EXPERIMENT NO. 1

UNIX COMMANDS

AIM: To study and execute the commands in Unix.

COMMAND:

1. Date Command: used to display the current date and time.

Syntax:

$date

$date +%ch

Options: -

a = Abbrevated weekday.
A = Full weekday.
b = Abbrevated month.
B = Full month.
c = Current day and time.
C = Display the century as a decimal number.
d = Day of the month.
D = Day in „mm/dd/yy" format
h = Abbrevated month day.
H = Display the hour.
L = Day of the year.
m = Month of the year.
M = Minute.
P = Display AM or PM
S = Seconds
T = HH:MM: SS format
u = Week of the year.
y = Display the year in 2 digits.
Y = Display the full year.
Z = Time zone.

To change the format:

Syntax:

$date „+%H-%M-%S"

2. Calender Command: used to display the calendar of the year or the particular month of a calendar year.

Syntax:

a. $cal <year>
b. $cal <month> <year>

- Here the first syntax gives the entire calendar for a given year & the second Syntax gives the calendar of the reserved month of that year.

3. Echo Command: used to print the arguments on the screen.

Syntax: $echo <text>

Multi-line echo command: To have the output in the same line, the following commands can be used.

Syntax: $echo <text\>text

To have the output in different line, the following command can be used.

Syntax: $echo "text
>line2
>line3"

4. Banner Command: used to display the arguments in „#" symbol.

Syntax: **$banner <arguments>**

EXPERIMENT NO. 2

THREAD AND PROCESS

AIM: To write a program for the implementation of Thread and Process.

THEORY:

Process: A process is an instance of a program that is being executed. It includes the program's code, current activity, and allocated resources (like memory and file handles). The operating system manages processes by designing, scheduling, and terminating them as needed, ensuring efficient use of CPU resources.

Thread: A thread is the smallest unit of execution within a process. Threads can run independently, but they share the same memory space within the process. A scheduler in the operating system manages threads, allowing for parallel execution and better utilization of CPU cores.

SOURCE CODE:

Thread Implementation:

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

void* threadFunction(void* arg) {
    int *id = (int *)arg;
    printf("Thread %d is running...\n", *id);
    free(id);
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    int *id1 = malloc(sizeof(int));
    int *id2 = malloc(sizeof(int));
    *id1 = 1;
    *id2 = 2;

    pthread_create(&thread1, NULL, threadFunction, id1);
    pthread_create(&thread2, NULL, threadFunction, id2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Both threads are completed.\n");
    return 0;
}
```

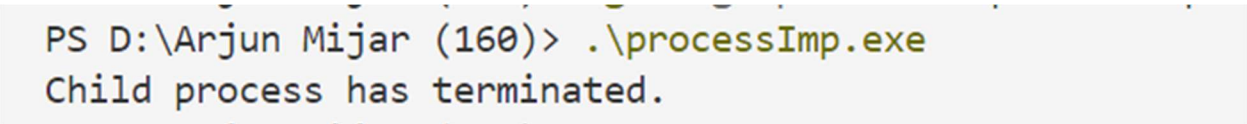
OUTPUT:

```
PS D:\Arjun Mijar (160)> .\threadImp.exe
Thread 1 is running...
Thread 2 is running...
Both threads are completed.
```

Process Implementation:

```
#include <windows.h>
#include <stdio.h>
int main() {

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    if (!CreateProcess(
            NULL,
            "notepad.exe",
            NULL,
            NULL,
            FALSE,
            0,
            NULL,
            NULL,
            &si,
            &pi)
    ) {
        printf("CreateProcess failed (%d).\n", GetLastError());
        return 1;
    }

    WaitForSingleObject(pi.hProcess, INFINITE);

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

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

    return 0;
}
```

OUTPUT:



EXPERIMENT NO. 3

PRODUCER-CONSUMER PROBLEM

AIM: To write a C program for the implementation of a producer-consumer problem using Semaphores.

THEORY:

The Producer-Consumer problem is a classical synchronization issue in operating systems. It involves two types of processes, the producer and the consumer, which share a common, fixed-size buffer. The producer's job is to generate data and place it into the buffer, while the consumer's job is to retrieve and process the data from the buffer.

To prevent race conditions and ensure synchronization between these processes, semaphores are used. Without proper synchronization, the producer may try to add data to a full buffer, or the consumer may try to consume data from an empty buffer, leading to data inconsistency or system failure.

Semaphores manage the access to shared resources (the buffer) between the producer and consumer by enforcing mutual exclusion and controlling the buffer's empty and full slots.

ALGORITHM:

1. **Producer:**

- Wait for the empty semaphore (which tracks the number of empty slots).
- Wait for the mutex semaphore (which ensures exclusive access to the buffer).
- Add an item to the buffer.
- Signal the mutex semaphore (release exclusive access to the buffer).
- Signal the full semaphore (which tracks the number of full slots).

2. **Consumer:**

- Wait for the full semaphore (which tracks the number of full slots).
- Wait for the mutex semaphore (which ensures exclusive access to the buffer).
- Remove an item from the buffer.
- Signal the mutex semaphore (release exclusive access to the buffer).
- Signal the empty semaphore (which tracks the number of empty slots).
-

The main components are:

- **Empty Semaphore**: Tracks the available empty slots in the buffer.
- **Full Semaphore**: Tracks the available filled slots in the buffer.
- **Mutex Semaphore**: Ensures that only one process (either producer or consumer) can access the buffer at any given time, providing mutual exclusion.

SOURCE CODE:

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

#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t empty, full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    int item;
    for (int i = 0; i < 3; i++) {
        item = rand() % 100;
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);


        buffer[in] = item;
        printf("Producer produced: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&full);
        sleep(1);
    }
}

// Function for consumer
void* consumer(void* arg) {
    int item;
    for (int i = 0; i < 3; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        item = buffer[out];
        printf("Consumer consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        sleep(1);
    }
}
```

```
int main() {

    pthread_t prod_thread, cons_thread;
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\proceConsumer.exe
Producer produced: 41
Consumer consumed: 41
Producer produced: 67
Consumer consumed: 67
Producer produced: 34
Consumer consumed: 34
```

EXPERIMENT NO. 4

DINING-PHILOSOPHERS PROBLEM

AIM: To write a C program to Simulate the concept of the Dining-philosophers problem.

THEORY:

The Dining Philosophers problem is a classical synchronization problem used to illustrate issues like resource contention, deadlock, and starvation. It involves five philosophers seated around a circular table, each alternating between thinking and eating.

A bowl of noodles is placed in the center, and five chopsticks (or forks) are placed between each pair of philosophers. A philosopher can only eat if both the left and right chopsticks are available. If a philosopher can't acquire both chopsticks, they must put down any chopstick they've picked up and return to thinking.

This problem demonstrates potential **deadlock** (when all philosophers hold one chopstick and wait indefinitely for the second) and **starvation** (when a philosopher never gets to eat). Solutions typically focus on preventing these issues.

ALGORITHM:

Step 1: **Initialization**

- Initialize one semaphore for each chopstick (or fork), all set to 1, indicating availability.
- Initialize a mutex (or another mechanism) to control access and avoid deadlock.
- Assign each philosopher a unique ID.

Step 2: **Philosopher Routine**

1. **Thinking**:
    - The philosopher "thinks" for a random amount of time, simulating this behavior with sleep().
2. **Picking up chopsticks**:
    - Lock the mutex to ensure mutual exclusion and prevent deadlock while acquiring the chopsticks.
    - Wait to acquire the left chopstick (sem_wait(left_fork)).
    - Wait to acquire the right chopstick (sem_wait(right_fork)).
    - Unlock the mutex after acquiring both chopsticks.
3. **Eating**:
    - The philosopher "eats" for a random time (simulated with sleep()).
4. **Putting down chopsticks**:
    - Release the left chopstick (sem_post(left_fork)).
    - Release the right chopstick (sem_post(right_fork)).
5. **Repeat**:
    - Go back to the thinking phase and repeat the cycle.

SOURCE CODE:

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

#define N 2
sem_t forks[N];
pthread_mutex_t mutex;

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

while (1) {
printf("Philosopher %d is thinking...\n", id);
sleep(1);

pthread_mutex_lock(&mutex);
sem_wait(&forks[id]);
sem_wait(&forks[(id + 1) % N]);
pthread_mutex_unlock(&mutex);

printf("Philosopher %d is eating...\n", id);
sleep(2);

sem_post(&forks[id]);
sem_post(&forks[(id + 1) % N]);

printf("Philosopher %d finished eating and is thinking again...\n", id);
}
}

int main() {
pthread_t philosophers[N];
int ids[N];
pthread_mutex_init(&mutex, NULL);

for (int i = 0; i < N; i++) {
sem_init(&forks[i], 0, 1);
ids[i] = i;
}

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

for (int i = 0; i < N; i++) {
pthread_join(philosophers[i], NULL);
}

for (int i = 0; i < N; i++) {
```

```
sem_destroy(&forks[i]);
}
pthread_mutex_destroy(&mutex);
return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\diningPhilosopher.exe
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 0 is eating...
Philosopher 0 finished eating and is thinking again...
Philosopher 0 is thinking...
Philosopher 1 is eating...
Philosopher 1 finished eating and is thinking again...
Philosopher 1 is thinking...
Philosopher 0 is eating...
Philosopher 1 is eating...
Philosopher 0 finished eating and is thinking again...
Philosopher 0 is thinking...
Philosopher 1 finished eating and is thinking again...
Philosopher 0 is eating...
Philosopher 1 is thinking...
Philosopher 0 finished eating and is thinking again...
Philosopher 0 is thinking...
Philosopher 1 is eating...
Philosopher 1 finished eating and is thinking again...
Philosopher 1 is thinking...
Philosopher 0 is eating...
Philosopher 0 finished eating and is thinking again...
```

EXPERIMENT NO. 5

NON-PREEMPTIVE CPU SCHEDULING ALGORITHMS

AIM: Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.

FCFS b) SJF c) Priority

THEORY:

CPU scheduling is a fundamental aspect of operating systems, determining the order in which processes are executed. In **non-preemptive scheduling**, once a process starts execution, it runs until completion without being interrupted. Here, we explore three non-preemptive scheduling algorithms: **First Come First Serve (FCFS)**, **Shortest Job First (SJF)**, and **Priority Scheduling**. These algorithms focus on minimizing important metrics such as **Turnaround Time** and **Waiting Time**.

Key Definitions:

Turnaround Time: The total time taken from the moment a process enters the ready queue until its completion.

Turnaround Time = Completion Time − Arrival Time

Waiting Time: The time a process spends waiting in the ready queue before it gets the CPU.

Waiting Time = Turnaround Time − Burst Time

a)First Come First Serve (FCFS)

- FCFS is the simplest scheduling algorithm where processes are executed in the order they arrive.
- It is a **non-preemptive** algorithm, meaning once a process starts executing, it cannot be stopped until it finishes.
- **Disadvantages**: It may lead to a **convoy effect**, where shorter processes wait for long processes to complete, causing higher waiting times.

ALGORITHM:

1. **Sort** the processes by their arrival time.
2. **Execute** the processes in the order they arrive.
3. **Calculate** waiting time and turnaround time for each process:
    - Waiting Time = Current Time - Arrival Time
    - Turnaround Time = Completion Time - Arrival Time

b)Shortest Job First (SJF)

- In SJF, the process with the **shortest burst time** is executed next.
- SJF is optimal in terms of minimizing the average waiting time, but it can suffer from **starvation** if shorter processes keep arriving, causing longer processes to wait indefinitely.
- In non-preemptive SJF, once a process is selected for execution, it runs to completion.

ALGORITHM:

1. **Sort** the processes by their burst time.
2. **Execute** the process with the shortest burst time.
3. **Calculate** waiting time and turnaround time for each process:
    - Waiting Time = Current Time - Arrival Time
    - Turnaround Time = Completion Time - Arrival Time

c)Priority Scheduling

- In **Priority Scheduling**, each process is assigned a **priority**, and the process with the highest priority (usually the lowest number) is executed first.
- It is non-preemptive, so once a process is selected for execution, it runs until completion.
- **Disadvantages**: It may lead to **starvation** if high-priority processes continuously enter the system, delaying lower-priority processes indefinitely. A common solution to this is **aging**, where a process's priority is increased the longer it waits.

ALGORITHM*:*

1. **Sort** the processes based on their priority.
2. **Execute** the process with the highest priority.
3. **Calculate** waiting time and turnaround time for each process:
    - Waiting Time = Current Time - Arrival Time
    - Turnaround Time = Completion Time - Arrival Time

SOURCE CODE:

```
#include <stdio.h>
typedef struct {
    int pid;        // Process ID
    int burstTime;  // Burst Time
    int arrivalTime;// Arrival Time
    int priority;   // Priority (for Priority Scheduling)
    int waitingTime;
    int turnaroundTime;
} Process;

// Function to calculate waiting time and turnaround time
void calculateTimes(Process p[], int n) {
    int currentTime = 0;
    for (int i = 0; i < n; i++) {
        p[i].waitingTime = currentTime - p[i].arrivalTime;
        currentTime += p[i].burstTime;
        p[i].turnaroundTime = currentTime - p[i].arrivalTime;
    }
}

// FCFS Scheduling Algorithm
void fcfsScheduling(Process p[], int n) {
    printf("\n--- FCFS Scheduling ---\n");
    calculateTimes(p, n);
    for (int i = 0; i < n; i++) {
        printf("Process %d: Waiting Time = %d, Turnaround Time = %d\n", p[i].pid, p[i].waitingTime,
p[i].turnaroundTime);
    }
}
```

```
// SJF Scheduling Algorithm (non-preemptive)
void sjfScheduling(Process p[], int n) {
    printf("\n--- SJF Scheduling ---\n");
    // Sort by burst time
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            if (p[i].burstTime > p[j].burstTime) {
                Process temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
    calculateTimes(p, n);
    for (int i = 0; i < n; i++) {
        printf("Process %d: Waiting Time = %d, Turnaround Time = %d\n", p[i].pid, p[i].waitingTime,
p[i].turnaroundTime);
    }
}

// Priority Scheduling Algorithm (non-preemptive)
void priorityScheduling(Process p[], int n) {
    printf("\n--- Priority Scheduling ---\n");
    // Sort by priority (higher number = lower priority)
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            if (p[i].priority > p[j].priority) {
                Process temp = p[i];
                p[i] = p[j];
```

```
            p[j] = temp;
        }
    }
  }
  calculateTimes(p, n);
  for (int i = 0; i < n; i++) {
    printf("Process %d: Waiting Time = %d, Turnaround Time = %d\n", p[i].pid, p[i].waitingTime,
p[i].turnaroundTime);
  }
}

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

  Process processes[n];

  for (int i = 0; i < n; i++) {
    processes[i].pid = i+1;
    printf("Enter burst time for Process %d: ", i+1);
    scanf("%d", &processes[i].burstTime);
    printf("Enter arrival time for Process %d: ", i+1);
    scanf("%d", &processes[i].arrivalTime);
    printf("Enter priority for Process %d: ", i+1);
    scanf("%d", &processes[i].priority);
  }

  // FCFS Scheduling
  fcfsScheduling(processes, n);
```

```
    // SJF Scheduling
    sjfScheduling(processes, n);

    // Priority Scheduling
    priorityScheduling(processes, n);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\nonPreemCPUScheduling.exe
Enter the number of processes: 3
Enter burst time for Process 1: 6
Enter arrival time for Process 1: 1
Enter priority for Process 1: 3
Enter burst time for Process 2: 8
Enter arrival time for Process 2: 2
Enter priority for Process 2: 1
Enter burst time for Process 3: 7
Enter arrival time for Process 3: 3
Enter priority for Process 3: 2

--- FCFS Scheduling ---
Process 1: Waiting Time = -1, Turnaround Time = 5
Process 2: Waiting Time = 4, Turnaround Time = 12
Process 3: Waiting Time = 11, Turnaround Time = 18

--- SJF Scheduling ---
Process 1: Waiting Time = -1, Turnaround Time = 5
Process 3: Waiting Time = 3, Turnaround Time = 10
Process 2: Waiting Time = 11, Turnaround Time = 19

--- Priority Scheduling ---
Process 2: Waiting Time = -2, Turnaround Time = 6
Process 3: Waiting Time = 5, Turnaround Time = 12
Process 1: Waiting Time = 14, Turnaround Time = 20
```

EXPERIMENT NO. 6

PREEMPTIVE CPU SCHEDULING ALGORITHMS

AIM: Write a C program to simulate the following preemptive CPU scheduling algorithms to find turnaround time and waiting time.

a) Round Robin b) Priority

THEORY:

In **preemptive CPU scheduling**, processes can be interrupted and moved back to the ready queue before their execution is complete. This allows the operating system to allocate CPU time more dynamically, ensuring that higher-priority processes or time-sharing processes receive appropriate attention. In this scenario, we explore two preemptive scheduling algorithms: **Round Robin (RR)** and **Preemptive Priority Scheduling**.

Key Definitions:

1. **Turnaround Time**: The total time taken from the moment a process enters the ready queue until its completion.

   Turnaround Time=Completion Time−Arrival Time

2. **Waiting Time**: The time a process spends waiting in the ready queue before getting the CPU.

   Waiting Time=Turnaround Time−Burst Time

a) Round Robin (RR)

- **Round Robin (RR)** is one of the most widely used preemptive scheduling algorithms.
- Each process is assigned a **time quantum** (also called a time slice), during which it executes. If the process doesn't finish during that time, it is preempted and moved to the back of the ready queue, and the CPU is given to the next process.
- RR is particularly effective in **time-sharing systems** where fair CPU allocation among processes is necessary.
- **Advantages**: It is fair because each process gets an equal share of the CPU, and no process waits too long.
- **Disadvantages**: If the time quantum is too large, it behaves like **FCFS**; if too small, it leads to high context switching, which can reduce CPU efficiency.

ALGORITHM:

1. **Set the time quantum**.
2. **Put processes in the ready queue**.
3. **Execute each process for the time quantum**. If a process doesn't finish within the time quantum, move it to the back of the queue.
4. **Repeat** until all processes are finished.
5. **Calculate** waiting time and turnaround time for each process:
    - Waiting Time = Time spent in the ready queue.
    - Turnaround Time = Completion Time - Arrival Time.

b) Preemptive Priority Scheduling

- In **Preemptive Priority Scheduling**, each process is assigned a **priority**, and the process with the highest priority (usually the lowest number) gets the CPU. If a process with a higher priority arrives while another process is running, the current process is preempted, and the CPU is allocated to the higher-priority process.
- **Advantages**: Important or high-priority tasks are handled quickly.
- **Disadvantages**: It may lead to **starvation** if high-priority processes continuously arrive, leaving low-priority processes waiting indefinitely. This can be mitigated using techniques like **aging** to increase the priority of waiting processes over time.

ALGORITHM:

1. **Assign a priority** to each process.
2. **Execute the highest-priority process**. If a new process with a higher priority arrives, preempt the current process.
3. **Repeat** until all processes are completed.
4. **Calculate** waiting time and turnaround time for each process:
    - Waiting Time = Time spent in the ready queue.
    - Turnaround Time = Completion Time - Arrival Time.

SOURCE CODE:

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

typedef struct {
    int pid;        // Process ID
    int burstTime;  // Burst Time
    int arrivalTime;// Arrival Time
    int priority;   // Priority (for Priority Scheduling)
    int remainingTime; // Remaining Time (for preemptive scheduling)
    int waitingTime;
    int turnaroundTime;
    int completed;  // Completion flag (used in Priority Scheduling)
} Process;

// Function to calculate Turnaround Time and Waiting Time for Round Robin
void calculateRR(Process p[], int n, int timeQuantum) {
    int currentTime = 0;
    int remainingProcesses = n;

    while (remainingProcesses > 0) {
        for (int i = 0; i < n; i++) {
            if (p[i].remainingTime > 0) {
                if (p[i].remainingTime <= timeQuantum) {
                    currentTime += p[i].remainingTime;
                    p[i].remainingTime = 0;
                    p[i].turnaroundTime = currentTime - p[i].arrivalTime;
                    p[i].waitingTime = p[i].turnaroundTime - p[i].burstTime;
                    remainingProcesses--;
                } else {
```

```
                currentTime += timeQuantum;
                p[i].remainingTime -= timeQuantum;
            }
        }
    }
}

// Function to calculate Turnaround Time and Waiting Time for Preemptive Priority
void calculatePriority(Process p[], int n) {
    int currentTime = 0;
    int completedProcesses = 0;

    while (completedProcesses < n) {
        int idx = -1;
        int highestPriority = 9999;

        for (int i = 0; i < n; i++) {
            if (p[i].arrivalTime <= currentTime && p[i].remainingTime > 0 && p[i].priority <
highestPriority) {
                highestPriority = p[i].priority;
                idx = i;
            }
        }

        if (idx != -1) {
            p[idx].remainingTime--;
            currentTime++;

            if (p[idx].remainingTime == 0) {
```

```
            completedProcesses++;
            p[idx].turnaroundTime = currentTime - p[idx].arrivalTime;
            p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;
        }
    } else {
        currentTime++;
    }
  }
}

// Display the results
void displayResults(Process p[], int n) {
  printf("\nProcess\tBurst Time\tArrival Time\tPriority\tWaiting Time\tTurnaround Time\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
           p[i].pid, p[i].burstTime, p[i].arrivalTime, p[i].priority, p[i].waitingTime, p[i].turnaroundTime);
  }
}

int main() {
  int n, choice, timeQuantum;

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

  Process processes[n];

  for (int i = 0; i < n; i++) {
    processes[i].pid = i + 1;
    printf("Enter burst time for Process %d: ", i + 1);
```

```
        scanf("%d", &processes[i].burstTime);
        printf("Enter arrival time for Process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter priority for Process %d (higher number = lower priority): ", i + 1);
        scanf("%d", &processes[i].priority);
        processes[i].remainingTime = processes[i].burstTime;
        processes[i].completed = 0;
    }

    printf("\nSelect the scheduling algorithm:\n1. Round Robin\n2. Preemptive Priority Scheduling\n");
    scanf("%d", &choice);

    if (choice == 1) {
        printf("Enter time quantum for Round Robin: ");
        scanf("%d", &timeQuantum);
        calculateRR(processes, n, timeQuantum);
    } else if (choice == 2) {
        calculatePriority(processes, n);
    } else {
        printf("Invalid choice!\n");
        return 1;
    }

    displayResults(processes, n);

    return 0;
}
```

OUTPUT:

Round Robin:

```
PS D:\Arjun Mijar (160)> .\preemCPUScheduling.exe
Enter the number of processes: 3
Enter burst time for Process 1: 10
Enter arrival time for Process 1: 0
Enter priority for Process 1 (higher number = lower priority): 2
Enter burst time for Process 2: 5
Enter arrival time for Process 2: 2
Enter priority for Process 2 (higher number = lower priority): 1
Enter burst time for Process 3: 8
Enter arrival time for Process 3: 4
Enter priority for Process 3 (higher number = lower priority): 3

Select the scheduling algorithm:
1. Round Robin
2. Preemptive Priority Scheduling
1
Enter time quantum for Round Robin: 4

Process Burst Time      Arrival Time    Priority        Waiting Time    Turnaround Time
1               10              0               2               13              23
2               5               2               1               10              15
3               8               4               3               9               17
```

Priority Scheduling

```
PS D:\Arjun Mijar (160)> .\preemCPUScheduling.exe
Enter the number of processes: 3
Enter burst time for Process 1: 10
Enter arrival time for Process 1: 0
Enter priority for Process 1 (higher number = lower priority): 2
Enter burst time for Process 2: 5
Enter arrival time for Process 2: 2
Enter priority for Process 2 (higher number = lower priority): 1
Enter burst time for Process 3: 8
Enter arrival time for Process 3: 4
Enter priority for Process 3 (higher number = lower priority): 3

Select the scheduling algorithm:
1. Round Robin
2. Preemptive Priority Scheduling
2

Process Burst Time      Arrival Time    Priority        Waiting Time    Turnaround Time
1               10              0               2               5               15
2               5               2               1               0               5
3               8               4               3               11              19
```

EXPERIMENT NO. 7

DEADLOCK AVOIDANCE

AIM: Program to Simulate Bankers Algorithm for Dead Lock Avoidance.

THEORY:

The **Banker's Algorithm**, developed by Edsger Dijkstra, is a well-known algorithm used to avoid deadlock in operating systems. It is primarily used in a multi-tasking environment where processes request resources from the operating system. The algorithm ensures that the system remains in a safe state by granting resources only if the system can still allocate resources to all processes without leading to a deadlock.

The Banker's Algorithm is named so because it simulates a banking system where customers (processes) request cash (resources). Before granting a request, the banker (operating system) checks whether it can still satisfy all future requests without running out of resources and getting into a situation where customers cannot be satisfied (i.e., a deadlock).

***Key Concepts**:*

1. **Safe State**: A state is safe if the system can allocate resources to each process in some order without leading to a deadlock.
2. **Unsafe State**: If, after allocating resources to a process, the system cannot guarantee that all processes can finish, the state is considered unsafe.
3. **Request**: Processes make resource requests that must be fulfilled if the system is to remain in a safe state.

***Components**:*

- **Max Matrix**: Maximum number of resources each process may need.
- **Allocation Matrix**: Resources currently allocated to each process.
- **Need Matrix**: Resources each process still needs to complete its execution.
    - $Need = Max - Allocation$
- **Available Resources**: Total resources available for allocation.
- **Request Vector**: The resources requested by a process at a given time.

ALGORITHM:

1. **Initialization**:
    - Define the Max, Allocation, and Available matrices.
    - Calculate the Need matrix using the formula:
        - $Need[i][j] = Max[i][j] - Allocation[i][j]$
2. **Resource Request**:
    - When a process P requests resources, check if:
        - The request is less than or equal to the need of the process.
        - The request is less than or equal to the available resources.

3. **Pretend Allocation**:
    - Temporarily allocate the requested resources to the process and reduce the available resources.
    - Update the Allocation and Need matrices.
4. **Safety Check**:
    - Check if the system is in a safe state by determining if there's a sequence in which all processes can be executed to completion.
    - A process can proceed if its **Need** is less than or equal to the **Available** resources. If it finishes, release its allocated resources, and add them back to the **Available** pool.
5. **Grant or Deny Request**:
    - If the system is in a **safe state**, grant the request.
    - If not, **deny the request** and revert the allocation.
6. **Repeat**:
    - Continue this process until all processes have finished, ensuring that the system never enters an unsafe state.

SOURCE CODE:

```
#include <stdio.h>

#define MAX_PROCESSES 10

#define MAX_RESOURCES 10

int allocation[MAX_PROCESSES][MAX_RESOURCES],
max[MAX_PROCESSES][MAX_RESOURCES], need[MAX_PROCESSES][MAX_RESOURCES],
available[MAX_RESOURCES];

int n, m; // n = number of processes, m = number of resources

void calculateNeed() {

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

      for (int j = 0; j < m; j++) {

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

      }

   }

}
```

```
int checkSafeState(int work[], int finish[], int safeSequence[]) {
    int count = 0;
   while (count < n) {
        int found = 0;
        for (int i = 0; i < n; i++) {
            if (finish[i] == 0) {
                int j;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > work[j])
                        break;
                }

                if (j == m) {
                    for (int k = 0; k < m; k++)
                        work[k] += allocation[i][k];
                    safeSequence[count++] = i;
                    finish[i] = 1;
                    found = 1;
                }
            }
        }
        if (found == 0) {
            printf("The system is in an unsafe state!\n");
            return 0;
```

```
        }
    }

    printf("The system is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", safeSequence[i]);
    }
    printf("\n");
    return 1;
}
int bankerAlgorithm() {
    int work[MAX_RESOURCES];
    int finish[MAX_PROCESSES] = {0};
    int safeSequence[MAX_PROCESSES];
    for (int i = 0; i < m; i++) {
        work[i] = available[i];
    }
    return checkSafeState(work, finish, safeSequence);
}
void requestResources(int processID) {
    int request[MAX_RESOURCES];
    printf("Enter the request for resources by process %d: \n", processID);
    for (int i = 0; i < m; i++) {
        printf("Resource %d: ", i);
```

```
    scanf("%d", &request[i]);
}
for (int i = 0; i < m; i++) {
    if (request[i] > need[processID][i]) {
        printf("Error: Process has exceeded its maximum claim.\n");
        return;
    }
}
for (int i = 0; i < m; i++) {
    if (request[i] > available[i]) {
        printf("Resources are not available right now. Process %d must wait.\n", processID);
        return;
    }
}
for (int i = 0; i < m; i++) {
    available[i] -= request[i];
    allocation[processID][i] += request[i];
    need[processID][i] -= request[i];
}
if (bankerAlgorithm() == 0) {
    for (int i = 0; i < m; i++) {
        available[i] += request[i];
        allocation[processID][i] -= request[i];
        need[processID][i] += request[i];
```

```
        }

        printf("Request cannot be granted as it would lead to an unsafe state. Process %d must wait.\n",
processID);

    } else {

        printf("Request granted to process %d.\n", processID);

    }

}

int main() {

    printf("Enter the number of processes: ");

    scanf("%d", &n);



    printf("Enter the number of resource types: ");

    scanf("%d", &m);



    printf("Enter the Allocation Matrix:\n");

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

        for (int j = 0; j < m; j++) {

            scanf("%d", &allocation[i][j]);

        }

    }

    printf("Enter the Max Matrix:\n");

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

        for (int j = 0; j < m; j++) {

            scanf("%d", &max[i][j]);

        }
```

```
    }
    printf("Enter the Available Resources:\n");
    for (int i = 0; i < m; i++) {
        scanf("%d", &available[i]);
    }
    calculateNeed();
    if (bankerAlgorithm()) {
        int processID;
        printf("Do you want to request resources for any process? Enter process ID (-1 to exit): ");
        scanf("%d", &processID);
        while (processID != -1) {
            requestResources(processID);
            printf("Enter process ID to request resources (-1 to exit): ");
            scanf("%d", &processID);
        }
    }
    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\bankerAlgorithm.exe
Enter the number of processes: 5
Enter the number of resource types: 3
Enter the Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Available Resources:
3 3 2
The system is in a safe state.
Safe sequence is: 1 3 4 0 2
Do you want to request resources for any process? Enter process ID (-1 to exit): 1
Enter the request for resources by process 1:
Resource 0: 1
Resource 1: 0
Resource 2: 2
The system is in a safe state.
Safe sequence is: 1 3 4 0 2
Request granted to process 1.
```

EXPERIMENT NO. 8

DEADLOCK PREVENTION

AIM: Program to Simulate Bankers Algorithm for Dead Lock Prevention.

THEORY:

The Banker's Algorithm for deadlock prevention is a resource allocation and deadlock avoidance algorithm that tests for the safety of resource allocation to a process. It ensures that the system will remain in a safe state by only allowing the allocation of resources if it can guarantee that all processes can be completed without causing a deadlock.

In contrast to deadlock detection (which identifies deadlock after it occurs), the Banker's Algorithm works proactively to prevent deadlock from occurring in the first place. The algorithm simulates resource allocation for each process request and checks if granting the requested resources will leave the system in a safe state.

The algorithm is referred to as the **Banker's Algorithm** because it mimics a banker deciding whether or not to grant a loan (resources) to a customer (process) while ensuring that there are sufficient resources for all customers to be eventually satisfied.

***Key Terms**:*

- **Safe State**: A state is safe if there exists a sequence of processes where each process can be allocated resources, complete its execution, and release the resources without causing a deadlock.
- **Unsafe State**: If the system allocates resources to a process, but cannot guarantee that all processes can finish their execution without leading to a deadlock, the state is considered unsafe.
- **Max Matrix**: The maximum number of each type of resource that each process may request.
- **Allocation Matrix**: The number of resources of each type currently allocated to each process.
- **Need Matrix**: The resources each process still needs to complete its task.
    - Need=Max−Allocation
- **Available Resources**: The number of available instances of each resource type.

***Assumptions**:*

- The number of instances of each resource type is fixed.
- The processes are well-behaved and will not exceed their declared maximum resource needs.

ALGORITHM:

1. **Initialization**:
    - Define the **Max**, **Allocation**, **Need**, and **Available** matrices.
    - Initialize the available resources for each type of resource.
2. **Request Resources**:
    - When a process makes a resource request, check if:
        - The requested resources are less than or equal to the **Need** for that process.
        - The requested resources are less than or equal to the **Available** resources in the system.
3. **Pretend Allocation**:
    - Temporarily allocate the requested resources to the process:
        - Subtract the requested resources from the **Available** vector.
        - Add the requested resources to the **Allocation** matrix for that process.
        - Subtract the requested resources from the **Need** matrix for that process.
4. **Check Safe State**:
    - Perform the **Safety Algorithm** to check if the system is still in a **safe state**:
        1. **Work** and **Finish Arrays**: Initialize Work to be a copy of the **Available** resources and Finish[i] (for each process) as false.
        2. Find a process P that is not yet finished (Finish[i] == false) and whose **Need** is less than or equal to the available **Work** resources.
        3. If such a process is found, assume that process finishes:
            - Add its currently allocated resources to the **Work** array.
            - Set Finish[i] to true for that process.
        4. Repeat steps 2 and 3 until all processes are finished or no such process can be found.
    - If all processes can be finished in some order, the system is in a **safe state**.
5. **Grant or Deny Request**:
    - If the system remains in a **safe state**, grant the resource request to the process.
    - If the system enters an **unsafe state**, deny the request, and roll back the temporary allocation.
6. **Reclaim Resources**:
    - After each process finishes, it releases the allocated resources, which are added back to the **Available** vector.

SOURCE CODE:

```
#include <stdio.h>


#define MAX_PROCESSES 10

#define MAX_RESOURCES 10


int allocation[MAX_PROCESSES][MAX_RESOURCES],
max[MAX_PROCESSES][MAX_RESOURCES], need[MAX_PROCESSES][MAX_RESOURCES],
available[MAX_RESOURCES];

int n, m; // n = number of processes, m = number of resources


void calculateNeed() {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         need[i][j] = max[i][j] - allocation[i][j];
      }
   }
}


int isSafe() {
   int work[MAX_RESOURCES], finish[MAX_PROCESSES] = {0},
safeSequence[MAX_PROCESSES];
   for (int i = 0; i < m; i++) {
      work[i] = available[i];
   }


   int count = 0;
   while (count < n) {
      int found = 0;
      for (int i = 0; i < n; i++) {
         if (!finish[i]) {
```

```
            int j;
            for (j = 0; j < m; j++) {
                if (need[i][j] > work[j]) {
                    break;
                }
            }

            if (j == m) {
                for (int k = 0; k < m; k++) {
                    work[k] += allocation[i][k];
                }
                safeSequence[count++] = i;
                finish[i] = 1;
                found = 1;
            }
        }
    }

    if (!found) {
        printf("System is in an unsafe state!\n");
        return 0;
    }
}

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

```
}

void requestResources(int processID) {
    int request[MAX_RESOURCES];
    printf("Enter the request for resources by process %d: \n", processID);
    for (int i = 0; i < m; i++) {
        printf("Resource %d: ", i);
        scanf("%d", &request[i]);
    }

    for (int i = 0; i < m; i++) {
        if (request[i] > need[processID][i]) {
            printf("Error: Request exceeds the process's maximum claim.\n");
            return;
        }
    }

    for (int i = 0; i < m; i++) {
        if (request[i] > available[i]) {
            printf("Resources are not available right now. Process %d must wait.\n", processID);
            return;
        }
    }

    for (int i = 0; i < m; i++) {
        available[i] -= request[i];
        allocation[processID][i] += request[i];
        need[processID][i] -= request[i];
    }
```

```
    if (!isSafe()) {
        for (int i = 0; i < m; i++) {
            available[i] += request[i];
            allocation[processID][i] -= request[i];
            need[processID][i] += request[i];
        }
        printf("Request cannot be granted as it would lead to an unsafe state. Process %d must wait.\n",
processID);
    } else {
        printf("Request granted to process %d.\n", processID);
    }
}

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

    printf("Enter the number of resource types: ");
    scanf("%d", &m);

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

    printf("Enter the Max Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
```

```
            scanf("%d", &max[i][j]);
        }
    }

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

    calculateNeed();

    if (isSafe()) {
        int processID;
        printf("Do you want to request resources for any process? Enter process ID (-1 to exit): ");
        scanf("%d", &processID);

        while (processID != -1) {
            requestResources(processID);
            printf("Enter process ID to request resources (-1 to exit): ");
            scanf("%d", &processID);
        }
    }

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\bankerPreventAlgorithm.exe
Enter the number of processes: 5
Enter the number of resource types: 3
Enter the Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Available Resources:
3 3 2
System is in a safe state. Safe sequence is: 1 3 4 0 2
Do you want to request resources for any process? Enter process ID (-1 to exit): 1
Enter the request for resources by process 1:
Resource 0: 1
Resource 1: 0
Resource 2: 2
System is in a safe state. Safe sequence is: 1 3 4 0 2
Request granted to process 1.
```

EXPERIMENT NO. 9

MEMORY MANAGEMENT TECHNIQUES

AIM: Program to Simulate the MVT and MFT memory management techniques.

THEORY:

Memory management is a critical function of an operating system, involving the allocation and management of memory resources. Two classic memory management techniques are **MVT (Multiprogramming with Variable Tasks)** and **MFT (Multiprogramming with Fixed Tasks)**.

1. **Multiprogramming with Fixed Tasks (MFT)**

MFT is a memory management technique where the memory is divided into fixed-size partitions. Each partition can hold only one process at a time, and processes are assigned to partitions based on their size.

***Key Concepts**:*

Fixed Partitioning: Memory is divided into a fixed number of partitions.

Internal Fragmentation: If a process occupies less memory than the partition size, the remaining space in the partition is wasted (internal fragmentation).

Process Allocation: Processes are allocated to a partition that is large enough to fit them. If no partition is available, the process must wait.

Advantages:

- Simple to implement and manage.
- Easy to allocate memory by simply assigning processes to fixed partitions.

Disadvantages:

- Internal fragmentation due to the fixed partition sizes.
- Not efficient for systems with processes of varying memory requirements.

ALGORITHMS:

1. **Initialization**:
    - Define fixed-size memory partitions.
    - Keep track of the size of each partition and whether it is free or occupied.
2. **Process Arrival**:
    - When a process arrives, check for a free partition large enough to accommodate the process.
    - If a partition is found:
        - Assign the process to the partition.
        - Mark the partition as occupied.
    - If no partition is found, the process must wait.

3. **Process Execution**:
    - Execute the process until completion.
    - Once completed, free the partition and mark it as available.
4. **Repeat**:
    - Continue until all processes are allocated and executed.

2. **Multiprogramming with Variable Tasks (MVT)**

MVT is a memory management technique where memory is allocated dynamically, based on the size of the process. This allows for more efficient use of memory as processes of varying sizes can be accommodated without wasting memory.

***Key Concepts**:*

Variable Partitioning: Memory is divided into variable-sized partitions, depending on the process's memory requirement.

External Fragmentation: Since partitions are variable, memory blocks may become scattered, causing unused memory gaps (external fragmentation).

Process Allocation: When a process arrives, it is allocated a block of memory just large enough to hold it. The system keeps track of free memory blocks and assigns processes to available blocks.

Advantages:

- More efficient memory usage compared to MFT as memory is allocated exactly as required.
- Reduces internal fragmentation since memory is dynamically allocated.

Disadvantages:

- External fragmentation, where memory becomes scattered and fragmented.
- More complex memory management since free memory blocks need to be continuously tracked and coalesced.

ALGORITHM

1. **Initialization**:
    - Define the total available memory.
    - Maintain a free memory list that tracks available memory blocks.
2. **Process Arrival**:
    - When a process arrives, check the free memory list for a block large enough to accommodate the process.
    - If a suitable block is found:
        - Allocate memory to the process, and subtract the allocated memory from the free list.
        - Keep track of the memory blocks that are currently in use.
    - If no suitable block is found, the process must wait.
3. **Process Execution**:

- Execute the process until completion.
- Once completed, release the memory block and return it to the free memory list.

4. **Compaction (optional)**:
   - If external fragmentation becomes excessive, perform memory compaction, which merges adjacent free blocks into a single block.

SOURCE CODE:

For MFT memory management techniques

```
#include <stdio.h>


int main() {
   int partitionSize[10], processSize[10], allocated[10], numPartitions, numProcesses;
   int internalFragmentation = 0;


   // Get number of partitions
   printf("Enter the number of memory partitions: ");
   scanf("%d", &numPartitions);


   // Get partition sizes
   printf("Enter the size of each partition:\n");
   for (int i = 0; i < numPartitions; i++) {
      printf("Partition %d size: ", i + 1);
      scanf("%d", &partitionSize[i]);
      allocated[i] = 0;  // No process is allocated initially
   }


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


   // Get process sizes
```

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

    // Allocating processes to partitions
    for (int i = 0; i < numProcesses; i++) {
        int allocatedFlag = 0;
        for (int j = 0; j < numPartitions; j++) {
            if (!allocated[j] && partitionSize[j] >= processSize[i]) {
                allocated[j] = 1;
                internalFragmentation += (partitionSize[j] - processSize[i]);
                printf("Process %d of size %d allocated to partition %d of size %d\n", i + 1, processSize[i], j +
1, partitionSize[j]);
                allocatedFlag = 1;
                break;
            }
        }
        if (!allocatedFlag) {
            printf("Process %d of size %d cannot be allocated to any partition.\n", i + 1, processSize[i]);
        }
    }

    printf("Total Internal Fragmentation: %d\n", internalFragmentation);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\mft.exe
Enter the number of memory partitions: 3
Enter the size of each partition:
Partition 1 size: 100
Partition 2 size: 200
Partition 3 size: 300
Enter the number of processes: 4
Enter the size of each process:
Process 1 size: 90
Process 2 size: 210
Process 3 size: 150
Process 4 size: 290
Process 1 of size 90 allocated to partition 1 of size 100
Process 2 of size 210 allocated to partition 3 of size 300
Process 3 of size 150 allocated to partition 2 of size 200
Process 4 of size 290 cannot be allocated to any partition.
Total Internal Fragmentation: 150
```

For MVT memory management techniques

```
#include <stdio.h>

int main() {
    int totalMemory, processSize[10], numProcesses, memoryAllocated = 0;

    // Get total memory available
    printf("Enter the total available memory: ");
    scanf("%d", &totalMemory);

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

    // Get process sizes
    printf("Enter the size of each process:\n");
```

```
    for (int i = 0; i < numProcesses; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &processSize[i]);
    }

    // Allocating processes to memory
    for (int i = 0; i < numProcesses; i++) {
        if (memoryAllocated + processSize[i] <= totalMemory) {
            memoryAllocated += processSize[i];
            printf("Process %d of size %d allocated. Memory used: %d/%d\n", i + 1, processSize[i],
memoryAllocated, totalMemory);
        } else {
            printf("Process %d of size %d cannot be allocated due to insufficient memory.\n", i + 1,
processSize[i]);
        }
    }

    printf("Total memory allocated: %d\n", memoryAllocated);
    printf("Total external fragmentation (remaining memory): %d\n", totalMemory - memoryAllocated);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\mvt.exe
Enter the total available memory: 600
Enter the number of processes: 4
Enter the size of each process:
Process 1 size: 100
Process 2 size: 200
Process 3 size: 150
Process 4 size: 250
Process 1 of size 100 allocated. Memory used: 100/600
Process 2 of size 200 allocated. Memory used: 300/600
Process 3 of size 150 allocated. Memory used: 450/600
Process 4 of size 250 cannot be allocated due to insufficient memory.
Total memory allocated: 450
Total external fragmentation (remaining memory): 150
```

EXPERIMENT NO. 10

MEMORY MANAGEMENT TECHNIQUES

AIM: Simulate the paging technique of memory management.

THEORY:

Paging is a memory management technique that eliminates the need for contiguous memory allocation, preventing the problem of external fragmentation and varying memory size requests. In paging, the entire memory is divided into fixed-size blocks called **pages** (in logical memory) and **frames** (in physical memory). The process is divided into pages of the same size, and these pages can be stored in any frame in the physical memory.

A **page table** is used to map the logical pages to the physical frames, keeping track of where each page is located in physical memory. The operating system uses the page table to translate logical addresses to physical addresses.

Key Concepts:

- **Pages**: Fixed-size blocks of logical memory.
- **Frames**: Fixed-size blocks of physical memory.
- **Page Table**: A table used to map the logical page number to a physical frame number.
- **Logical Address**: Address generated by the CPU, which is divided into a page number and an offset.
- **Physical Address**: Actual address in memory, formed by combining the frame number with the offset.

Advantages:

1. **No External Fragmentation**: Since processes are allocated to available frames, there's no issue with free memory being scattered.
2. **Efficient Memory Use**: Memory is used more efficiently as processes can occupy non-contiguous blocks of memory.
3. **Supports Virtual Memory**: Paging makes it easier to implement virtual memory, allowing the process to use more memory than is physically available.

Disadvantages:

1. **Page Table Overhead**: Each process needs its own page table, which requires additional memory and processing.
2. **Internal Fragmentation**: Since pages are fixed size, if a page isn’t fully used, the remaining space within the page is wasted.

ALGORITHM:

1. **Initialization**:
    - Divide the logical memory into equal-sized pages.
    - Divide the physical memory into equal-sized frames.
    - Initialize a page table to map each logical page to a physical frame.
2. **Process Arrival**:
    - When a process arrives, calculate the number of pages required to hold the entire process.
    - Allocate available frames to each page of the process.
    - Update the page table with the mapping of pages to frames.
3. **Address Translation**:
    - A logical address generated by the CPU is divided into two parts: the page number and the offset.
    - Use the page number to look up the corresponding frame number in the page table.
    - The physical address is then calculated by combining the frame number with the offset.
4. **Access Memory**:
    - Once the physical address is determined, the data can be fetched from memory.
5. **Repeat**:
    - Continue translating logical addresses to physical addresses as the process executes.

SOURCE CODE:

```
#include <stdio.h>

#include <stdlib.h>


#define MAX 100


int main() {

   int pageTable[MAX], pageSize, numPages, logicalAddress, pageNumber, offset, frameNumber, physicalAddress;


   // Input page size and number of pages

   printf("Enter the page size (in words): ");

   scanf("%d", &pageSize);


   printf("Enter the number of pages: ");

   scanf("%d", &numPages);
```

```
// Populate page table with frame numbers for each page
printf("Enter the frame number for each page:\n");
for (int i = 0; i < numPages; i++) {
    printf("Page %d: ", i);
    scanf("%d", &pageTable[i]);
}

// Input the logical address
printf("\nEnter a logical address (in words): ");
scanf("%d", &logicalAddress);

// Calculate the page number and offset
pageNumber = logicalAddress / pageSize;
offset = logicalAddress % pageSize;

// Check if the page number is valid
if (pageNumber >= numPages) {
    printf("Error: Invalid logical address!\n");
    return 1;
}

// Get the frame number from the page table
frameNumber = pageTable[pageNumber];

// Calculate the physical address
physicalAddress = frameNumber * pageSize + offset;

// Output the results
printf("\nLogical Address: %d\n", logicalAddress);
printf("Page Number: %d\n", pageNumber);
```

```
    printf("Offset: %d\n", offset);
    printf("Frame Number: %d\n", frameNumber);
    printf("Physical Address: %d\n", physicalAddress);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\pagingMemoryManage.exe
Enter the page size (in words): 100
Enter the number of pages: 4
Enter the frame number for each page:
Page 0: 3
Page 1: 1
Page 2: 4
Page 3: 2

Enter a logical address (in words): 230

Logical Address: 230
Page Number: 2
Offset: 30
Frame Number: 4
Physical Address: 430
```

EXPERIMENT NO. 11

MEMORY ALLOCATION TECHNIQUES

AIM: Write a C program to simulate the FIRST-FIT contiguous memory allocation technique.

THEORY:

First-fit is one of the simplest memory allocation techniques in contiguous memory allocation. The memory is divided into different fixed or variable-sized blocks, and each process is allocated the first available block that is large enough to fulfill its memory requirement. This technique is a form of dynamic memory allocation where the size of blocks can vary, and processes are placed into these blocks based on their size requirements.

The main idea behind First-Fit is to allocate the first available block of memory that is large enough to accommodate the process. Once a block is allocated to a process, the remaining memory (if any) becomes available for future allocations.

Advantages:

1. **Simple and Fast**: The First-Fit technique is quick to implement since it simply traverses the list of free memory blocks and allocates the first one large enough to fit the process.
2. **Efficient Memory Usage**: It can lead to reasonably efficient memory utilization as processes are allocated to the first free block available.

Disadvantages:

1. **Fragmentation**: This technique can lead to **external fragmentation**, where small unusable memory spaces are left between allocated blocks.
2. **Wasted Space**: Large free blocks may be partially wasted when only a small part of them is allocated to a process.

ALGORITHM:

1. **Initialization**:
    - Define a list of available memory blocks with their sizes.
    - Define a list of processes with their memory requirements.
2. **Memory Allocation**:
    - For each process, start checking from the beginning of the memory block list.
    - Allocate the first block that is large enough to accommodate the process.
    - If the block is larger than the process, split the block into allocated memory for the process and the remaining memory as a new block.
    - Mark the allocated block as "used."
3. **Check for Memory Availability**:
    - If no block is large enough for a process, display a message indicating that the process could not be allocated memory.
4. **Repeat**:
    - Continue this process for all the processes.

5. **Output**:
    - Display the process, memory block assigned, and remaining free blocks.

SOURCE CODE:

```
#include <stdio.h>


void firstFit(int blockSize[], int m, int processSize[], int n) {
   // Stores block id of the block allocated to a process
   int allocation[n];

   // Initially no block is assigned to any process
   for (int i = 0; i < n; i++) {
      allocation[i] = -1;
   }

   // Pick each process and find the first block that can fit it
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         // If block j can accommodate process i
         if (blockSize[j] >= processSize[i]) {
            // Allocate block j to process i
            allocation[i] = j;

            // Reduce available memory in this block
            blockSize[j] -= processSize[i];

            break; // Move to the next process
         }
      }
   }
```

```
    // Display allocation details
    printf("\nProcess 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);  // Display block number (1-based index)
        } else {
            printf("Not Allocated\n");
        }
    }
}

int main() {
    int m, n;

    // Input number of blocks
    printf("Enter the number of memory blocks: ");
    scanf("%d", &m);
    int blockSize[m];

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

    // Input number of processes
    printf("\nEnter the number of processes: ");
```

```
    scanf("%d", &n);
    int processSize[n];

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

    // Call first fit allocation function
    firstFit(blockSize, m, processSize, n);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\firstFitConMemAllocation.exe
Enter the number of memory blocks: 5
Enter the size of each memory block:
Block 1: 200
Block 2: 500
Block 3: 100
Block 4: 300
Block 5: 600

Enter the number of processes: 4
Enter the size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 236

Process No.     Process Size    Block No.
 1              212             2
 2              417             5
 3              112             1
 4              236             2
```

EXPERIMENT NO. 12

MEMORY ALLOCATION TECHNIQUES

AIM: Write a C program to simulate the BEST FIT contiguous memory allocation technique.

THEORY:

Best-Fit is a memory allocation technique that aims to minimize wasted space by allocating the smallest available block that is large enough to accommodate the process. In contrast to the First-Fit strategy, which allocates the first block that fits, the Best-Fit technique searches through all available blocks and selects the one that leaves the least amount of wasted space. This approach helps to reduce external fragmentation and maximize the use of available memory.

Advantages:

1. **Minimized Wasted Space**: By selecting the smallest suitable block, Best-Fit reduces the amount of memory left unused after allocation.
2. **Efficient Memory Usage**: It often results in better memory utilization compared to First-Fit, especially in systems with varying sizes of memory blocks.

Disadvantages:

1. **Complexity and Overhead**: The Best-Fit algorithm requires searching through all available blocks, which can be time-consuming and computationally expensive.
2. **Increased Fragmentation**: While it minimizes wasted space, it can lead to increased fragmentation over time as small free spaces accumulate.

ALGORITHM:

1. **Initialization**:
    - Define a list of available memory blocks with their sizes.
    - Define a list of processes with their memory requirements.
2. **Memory Allocation**:
    - For each process, search through all available memory blocks to find the smallest block that is large enough to fit the process.
    - Allocate the process to the selected block.
    - If the block is larger than the process size, split the block into allocated memory for the process and the remaining memory as a new free block.
3. **Check for Memory Availability**:
    - If no block is found that can accommodate the process, display a message indicating that the process could not be allocated memory.
4. **Repeat**:
    - Continue this process for all the processes.
5. **Output**:
    - Display which memory block is assigned to which process.
    - Display the remaining free blocks and their sizes.

SOURCE CODE:

```
#include <stdio.h>

#define MAX 100

// Function to allocate memory using Best-Fit algorithm
void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    int i, j;

    // Initialize all allocations as -1 (indicating no allocation)
    for (i = 0; i < n; i++) {
        allocation[i] = -1;
    }

    // Traverse each process
    for (i = 0; i < n; i++) {
        // Find the best fit block for this process
        int bestIdx = -1;
        for (j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx]) {
                    bestIdx = j;
                }
            }
        }

        // If a suitable block is found
        if (bestIdx != -1) {
            // Allocate the block to the process
```

```
            allocation[i] = bestIdx;
            // Reduce the size of the available block
            blockSize[bestIdx] -= processSize[i];
        }
    }

    // Print the allocation results
    printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");
    for (i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, processSize[i], allocation[i] + 1, blockSize[allocation[i]]);
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i + 1, processSize[i]);
        }
    }
}

int main() {
    int m, n;

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

```
    // Input number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int processSize[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]);
    }

    // Perform Best-Fit memory allocation
    bestFit(blockSize, m, processSize, n);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\bestFitConMemAllocation.exe
Enter the number of memory blocks: 5
Enter the size of each memory block:
Block 1: 200
Block 2: 500
Block 3: 100
Block 4: 300
Block 5: 600
Enter the number of processes: 4
Enter the size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

Process No.     Process Size    Block No.       Block Size
1               212             4               88
2               417             2               83
3               112             1               88
4               426             5               174
```

EXPERIMENT NO. 13

MEMORY ALLOCATION TECHNIQUES

AIM: Write a C program to simulate the WORST-FIT contiguous memory allocation technique.

THEORY:

Worst-fit is a memory allocation technique designed to minimize external fragmentation by allocating memory to the largest available block. In this method, when a process requests memory, it is allocated to the largest free block available. The rationale behind Worst-Fit is to leave large blocks available for future allocations, assuming that the largest block will be less likely to become fragmented compared to smaller blocks.

Advantages:

1. **Minimizes Fragmentation**: By allocating the largest available block, Worst-Fit helps to ensure that larger blocks remain available for larger processes.
2. **Simple Implementation**: The algorithm is straightforward to implement as it only requires searching for the largest block.

Disadvantages:

1. **Inefficient Use of Memory**: The remaining free block might become too small for future allocations, potentially leading to inefficient use of memory.
2. **Increased Fragmentation**: Over time, Worst-Fit can lead to increased fragmentation and inefficient use of memory due to the leftover small blocks.

ALGORITHM

1. **Initialization**:
    - Define a list of available memory blocks with their sizes.
    - Define a list of processes with their memory requirements.
2. **Memory Allocation**:
    - For each process, search through all available memory blocks to find the largest block that is large enough to accommodate the process.
    - Allocate the process to this largest block.
    - If the block is larger than the process size, split the block into allocated memory for the process and the remaining memory as a new free block.
3. **Check for Memory Availability**:
    - If no block is found that can accommodate the process, display a message indicating that the process could not be allocated memory.
4. **Repeat**:
    - Continue this process for all the processes.
5. **Output**:
    - Display which memory block is assigned to which process.
    - Display the remaining free blocks and their sizes.

SOURCE CODE:

```
#include <stdio.h>

#define MAX 100

// Function to allocate memory using Worst-Fit algorithm
void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    int i, j;

    // Initialize all allocations as -1 (indicating no allocation)
    for (i = 0; i < n; i++) {
        allocation[i] = -1;
    }

    // Traverse each process
    for (i = 0; i < n; i++) {
        // Find the worst fit block for this process
        int worstIdx = -1;
        for (j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx]) {
                    worstIdx = j;
                }
            }
        }

        // If a suitable block is found
        if (worstIdx != -1) {
            // Allocate the block to the process
```

```
            allocation[i] = worstIdx;
            // Reduce the size of the available block
            blockSize[worstIdx] -= processSize[i];
        }
    }

    // Print the allocation results
    printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");
    for (i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, processSize[i], allocation[i] + 1, blockSize[allocation[i]]);
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i + 1, processSize[i]);
        }
    }
}

int main() {
    int m, n;

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

```
    // Input number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int processSize[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]);
    }

    // Perform Worst-Fit memory allocation
    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\worstFitConMemAllocation.exe
Enter the number of memory blocks: 5
Enter the size of each memory block:
Block 1: 200
Block 2: 100
Block 3: 500
Block 4: 300
Block 5: 600
Enter the number of processes: 4
Enter the size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

Process No.     Process Size    Block No.       Block Size
1               212             5               276
2               417             3               83
3               112             5               276
4               426             Not Allocated
```

EXPERIMENT NO. 14

PAGE REPLACEMENT ALGORITHM

AIM: Write a C program to simulate the FIFO page replacement algorithm

THEORY:

FIFO (First-In-First-Out) is a page replacement algorithm used in operating systems to manage memory pages. In FIFO, the page that has been in memory the longest is replaced first when a new page needs to be loaded into a full memory. This method uses a queue to keep track of the order in which pages are loaded into memory.

Advantages:

1. **Simplicity**: FIFO is straightforward to implement and understand.
2. **Fairness**: Each page gets a chance to stay in memory for a fixed duration before being replaced.

Disadvantages:

1. **Belady's Anomaly**: FIFO can lead to worse performance in some cases where increasing the number of page frames results in more page faults.
2. **Not Optimal**: It does not always choose the best page to replace, potentially leading to suboptimal performance.

ALGORITHM:

1. **Initialization**:
    - Define the number of page frames (memory slots) and the page reference string (sequence of pages to be accessed).
    - Initialize a queue to keep track of the order of pages in memory.
2. **Page Replacement**:
    - Traverse the page reference string.
    - For each page:
        - **If the page is already in memory**: Continue to the next page.
        - **If the page is not in memory**:
            - **If there is an empty frame**: Load the page into the empty frame.
            - **If all frames are full**: Remove the oldest page from memory (the one at the front of the queue) and replace it with the new page. Update the queue accordingly.
3. **Count Page Faults**:
    - Increment the page fault count whenever a page is loaded into memory.
4. **Output**:
    - Display the page reference string, the memory content after each page reference, and the total number of page faults.

SOURCE CODE:

```
#include <stdio.h>

#define MAX_FRAMES 10
#define MAX_PAGES 100

// Function to simulate FIFO page replacement algorithm
void fifoPageReplacement(int pages[], int numPages, int numFrames) {
    int frame[numFrames];
    int pageFaults = 0;
    int pageIndex = 0;

    // Initialize frames to -1 (indicating empty frames)
    for (int i = 0; i < numFrames; i++) {
        frame[i] = -1;
    }

    // FIFO queue initialization
    int queue[MAX_FRAMES];
    int front = 0, rear = 0;

    printf("Page Reference String: ");
    for (int i = 0; i < numPages; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    printf("Memory Content after each reference:\n");

    for (int i = 0; i < numPages; i++) {
```

```
int page = pages[i];
int pageFound = 0;

// Check if page is already in memory
for (int j = 0; j < numFrames; j++) {
    if (frame[j] == page) {
        pageFound = 1;
        break;
    }
}

// Page not found in memory, perform page replacement
if (!pageFound) {
    // If memory is full, remove the oldest page
    if (rear == numFrames) {
        frame[front] = page;
        front = (front + 1) % numFrames;
    } else {
        frame[rear] = page;
        rear++;
    }
    pageFaults++;
}

// Print memory content
printf("After page %d: ", page);
for (int j = 0; j < numFrames; j++) {
    if (frame[j] != -1) {
        printf("%d ", frame[j]);
    }
```

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

    // Print total page faults
    printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
    int numPages, numFrames;
    int pages[MAX_PAGES];

    // Input number of pages
    printf("Enter the number of pages: ");
    scanf("%d", &numPages);

    // Input page reference string
    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

    // Input number of frames
    printf("Enter the number of frames: ");
    scanf("%d", &numFrames);

    // Simulate FIFO page replacement
    fifoPageReplacement(pages, numPages, numFrames);
```

```
    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\fifoPageRepAlgorithm.exe
Enter the number of pages: 12
Enter the page reference string:
Page 1: 7
Page 2: 0
Page 3: 1
Page 4: 2
Page 5: 0
Page 6: 3
Page 7: 0
Page 8: 4
Page 9: 2
Page 10: 0
Page 11: 3
Page 12: 0
Enter the number of frames: 3
Page Reference String: 7 0 1 2 0 3 0 4 2 0 3 0
Memory Content after each reference:
After page 7: 7
After page 0: 7 0
After page 1: 7 0 1
After page 2: 2 0 1
After page 0: 2 0 1
After page 3: 2 3 1
After page 0: 2 3 0
After page 4: 4 3 0
After page 2: 4 2 0
After page 0: 4 2 0
After page 3: 4 2 3
After page 0: 0 2 3
Total Page Faults: 10
```

EXPERIMENT NO. 15

PAGE REPLACEMENT ALGORITHM

AIM: Write a C program to simulate the LRU page replacement algorithm.

THEORY:

LRU (Least Recently Used) is a page replacement algorithm used in operating systems to manage memory pages efficiently. LRU keeps track of the order in which pages are accessed and replaces the page that has not been used for the longest period of time when a new page needs to be loaded into memory.

Advantages:

1. **Efficient**: LRU is more efficient than FIFO because it replaces pages based on recent usage, which often results in fewer page faults.
2. **Minimizes Page Faults**: By keeping the most recently used pages in memory, LRU can effectively minimize the number of page faults in many scenarios.

Disadvantages:

1. **Implementation Complexity**: Implementing LRU can be more complex compared to simpler algorithms like FIFO.
2. **Overhead**: Maintaining the order of page accesses can require additional processing and memory.

ALGORITHM:

1. **Initialization**:
    - Define the number of page frames (memory slots) and the page reference string (sequence of pages to be accessed).
    - Initialize a data structure to keep track of page access times (or order).
2. **Page Replacement**:
    - Traverse the page reference string.
    - For each page:
        - **If the page is already in memory**: Update its access time or position.
        - **If the page is not in memory**:
            - **If there is an empty frame**: Load the page into the empty frame.
            - **If all frames are full**: Identify the page with the oldest access time (least recently used) and replace it with the new page.
3. **Count Page Faults**:
    - Increment the page fault count whenever a page is loaded into memory.
4. **Output**:
    - Display the page reference string, the memory content after each page reference, and the total number of page faults.

SOURCE CODE:

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

#define MAX_FRAMES 10
#define MAX_PAGES 100

// Function to simulate LRU page replacement algorithm
void lruPageReplacement(int pages[], int numPages, int numFrames) {
    int frame[numFrames];
    int pageFaults = 0;
    int lastUsed[numFrames];
    int currentTime = 0;

    // Initialize frames and last used times
    for (int i = 0; i < numFrames; i++) {
        frame[i] = -1;
        lastUsed[i] = -1;
    }

    printf("Page Reference String: ");
    for (int i = 0; i < numPages; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    printf("Memory Content after each reference:\n");

    for (int i = 0; i < numPages; i++) {
        int page = pages[i];
```

```
bool pageFound = false;
int oldestPageIndex = 0;

// Check if page is already in memory
for (int j = 0; j < numFrames; j++) {
    if (frame[j] == page) {
        pageFound = true;
        lastUsed[j] = currentTime;
        break;
    }
}

// Page not found in memory, perform page replacement
if (!pageFound) {
    int minTime = currentTime;
    for (int j = 0; j < numFrames; j++) {
        if (lastUsed[j] < minTime) {
            minTime = lastUsed[j];
            oldestPageIndex = j;
        }
    }

    frame[oldestPageIndex] = page;
    lastUsed[oldestPageIndex] = currentTime;
    pageFaults++;
}

// Print memory content
printf("After page %d: ", page);
for (int j = 0; j < numFrames; j++) {
```

```
            if (frame[j] != -1) {
                printf("%d ", frame[j]);
            }
        }
        printf("\n");

        currentTime++;
    }

    // Print total page faults
    printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
    int numPages, numFrames;
    int pages[MAX_PAGES];

    // Input number of pages
    printf("Enter the number of pages: ");
    scanf("%d", &numPages);

    // Input page reference string
    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

    // Input number of frames
    printf("Enter the number of frames: ");
```

```
    scanf("%d", &numFrames);

    // Simulate LRU page replacement
    lruPageReplacement(pages, numPages, numFrames);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\lruPageRepAlgorithm.exe
Enter the number of pages: 12
Enter the page reference string:
Page 1: 7
Page 2: 0
Page 3: 1
Page 4: 2
Page 5: 0
Page 6: 3
Page 7: 0
Page 8: 4
Page 9: 2
Page 10: 3
Page 11: 0
Page 12: 3
Enter the number of frames: 3
Page Reference String: 7 0 1 2 0 3 0 4 2 3 0 3
Memory Content after each reference:
After page 7: 7
After page 0: 7 0
After page 1: 7 0 1
After page 2: 2 0 1
After page 0: 2 0 1
After page 3: 2 0 3
After page 0: 2 0 3
After page 4: 4 0 3
After page 2: 4 0 2
After page 3: 4 3 2
After page 0: 0 3 2
After page 3: 0 3 2
Total Page Faults: 9
```

EXPERIMENT NO. 16

PAGE REPLACEMENT ALGORITHM

AIM: Write a C program to simulate LFU page replacement algorithm.

THEORY:

LFU (Least Frequently Used) is a page replacement algorithm that replaces the page which has the smallest access count, i.e., the page that has been used the least number of times. LFU assumes that if a page has been used frequently in the past, it is likely to be used again in the future, so it should be kept in memory.

Advantages:

1. **Historical Data**: It considers the frequency of page accesses, which can be beneficial in scenarios where some pages are accessed much more frequently than others.
2. **Adaptive**: Adjusts to the actual usage patterns of the pages, potentially reducing page faults in certain workloads.

Disadvantages:

1. **Complexity**: Keeping track of page frequencies and managing the data structures required for LFU can be complex.
2. **Implementation Overhead**: More overhead compared to simpler algorithms like FIFO or LRU due to additional bookkeeping.

ALGORITHM:

1. **Initialization**:
    - Define the number of page frames (memory slots) and the page reference string (sequence of pages to be accessed).
    - Initialize data structures to track page frequencies and manage pages in memory.
2. **Page Replacement**:
    - Traverse the page reference string.
    - For each page:
        - **If the page is already in memory**: Update its frequency count.
        - **If the page is not in memory**:
            - **If there is an empty frame**: Load the page into the empty frame and initialize its frequency count.
            - **If all frames are full**: Identify the page with the smallest frequency (least frequently used) and replace it with the new page. Update the frequency counts accordingly.
3. **Count Page Faults**:
    - Increment the page fault count whenever a page is loaded into memory.
4. **Output**:
    - Display the page reference string, the memory content after each page reference, and the total number of page faults.

SOURCE CODE:

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

#define MAX_FRAMES 10
#define MAX_PAGES 100

// Function to simulate LFU page replacement algorithm
void lfuPageReplacement(int pages[], int numPages, int numFrames) {
    int frame[numFrames];
    int frequency[numFrames];
    int pageFaults = 0;
    int pageIndex = 0;

    // Initialize frames and frequency counts
    for (int i = 0; i < numFrames; i++) {
        frame[i] = -1;
        frequency[i] = 0;
    }

    printf("Page Reference String: ");
    for (int i = 0; i < numPages; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    printf("Memory Content after each reference:\n");

    for (int i = 0; i < numPages; i++) {
        int page = pages[i];
```

```
int pageFound = 0;
int minFrequency = INT_MAX;
int replaceIndex = 0;

// Check if page is already in memory
for (int j = 0; j < numFrames; j++) {
    if (frame[j] == page) {
        pageFound = 1;
        frequency[j]++;
        break;
    }
}

// Page not found in memory, perform page replacement
if (!pageFound) {
    // If memory is full, replace the least frequently used page
    for (int j = 0; j < numFrames; j++) {
        if (frequency[j] < minFrequency) {
            minFrequency = frequency[j];
            replaceIndex = j;
        }
    }

    frame[replaceIndex] = page;
    frequency[replaceIndex] = 1;
    pageFaults++;
}

// Print memory content
printf("After page %d: ", page);
```

```
        for (int j = 0; j < numFrames; j++) {
            if (frame[j] != -1) {
                printf("%d ", frame[j]);
            }
        }
        printf("\n");
    }

    // Print total page faults
    printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
    int numPages, numFrames;
    int pages[MAX_PAGES];

    // Input number of pages
    printf("Enter the number of pages: ");
    scanf("%d", &numPages);

    // Input page reference string
    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

    // Input number of frames
    printf("Enter the number of frames: ");
    scanf("%d", &numFrames);
```

```
    // Simulate LFU page replacement
    lfuPageReplacement(pages, numPages, numFrames);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\lfuPageRepAlgorithm.exe
Enter the number of pages: 12
Enter the page reference string:
Page 1: 7
Page 2: 0
Page 3: 1
Page 4: 2
Page 5: 0
Page 6: 3
Page 7: 0
Page 8: 4
Page 9: 2
Page 10: 3
Page 11: 0
Page 12: 3
Enter the number of frames: 3
Page Reference String: 7 0 1 2 0 3 0 4 2 3 0 3
Memory Content after each reference:
After page 7: 7
After page 0: 7 0
After page 1: 7 0 1
After page 2: 2 0 1
After page 0: 2 0 1
After page 3: 3 0 1
After page 0: 3 0 1
After page 4: 4 0 1
After page 2: 2 0 1
After page 3: 3 0 1
After page 0: 3 0 1
After page 3: 3 0 1
Total Page Faults: 8
```

EXPERIMENT NO. 17

PAGE REPLACEMENT ALGORITHM

AIM: Write a C program to simulate Optimal page replacement algorithm.

THEORY:

The **Optimal Page Replacement** algorithm, also known as the **Optimal Page Replacement Policy** or **OPT**, is a theoretical page replacement strategy that aims to minimize the number of page faults by replacing the page that will not be used for the longest period of time in the future. It is considered the optimal page replacement algorithm because it has the lowest possible page fault rate for a given reference string.

However, because the algorithm requires future knowledge of page references, it is not practical for real-world systems but serves as a benchmark to evaluate other page replacement algorithms.

Advantages:

1. **Optimal Performance**: It provides the best possible performance in terms of page fault rate.
2. **Benchmark**: It is used as a benchmark to compare the performance of other page replacement algorithms.

Disadvantages:

1. **Future Knowledge**: Requires knowledge of future page references, which is not possible in real systems.
2. **Practicality**: Not feasible for implementation in real-world systems due to its requirement for future information.

ALGORITHM:

1. **Initialization**:
    - Define the number of page frames (memory slots) and the page reference string (sequence of pages to be accessed).
    - Initialize the page frames to an empty state.
2. **Page Replacement**:
    - Traverse the page reference string.
    - For each page:
        - **If the page is already in memory**: Continue to the next page reference.
        - **If the page is not in memory**:
            - **If there is an empty frame**: Load the page into the empty frame.
            - **If all frames are full**: Identify the page that will not be used for the longest period of time in the future and replace it with the new page. Update the frame and page fault count accordingly.
3. **Count Page Faults**:
    - Increment the page fault count whenever a page is loaded into memory.

4. **Output**:
    - Display the page reference string, the memory content after each page reference, and the total number of page faults.

SOURCE CODE:

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

#define MAX_FRAMES 10
#define MAX_PAGES 100

// Function to find the index of the page to replace
int findOptimalPageToReplace(int frame[], int pageIndex[], int numFrames, int currentPage) {
    int farthest = currentPage;
    int replaceIndex = -1;

    for (int i = 0; i < numFrames; i++) {
        int j;
        for (j = currentPage; j < MAX_PAGES; j++) {
            if (frame[i] == pageIndex[j]) {
                if (j > farthest) {
                    farthest = j;
                    replaceIndex = i;
                }
                break;
            }
        }
        if (j == MAX_PAGES) {
            return i;
        }
```

```
    }
    return replaceIndex;
}

// Function to simulate Optimal page replacement algorithm
void optimalPageReplacement(int pages[], int numPages, int numFrames) {
    int frame[numFrames];
    int pageIndex[MAX_PAGES];
    int pageFaults = 0;
    int pageHits = 0;

    // Initialize frames and page index
    for (int i = 0; i < numFrames; i++) {
        frame[i] = -1;
    }
    for (int i = 0; i < numPages; i++) {
        pageIndex[i] = pages[i];
    }

    printf("Page Reference String: ");
    for (int i = 0; i < numPages; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    printf("Memory Content after each reference:\n");

    for (int i = 0; i < numPages; i++) {
        int page = pages[i];
        int pageFound = 0;
```

```
    // Check if page is already in memory
    for (int j = 0; j < numFrames; j++) {
        if (frame[j] == page) {
            pageFound = 1;
            pageHits++;
            break;
        }
    }

    // Page not found in memory, perform page replacement
    if (!pageFound) {
        int replaceIndex = findOptimalPageToReplace(frame, pageIndex, numFrames, i);
        frame[replaceIndex] = page;
        pageFaults++;
    }

    // Print memory content
    printf("After page %d: ", page);
    for (int j = 0; j < numFrames; j++) {
        if (frame[j] != -1) {
            printf("%d ", frame[j]);
        }
    }
    printf("\n");
}

// Print total page faults
printf("Total Page Faults: %d\n", pageFaults);
printf("Total Page Hits: %d\n", pageHits);
```

```
}

int main() {
    int numPages, numFrames;
    int pages[MAX_PAGES];

    // Input number of pages
    printf("Enter the number of pages: ");
    scanf("%d", &numPages);

    // Input page reference string
    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

    // Input number of frames
    printf("Enter the number of frames: ");
    scanf("%d", &numFrames);

    // Simulate Optimal page replacement
    optimalPageReplacement(pages, numPages, numFrames);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\optimalPageRepAlgorithm.exe
Enter the number of pages: 12
Enter the page reference string:
Page 1: 7
Page 2: 0
Page 3: 1
Page 4: 2
Page 5: 0
Page 6: 3
Page 7: 0
Page 8: 2
Page 9: 2
Page 10: 3
Page 11: 0
Page 12: 3
Enter the number of frames: 3
Page Reference String: 7 0 1 2 0 3 0 2 2 3 0 3
Memory Content after each reference:
After page 7: 7
After page 0: 0
After page 1: 0 1
After page 2: 0 1 2
After page 0: 0 1 2
After page 3: 0 3 2
After page 0: 0 3 2
After page 2: 0 3 2
After page 2: 0 3 2
After page 3: 0 3 2
After page 0: 0 3 2
After page 3: 0 3 2
Total Page Faults: 5
Total Page Hits: 7
```

EXPERIMENT NO. 18

PAGE REPLACEMENT ALGORITHM

AIM: Write a C program to simulate the following file organization techniques

a) Single level directory b) Two level directory c) Hierarchical

THEORY:

a) **Single-level directory**

Single-Level Directory is the simplest form of file organization where all files are stored in a single directory. Each file is uniquely identified by its filename, and there is no hierarchy or subdirectories. This method is straightforward but may lead to inefficiencies as the number of files increases since all files are managed in a single directory.

Advantages:

1. **Simplicity**: Easy to implement and manage.
2. **No Overhead**: No additional overhead for managing directories.

Disadvantages:

1. **Scalability Issues**: Managing a large number of files can become cumbersome.
2. **Lack of Organization**: No way to group related files, which can make file management difficult.

ALGORITHM:

1. Initialize a single directory structure to store file names and their data.
2. Provide operations to create, delete, and list files within a single directory.
3. Each file is accessed by its unique name.

b) **Two-Level Directory**

Theory:

- **Two-Level Directory** organizes files into a two-tier hierarchy: the root directory and user directories.
- Each user has their own directory under the root directory, and all files for that user are stored in their specific directory.
- This approach helps in managing files more effectively by grouping them based on users.

Advantages:

1. **Better Organization**: Files are grouped by users, reducing clutter in a single directory.
2. **Improved Scalability**: Easier to manage files as the number of users grows.

Disadvantages:

1. **Complexity**: Slightly more complex than single-level directories.
2. **Limited Hierarchy**: Only provides one level of grouping (user directories).

ALGORITHM:

1. Initialize a root directory and user directories under it.
2. Provide operations to create, delete, and list files within user directories.
3. Access files through user-specific directories.

c) **Hierarchical Directory**

Theory:

- **Hierarchical Directory** organizes files into a tree-like structure with multiple levels of directories.
- Each directory can contain subdirectories, creating a hierarchy that can reflect the organization of files and folders more intuitively.
- This method is common in modern operating systems and provides flexible and scalable file management.

Advantages:

1. **Flexible Organization**: Supports complex directory structures and nesting.
2. **Efficient File Management**: Allows logical grouping of files and directories, making file management easier.

Disadvantages:

1. **Complexity**: More complex to implement and manage.
2. **Overhead**: Additional overhead for managing multiple levels of directories.

ALGORITHM:

1. Initialize a root directory and provide operations to create, delete, and navigate directories.
2. Each directory can contain files or other subdirectories.
3. Support operations to traverse the directory tree and manage files within any directory.

SOURCE CODE:

```
#include <stdio.h>

#include <string.h>

#define MAX_FILES 10

#define MAX_NAME_LENGTH 100

#define MAX_CONTENT_LENGTH 100

// Define a file structure

typedef struct {

    char name[MAX_NAME_LENGTH];

    char content[MAX_CONTENT_LENGTH];

} File;

// Define a single-level directory structure

typedef struct {

    File files[MAX_FILES];

    int count; // Number of files currently in the directory

} SingleLevelDirectory;


// Initialize the directory

void initSingleLevelDirectory(SingleLevelDirectory *dir) {

    dir->count = 0;

}


// Add a file to the directory

void addFileToSingleLevelDirectory(SingleLevelDirectory *dir, const char *name, const char *content) {
```

```
    if (dir->count < MAX_FILES) {
        strcpy(dir->files[dir->count].name, name);
        strcpy(dir->files[dir->count].content, content);
        dir->count++;
        printf("File '%s' added successfully.\n", name);
    } else {
        printf("Directory is full! Cannot add more files.\n");
    }
}
// Delete a file from the directory
void deleteFileFromSingleLevelDirectory(SingleLevelDirectory *dir, const char *name) {
    int index = -1;
    for (int i = 0; i < dir->count; i++) {
        if (strcmp(dir->files[i].name, name) == 0) {
            index = i;
            break;
        }
    }
    if (index == -1) {
        printf("File '%s' not found.\n", name);
        return;
    }
    // Shift files to remove the deleted file
    for (int i = index; i < dir->count - 1; i++) {
```

```
        dir->files[i] = dir->files[i + 1];
    }
    dir->count--;
    printf("File '%s' deleted successfully.\n", name);
}
// List all files in the directory
void listFilesInSingleLevelDirectory(SingleLevelDirectory *dir) {
    if (dir->count == 0) {
        printf("Directory is empty.\n");
        return;
    }
    printf("Files in Single-Level Directory:\n");
    for (int i = 0; i < dir->count; i++) {
        printf("File Name: %s\n", dir->files[i].name);
        printf("Content: %s\n", dir->files[i].content);
        printf("------------------------------\n");
    }
}
int main() {
    SingleLevelDirectory directory;
    initSingleLevelDirectory(&directory);
    // Adding files to the directory
    addFileToSingleLevelDirectory(&directory, "file1.txt", "Content of file1");
    addFileToSingleLevelDirectory(&directory, "file2.txt", "Content of file2");
```

```
    // Listing files
    listFilesInSingleLevelDirectory(&directory);
    // Deleting a file
    deleteFileFromSingleLevelDirectory(&directory, "file1.txt");
    // Listing files again
    listFilesInSingleLevelDirectory(&directory);
    return 0;
}
```

Single Leve Directory:

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\singleLevFileOrgTechnique.exe
File 'file1.txt' added successfully.
File 'file2.txt' added successfully.
Files in Single-Level Directory:
File Name: file1.txt
Content: Content of file1
------------------------------
File Name: file2.txt
Content: Content of file2
------------------------------
File 'file1.txt' deleted successfully.
Files in Single-Level Directory:
File Name: file2.txt
Content: Content of file2
------------------------------
```

Two Leve Directory:

```
#include <stdio.h>
#include <string.h>
#define MAX_USERS 5
#define MAX_FILES 10
#define MAX_NAME_LENGTH 100
#define MAX_CONTENT_LENGTH 100
```

```
// Define a file structure
typedef struct {
    char name[MAX_NAME_LENGTH];
    char content[MAX_CONTENT_LENGTH];
} File;
// Define a user directory structure
typedef struct {
    File files[MAX_FILES];
    int count; // Number of files currently in the user directory
} UserDirectory;
// Define a two-level directory structure
typedef struct {
    UserDirectory userDirs[MAX_USERS];
    int userCount; // Number of user directories
} TwoLevelDirectory;
// Initialize the two-level directory
void initTwoLevelDirectory(TwoLevelDirectory *dir) {
    dir->userCount = 0;
}
// Initialize a user directory
void initUserDirectory(UserDirectory *userDir) {
    userDir->count = 0;
}
// Add a file to a user directory
void addFileToUserDirectory(UserDirectory *userDir, const char *name, const char *content) {
    if (userDir->count < MAX_FILES) {
        strcpy(userDir->files[userDir->count].name, name);
        strcpy(userDir->files[userDir->count].content, content);
        userDir->count++;
        printf("File '%s' added successfully.\n", name);
```

```
    } else {
        printf("User directory is full! Cannot add more files.\n");
    }
}
// Delete a file from a user directory
void deleteFileFromUserDirectory(UserDirectory *userDir, const char *name) {
    int index = -1;
    for (int i = 0; i < userDir->count; i++) {
        if (strcmp(userDir->files[i].name, name) == 0) {
            index = i;
            break;
        }
    }
    if (index == -1) {
        printf("File '%s' not found.\n", name);
        return;
    }
    // Shift files to remove the deleted file
    for (int i = index; i < userDir->count - 1; i++) {
        userDir->files[i] = userDir->files[i + 1];
    }
    userDir->count--;
    printf("File '%s' deleted successfully.\n", name);
}
// List all files in a user directory
void listFilesInUserDirectory(UserDirectory *userDir) {
    if (userDir->count == 0) {
        printf("User directory is empty.\n");
        return;
    }
```

```
    printf("Files in User Directory:\n");
    for (int i = 0; i < userDir->count; i++) {
        printf("File Name: %s\n", userDir->files[i].name);
        printf("Content: %s\n", userDir->files[i].content);
        printf("------------------------------\n");
    }
}
// Add a user directory to the two-level directory
void addUserDirectory(TwoLevelDirectory *dir) {
    if (dir->userCount < MAX_USERS) {
        initUserDirectory(&dir->userDirs[dir->userCount]);
        dir->userCount++;
        printf("User directory added successfully.\n");
    } else {
        printf("Two-level directory is full! Cannot add more user directories.\n");
    }
}

// List all user directories in the two-level directory
void listUserDirectories(TwoLevelDirectory *dir) {
    if (dir->userCount == 0) {
        printf("No user directories available.\n");
        return;
    }
    for (int i = 0; i < dir->userCount; i++) {
        printf("User Directory %d:\n", i);
        listFilesInUserDirectory(&dir->userDirs[i]);
    }
}
int main() {
```

```
    TwoLevelDirectory directory;
    initTwoLevelDirectory(&directory);
    // Adding user directories
    addUserDirectory(&directory);
    addUserDirectory(&directory);
    // Adding files to user directories
    addFileToUserDirectory(&directory.userDirs[0], "user1file1.txt", "Content of user1 file1");
    addFileToUserDirectory(&directory.userDirs[0], "user1file2.txt", "Content of user1 file2");
    addFileToUserDirectory(&directory.userDirs[1], "user2file1.txt", "Content of user2 file1");
    // Listing files in user directories
    printf("\nListing files in all user directories:\n");
    listUserDirectories(&directory);

    // Deleting a file
    deleteFileFromUserDirectory(&directory.userDirs[0], "user1file1.txt");
    // Listing files again
    printf("\nListing files in all user directories after deletion:\n");
    listUserDirectories(&directory);
    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\twoLevelFileOrgTechnique.exe
User directory added successfully.
User directory added successfully.
File 'user1file1.txt' added successfully.
File 'user1file2.txt' added successfully.
File 'user2file1.txt' added successfully.

Listing files in all user directories:
User Directory 0:
Files in User Directory:
File Name: user1file1.txt
Content: Content of user1 file1
------------------------------
File Name: user1file2.txt
Content: Content of user1 file2
------------------------------
User Directory 1:
Files in User Directory:
File Name: user2file1.txt
Content: Content of user2 file1
------------------------------
File 'user1file1.txt' deleted successfully.

Listing files in all user directories after deletion:
User Directory 0:
Files in User Directory:
File Name: user1file2.txt
Content: Content of user1 file2
------------------------------
User Directory 1:
Files in User Directory:
File Name: user2file1.txt
Content: Content of user2 file1
```

Hierarchical Directory

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

#define MAX_NAME_LENGTH 100
#define MAX_CONTENT_LENGTH 100
#define MAX_FILES 10
#define MAX_SUBDIRS 10

typedef struct {
    char name[MAX_NAME_LENGTH];
    char content[MAX_CONTENT_LENGTH];
} File;

typedef struct Directory {
    char name[MAX_NAME_LENGTH];
    File files[MAX_FILES];
    int fileCount;
    struct Directory* subdirs[MAX_SUBDIRS];
    int subdirCount;
} Directory;

// Initialize a directory
void initDirectory(Directory* dir, const char* name) {
    strncpy(dir->name, name, MAX_NAME_LENGTH);
    dir->fileCount = 0;
    dir->subdirCount = 0;
}
```

```
// Add a file to a directory
void addFileToDirectory(Directory* dir, const char* fileName, const char* content) {
    if (dir->fileCount < MAX_FILES) {
        strncpy(dir->files[dir->fileCount].name, fileName, MAX_NAME_LENGTH);
        strncpy(dir->files[dir->fileCount].content, content, MAX_CONTENT_LENGTH);
        dir->fileCount++;
    } else {
        printf("Directory '%s' is full!\n", dir->name);
    }
}

// Add a subdirectory to a directory
void addSubdirectoryToDirectory(Directory* dir, Directory* subdir) {
    if (dir->subdirCount < MAX_SUBDIRS) {
        dir->subdirs[dir->subdirCount] = subdir;
        dir->subdirCount++;
    } else {
        printf("Directory '%s' is full of subdirectories!\n", dir->name);
    }
}

// List all files in a directory
void listFilesInDirectory(const Directory* dir) {
    if (dir->fileCount > 0) {
        printf("Files in '%s':\n", dir->name);
        for (int i = 0; i < dir->fileCount; i++) {
            printf("  %s: %s\n", dir->files[i].name, dir->files[i].content);
        }
    } else {
        printf("Directory '%s' is empty.\n", dir->name);
```

```
    }
}

// Recursively list all files and directories
void listAllInDirectory(const Directory* dir, int level) {
    for (int i = 0; i < level; i++) printf("  ");
    printf("Directory '%s':\n", dir->name);
    listFilesInDirectory(dir);

    for (int i = 0; i < dir->subdirCount; i++) {
        listAllInDirectory(dir->subdirs[i], level + 1);
    }
}

int main() {
    Directory root;
    initDirectory(&root, "root");

    Directory subdir1, subdir2;
    initDirectory(&subdir1, "subdir1");
    initDirectory(&subdir2, "subdir2");

    addFileToDirectory(&root, "file1.txt", "Content of file1");
    addFileToDirectory(&root, "file2.txt", "Content of file2");
    addFileToDirectory(&subdir1, "subfile1.txt", "Content of subdir1 file1");
    addFileToDirectory(&subdir2, "subfile2.txt", "Content of subdir2 file1");

    addSubdirectoryToDirectory(&root, &subdir1);
    addSubdirectoryToDirectory(&root, &subdir2);
```

```
    printf("Hierarchical Directory Structure:\n");
    listAllInDirectory(&root, 0);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\hireFileOrgTechnique.exe
File 'file1.txt' added to directory 'root'.
File 'file2.txt' added to directory 'root'.
File 'subfile1.txt' added to directory 'subdir1'.
File 'subfile2.txt' added to directory 'subdir2'.
Subdirectory 'subdir1' added to directory 'root'.
Subdirectory 'subdir2' added to directory 'root'.
Hierarchical Directory Structure:
Directory 'root':
Files in directory 'root':
File Name: file1.txt
Content: Content of root file1
-----------------------------
File Name: file2.txt
Content: Content of root file2
-----------------------------
  Directory 'subdir1':
Files in directory 'subdir1':
File Name: subfile1.txt
Content: Content of subdir1 file1
-----------------------------
  Directory 'subdir2':
Files in directory 'subdir2':
File Name: subfile2.txt
Content: Content of subdir2 file1
-----------------------------
```

EXPERIMENT NO. 19

FILE ALLOCATION STRATEGIES

AIM: Write a program to Simulate all file allocation strategies

a) Sequential b) Indexed c) Linked

THEORY:

a) Sequential Allocation

In sequential allocation, files are stored in contiguous blocks on the disk. This means that the entire file is written in a contiguous block of disk space. Sequential allocation is simple and efficient for reading files sequentially but can lead to fragmentation when files are deleted or resized.

ALGORITHM:

1. **Initialization**:
   - Initialize a disk space array with all blocks marked as free.
   - Allocate a contiguous block of free space for a new file.
2. **File Allocation**:
   - Find a contiguous block of free space that is large enough to hold the file.
   - Allocate the required space to the file.
   - Mark the allocated blocks as occupied.
3. **File Access**:
   - To read or write a file, access the blocks in the allocated range.

b) Indexed Allocation

Indexed allocation uses an index block to keep track of all the blocks that are used by a file. Each file has an index block (or table) that contains pointers to the actual data blocks. This method avoids fragmentation and allows for efficient random access but requires extra space for the index block.

ALGORITHM:

1. **Initialization**:
   - Initialize a disk space array with all blocks marked as free.
   - Create an index block for the file.
2. **File Allocation**:
   - Allocate data blocks for the file and update the index block with pointers to these blocks.
   - Mark the allocated blocks as occupied.
3. **File Access**:
   - To read or write a file, use the index block to locate the data blocks.

c) Linked Allocation

Linked allocation uses a linked list to keep track of blocks used by a file. Each block contains a pointer to the next block in the file. This method provides flexibility in file size and efficient use of disk space but can result in slower access times due to the need to traverse the list.

ALGORITHM:

1. **Initialization**:
    - Initialize a disk space array with all blocks marked as free.
2. **File Allocation**:
    - Allocate blocks for the file and set up pointers in each block to the next block.
    - Mark the allocated blocks as occupied.
3. **File Access**:
    - To read or write a file, follow the pointers from the first block to access subsequent blocks.

SOURCE CODE:

For Sequential Allocation

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX_BLOCKS 100

#define BLOCK_SIZE 10

#define MAX_FILES 5

// Disk space

int disk[MAX_BLOCKS];

int disk_size = MAX_BLOCKS;

// File structure

typedef struct {

   char name[20];

   int size;

   int start_block;  // For Sequential Allocation

} File;

// Initialize the disk

void initializeDisk() {
```

```
    for (int i = 0; i < disk_size; i++) {
        disk[i] = 0; // 0 means free
    }
}
// Print the disk status
void printDiskStatus() {
    printf("Disk Status:\n");
    for (int i = 0; i < disk_size; i++) {
        printf("Block %d: %s\n", i, disk[i] == 0 ? "Free" : "Occupied");
    }
}
// Sequential Allocation
void allocateSequential(File *file) {
    int start = -1;
    int count = 0;
    for (int i = 0; i < disk_size; i++) {
        if (disk[i] == 0) {
            if (count == 0) start = i;
            count++;
            if (count == file->size) {
                file->start_block = start;
                for (int j = start; j < start + file->size; j++) {
                    disk[j] = 1;
                }
                printf("File '%s' allocated sequentially from block %d.\n", file->name, start);
                return;
            }
        } else {
            count = 0;
        }
```

```
    }
    printf("Not enough contiguous space for file '%s'.\n", file->name);
}
// Free allocated blocks
void freeBlocks(File *file) {
    for (int i = file->start_block; i < file->start_block + file->size; i++) {
        disk[i] = 0;
    }
}
int main() {
    File files[MAX_FILES] = {
        {"File1", 5},
        {"File2", 3},
        {"File3", 7},
        {"File4", 2},
        {"File5", 4}
    };
    // Initialize disk
    initializeDisk();
    printDiskStatus();
    // Allocate files
    printf("\nSequential Allocation:\n");
    for (int i = 0; i < MAX_FILES; i++) {
        allocateSequential(&files[i]);
    }
    printDiskStatus();
    // Free allocated blocks
    printf("\nFreeing Allocated Blocks:\n");
    for (int i = 0; i < MAX_FILES; i++) {
        freeBlocks(&files[i]);
```

```
    }
    printDiskStatus();
    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\seqFileAllocStrategy.exe
Disk Status:
Block 0: Free
Block 1: Free
Block 2: Free
Block 3: Free
Block 4: Free
Block 5: Free
Block 6: Free

Sequential Allocation:
File 'File1' allocated sequentially from block 0.
Not enough contiguous space for file 'File2'.
Not enough contiguous space for file 'File3'.
File 'File4' allocated sequentially from block 5.
Not enough contiguous space for file 'File5'.
Disk Status:
Block 0: Occupied
Block 1: Occupied
Block 2: Occupied
Block 3: Occupied
Block 4: Occupied
Block 5: Occupied
Block 6: Occupied

Freeing Allocated Blocks:
Disk Status:
Block 0: Free
Block 1: Free
Block 2: Free
Block 3: Free
Block 4: Free
Block 5: Free
Block 6: Free
```

For Indexed Allocation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_BLOCKS 100
#define MAX_FILES 5
// Disk space
int disk[MAX_BLOCKS];
int disk_size = MAX_BLOCKS;
// File structure
typedef struct {
    char name[20];
    int size;
    int *index_blocks; // For Indexed Allocation
} File;
// Initialize the disk
void initializeDisk() {
    for (int i = 0; i < disk_size; i++) {
        disk[i] = 0; // 0 means free
    }
}
// Print the disk status
void printDiskStatus() {
    printf("Disk Status:\n");
    for (int i = 0; i < disk_size; i++) {
        printf("Block %d: %s\n", i, disk[i] == 0 ? "Free" : "Occupied");
    }
}
// Indexed Allocation
void allocateIndexed(File *file) {
```

```
    file->index_blocks = (int *)malloc(file->size * sizeof(int));
    if (!file->index_blocks) {
        printf("Memory allocation failed for index blocks.\n");
        return;
    }
    int count = 0;
    for (int i = 0; i < disk_size; i++) {
        if (disk[i] == 0) {
            file->index_blocks[count] = i;
            disk[i] = 1;
            count++;
            if (count == file->size) {
                printf("File '%s' allocated with index blocks: ", file->name);
                for (int j = 0; j < file->size; j++) {
                    printf("%d ", file->index_blocks[j]);
                }
                printf("\n");
                return;
            }
        }
    }
    printf("Not enough free blocks for file '%s'.\n", file->name);
}

// Free allocated blocks
void freeBlocks(File *file) {
    for (int i = 0; i < file->size; i++) {
        disk[file->index_blocks[i]] = 0;
    }
    free(file->index_blocks);
```

```
}
int main() {
    File files[MAX_FILES] = {
        {"File1", 5},
        {"File2", 3},
        {"File3", 7},
        {"File4", 2},
        {"File5", 4}
    };
    // Initialize disk
    initializeDisk();
    printDiskStatus();
    // Allocate files
    printf("\nIndexed Allocation:\n");
    for (int i = 0; i < MAX_FILES; i++) {
        allocateIndexed(&files[i]);
    }
    printDiskStatus();

    // Free allocated blocks
    printf("\nFreeing Allocated Blocks:\n");
    for (int i = 0; i < MAX_FILES; i++) {
        freeBlocks(&files[i]);
    }
    printDiskStatus();

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\indexedFileAllocStrategy.exe
Disk Status:
Block 0: Free
Block 1: Free
Block 2: Free
Block 3: Free
Block 4: Free
Block 5: Free

Indexed Allocation:
File 'File1' allocated with index blocks: 0 1 2 3 4
Not enough free blocks for file 'File2'.
Not enough free blocks for file 'File3'.
Not enough free blocks for file 'File4'.
Not enough free blocks for file 'File5'.
Disk Status:
Block 0: Occupied
Block 1: Occupied
Block 2: Occupied
Block 3: Occupied
Block 4: Occupied
Block 5: Occupied

Freeing Allocated Blocks:
```

For Linked Allocation

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_BLOCKS 100

#define MAX_FILES 5


// Disk space
```

```
int disk[MAX_BLOCKS];
int disk_size = MAX_BLOCKS;

// File structure
typedef struct File {
    char name[20];
    int size;
    int *linked_blocks; // For Linked Allocation
} File;

// Initialize the disk
void initializeDisk() {
    for (int i = 0; i < disk_size; i++) {
        disk[i] = 0; // 0 means free
    }
}

// Print the disk status
void printDiskStatus() {
    printf("Disk Status:\n");
    for (int i = 0; i < disk_size; i++) {
        printf("Block %d: %s\n", i, disk[i] == 0 ? "Free" : "Occupied");
    }
}

// Linked Allocation
void allocateLinked(File *file) {
    file->linked_blocks = (int *)malloc(file->size * sizeof(int));
    if (!file->linked_blocks) {
        printf("Memory allocation failed for linked blocks.\n");
```

```
        return;
    }

    int count = 0;
    int prev_block = -1;
    for (int i = 0; i < disk_size; i++) {
        if (disk[i] == 0) {
            file->linked_blocks[count] = i;
            disk[i] = 1;
            if (prev_block != -1) {
                printf("Block %d -> Block %d\n", file->linked_blocks[count - 1], i);
            }
            prev_block = i;
            count++;
            if (count == file->size) {
                printf("File '%s' allocated with linked blocks: ", file->name);
                for (int j = 0; j < file->size; j++) {
                    printf("%d ", file->linked_blocks[j]);
                }
                printf("\n");
                return;
            }
        }
    }
    printf("Not enough free blocks for file '%s'.\n", file->name);
}

// Free allocated blocks
void freeBlocks(File *file) {
    for (int i = 0; i < file->size; i++) {
```

```
        disk[file->linked_blocks[i]] = 0;
    }
    free(file->linked_blocks);
}

int main() {
    File files[MAX_FILES] = {
        {"File1", 5},
        {"File2", 3},
        {"File3", 7},
        {"File4", 2},
        {"File5", 4}
    };

    // Initialize disk
    initializeDisk();
    printDiskStatus();

    // Allocate files
    printf("\nLinked Allocation:\n");
    for (int i = 0; i < MAX_FILES; i++) {
        allocateLinked(&files[i]);
    }
    printDiskStatus();

    // Free allocated blocks
    printf("\nFreeing Allocated Blocks:\n");
    for (int i = 0; i < MAX_FILES; i++) {
        freeBlocks(&files[i]);
    }
```

```
    printDiskStatus();

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\linkedFileAllocStrategy.exe
Disk Status:
Block 0: Free
Block 1: Free
Block 2: Free
Block 3: Free
Block 4: Free
Block 5: Free

Linked Allocation:
Block 0 -> Block 1
Block 1 -> Block 2
Block 2 -> Block 3
Block 3 -> Block 4
File 'File1' allocated with linked blocks: 0 1 2 3 4
Not enough free blocks for file 'File2'.
Not enough free blocks for file 'File3'.
Not enough free blocks for file 'File4'.
Not enough free blocks for file 'File5'.
Disk Status:
Block 0: Occupied
Block 1: Occupied
Block 2: Occupied
Block 3: Occupied
Block 4: Occupied
Block 5: Occupied

Freeing Allocated Blocks:
```

EXPERIMENT NO. 20

DISK SCHEDULING ALGORITHMS

AIM: Write a C program to simulate disk scheduling algorithms.

a) FCFS b) SCAN c) LOOK

THEORY:

Disk scheduling algorithms are used to determine the order in which disk I/O requests are processed. These algorithms aim to optimize disk access times and improve overall system performance. Below are the theories and algorithms for three common disk scheduling techniques: FCFS, SCAN, and LOOK.

a) FCFS (First-Come, First-Served)

The FCFS disk scheduling algorithm processes disk I/O requests in the order they arrive. It is the simplest disk scheduling algorithm and ensures fairness as each request is handled in the order it was requested, without any reordering. However, it can lead to the "convoy effect," where short requests might be delayed behind longer ones, potentially causing inefficiencies.

ALGORITHM:

1. Maintain a queue of disk I/O requests.
2. Start from the initial position of the disk head.
3. Process each request in the order they are queued.
4. Move the disk head to the track of the current request.
5. Complete the request and move to the next request in the queue.
6. Repeat until all requests are processed.

b) SCAN

The SCAN disk scheduling algorithm, also known as the "Elevator Algorithm," moves the disk head in one direction (e.g., towards higher track numbers) and services all requests in that direction until it reaches the end of the disk. It then reverses direction and services requests on the return trip. This approach aims to reduce the average seek time compared to FCFS.

ALGORITHM:

1. Initialize the disk head position and direction of movement.
2. Move the disk head in the current direction (e.g., upward) and service all requests in that direction.
3. Once the end of the disk is reached, reverse the direction of movement.
4. Continue servicing requests in the new direction until all requests are processed.
5. Repeat the process until all requests are completed.

c) LOOK

The LOOK disk scheduling algorithm is similar to SCAN but with a crucial difference: it only moves the disk head between the outermost requests in the current direction, rather than traveling to the end of the disk. This reduces unnecessary movement when no requests exist in the direction of travel, thus optimizing seek times.

ALGORITHM:

1. Initialize the disk head position and direction of movement.
2. Move the disk head in the current direction and service all requests in that direction.
3. When no more requests are present in the current direction, reverse the direction of movement.
4. Move the disk head in the new direction and service all requests.
5. Continue this process until all requests are serviced.

SOURCE CODE:

For FCFS (First-Come, First-Served)

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_REQUESTS 100


// Function to calculate total seek time and print the sequence

void fcfsDiskScheduling(int requests[], int n, int initial_position) {

    int total_seek_time = 0;

    int current_position = initial_position;


    printf("FCFS Disk Scheduling:\n");

    printf("Initial Disk Head Position: %d\n", initial_position);


    // Print the sequence of requests

    printf("Request Sequence: ");

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

        printf("%d ", requests[i]);

    }
```

```
    printf("\n");

    // Calculate the total seek time
    for (int i = 0; i < n; i++) {
        int seek_time = abs(requests[i] - current_position);
        total_seek_time += seek_time;
        printf("Move from %d to %d: Seek Time = %d\n", current_position, requests[i], seek_time);
        current_position = requests[i];
    }

    printf("Total Seek Time: %d\n", total_seek_time);
}

int main() {
    int n, initial_position;

    // Input number of requests and initial disk head position
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX_REQUESTS) {
        printf("Invalid number of requests. Exiting...\n");
        return 1;
    }

    int requests[n];

    printf("Enter the disk requests:\n");
    for (int i = 0; i < n; i++) {
        printf("Request %d: ", i + 1);
```

```
        scanf("%d", &requests[i]);
    }

    printf("Enter the initial disk head position: ");
    scanf("%d", &initial_position);

    // Simulate FCFS Disk Scheduling
    fcfsDiskScheduling(requests, n, initial_position);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\fcfsDiskSchAlgo.exe
Enter the number of disk requests: 5
Enter the disk requests:
Request 1: 98
Request 2: 183
Request 3: 37
Request 4: 122
Request 5: 14
Enter the initial disk head position: 50
FCFS Disk Scheduling:
Initial Disk Head Position: 50
Request Sequence: 98 183 37 122 14
Move from 50 to 98: Seek Time = 48
Move from 98 to 183: Seek Time = 85
Move from 183 to 37: Seek Time = 146
Move from 37 to 122: Seek Time = 85
Move from 122 to 14: Seek Time = 108
Total Seek Time: 472
```

For SCAN

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_REQUESTS 100
#define DISK_SIZE 200 // Define the size of the disk
// Function to compare two integers used for sorting requests
int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}
// Function to simulate SCAN Disk Scheduling
void scanDiskScheduling(int requests[], int n, int initial_position, int direction) {
    int total_seek_time = 0;
    int current_position = initial_position;
    int seek_sequence[MAX_REQUESTS];
    int count = 0;
    // Sort the requests in ascending order
    qsort(requests, n, sizeof(int), compare);
    // Add the initial position to the sequence
    seek_sequence[count++] = initial_position;
    // Handle direction
    if (direction == 1) { // Move right
        for (int i = 0; i < n; i++) {
            if (requests[i] >= current_position) {
                seek_sequence[count++] = requests[i];
            }
        }
        // Move to the end of the disk
        if (current_position < DISK_SIZE) {
```

```
        seek_sequence[count++] = DISK_SIZE - 1;
    }
    // Move left
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] < DISK_SIZE && requests[i] < current_position) {
            seek_sequence[count++] = requests[i];
        }
    }
} else { // Move left
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] <= current_position) {
            seek_sequence[count++] = requests[i];
        }
    }
    // Move to the beginning of the disk
    if (current_position > 0) {
        seek_sequence[count++] = 0;
    }
    // Move right
    for (int i = 0; i < n; i++) {
        if (requests[i] > 0 && requests[i] > current_position) {
            seek_sequence[count++] = requests[i];
        }
    }
}
// Calculate the total seek time
printf("SCAN Disk Scheduling:\n");
printf("Seek Sequence: ");
```

```
    for (int i = 0; i < count; i++) {
        printf("%d ", seek_sequence[i]);
        if (i > 0) {
            total_seek_time += abs(seek_sequence[i] - seek_sequence[i - 1]);
        }
    }
    printf("\nTotal Seek Time: %d\n", total_seek_time);
}
int main() {
    int n, initial_position, direction;
    // Input number of requests and initial disk head position
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);
    if (n <= 0 || n > MAX_REQUESTS) {
        printf("Invalid number of requests. Exiting...\n");
        return 1;
    }
    int requests[n];
    printf("Enter the disk requests:\n");
    for (int i = 0; i < n; i++) {
        printf("Request %d: ", i + 1);
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial disk head position: ");
    scanf("%d", &initial_position);
    printf("Enter the direction of head movement (1 for right, 0 for left): ");
    scanf("%d", &direction);
```

```
    if (direction != 0 && direction != 1) {
        printf("Invalid direction. Use 1 for right or 0 for left.\n");
        return 1;
    }
    // Simulate SCAN Disk Scheduling
    scanDiskScheduling(requests, n, initial_position, direction);
    return 0;
}
```

OUTPUT:

```
Enter the number of disk requests: 8
Enter the disk requests:
Request 1: 55
Request 2: 58
Request 3: 39
Request 4: 18
Request 5: 90
Request 6: 160
Request 7: 150
Request 8: 12
Enter the initial disk head position: 50
Enter the direction of head movement (1 for right, 0 for left): 1
SCAN Disk Scheduling:
Seek Sequence: 50 55 58 90 150 160 199 39 18 12
Total Seek Time: 336
```

For LOOK

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

#define MAX_REQUESTS 100
#define DISK_SIZE 200 // Define the size of the disk

// Function to compare two integers used for sorting requests
int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

// Function to simulate LOOK Disk Scheduling
void lookDiskScheduling(int requests[], int n, int initial_position, int direction) {
    int total_seek_time = 0;
    int current_position = initial_position;
    int seek_sequence[MAX_REQUESTS];
    int count = 0;

    // Sort the requests in ascending order
    qsort(requests, n, sizeof(int), compare);

    // Add the initial position to the sequence
    seek_sequence[count++] = initial_position;

    // Handle direction
    if (direction == 1) { // Move right
        // Move right to the highest request in the right direction
        for (int i = 0; i < n; i++) {
            if (requests[i] >= current_position) {
```

```
            seek_sequence[count++] = requests[i];
        }
    }
    // Move left after reaching the highest request
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] < current_position) {
            seek_sequence[count++] = requests[i];
        }
    }
} else { // Move left
    // Move left to the lowest request in the left direction
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] <= current_position) {
            seek_sequence[count++] = requests[i];
        }
    }
    // Move right after reaching the lowest request
    for (int i = 0; i < n; i++) {
        if (requests[i] > current_position) {
            seek_sequence[count++] = requests[i];
        }
    }
}

// Calculate the total seek time
printf("LOOK Disk Scheduling:\n");
printf("Seek Sequence: ");
for (int i = 0; i < count; i++) {
    printf("%d ", seek_sequence[i]);
    if (i > 0) {
```

```
            total_seek_time += abs(seek_sequence[i] - seek_sequence[i - 1]);
        }
    }
    printf("\nTotal Seek Time: %d\n", total_seek_time);
}

int main() {
    int n, initial_position, direction;

    // Input number of requests and initial disk head position
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX_REQUESTS) {
        printf("Invalid number of requests. Exiting...\n");
        return 1;
    }

    int requests[n];

    printf("Enter the disk requests:\n");
    for (int i = 0; i < n; i++) {
        printf("Request %d: ", i + 1);
        scanf("%d", &requests[i]);
    }

    printf("Enter the initial disk head position: ");
    scanf("%d", &initial_position);

    printf("Enter the direction of head movement (1 for right, 0 for left): ");
```

```
    scanf("%d", &direction);

    if (direction != 0 && direction != 1) {
        printf("Invalid direction. Use 1 for right or 0 for left.\n");
        return 1;
    }

    // Simulate LOOK Disk Scheduling
    lookDiskScheduling(requests, n, initial_position, direction);

    return 0;
}
```

OUTPUT:

```
PS D:\Arjun Mijar (160)> .\lookDiskSchAlgo.exe
Enter the number of disk requests: 8
Enter the disk requests:
Request 1: 55
Request 2: 58
Request 3: 40
Request 4: 18
Request 5: 90
Request 6: 160
Request 7: 150
Request 8: 12
Enter the initial disk head position: 50
Enter the direction of head movement (1 for right, 0 for left): 1
LOOK Disk Scheduling:
Seek Sequence: 50 55 58 90 150 160 40 18 12
Total Seek Time: 258
```