEN 25

void fifo(int ref_str[], int ref_len, int frames);
void lru(int ref_str[], int ref_len, int frames);
void optimal(int ref_str[], int ref_len, int frames);

int main() {
    int ref_str[MAX_REF_STR_LEN] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
    int ref_len = 20;
    int frames = 3;

    printf("FIFO Page Replacement Algorithm:\n");
    fifo(ref_str, ref_len, frames);

    printf("\nLRU Page Replacement Algorithm:\n");
    lru(ref_str, ref_len, frames);

    printf("\nOptimal Page Replacement Algorithm:\n");
    optimal(ref_str, ref_len, frames);

    return 0;
}

void fifo(int ref_str[], int ref_len, int frames) {
    int page_faults = 0;
    int frame[MAX_FRAMES];
    int index = 0;

    for (int i = 0; i < frames; i++) {
        frame[i] = -1;
    }

    for (int i = 0; i < ref_len; i++) {
        bool found = false;

        for (int j = 0; j < frames; j++) {
            if (frame[j] == ref_str[i]) {
```

```
                found = true;
                break;
            }
        }

        if (!found) {
            frame[index] = ref_str[i];
            index = (index + 1) % frames;
            page_faults++;
        }

        printf("Frame state: ");
        for (int j = 0; j < frames; j++) {
            if (frame[j] != -1) {
                printf("%d ", frame[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }

    printf("Total Page Faults: %d\n", page_faults);
}

void lru(int ref_str[], int ref_len, int frames) {
    int page_faults = 0;
    int frame[MAX_FRAMES];
    int use_time[MAX_FRAMES];

    for (int i = 0; i < frames; i++) {
        frame[i] = -1;
        use_time[i] = 0;
    }

    for (int i = 0; i < ref_len; i++) {
        bool found = false;

        for (int j = 0; j < frames; j++) {
            if (frame[j] == ref_str[i]) {
                found = true;
                use_time[j] = i;
                break;
            }
        }

        if (!found) {
```

```
                int lru_index = 0;
                for (int j = 1; j < frames; j++) {
                    if (use_time[j] < use_time[lru_index]) {
                        lru_index = j;
                    }
                }
                frame[lru_index] = ref_str[i];
                use_time[lru_index] = i;
                page_faults++;
            }

            printf("Frame state: ");
            for (int j = 0; j < frames; j++) {
                if (frame[j] != -1) {
                    printf("%d ", frame[j]);
                } else {
                    printf("- ");
                }
            }
            printf("\n");
        }

        printf("Total Page Faults: %d\n", page_faults);
    }

    void optimal(int ref_str[], int ref_len, int frames) {
        int page_faults = 0;
        int frame[MAX_FRAMES];

        for (int i = 0; i < frames; i++) {
            frame[i] = -1;
        }

        for (int i = 0; i < ref_len; i++) {
            bool found = false;

            for (int j = 0; j < frames; j++) {
                if (frame[j] == ref_str[i]) {
                    found = true;
                    break;
                }
            }

            if (!found) {
                int optimal_index = -1;
                int farthest = i + 1;
                for (int j = 0; j < frames; j++) {
```

```
                int next_use = ref_len;
                for (int k = i + 1; k < ref_len; k++) {
                    if (frame[j] == ref_str[k]) {
                        next_use = k;
                        break;
                    }
                }
                if (next_use > farthest) {
                    farthest = next_use;
                    optimal_index = j;
                }
            }
            if (optimal_index == -1) {
                for (int j = 0; j < frames; j++) {
                    if (frame[j] == -1) {
                        optimal_index = j;
                        break;
                    }
                }
            }
            frame[optimal_index] = ref_str[i];
            page_faults++;
        }

        printf("Frame state: ");
        for (int j = 0; j < frames; j++) {
            if (frame[j] != -1) {
                printf("%d ", frame[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }

    printf("Total Page Faults: %d\n", page_faults);
}
```

**Ex. No:15 IMPLEMENT MEMORY MANAGEMENT SCHEME**

**Program:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MEMORY_SIZE 1000  // Define the total size of memory
#define MAX_BLOCKS 100    // Define the maximum number of memory blocks
```

```c
typedef struct {
    int start;
    int size;
    bool is_free;
} MemoryBlock;

MemoryBlock memory[MAX_BLOCKS];
int memory_count = 0;

void init_memory() {
    // Initialize the memory with a single free block
    memory[0].start = 0;
    memory[0].size = MEMORY_SIZE;
    memory[0].is_free = true;
    memory_count = 1;
}

void display_memory() {
    printf("Memory Blocks:\n");
    for (int i = 0; i < memory_count; i++) {
        printf("Block %d: Start = %d, Size = %d, %s\n", i, memory[i].start, memory[i].size,
memory[i].is_free ? "Free" : "Allocated");
    }
}

void merge_free_blocks() {
    for (int i = 0; i < memory_count - 1; i++) {
        if (memory[i].is_free && memory[i + 1].is_free) {
            memory[i].size += memory[i + 1].size;
            for (int j = i + 1; j < memory_count - 1; j++) {
                memory[j] = memory[j + 1];
            }
            memory_count--;
            i--;
        }
```

```
    }
}

void* allocate_memory(int size) {
    for (int i = 0; i < memory_count; i++) {
        if (memory[i].is_free && memory[i].size >= size) {
            if (memory[i].size > size) {
                // Split the block
                for (int j = memory_count; j > i; j--) {
                    memory[j] = memory[j - 1];
                }
                memory[i + 1].start = memory[i].start + size;
                memory[i + 1].size = memory[i].size - size;
                memory[i + 1].is_free = true;
                memory[i].size = size;
                memory_count++;
            }
            memory[i].is_free = false;
            return (void*)(memory[i].start);
        }
    }
    printf("Error: Not enough memory to allocate %d bytes\n", size);
    return NULL;
}

void free_memory(void* ptr) {
    int address = (int)ptr;
    for (int i = 0; i < memory_count; i++) {
        if (memory[i].start == address) {
            memory[i].is_free = true;
            merge_free_blocks();
            return;
        }
    }
    printf("Error: Invalid memory address %d\n", address);
}
```

```
int main() {
    init_memory();
    display_memory();

    void* block1 = allocate_memory(200);
    display_memory();

    void* block2 = allocate_memory(300);
    display_memory();

    free_memory(block1);
    display_memory();

    void* block3 = allocate_memory(100);
    display_memory();

    free_memory(block2);
    display_memory();

    free_memory(block3);
    display_memory();

    return 0;
}
```