EXPERIMENT NO. 1

Study of Basic Commands in Linux Operating System

File and Directory Management

`ls`

- Description: Lists files and directories in the current directory.
- Usage: `ls [options] [directory]`
- Examples:
  - `ls` - List files and directories.
  - `ls -l` - List with detailed information.
  - `ls -a` - List all files, including hidden ones.

`cd`

- Description: Changes the current directory.
- Usage: `cd [directory]`
- Examples:
  - `cd /home/user` - Change to `/home/user` directory.
  - `cd ..` - Move up one directory.
  - `cd ~` - Move to the home directory.

`pwd`

- Description: Prints the current working directory.
- Usage: `pwd`
- Examples:
  - `pwd` - Display the full path of the current directory.

`mkdir`

- Description: Creates a new directory.
- Usage: `mkdir [options] directory_name`

- Examples:
  - `mkdir new_folder` - Create a directory named `new_folder`.
  - `mkdir -p parent_folder/child_folder` - Create parent and child directories.

`rmdir`

- Description: Removes an empty directory.
- Usage: `rmdir directory_name`
- Examples:
  - `rmdir old_folder` - Remove the empty directory `old_folder`.

`rm`

- Description: Removes files or directories.
- Usage: `rm [options] file_name`
- Examples:
  - `rm file.txt` - Remove `file.txt`.
  - `rm -r folder_name` - Remove `folder_name` and its contents recursively.

`cp`

- Description: Copies files or directories.
- Usage: `cp [options] source destination`
- Examples:
  - `cp file1.txt file2.txt` - Copy `file1.txt` to `file2.txt`.
  - `cp -r dir1 dir2` - Copy `dir1` to `dir2` recursively.

`mv`

- Description: Moves or renames files or directories.
- Usage: `mv [options] source destination`
- Examples:
  - `mv file1.txt /home/user/` - Move `file1.txt` to `/home/user/`.
  - `mv old_name.txt new_name.txt` - Rename `old_name.txt` to `new_name.txt`.

File Viewing and Editing

`cat`

- Description: Concatenates and displays file content.
- Usage: `cat [options] file_name`
- Examples:
  - `cat file.txt` - Display the content of `file.txt`.
  - `cat file1.txt file2.txt` - Concatenate and display both files.

`less`

- Description: Views file content with backward and forward navigation.
- Usage: `less file_name`
- Examples:
  - `less file.txt` - View and navigate through `file.txt`.

`free`

- Description: Displays memory usage.
- Usage: `free [options]`
- Examples:
  - `free` - Show memory usage.
  - `free -h` - Show memory usage in human-readable format.

Process Management

`ps`

- Description: Displays information about active processes.
- Usage: `ps [options]`
- Examples:
  - `ps` - Display processes for the current user.
  - `ps aux` - Display detailed information about all processes.

Networking

`ping`

- Description: Tests network connectivity to a host.
- Usage: `ping [options] destination`
- Examples:
  - `ping google.com` - Send ICMP packets to `google.com`.

`ifconfig`

- Description: Displays or configures network interfaces.
- Usage: `ifconfig [interface] [options]`
- Examples:
  - `ifconfig` - Display network interfaces and their details.

`netstat`

- Description: Displays network connections and routing tables.
- Usage: `netstat [options]`
- Examples:
  - `netstat -tuln` - Display listening ports and network connections.

User Management

`whoami`

- Description: Displays the currently logged-in user.
- Usage: `whoami`
- Examples:
  - `whoami` - Show the username of the current user.

`adduser`

- Description: Adds a new user to the system.
- Usage: `adduser username`
- Examples:
  - `adduser newuser` - Add a new user named `newuser`.

File Permissions

`chmod`

- Description: Changes file permissions.
- Usage: `chmod [options] mode file_name`
- Examples:
  - `chmod 755 file.txt` - Set permissions of `file.txt` to `rwxr-xr-x`.

Experiment No.2

Aim:

Write a program to implement Thread and Process

Theory:

A thread is a lightweight process sharing memory with other threads in the same process, while a process is an independent program with its own memory space. Threads enable concurrent execution within a process, and process management involves handling separate program instances

Code

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

void *thread_function(void *arg) {
  printf("Thread: Running in thread\n");
  return NULL;
}

void create_thread() {
  pthread_t thread;
  if (pthread_create(&thread, NULL, thread_function, NULL)) {
    printf("Error creating thread\n");
    return;
  }
  pthread_join(thread, NULL);
  printf("Thread: Finished execution\n");
}

void create_process() {
  pid_t pid = fork();

  if (pid == 0) {
    printf("Process: This is the child process\n");
  } else if (pid > 0) {
    printf("Process: This is the parent process\n");
  } else {
    printf("Process: Fork failed\n");
  }
}

int main() {
  printf("Main: Starting thread and process examples\n");
  create_thread();
  create_thread();
  create_process();
  return 0;
}
```

Experiment NO.3

Aim:

Write a C program to implement the producer-consumer problem using Semaphores

Theory:

The producer-consumer problem involves two types of processes: producers that generate data and consumers that process it. Semaphores are used to synchronize access to a shared buffer, preventing race conditions and ensuring that producers and consumers operate efficiently without conflicts.

Code

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

#define BUFFER_SIZE 10
#define PRODUCER_COUNT 1
#define CONSUMER_COUNT 1
#define PRODUCE_COUNT 20

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t empty;
sem_t full;
pthread_mutex_t mutex;

void *producer(void *arg) {
  for (int i = 0; i < PRODUCE_COUNT; i++) {
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

    buffer[in] = i;
    printf("Produced: %d ", buffer[in]);
    in = (in + 1) % BUFFER_SIZE;

    pthread_mutex_unlock(&mutex);
    sem_post(&full);

    usleep(rand() % 100000); // Simulate variable production time
  }
  return NULL;
}

void *consumer(void *arg) {
```

```c
  for (int i = 0; i < PRODUCE_COUNT; i++) {
    sem_wait(&full);
    pthread_mutex_lock(&mutex);

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

    pthread_mutex_unlock(&mutex);
    sem_post(&empty);

    usleep(rand() % 1000000); // Simulate variable consumption time
  }
  return NULL;
}

int main() {
  pthread_t producers[PRODUCER_COUNT];
  pthread_t consumers[CONSUMER_COUNT];

  sem_init(&empty, 0, BUFFER_SIZE);
  sem_init(&full, 0, 0);
  pthread_mutex_init(&mutex, NULL);

  for (int i = 0; i < PRODUCER_COUNT; i++) {
    pthread_create(&producers[i], NULL, producer, NULL);
  }

  for (int i = 0; i < CONSUMER_COUNT; i++) {
    pthread_create(&consumers[i], NULL, consumer, NULL);
  }

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

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

  sem_destroy(&empty);
  sem_destroy(&full);
  pthread_mutex_destroy(&mutex);

  return 0;
}
```

Experiment NO.4

Aim:

Write a C program to simulate the concept of the Dining Philosophers problem

Theory:

The Dining Philosophers problem is a classic synchronization problem where philosophers alternate between thinking and eating from a shared set of forks. The challenge is to avoid deadlock and ensure that each philosopher gets a chance to eat without resource starvation.

Code

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

Experiment NO.5

Aim:

To simulate and calculate the turnaround time and waiting time for the following non-preemptive CPU scheduling algorithms:
a) FCFS (First-Come, First-Served)
b) SJF (Shortest Job First)
c) Priority Scheduling

Theory:

FCFS (First-Come, First-Served):
Description: Processes are executed in the order they arrive. The process that arrives first gets executed first.
Turnaround Time: The total time taken from arrival to completion of the process.
Waiting Time: The total time a process spends waiting in the queue before getting executed.

SJF (Shortest Job First):
Description: The process with the shortest burst time (execution time) is executed next. This algorithm can be either preemptive or non-preemptive, but here we'll consider the non-preemptive version.
Turnaround Time: Time from the arrival of the process to its completion.
Waiting Time: The total time a process spends waiting in the queue before it starts executing.

Priority Scheduling:
Description: Each process is assigned a priority. The process with the highest priority (typically the smallest number) is executed first. In case of a tie, FCFS is used.
Turnaround Time: Time from the arrival of the process to its completion.
Waiting Time: The total time a process spends waiting in the queue before it starts executing.

Algorithm:

FCFS Scheduling:
1.1 Read the number of processes.
1.2 Read the arrival and burst times for each process.
1.3 Calculate the completion, turnaround, and waiting times:

Completion Time (CT) = Arrival Time + Burst Time
Turnaround Time (TAT) = Completion Time - Arrival Time
Waiting Time (WT) = Turnaround Time - Burst Time

SJF Scheduling:
2.1 Read the number of processes.
2.2 Read the arrival and burst times for each process.
2.3 Sort processes by burst time.
2.4 Calculate completion, turnaround, and waiting times:
Completion Time (CT) is computed based on the sorted order.
Turnaround Time (TAT) = Completion Time - Arrival Time
Waiting Time (WT) = Turnaround Time - Burst Time

Priority Scheduling:
3.1 Read the number of processes.
3.2 Read the arrival time, burst time, and priority for each process.
3.3 Sort processes by priority (lower number indicates higher priority).
3.4 Calculate completion, turnaround, and waiting times:
Completion Time (CT) is computed based on the priority order.
Turnaround Time (TAT) = Completion Time - Arrival Time
Waiting Time (WT) = Turnaround Time - Burst Time

Code

```c
#include <stdio.h>

#define MAX 10

typedef struct {
  int id;
  int arrival;
  int burst;
  int priority;
  int waiting;
  int turnaround;
} Process;

void swap(Process *a, Process *b) {
  Process temp = *a;
  *a = *b;
  *b = temp;
}

void sortByArrival(Process proc[], int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
      if (proc[i].arrival > proc[j].arrival) {
        swap(&proc[i], &proc[j]);
      }
    }
  }
}
```

```c
void sortByBurst(Process proc[], int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
      if (proc[i].burst > proc[j].burst) {
        swap(&proc[i], &proc[j]);
      }
    }
  }
}

void sortByPriority(Process proc[], int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
      if (proc[i].priority > proc[j].priority) {
        swap(&proc[i], &proc[j]);
      }
    }
  }
}

void FCFS(Process proc[], int n) {
  int total_wt = 0, total_tat = 0;
  proc[0].waiting = 0;
  proc[0].turnaround = proc[0].burst;

  for (int i = 1; i < n; i++) {
    proc[i].waiting = proc[i - 1].waiting + proc[i - 1].burst - proc[i].arrival;
    if (proc[i].waiting < 0)
      proc[i].waiting = 0;
    proc[i].turnaround = proc[i].waiting + proc[i].burst;
    total_wt += proc[i].waiting;
    total_tat += proc[i].turnaround;
  }

  printf("\nFCFS Scheduling:\n");
  for (int i = 0; i < n; i++) {
    printf("Process %d: Waiting Time = %d, Turnaround Time = %d\n", proc[i].id,
           proc[i].waiting, proc[i].turnaround);
  }
  printf("Average Waiting Time = %.2f\n", (float)total_wt / n);
  printf("Average Turnaround Time = %.2f\n", (float)total_tat / n);
}

void SJF(Process proc[], int n) {
  int total_wt = 0, total_tat = 0;
  int time = 0;
  proc[0].waiting = 0;
  proc[0].turnaround = proc[0].burst;

  for (int i = 1; i < n; i++) {
    proc[i].waiting = time - proc[i].arrival;
    if (proc[i].waiting < 0)
      proc[i].waiting = 0;
    proc[i].turnaround = proc[i].waiting + proc[i].burst;
    total_wt += proc[i].waiting;
    total_tat += proc[i].turnaround;
    time += proc[i].burst;
  }

  printf("\nSJF Scheduling:\n");
  for (int i = 0; i < n; i++) {
    printf("Process %d: Waiting Time = %d, Turnaround Time = %d\n", proc[i].id,
           proc[i].waiting, proc[i].turnaround);
  }
```

```
  printf("Average Waiting Time = %.2f\n", (float)total_wt / n);
  printf("Average Turnaround Time = %.2f\n", (float)total_tat / n);
}

void PriorityScheduling(Process proc[], int n) {
  int total_wt = 0, total_tat = 0;
  int time = 0;
  proc[0].waiting = 0;
  proc[0].turnaround = proc[0].burst;

  for (int i = 1; i < n; i++) {
    proc[i].waiting = time - proc[i].arrival;
    if (proc[i].waiting < 0)
      proc[i].waiting = 0;
    proc[i].turnaround = proc[i].waiting + proc[i].burst;
    total_wt += proc[i].waiting;
    total_tat += proc[i].turnaround;
    time += proc[i].burst;
  }

  printf("\nPriority Scheduling:\n");
  for (int i = 0; i < n; i++) {
    printf("Process %d: Waiting Time = %d, Turnaround Time = %d\n", proc[i].id,
           proc[i].waiting, proc[i].turnaround);
  }
  printf("Average Waiting Time = %.2f\n", (float)total_wt / n);
  printf("Average Turnaround Time = %.2f\n", (float)total_tat / n);
}

int main() {
  Process proc[MAX];
  int n;

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

  for (int i = 0; i < n; i++) {
    printf("Enter arrival time, burst time, and priority for process %d: ",
           i + 1);
    proc[i].id = i + 1;
    scanf("%d %d %d", &proc[i].arrival, &proc[i].burst, &proc[i].priority);
  }

  sortByArrival(proc, n);
  FCFS(proc, n);

  sortByArrival(proc, n);
  sortByBurst(proc, n);
  SJF(proc, n);

  sortByArrival(proc, n);
  sortByPriority(proc, n);
  PriorityScheduling(proc, n);

  return 0;
}
```

Experiment NO.6

Aim:

To simulate the following preemptive CPU scheduling algorithms to find turnaround time and waiting time. a) Round Robin To simulate and calculate the turnaround time and waiting time for the following preemptive CPU scheduling algorithms:
a) Round Robin
b) Priority

b) Priority

Theory:

Round Robin:
Description: Each process is assigned a fixed time quantum (or time slice). Processes are executed in a circular order, with each process getting a chance to execute for the duration of the time quantum. If a process does not complete within its time quantum, it is placed at the end of the queue to wait for its next turn.
Turnaround Time: Time from the arrival of the process to its completion.
Waiting Time: The total time a process spends waiting in the queue before it gets executed.

Priority Scheduling:
Description: Each process is assigned a priority. The process with the highest priority (typically the smallest number) is executed first. In preemptive priority scheduling, if a new process arrives with a higher priority than the currently running process, the current process is preempted, and the new process is executed.
Turnaround Time: Time from the arrival of the process to its completion.
Waiting Time: The total time a process spends waiting in the queue before it starts executing.

Algorithm:

Round Robin Scheduling:
1.1 Read the number of processes and the time quantum.
1.2 Read the arrival and burst times for each process.
1.3 Initialize the queue and start processing each process for

the duration of the time quantum.
1.4 If a process's burst time is greater than the time quantum, update the burst time and move it to the end of the queue.
1.5 Calculate completion, turnaround, and waiting times:
    Completion Time (CT) is updated as the process finishes execution.
    Turnaround Time (TAT) = Completion Time - Arrival Time
    Waiting Time (WT) = Turnaround Time - Burst Time

Priority Scheduling:
2.1 Read the number of processes.
2.2 Read the arrival time, burst time, and priority for each process.
2.3 Sort processes by priority (lower number indicates higher priority).
2.4 Implement the preemptive scheduling by maintaining a priority queue and preempting the current process if a new process with higher priority arrives.
2.5 Calculate completion, turnaround, and waiting times:
    Completion Time (CT) is updated based on process execution and preemption.
    Turnaround Time (TAT) = Completion Time - Arrival Time
    Waiting Time (WT) = Turnaround Time - Burst Time

Code

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

#define MAX 10

typedef struct {
  int id;
  int arrival_time;
  int burst_time;
  int priority;
  int remaining_time;
  int turnaround_time;
  int waiting_time;
} Process;

void round_robin(Process processes[], int n, int quantum) {
  int time = 0, i, flag;
  int remaining_processes = n;

  while (remaining_processes > 0) {
    flag = 0;
    for (i = 0; i < n; i++) {
      if (processes[i].arrival_time <= time &&
          processes[i].remaining_time > 0) {
        flag = 1;
        if (processes[i].remaining_time <= quantum) {
          time += processes[i].remaining_time;
          processes[i].turnaround_time = time - processes[i].arrival_time;
          processes[i].waiting_time =
```

```
            processes[i].turnaround_time - processes[i].burst_time;
          processes[i].remaining_time = 0;
          remaining_processes--;
        } else {
          time += quantum;
          processes[i].remaining_time -= quantum;
        }
      }
    }
    if (flag == 0) {
      time++;
    }
  }
}

void priority_scheduling(Process processes[], int n) {
  int time = 0, i, min_priority_index;
  int completed = 0;

  while (completed < n) {
    min_priority_index = -1;
    for (i = 0; i < n; i++) {
      if (processes[i].arrival_time <= time &&
          processes[i].remaining_time > 0) {
        if (min_priority_index == -1 ||
            processes[i].priority < processes[min_priority_index].priority) {
          min_priority_index = i;
        }
      }
    }
    if (min_priority_index != -1) {
      processes[min_priority_index].remaining_time--;
      time++;
      if (processes[min_priority_index].remaining_time == 0) {
        processes[min_priority_index].turnaround_time =
            time - processes[min_priority_index].arrival_time;
        processes[min_priority_index].waiting_time =
            processes[min_priority_index].turnaround_time -
            processes[min_priority_index].burst_time;
        completed++;
      }
    } else {
      time++;
    }
  }
}

void print_results(Process processes[], int n, const char *algorithm) {
  int i;
  float total_turnaround_time = 0, total_waiting_time = 0;
  printf("%s Scheduling Results:\n", algorithm);
  printf("ID\tArrival\tBurst\tPriority\tTurnaround\tWaiting\n");
  for (i = 0; i < n; i++) {
    total_turnaround_time += processes[i].turnaround_time;
    total_waiting_time += processes[i].waiting_time;
    printf("%d\t%d\t%d\t%d\t\t%d\t\t%d\n", processes[i].id,
           processes[i].arrival_time, processes[i].burst_time,
           processes[i].priority, processes[i].turnaround_time,
           processes[i].waiting_time);
  }
  printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
  printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}
```

```c
int main() {
  int n, i, quantum;
  Process processes[MAX];

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

  for (i = 0; i < n; i++) {
    printf("Enter arrival time, burst time, and priority for process %d: ",
           i + 1);
    processes[i].id = i + 1;
    scanf("%d %d %d", &processes[i].arrival_time, &processes[i].burst_time,
          &processes[i].priority);
    processes[i].remaining_time = processes[i].burst_time;
    processes[i].turnaround_time = 0;
    processes[i].waiting_time = 0;
  }

  printf("Enter time quantum for Round Robin: ");
  scanf("%d", &quantum);

  Process rr_processes[MAX];
  Process pr_processes[MAX];

  for (i = 0; i < n; i++) {
    rr_processes[i] = processes[i];
    pr_processes[i] = processes[i];
  }

  round_robin(rr_processes, n, quantum);
  print_results(rr_processes, n, "Round Robin");

  priority_scheduling(pr_processes, n);
  print_results(pr_processes, n, "Priority");

  return 0;
}
```

Experiment NO.7

Aim:

To simulate the Banker's Algorithm to ensure safe allocation of resources and avoid deadlocks in a system with multiple processes and resource types.

Theory:

Banker's Algorithm:
Description: The Banker's Algorithm is used to allocate resources to processes in a way that ensures the system is always in a safe state. It checks for safe states using a request matrix and allocation matrix, which track resource allocations and requests. The algorithm calculates if resource allocation to processes can lead to a safe sequence, where all processes can complete without causing a deadlock.
Safe State: A state is considered safe if there exists a sequence of processes that allows all of them to complete successfully without causing a deadlock. The Banker's Algorithm ensures that every resource request can be granted while maintaining this safe state.
Request Matrix: Represents the maximum demand of each process for each resource.
Allocation Matrix: Represents the current allocation of resources to each process.
Available Resources: Represents the resources available in the system that are not currently allocated.

Algorithm:

Banker's Algorithm:
1.1 Initialize matrices for available resources, maximum demands, and current allocations.
1.2 For each resource request by a process:
1.2.1 Check if the request is less than or equal to the process's maximum demand.
1.2.2 Check if the request is less than or equal to the available resources.
1.2.3 Temporarily allocate the requested resources and check if the system is in a safe state by simulating the execution of processes with the new allocation.
1.2.4 If the system is in a safe state, grant the request.

Otherwise, revert to the previous state and deny the request.
1.3 Repeat the above steps for all resource requests and process executions.

Code

```c
#include <stdio.h>

#define MAX 10
#define RESOURCE_TYPES 3

void calculate_need(int need[MAX][RESOURCE_TYPES], int max[MAX][RESOURCE_TYPES],
                    int allot[MAX][RESOURCE_TYPES], int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < RESOURCE_TYPES; j++) {
      need[i][j] = max[i][j] - allot[i][j];
    }
  }
}

int is_safe(int processes[], int n, int m, int available[],
            int max[][RESOURCE_TYPES], int allot[][RESOURCE_TYPES]) {
  int need[MAX][RESOURCE_TYPES];
  calculate_need(need, max, allot, n);

  int work[RESOURCE_TYPES];
  int finish[MAX];

  for (int i = 0; i < m; i++) {
    work[i] = available[i];
  }

  for (int i = 0; i < n; i++) {
    finish[i] = 0;
  }

  int safeSeq[MAX];
  int count = 0;

  while (count < n) {
    int found = 0;
    for (int p = 0; p < n; p++) {
      if (!finish[p]) {
        int j;
        for (j = 0; j < m; j++) {
          if (need[p][j] > work[j]) {
            break;
          }
        }
        if (j == m) {
          for (int k = 0; k < m; k++) {
            work[k] += allot[p][k];
          }
          safeSeq[count++] = p;
          finish[p] = 1;
          found = 1;
        }
      }
    }
    if (!found) {
      printf("System is not in a safe state.\n");
      return 0;
```

```
    }
  }

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

int main() {
  int n, m;
  int processes[MAX];
  int available[RESOURCE_TYPES];
  int max[MAX][RESOURCE_TYPES];
  int allot[MAX][RESOURCE_TYPES];

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

  printf("Enter the number of available instances for each resource:\n");
  for (int i = 0; i < m; i++) {
    printf("Resource %d: ", i + 1);
    scanf("%d", &available[i]);
  }

  printf("Enter maximum matrix:\n");
  for (int i = 0; i < n; i++) {
    processes[i] = i;
    printf("Process %d:\n", i + 1);
    for (int j = 0; j < m; j++) {
      printf("Maximum resource %d: ", j + 1);
      scanf("%d", &max[i][j]);
    }
  }

  printf("Enter allocation matrix:\n");
  for (int i = 0; i < n; i++) {
    printf("Process %d:\n", i + 1);
    for (int j = 0; j < m; j++) {
      printf("Allocated resource %d: ", j + 1);
      scanf("%d", &allot[i][j]);
    }
  }

  is_safe(processes, n, m, available, max, allot);

  return 0;
}
```

Experiment NO.8

Aim:

Aim:
To simulate the Banker's Algorithm for deadlock prevention, ensuring that resources are allocated in a way that prevents the system from entering an unsafe state that could lead to a deadlock.

Theory:

Banker's Algorithm:
Description: The Banker's Algorithm is used to prevent deadlocks by ensuring that resource allocations are made in such a way that the system always remains in a safe state. This involves checking whether a resource allocation will result in a safe or unsafe state before granting the request. If granting a request results in a safe state, the request is granted; otherwise, it is denied.
Safe State: A system is in a safe state if there is a sequence of processes that allows all processes to complete without causing a deadlock. The algorithm ensures that resource allocations do not lead to a state where no such sequence exists.
Request Matrix: Represents the maximum resources each process may need.
Allocation Matrix: Represents the current allocation of resources to each process.
Available Resources: Represents the resources currently available in the system.

Algorithm:

Banker's Algorithm for Deadlock Prevention:
1.1 Initialize the matrices for available resources, maximum demands, and current allocations.
1.2 For each resource request by a process:
1.2.1 Check if the request is less than or equal to the process's maximum demand.
1.2.2 Check if the request is less than or equal to the available resources.
1.2.3 Temporarily allocate the requested resources and check if the system remains in a safe state using a safety algorithm:
1.2.3.1 Calculate the new available resources, allocation matrix, and need matrix.

1.2.3.2 Use a safety algorithm to determine if there exists a safe sequence.
1.2.4 If the system is in a safe state, grant the request. Otherwise, revert to the previous state and deny the request.
1.3 Repeat the above steps for all resource requests and process executions.

Code

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

#define MAX 10
#define RESOURCE_TYPES 3

void calculate_need(int need[MAX][RESOURCE_TYPES], int max[MAX][RESOURCE_TYPES],
                    int allot[MAX][RESOURCE_TYPES], int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < RESOURCE_TYPES; j++) {
      need[i][j] = max[i][j] - allot[i][j];
    }
  }
}

bool is_safe(int processes[], int n, int m, int available[],
             int max[][RESOURCE_TYPES], int allot[][RESOURCE_TYPES],
             int need[MAX][RESOURCE_TYPES]) {
  int work[RESOURCE_TYPES];
  bool finish[MAX];

  for (int i = 0; i < m; i++) {
    work[i] = available[i];
  }

  for (int i = 0; i < n; i++) {
    finish[i] = false;
  }

  int safeSeq[MAX];
  int count = 0;

  while (count < n) {
    bool found = false;
    for (int p = 0; p < n; p++) {
      if (!finish[p]) {
        int j;
        for (j = 0; j < m; j++) {
          if (need[p][j] > work[j]) {
            break;
          }
        }
        if (j == m) {
          for (int k = 0; k < m; k++) {
            work[k] += allot[p][k];
          }
          safeSeq[count++] = p;
          finish[p] = true;
          found = true;
        }
      }
    }
```

```
    if (!found) {
      return false;
    }
  }

  printf("System is in a safe state.\nSafe sequence is: ");
  for (int i = 0; i < n; i++) {
    printf("P%d ", safeSeq[i]);
  }
  printf("\n");
  return true;
}

void request_resources(int processes[], int n, int m, int available[],
                       int max[][RESOURCE_TYPES], int allot[][RESOURCE_TYPES],
                       int need[MAX][RESOURCE_TYPES]) {
  int request[MAX][RESOURCE_TYPES];
  int i, j, process_id;

  printf("Enter process number making request: ");
  scanf("%d", &process_id);

  if (process_id < 0 || process_id >= n) {
    printf("Invalid process number.\n");
    return;
  }

  printf("Enter request for resources:\n");
  for (i = 0; i < m; i++) {
    printf("Resource %d: ", i + 1);
    scanf("%d", &request[process_id][i]);
  }

  for (i = 0; i < m; i++) {
    if (request[process_id][i] > need[process_id][i]) {
      printf("Request exceeds maximum claim.\n");
      return;
    }
  }

  for (i = 0; i < m; i++) {
    if (request[process_id][i] > available[i]) {
      printf("Resources not available.\n");
      return;
    }
  }

  for (i = 0; i < m; i++) {
    available[i] -= request[process_id][i];
    allot[process_id][i] += request[process_id][i];
    need[process_id][i] -= request[process_id][i];
  }

  if (is_safe(processes, n, m, available, max, allot, need)) {
    printf("Request granted.\n");
  } else {
    printf("Request cannot be granted due to unsafe state. Rolling back.\n");

    for (i = 0; i < m; i++) {
      available[i] += request[process_id][i];
      allot[process_id][i] -= request[process_id][i];
      need[process_id][i] += request[process_id][i];
    }
  }
```

```c
}

int main() {
  int n, m;
  int processes[MAX];
  int available[RESOURCE_TYPES];
  int max[MAX][RESOURCE_TYPES];
  int allot[MAX][RESOURCE_TYPES];
  int need[MAX][RESOURCE_TYPES];

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

  printf("Enter the number of available instances for each resource:\n");
  for (int i = 0; i < m; i++) {
    printf("Resource %d: ", i + 1);
    scanf("%d", &available[i]);
  }

  printf("Enter maximum matrix:\n");
  for (int i = 0; i < n; i++) {
    processes[i] = i;
    printf("Process %d:\n", i + 1);
    for (int j = 0; j < m; j++) {
      printf("Maximum resource %d: ", j + 1);
      scanf("%d", &max[i][j]);
    }
  }

  printf("Enter allocation matrix:\n");
  for (int i = 0; i < n; i++) {
    printf("Process %d:\n", i + 1);
    for (int j = 0; j < m; j++) {
      printf("Allocated resource %d: ", j + 1);
      scanf("%d", &allot[i][j]);
    }
  }

  calculate_need(need, max, allot, n);

  request_resources(processes, n, m, available, max, allot, need);

  return 0;
}
```

Experiment NO.9

Aim:

Program to simulate the MVT and MFT memory management techniques

Theory:

MVT (Multi-Programming with Variable Partitions): MVT is a memory management technique where the memory is divided into variable-sized partitions. Each partition is allocated to a process based on its needs. The partitions can grow or shrink dynamically as processes are loaded or terminated, allowing more efficient use of memory and accommodating varying process sizes.

MFT (Multi-Programming with Fixed Partitions): MFT divides memory into fixed-size partitions. Each partition is allocated to a process, and processes are limited to the partition size. This method simplifies memory management but can lead to internal fragmentation if a process does not fully use its allocated partition.

Code

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

#define TOTAL_MEMORY 100
#define MFT_BLOCKS 50

void printMemory(int mft[], int mvt[], int mft_size, int mvt_size) {
  printf("MFT Memory: ");
  for (int i = 0; i < mft_size; i++) {
    printf("%d ", mft[i]);
  }
  printf("\n");

  printf("MVT Memory: ");
  for (int i = 0; i < mvt_size; i++) {
    if (mvt[i] == -1) {
      printf(". ");
    } else {
      printf("%d ", mvt[i]);
    }
  }
  printf("\n");
}

int allocateMFT(int mft[], int task_size, int mft_size) {
  for (int i = 0; i <= mft_size - task_size; i++) {
    bool fit = true;
    for (int j = i; j < i + task_size; j++) {
```

```
      if (mft[j] != 0) {
        fit = false;
        break;
      }
    }
    if (fit) {
      for (int j = i; j < i + task_size; j++) {
        mft[j] = 1;
      }
      return i;
    }
  }
  return -1;
}

void deallocateMFT(int mft[], int start, int task_size) {
  for (int i = start; i < start + task_size; i++) {
    mft[i] = 0;
  }
}

int allocateMVT(int mvt[], int task_size, int mvt_size) {
  for (int i = 0; i <= mvt_size - task_size; i++) {
    bool fit = true;
    for (int j = i; j < i + task_size; j++) {
      if (mvt[j] != -1) {
        fit = false;
        break;
      }
    }
    if (fit) {
      for (int j = i; j < i + task_size; j++) {
        mvt[j] = 1;
      }
      return i;
    }
  }
  return -1;
}

void deallocateMVT(int mvt[], int start, int task_size) {
  for (int i = start; i < start + task_size; i++) {
    mvt[i] = -1;
  }
}

int main() {
  int mft[MFT_BLOCKS] = {0};
  int mvt[TOTAL_MEMORY] = {-1};

  printf("Initial State:\n");
  printMemory(mft, mvt, MFT_BLOCKS, TOTAL_MEMORY);

  printf("\nAllocating tasks:\n");
  int task_size = 10;
  printf("Allocating task of size %d in MFT\n", task_size);
  int start = allocateMFT(mft, task_size, MFT_BLOCKS);
  if (start != -1) {
    printf("Task allocated at index %d\n", start);
  } else {
    printf("Failed to allocate task in MFT\n");
  }

  printf("Allocating task of size %d in MVT\n", task_size);
```

```
  start = allocateMVT(mvt, task_size, TOTAL_MEMORY);
  if (start != -1) {
    printf("Task allocated at index %d\n", start);
  } else {
    printf("Failed to allocate task in MVT\n");
  }

  printf("\nCurrent State:\n");
  printMemory(mft, mvt, MFT_BLOCKS, TOTAL_MEMORY);

  printf("\nDeallocating tasks:\n");
  printf("Deallocating task of size %d from MFT starting at index %d\n",
         task_size, 0);
  deallocateMFT(mft, 0, task_size);

  printf("Deallocating task of size %d from MVT starting at index %d\n",
         task_size, 0);
  deallocateMVT(mvt, 0, task_size);

  printf("\nFinal State:\n");
  printMemory(mft, mvt, MFT_BLOCKS, TOTAL_MEMORY);

  return 0;
}
```

Experiment NO.10

Aim:

To simulate the paging technique of memory management

Theory:

Paging: Paging is a memory management technique that divides memory into fixed-size blocks called pages. Processes are also divided into pages of the same size. When a process is executed, its pages are loaded into any available memory frames. This allows for efficient use of memory and helps in managing processes that are larger than the available physical memory. Paging eliminates external fragmentation and simplifies memory allocation.

Code

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

#define PAGE_SIZE 256
#define PHYSICAL_MEM_SIZE 1024
#define LOGICAL_MEM_SIZE 2048
#define NUM_FRAMES (PHYSICAL_MEM_SIZE / PAGE_SIZE)
#define NUM_PAGES (LOGICAL_MEM_SIZE / PAGE_SIZE)

int physical_memory[PHYSICAL_MEM_SIZE];
int page_table[NUM_PAGES];

void initialize_page_table() {
  for (int i = 0; i < NUM_PAGES; i++) {
    page_table[i] = -1;
  }
}

int allocate_frame(int page_number) {
  int frame_number = rand() % NUM_FRAMES;
  page_table[page_number] = frame_number;
  return frame_number;
}

int translate_address(int logical_address) {
  int page_number = logical_address / PAGE_SIZE;
  int offset = logical_address % PAGE_SIZE;

  int frame_number = page_table[page_number];
  if (frame_number == -1) {
    frame_number = allocate_frame(page_number);
  }
```

```c
  return frame_number * PAGE_SIZE + offset;
}

void write_memory(int logical_address, int value) {
  int physical_address = translate_address(logical_address);
  physical_memory[physical_address] = value;
}

int read_memory(int logical_address) {
  int physical_address = translate_address(logical_address);
  return physical_memory[physical_address];
}

int main() {
  srand(time(0));
  initialize_page_table();

  write_memory(500, 42);
  write_memory(1200, 84);

  printf("Value at logical address 500: %d\n", read_memory(500));
  printf("Value at logical address 1200: %d\n", read_memory(1200));

  return 0;
}
```

Experiment NO.11

Aim:

To simulate the FIRST-FIT contiguous memory allocation technique

Theory:

FIRST-FIT: The FIRST-FIT memory allocation technique allocates the first available block of memory that is large enough to satisfy the request. The memory is scanned from the beginning, and the first sufficiently sized block is chosen for allocation. This method is simple and fast but can lead to fragmentation over time.

Code

```c
#include <stdio.h>

#define MEMORY_SIZE 1000
#define NUM_BLOCKS 5

int memory[MEMORY_SIZE];
int block_size[NUM_BLOCKS] = {100, 200, 300, 150, 250};
int block_allocated[NUM_BLOCKS] = {0};

void allocate_memory(int process_size, int process_id) {
  for (int i = 0; i < NUM_BLOCKS; i++) {
    if (!block_allocated[i] && block_size[i] >= process_size) {
      block_allocated[i] = process_id;
      printf("Process %d allocated to block %d\n", process_id, i);
      return;
    }
  }
  printf("Process %d cannot be allocated\n", process_id);
}

void deallocate_memory(int process_id) {
  for (int i = 0; i < NUM_BLOCKS; i++) {
    if (block_allocated[i] == process_id) {
      block_allocated[i] = 0;
      printf("Process %d deallocated from block %d\n", process_id, i);
      return;
    }
  }
  printf("Process %d not found in memory\n", process_id);
}

int main() {
  allocate_memory(120, 1);
  allocate_memory(180, 2);
  allocate_memory(90, 3);
  allocate_memory(200, 4);
  allocate_memory(80, 5);
```

```
  deallocate_memory(2);

  allocate_memory(180, 6);

  return 0;
}
```

Experiment NO.12

Aim:

To simulate the BEST-FIT contiguous memory allocation technique

Theory:

BEST-FIT: The BEST-FIT memory allocation technique allocates the smallest available block of memory that is large enough to satisfy the request. The memory is searched for the block that most closely matches the size of the request, aiming to minimize wasted space. This method can reduce fragmentation but may require more time to find the best fit.

Code

```c
#include <stdio.h>

#define MEMORY_SIZE 1000
#define NUM_BLOCKS 5

int memory[MEMORY_SIZE];
int block_size[NUM_BLOCKS] = {100, 200, 300, 150, 250};
int block_allocated[NUM_BLOCKS] = {0};

void allocate_memory(int process_size, int process_id) {
  int best_index = -1;
  for (int i = 0; i < NUM_BLOCKS; i++) {
    if (!block_allocated[i] && block_size[i] >= process_size) {
      if (best_index == -1 || block_size[i] < block_size[best_index]) {
        best_index = i;
      }
    }
  }

  if (best_index != -1) {
    block_allocated[best_index] = process_id;
    printf("Process %d allocated to block %d\n", process_id, best_index);
  } else {
    printf("Process %d cannot be allocated\n", process_id);
  }
}

void deallocate_memory(int process_id) {
  for (int i = 0; i < NUM_BLOCKS; i++) {
    if (block_allocated[i] == process_id) {
      block_allocated[i] = 0;
      printf("Process %d deallocated from block %d\n", process_id, i);
      return;
    }
  }
  printf("Process %d not found in memory\n", process_id);
}
```

```c
int main() {
  allocate_memory(120, 1);
  allocate_memory(180, 2);
  allocate_memory(90, 3);
  allocate_memory(200, 4);
  allocate_memory(80, 5);

  deallocate_memory(2);

  allocate_memory(180, 6);

  return 0;
}
```

Experiment NO.13

Aim:

To simulate the WORST-FIT contiguous memory allocation technique

Theory:

WORST-FIT: The WORST-FIT memory allocation technique allocates the largest available block of memory to a process. By choosing the biggest block, it aims to leave significant free space for future allocations. However, this can lead to higher fragmentation compared to other techniques like FIRST-FIT and BEST-FIT.

Code

```c
#include <stdio.h>

#define MEMORY_SIZE 1000
#define NUM_BLOCKS 5

int memory[MEMORY_SIZE];
int block_size[NUM_BLOCKS] = {100, 200, 300, 150, 250};
int block_allocated[NUM_BLOCKS] = {0};

void allocate_memory(int process_size, int process_id) {
  int worst_index = -1;
  for (int i = 0; i < NUM_BLOCKS; i++) {
    if (!block_allocated[i] && block_size[i] >= process_size) {
      if (worst_index == -1 || block_size[i] > block_size[worst_index]) {
        worst_index = i;
      }
    }
  }

  if (worst_index != -1) {
    block_allocated[worst_index] = process_id;
    printf("Process %d allocated to block %d\n", process_id, worst_index);
  } else {
    printf("Process %d cannot be allocated\n", process_id);
  }
}

void deallocate_memory(int process_id) {
  for (int i = 0; i < NUM_BLOCKS; i++) {
    if (block_allocated[i] == process_id) {
      block_allocated[i] = 0;
      printf("Process %d deallocated from block %d\n", process_id, i);
      return;
    }
  }
  printf("Process %d not found in memory\n", process_id);
}
```

```c
int main() {
  allocate_memory(120, 1);
  allocate_memory(180, 2);
  allocate_memory(90, 3);
  allocate_memory(200, 4);
  allocate_memory(80, 5);

  deallocate_memory(2);

  allocate_memory(180, 6);

  return 0;
}
```

Experiment NO.14

Aim:

To simulate the FIFO page replacement algorithm

Theory:

FIFO (First-In, First-Out): FIFO is a page replacement algorithm where the oldest page in memory is replaced when a new page needs to be loaded. Pages are placed in a queue, and the first page to enter is the first one removed when memory is full. This simple approach can lead to suboptimal performance, especially in cases where the oldest page is frequently used.

Algorithm:

1. Initialize: Maintain a queue to track the pages in memory.
2. Page Requests: For each page request, check if the page is already in memory.
3. Page Replacement: If the page is not in memory and memory is full, remove the page at the front of the queue (oldest) and load the new page.
4. Repeat: Continue this process for all incoming page requests.

Code

```c
#include <stdio.h>

#define MAX_FRAMES 3
#define MAX_PAGES 10

int frames[MAX_FRAMES], pages[MAX_PAGES];
int front = 0, rear = 0, num_frames_filled = 0;

int is_page_in_frames(int page) {
  for (int i = 0; i < MAX_FRAMES; i++) {
    if (frames[i] == page) {
      return 1;
    }
  }
  return 0;
}

void print_frames() {
  for (int i = 0; i < MAX_FRAMES; i++) {
    if (frames[i] == -1) {
      printf("- ");
```

```
    } else {
      printf("%d ", frames[i]);
    }
  }
  printf("\n");
}

void fifo_page_replacement(int num_pages) {
  for (int i = 0; i < num_pages; i++) {
    if (!is_page_in_frames(pages[i])) {
      if (num_frames_filled < MAX_FRAMES) {
        frames[rear] = pages[i];
        rear = (rear + 1) % MAX_FRAMES;
        num_frames_filled++;
      } else {
        frames[front] = pages[i];
        front = (front + 1) % MAX_FRAMES;
        rear = (rear + 1) % MAX_FRAMES;
      }
      print_frames();
    }
  }
}

int main() {
  int num_pages;

  printf("Enter number of pages: ");
  scanf("%d", &num_pages);

  printf("Enter the page sequence: ");
  for (int i = 0; i < num_pages; i++) {
    scanf("%d", &pages[i]);
  }

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

  fifo_page_replacement(num_pages);

  return 0;
}
```

Experiment NO.15

Aim:

To simulate the LRU page replacement algorithm.

Theory:

LRU (Least Recently Used): The LRU page replacement algorithm replaces the page that has not been used for the longest period of time. By tracking the order in which pages are accessed, the algorithm ensures that the least recently used page is removed when a new page needs to be loaded. This approach generally leads to better performance than FIFO, as it more closely mimics actual memory usage patterns.

Algorithm:

1. Initialize: Maintain a list or stack to track the order of page accesses.
2. Page Requests: For each page request, check if the page is already in memory.
3. Update List: If the page is in memory, move it to the most recent position in the list.
4. Page Replacement: If the page is not in memory and memory is full, remove the least recently used page (the oldest in the list) and load the new page.
5. Repeat: Continue this process for all incoming page requests.

Code

```c
#include <stdio.h>

#define MAX_FRAMES 3
#define MAX_PAGES 10

int frames[MAX_FRAMES], pages[MAX_PAGES];
int page_age[MAX_FRAMES];
int num_pages;

void initialize_frames() {
  for (int i = 0; i < MAX_FRAMES; i++) {
    frames[i] = -1;
    page_age[i] = 0;
  }
}
```

```c
void print_frames() {
  for (int i = 0; i < MAX_FRAMES; i++) {
    if (frames[i] == -1) {
      printf("- ");
    } else {
      printf("%d ", frames[i]);
    }
  }
  printf("\n");
}

int find_lru_index() {
  int min_age = page_age[0];
  int lru_index = 0;

  for (int i = 1; i < MAX_FRAMES; i++) {
    if (page_age[i] < min_age) {
      min_age = page_age[i];
      lru_index = i;
    }
  }
  return lru_index;
}

void lru_page_replacement() {
  int time = 0;

  for (int i = 0; i < num_pages; i++) {
    int page = pages[i];
    int found = 0;

    for (int j = 0; j < MAX_FRAMES; j++) {
      if (frames[j] == page) {
        page_age[j] = time++;
        found = 1;
        break;
      }
    }

    if (!found) {
      int lru_index = find_lru_index();
      frames[lru_index] = page;
      page_age[lru_index] = time++;
    }

    print_frames();
  }
}

int main() {
  printf("Enter number of pages: ");
  scanf("%d", &num_pages);

  printf("Enter the page sequence: ");
  for (int i = 0; i < num_pages; i++) {
    scanf("%d", &pages[i]);
  }

  initialize_frames();
  lru_page_replacement();

  return 0;
}
```

Experiment NO.16

Aim:

To simulate the LFU page replacement algorithm.

Theory:

LFU (Least Frequently Used): The LFU page replacement algorithm replaces the page that has been used the least number of times. It tracks the frequency of page accesses and evicts the page with the lowest usage count when a new page needs to be loaded. This technique assumes that pages accessed frequently in the past will continue to be accessed more often than others.

Algorithm:

1. Initialize: Maintain a count for each page that tracks the frequency of its access.
2. Page Requests: For each page request, check if the page is already in memory.
3. Update Frequency: If the page is in memory, increment its frequency count.
4. Page Replacement: If the page is not in memory and memory is full, remove the page with the lowest frequency count and load the new page.
5. Repeat: Continue this process for all incoming page requests.

Code

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

#define MAX_FRAMES 3
#define MAX_PAGES 10

int frames[MAX_FRAMES], pages[MAX_PAGES];
int page_frequency[MAX_FRAMES];
int num_pages;

void initialize_frames() {
  for (int i = 0; i < MAX_FRAMES; i++) {
    frames[i] = -1;
    page_frequency[i] = 0;
  }
}
```

```c
void print_frames() {
  for (int i = 0; i < MAX_FRAMES; i++) {
    if (frames[i] == -1) {
      printf("- ");
    } else {
      printf("%d ", frames[i]);
    }
  }
  printf("\n");
}

int find_lfu_index() {
  int min_freq = INT_MAX;
  int lfu_index = 0;

  for (int i = 0; i < MAX_FRAMES; i++) {
    if (page_frequency[i] < min_freq) {
      min_freq = page_frequency[i];
      lfu_index = i;
    }
  }
  return lfu_index;
}

void lfu_page_replacement() {
  for (int i = 0; i < num_pages; i++) {
    int page = pages[i];
    int found = 0;

    for (int j = 0; j < MAX_FRAMES; j++) {
      if (frames[j] == page) {
        page_frequency[j]++;
        found = 1;
        break;
      }
    }

    if (!found) {
      int lfu_index = find_lfu_index();
      frames[lfu_index] = page;
      page_frequency[lfu_index] = 1;
    }

    print_frames();
  }
}

int main() {
  printf("Enter number of pages: ");
  scanf("%d", &num_pages);

  printf("Enter the page sequence: ");
  for (int i = 0; i < num_pages; i++) {
    scanf("%d", &pages[i]);
  }

  initialize_frames();
  lfu_page_replacement();

  return 0;
}
```

Experiment NO.17

Aim:

To simulate the Optimal page replacement algorithm.

Theory:

Optimal Page Replacement: The Optimal page replacement algorithm replaces the page that will not be used for the longest period in the future. It provides the lowest possible page-fault rate but is not practical for implementation, as it requires future knowledge of the reference string. This algorithm is often used as a benchmark to compare the performance of other page replacement algorithms.

Algorithm:

1. Initialize: Maintain a list of future page requests.
2. Page Requests: For each page request, check if the page is already in memory.
3. Page Replacement: If the page is not in memory and memory is full, look ahead in the future reference string and replace the page that is not needed for the longest period of time.
4. Load the new page and continue processing the remaining page requests.

Code

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

#define MAX_FRAMES 3
#define MAX_PAGES 10

int frames[MAX_FRAMES], pages[MAX_PAGES];
int num_pages;

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

void print_frames() {
  for (int i = 0; i < MAX_FRAMES; i++) {
    if (frames[i] == -1) {
```

```
      printf("- ");
    } else {
      printf("%d ", frames[i]);
    }
  }
  printf("\n");
}

int find_optimal_index(int current_index) {
  int furthest_use = -1;
  int optimal_index = 0;

  for (int i = 0; i < MAX_FRAMES; i++) {
    int j;
    for (j = current_index; j < num_pages; j++) {
      if (frames[i] == pages[j]) {
        if (j > furthest_use) {
          furthest_use = j;
          optimal_index = i;
        }
        break;
      }
    }
    if (j == num_pages) {
      return i;
    }
  }
  return optimal_index;
}

void optimal_page_replacement() {
  int current_index = 0;

  for (int i = 0; i < num_pages; i++) {
    int page = pages[i];
    int found = 0;

    for (int j = 0; j < MAX_FRAMES; j++) {
      if (frames[j] == page) {
        found = 1;
        break;
      }
    }

    if (!found) {
      int optimal_index = find_optimal_index(i + 1);
      frames[optimal_index] = page;
    }

    print_frames();
  }
}

int main() {
  printf("Enter number of pages: ");
  scanf("%d", &num_pages);

  printf("Enter the page sequence: ");
  for (int i = 0; i < num_pages; i++) {
    scanf("%d", &pages[i]);
  }

  initialize_frames();
  optimal_page_replacement();
```

```
  return 0;
}
```

Experiment NO.18

Aim:

To simulate the following file organization techniques

Theory:

Single-Level Directory: In this technique, all files are stored in a single directory. All users and processes share the same directory space, which can lead to difficulties in organizing and managing files as the number of files grows.

Two-Level Directory: In this method, each user has their own directory, separate from others. The system maintains a master directory that contains pointers to each user's directory. This technique improves file organization but may still become complex for large systems.

Hierarchical Directory: A hierarchical (or tree-structured) directory system allows for multiple levels of directories. This technique enables users to create subdirectories within their directories, forming a tree-like structure. It is highly scalable and efficient for managing a large number of files and directories.

Code

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

#define MAX_FILES 10
#define MAX_SUBDIRS 5
#define MAX_FILE_NAME 20
#define MAX_DIR_NAME 20

typedef struct {
  char name[MAX_FILE_NAME];
} File;

typedef struct {
  char name[MAX_DIR_NAME];
  File files[MAX_FILES];
  int num_files;
} Directory;

typedef struct {
  char name[MAX_DIR_NAME];
  Directory subdirs[MAX_SUBDIRS];
  int num_subdirs;
} TwoLevelDirectory;
```

```c
typedef struct HierarchicalDirectory HierarchicalDirectory;

struct HierarchicalDirectory {
  char name[MAX_DIR_NAME];
  HierarchicalDirectory *subdirs[MAX_SUBDIRS];
  int num_subdirs;
  File files[MAX_FILES];
  int num_files;
};

void initialize_directory(Directory *dir) { dir->num_files = 0; }

void add_file(Directory *dir, const char *file_name) {
  if (dir->num_files < MAX_FILES) {
    strcpy(dir->files[dir->num_files].name, file_name);
    dir->num_files++;
  } else {
    printf("Directory is full.\n");
  }
}

void print_single_level_directory(Directory *dir) {
  printf("Files in directory '%s':\n", dir->name);
  for (int i = 0; i < dir->num_files; i++) {
    printf("  %s\n", dir->files[i].name);
  }
}

void initialize_two_level_directory(TwoLevelDirectory *two_level_dir) {
  two_level_dir->num_subdirs = 0;
}

void add_subdir(TwoLevelDirectory *two_level_dir, const char *subdir_name) {
  if (two_level_dir->num_subdirs < MAX_SUBDIRS) {
    strcpy(two_level_dir->subdirs[two_level_dir->num_subdirs].name,
           subdir_name);
    initialize_directory(&two_level_dir->subdirs[two_level_dir->num_subdirs]);
    two_level_dir->num_subdirs++;
  } else {
    printf("Two-level directory is full.\n");
  }
}

void print_two_level_directory(TwoLevelDirectory *two_level_dir) {
  printf("Two-Level Directory '%s':\n", two_level_dir->name);
  for (int i = 0; i < two_level_dir->num_subdirs; i++) {
    printf("  Subdirectory '%s':\n", two_level_dir->subdirs[i].name);
    for (int j = 0; j < two_level_dir->subdirs[i].num_files; j++) {
      printf("    %s\n", two_level_dir->subdirs[i].files[j].name);
    }
  }
}

void initialize_hierarchical_directory(HierarchicalDirectory *hier_dir,
                                       const char *name) {
  strcpy(hier_dir->name, name);
  hier_dir->num_files = 0;
  hier_dir->num_subdirs = 0;
}

void add_file_hierarchical(HierarchicalDirectory *hier_dir,
                           const char *file_name) {
  if (hier_dir->num_files < MAX_FILES) {
```

```
    strcpy(hier_dir->files[hier_dir->num_files].name, file_name);
    hier_dir->num_files++;
  } else {
    printf("Hierarchical directory is full.\n");
  }
}

void add_subdir_hierarchical(HierarchicalDirectory *hier_dir,
                             HierarchicalDirectory *subdir) {
  if (hier_dir->num_subdirs < MAX_SUBDIRS) {
    hier_dir->subdirs[hier_dir->num_subdirs] = subdir;
    hier_dir->num_subdirs++;
  } else {
    printf("Hierarchical directory has reached maximum subdirectories.\n");
  }
}

void print_hierarchical_directory(HierarchicalDirectory *hier_dir, int level) {
  for (int i = 0; i < level; i++) {
    printf("  ");
  }
  printf("Directory '%s':\n", hier_dir->name);
  for (int i = 0; i < hier_dir->num_files; i++) {
    for (int j = 0; j < level + 1; j++) {
      printf("  ");
    }
    printf("File: %s\n", hier_dir->files[i].name);
  }
  for (int i = 0; i < hier_dir->num_subdirs; i++) {
    print_hierarchical_directory(hier_dir->subdirs[i], level + 1);
  }
}

int main() {
  Directory single_level_dir;
  strcpy(single_level_dir.name, "Root");
  initialize_directory(&single_level_dir);
  add_file(&single_level_dir, "file1.txt");
  add_file(&single_level_dir, "file2.txt");
  print_single_level_directory(&single_level_dir);

  TwoLevelDirectory two_level_dir;
  strcpy(two_level_dir.name, "MainDir");
  initialize_two_level_directory(&two_level_dir);
  add_subdir(&two_level_dir, "SubDir1");
  add_file(&two_level_dir.subdirs[0], "file3.txt");
  add_file(&two_level_dir.subdirs[0], "file4.txt");
  add_subdir(&two_level_dir, "SubDir2");
  add_file(&two_level_dir.subdirs[1], "file5.txt");
  print_two_level_directory(&two_level_dir);

  HierarchicalDirectory root_dir;
  initialize_hierarchical_directory(&root_dir, "RootHier");
  HierarchicalDirectory subdir1;
  initialize_hierarchical_directory(&subdir1, "SubDirHier1");
  HierarchicalDirectory subdir2;
  initialize_hierarchical_directory(&subdir2, "SubDirHier2");

  add_file_hierarchical(&root_dir, "file6.txt");
  add_file_hierarchical(&root_dir, "file7.txt");
  add_file_hierarchical(&subdir1, "file8.txt");
  add_file_hierarchical(&subdir2, "file9.txt");

  add_subdir_hierarchical(&root_dir, &subdir1);
```

```
  add_subdir_hierarchical(&root_dir, &subdir2);

  print_hierarchical_directory(&root_dir, 0);

  return 0;
}
```

Experiment NO.19

Aim:

To simulate all file allocation strategies

Theory:

Sequential Allocation: In sequential allocation, files are stored in contiguous blocks on the disk. Each file occupies a set of consecutive disk blocks, making it easy to access sequentially. However, it can lead to external fragmentation and inefficient use of space for files that grow over time.

Indexed Allocation: In this method, an index block is created for each file. The index block contains pointers to the actual blocks where the file data is stored. This allows random access to file blocks, eliminating fragmentation but requiring extra space for index blocks.

Linked Allocation: Linked allocation stores file blocks in a linked list, where each block points to the next one. There is no need for contiguous allocation, reducing fragmentation. However, it can lead to slower access times due to the need to traverse the linked list to read or write a file.

Code

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

#define MAX_BLOCKS 10
#define MAX_FILES 5
#define MAX_FILE_NAME 20

int disk[MAX_BLOCKS];

typedef struct {
  char name[MAX_FILE_NAME];
  int start_block;
  int size;
} SequentialFile;

typedef struct {
  char name[MAX_FILE_NAME];
  int index_blocks[MAX_BLOCKS];
  int num_blocks;
} IndexedFile;

typedef struct Node {
```

```c
  int block;
  struct Node *next;
} LinkedNode;

typedef struct {
  char name[MAX_FILE_NAME];
  LinkedNode *head;
} LinkedFile;

void initialize_disk() {
  for (int i = 0; i < MAX_BLOCKS; i++) {
    disk[i] = 0;
  }
}

void print_disk() {
  printf("Disk blocks: ");
  for (int i = 0; i < MAX_BLOCKS; i++) {
    printf("%d ", disk[i]);
  }
  printf("\n");
}

void allocate_sequential(SequentialFile *file, int start_block, int size) {
  if (start_block + size > MAX_BLOCKS) {
    printf("Error: Not enough space on the disk.\n");
    return;
  }
  for (int i = start_block; i < start_block + size; i++) {
    if (disk[i] != 0) {
      printf("Error: Space already allocated.\n");
      return;
    }
  }
  file->start_block = start_block;
  file->size = size;
  strcpy(file->name, "SequentialFile");
  for (int i = start_block; i < start_block + size; i++) {
    disk[i] = 1;
  }
  printf("Sequential file '%s' allocated from block %d to %d.\n", file->name,
         start_block, start_block + size - 1);
}

void allocate_indexed(IndexedFile *file, int blocks[], int num_blocks) {
  if (num_blocks > MAX_BLOCKS) {
    printf("Error: Too many blocks requested.\n");
    return;
  }
  for (int i = 0; i < num_blocks; i++) {
    if (blocks[i] >= MAX_BLOCKS || disk[blocks[i]] != 0) {
      printf("Error: Block %d is not available.\n", blocks[i]);
      return;
    }
  }
  file->num_blocks = num_blocks;
  strcpy(file->name, "IndexedFile");
  for (int i = 0; i < num_blocks; i++) {
    disk[blocks[i]] = 1; // Mark block as allocated
    file->index_blocks[i] = blocks[i];
  }
  printf("Indexed file '%s' allocated at blocks: ", file->name);
  for (int i = 0; i < num_blocks; i++) {
    printf("%d ", file->index_blocks[i]);
```

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

void allocate_linked(LinkedFile *file, int blocks[], int num_blocks) {
  LinkedNode *prev = NULL;
  LinkedNode *head = NULL;
  for (int i = 0; i < num_blocks; i++) {
    if (blocks[i] >= MAX_BLOCKS || disk[blocks[i]] != 0) {
      printf("Error: Block %d is not available.\n", blocks[i]);
      while (head != NULL) {
        LinkedNode *temp = head;
        head = head->next;
        free(temp);
      }
      return;
    }
    disk[blocks[i]] = 1;
    LinkedNode *node = (LinkedNode *)malloc(sizeof(LinkedNode));
    node->block = blocks[i];
    node->next = NULL;
    if (prev == NULL) {
      head = node;
    } else {
      prev->next = node;
    }
    prev = node;
  }
  file->head = head;
  strcpy(file->name, "LinkedFile");
  printf("Linked file '%s' allocated with blocks: ", file->name);
  LinkedNode *current = head;
  while (current != NULL) {
    printf("%d ", current->block);
    current = current->next;
  }
  printf("\n");
}

int main() {
  initialize_disk();
  print_disk();

  SequentialFile seq_file;
  int start_block = 2;
  int size = 4;
  allocate_sequential(&seq_file, start_block, size);
  print_disk();

  IndexedFile idx_file;
  int index_blocks[] = {6, 7, 8};
  allocate_indexed(&idx_file, index_blocks, 3);
  print_disk();

  LinkedFile lnk_file;
  int linked_blocks[] = {1, 3, 5};
  allocate_linked(&lnk_file, linked_blocks, 3);
  print_disk();

  return 0;
}
```

Experiment NO.20

Aim:

To simulate disk scheduling algorithms

Theory:

FCFS (First-Come, First-Served): FCFS is the simplest disk scheduling algorithm. Requests are handled in the order they arrive, without considering their proximity to the current head position. This can lead to high seek times if requests are scattered across the disk.

SCAN (Elevator Algorithm): In the SCAN algorithm, the disk arm moves in one direction, servicing all requests until it reaches the end of the disk. It then reverses direction and continues servicing requests. This approach reduces the variability in seek times compared to FCFS.

LOOK: The LOOK algorithm is a variation of SCAN. Instead of going all the way to the end of the disk, the arm only moves as far as the last request in each direction before reversing. This reduces unnecessary movement, improving efficiency.

Algorithm:

1. FCFS:
a. Initialize a queue of disk requests.
b. Process each request in the order it arrives.
c. Move the disk head to the requested track and process the request.

2. SCAN:
a. Start by moving the disk head in one direction (either toward the lowest or highest track).
b. Service all requests in that direction until the disk’s end is reached.
c. Reverse the direction and continue servicing requests.

3. LOOK:
a. Move the disk head in one direction.
b. Service requests up to the last request in that direction (don’t go to the disk’s end).
c. Reverse the direction and repeat the process.

Code

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

#define MAX_REQUESTS 10

void fcfs(int requests[], int num_requests, int start) {
  int seek_count = 0;
  int distance;
  int cur_track = start;

  printf("FCFS Disk Scheduling:\n");
  for (int i = 0; i < num_requests; i++) {
    distance = abs(requests[i] - cur_track);
    seek_count += distance;
    cur_track = requests[i];
    printf("Move to track %d\n", requests[i]);
  }
  printf("Total seek time: %d\n", seek_count);
}

void scan(int requests[], int num_requests, int start, int direction) {
  int seek_count = 0;
  int cur_track = start;
  int distance;

  printf("SCAN Disk Scheduling:\n");
  int direction_change = (direction == 1) ? 0 : 1;
  int max = 0, min = 0;
  for (int i = 0; i < num_requests; i++) {
    if (requests[i] > max)
      max = requests[i];
    if (requests[i] < min)
      min = requests[i];
  }
  if (direction == 1) {
    for (int i = 0; i < num_requests; i++) {
      if (requests[i] >= cur_track) {
        distance = abs(requests[i] - cur_track);
        seek_count += distance;
        cur_track = requests[i];
        printf("Move to track %d\n", requests[i]);
      }
    }
    if (cur_track != max) {
      seek_count += abs(max - cur_track);
      cur_track = max;
      printf("Move to track %d\n", max);
    }
    for (int i = 0; i < num_requests; i++) {
      if (requests[i] < cur_track) {
        distance = abs(requests[i] - cur_track);
        seek_count += distance;
        cur_track = requests[i];
        printf("Move to track %d\n", requests[i]);
      }
    }
  } else {
    for (int i = 0; i < num_requests; i++) {
      if (requests[i] <= cur_track) {
```

```
      distance = abs(requests[i] - cur_track);
      seek_count += distance;
      cur_track = requests[i];
      printf("Move to track %d\n", requests[i]);
    }
  }
  if (cur_track != min) {
    seek_count += abs(min - cur_track);
    cur_track = min;
    printf("Move to track %d\n", min);
  }
  for (int i = 0; i < num_requests; i++) {
    if (requests[i] > cur_track) {
      distance = abs(requests[i] - cur_track);
      seek_count += distance;
      cur_track = requests[i];
      printf("Move to track %d\n", requests[i]);
    }
  }
}
printf("Total seek time: %d\n", seek_count);
}

void look(int requests[], int num_requests, int start, int direction) {
  int seek_count = 0;
  int cur_track = start;
  int distance;
  int max = 0, min = 0;

  printf("LOOK Disk Scheduling:\n");
  for (int i = 0; i < num_requests; i++) {
    if (requests[i] > max)
      max = requests[i];
    if (requests[i] < min)
      min = requests[i];
  }
  if (direction == 1) {
    for (int i = 0; i < num_requests; i++) {
      if (requests[i] >= cur_track) {
        distance = abs(requests[i] - cur_track);
        seek_count += distance;
        cur_track = requests[i];
        printf("Move to track %d\n", requests[i]);
      }
    }
    for (int i = 0; i < num_requests; i++) {
      if (requests[i] < cur_track) {
        distance = abs(requests[i] - cur_track);
        seek_count += distance;
        cur_track = requests[i];
        printf("Move to track %d\n", requests[i]);
      }
    }
  } else {
    for (int i = 0; i < num_requests; i++) {
      if (requests[i] <= cur_track) {
        distance = abs(requests[i] - cur_track);
        seek_count += distance;
        cur_track = requests[i];
        printf("Move to track %d\n", requests[i]);
      }
    }
    for (int i = 0; i < num_requests; i++) {
      if (requests[i] > cur_track) {
```

```
        distance = abs(requests[i] - cur_track);
        seek_count += distance;
        cur_track = requests[i];
        printf("Move to track %d\n", requests[i]);
      }
    }
  }
  printf("Total seek time: %d\n", seek_count);
}

int main() {
  int requests[] = {55, 58, 39, 18, 90, 160, 150};
  int num_requests = sizeof(requests) / sizeof(requests[0]);
  int start = 50;
  int direction = 1;

  printf("Disk Scheduling Algorithms Simulation\n");

  fcfs(requests, num_requests, start);

  printf("\n");
  scan(requests, num_requests, start, direction);

  printf("\n");
  look(requests, num_requests, start, direction);

  return 0;
}
```