JSS MAHAVIDYAPEETHA
JSS ACADEMY OF TECHNICAL EDUCATION, BENGALURU-60
DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING
JSSATE Campus, Dr. Vishnuvardhana Main Road, Bengaluru – 560060

III SEMESTER

OPERATING SYSTEM LABORATORY LAB MANUAL

[BCS303]

Compiled By:

Dr. D V ASHOKA	Mrs. B U Patil
Professor, Dept of Information Science & Engineering, JSSATEB	Assistant Professor, Dept of Information Science & Engineering, JSSATEB

Signature of Faculty **Signature of HOD**

JSS MAHAVIDYAPEETHA

JSS ACADEMY OF TECHNICAL EDUCATION, BENGALURU-60
DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

VISION

To emerge as a center for achieving academic excellence, by producing competent professionals to meet the global challenges in the field of Information science and Technology.

MISSION

M1: To prepare the students as competent professionals to meet the advancements in the industry and academia by imparting quality technical education.

M2: To enrich the technical ability of students to face the world with confidence, commitment, and teamwork

.

M3: To inculcate and practice strong techno-ethical values to serve the society.

Program Educational Objectives (PEOs):

PEO1: To demonstrate analytical and technical problem-solving abilities.

PEO2: To be conversant in the developments of Information Science and Engineering, leading towards the employability and higher studies.

PEO3: To engage in research and development leading to new innovations and products.

Program Specific Outcomes (PSOs):

PSO1: Apply the mathematical concepts for solving engineering problems by using appropriate programming constructs

.

PSO2: Adaptability to software development methodologies.

PSO3: Demonstrate the knowledge towards the domain specific initiatives of Information Science and Engineering.

Program Outcomes (POs):

Information Science and Engineering Graduates will be able to:

PO1	Apply the knowledge of mathematics, science, engineering fundamentals, and an Engineering specialization to the solution of complex engineering problems.
PO2	Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues, and the consequent responsibilities relevant to the professional engineering practice.
PO7	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Lesson plan

Course Name: OPERATING SYSTEMS	**Course Code: BCS303**	**Sem/Sec:** 3
	Contact Hrs: 2 Hours/Week	**Total Hrs.**: 20
SEE Marks:	**CIE Marks:**25	**Exam Duration:** 3 hrs

Course Outcomes:
At the end of the course, students will be able to

CO#	CO Statement	BLL
CO1	Identify the process concept, structure and design of the operating system.	L3
CO2	Experiment the concepts of threads, process synchronization and CPU scheduling algorithms.	L3
CO3	Identify causes of deadlocks and solutions for eliminating deadlock.	L3
CO4	Analyze the virtual memory management, file system implementation , storage structure, disk scheduling and protection associated with OS	L4

Pre-requisites: Basic Knowledge of Hardware System Components.
Course overview: Students will be able to studyconcepts and terminology used in Operating System and understands Memory Management, File system and storage techniques.
Pedagogical/Innovative methods planned: Charts / Posters/Projects Presentations by the students

Class No.	Topics to be covered	Reference material and CO
1	Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)	T1 CO1
2	Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.	T1 CO2
3	Develop a C program to simulate producer-consumer problem using semaphores.	T1 CO2
4	Develop a C program which demonstrates inter-process communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.	T1 CO2
5	Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.	T1 CO3
6	Test	
7	Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.	T1 CO4

8	Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU	T1 CO4
9	Simulate following File Organization Techniques a) Single level directory b) Two level directory	T1 CO4
10	Develop a C program to simulate the Linked file allocation strategies.	T1 CO4
11	Develop a C program to simulate SCAN disk scheduling algorithm.	T1 CO4
12	Test	

Total No. of Hours required as per VTU: 20
Total No. of classes planned: 10*2=20 hours

TEXT BOOKS (TB):

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating System Principles 8th edition, Wiley-India, 2015

REFERENCE BOOKS (RB):

1. Ann McHoes Ida M Fylnn, Understanding Operating System, Cengage Learning, 6th Edition
2. D.M Dhamdhere, Operating Systems: A Concept Based Approach 3rd Ed, McGraw- Hill, 2013.
3. P.C.P. Bhatt, An Introduction to Operating Systems: Concepts and Practice 4th Edition, PHI(EEE), 2014.
4. William Stallings Operating Systems: Internals and Design Principles, 6th Edition, Pearson.

CIE for the practical component of the IPCC

- 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks.
- The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

Table of Content

Sl. No.	Content	Page No
1	Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)	7
2	Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.	9
3	Develop a C program to simulate producer-consumer problem using semaphores.	16
4	Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.	18
5	Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.	20
6	Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.	24
7	Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU	29
8	Simulate following File Organization Techniques a) Single level directory b) Two level directory	32
9	Develop a C program to simulate the Linked file allocation strategies.	42
10	Develop a C program to simulate SCAN disk scheduling algorithm.	46

1. **Develop a C program to implement process system calls fork(), exec(), wait() create process and terminate process.**

```
#include <stdio.h>      // include necessary header files
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {                  // Define main() function
   pid_t child_pid;           // Declare variables for the child process ID (child_pid) and status (status)
   int status;

   // Create a child process using fork()
   child_pid = fork();


   if (child_pid == -1) {
      perror("Fork failed");
      exit(1);
   }

   if (child_pid == 0) {
      // This code is executed by the child process

      printf("Child process: My PID is %d\n", getpid());

      // Execute a program in the child process using exec() ,
      char *args[] = {"ls", "-l", NULL};
      if (execvp("ls", args) == -1) {
         perror("Exec failed");
         exit(1);
      }

   } else {
      // This code is executed by the parent process

      printf("Parent process: My PID is %d\n", getpid());

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

      if (WIFEXITED(status)) {
         printf("Child process terminated with status %d\n", WEXITSTATUS(status));
      } else if (WIFSIGNALED(status)) {
         printf("Child process terminated due to signal %d\n", WTERMSIG(status));
      }
   }

   return 0;                  // Return 0 to exit the main function and the program:
```

```
}
```

Output:

```
Parent process: My PID is 128
Child process: My PID is 132
total 20
-rwxr-xr-x 1 14065 14065 16264 Oct 18 14:18 a.out
-rwxrwxrwx 1 root  root  1513 Oct 18 14:18 main.c
Child process terminated with status 0
```

(Upon successful completion, fork() returns 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, -1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

The pid_t data type represents process IDs. You can get the process ID of a process by calling getpid . The function getppid returns the process ID of the parent of the current process (this is also known as the parent process ID). Your program should include the header files `unistd. h' and `sys/types.

- The code checks whether the child process terminated normally (WIFEXITED) or due to a signal (WIFSIGNALED) and prints the corresponding status or signal number.)

```c
#include <stdio.h>

int main() {
    int num, originalNum, reversedNum = 0, remainder;

    // Input a number from the user
    printf("Enter an integer: ");
    scanf("%d", &num);

    // Store the original number
    originalNum = num;

    // Reverse the number
    while (num != 0) {
        remainder = num % 10;
        reversedNum = reversedNum * 10 + remainder;
        num /= 10;
    }

    // Check if the original number and reversed number are the same
    if (originalNum == reversedNum) {
        printf("%d is a palindrome.\n", originalNum);
    } else {
        printf("%d is not a palindrome.\n", originalNum);
    }

    return 0;
```

2. Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

a) FCFS Scheduling

```
#include <stdio.h>

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int turnaround_time;
    int waiting_time;
};

void calculateTimes(struct Process processes[], int n) {
    int current_time = 0;

    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time > current_time) {
            current_time = processes[i].arrival_time;
        }
        processes[i].waiting_time = current_time - processes[i].arrival_time;
        processes[i].turnaround_time = processes[i].waiting_time +
processes[i].burst_time;
        current_time += processes[i].burst_time;
    }
}

int main() {
    int n = 4; // Number of processes
    struct Process processes[n];

    // Initialize processes (you can modify these values)
    processes[0] = (struct Process){1, 0, 6, 0, 0};
    processes[1] = (struct Process){2, 1, 8, 0, 0};
    processes[2] = (struct Process){3, 2, 7, 0, 0};
    processes[3] = (struct Process){4, 3, 3, 0, 0};

    calculateTimes(processes, n);

    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time,
            processes[i].burst_time, processes[i].waiting_time,
processes[i].turnaround_time);
    }

    return 0;
```

```
}
```

Output:

Process	Arrival Time	Burst Time	Waiting Time	Turnaround Time
1	0	6	0	6
2	1	8	5	13
3	2	7	12	19
4	3	3	18	21

b) SJF Scheduling Algorithm

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

struct Process {
   int id;
   int arrival_time;
   int burst_time;
   int turnaround_time;
   int waiting_time;
};

// Function to calculate turnaround time and waiting time using SJF
void SJF(struct Process processes[], int n) {
   int remaining_time[n];
   int completed_processes = 0;
   int current_time = 0;

   // Initialize remaining_time array
   for (int i = 0; i < n; i++) {
      remaining_time[i] = processes[i].burst_time;
   }

   while (completed_processes < n) {
      int shortest_job = -1;
      int shortest_time = INT_MAX;

      for (int i = 0; i < n; i++) {
         if (processes[i].arrival_time <= current_time && remaining_time[i] <
shortest_time && remaining_time[i] > 0) {
            shortest_job = i;
            shortest_time = remaining_time[i];
         }
      }

      if (shortest_job == -1) {
         current_time++;
      } else {
         remaining_time[shortest_job]--;
         if (remaining_time[shortest_job] == 0) {
```

```
            completed_processes++;
            processes[shortest_job].turnaround_time = current_time -
processes[shortest_job].arrival_time + 1;
            processes[shortest_job].waiting_time =
processes[shortest_job].turnaround_time - processes[shortest_job].burst_time;
          }
          current_time++;
       }
    }
}

int main() {
    int n = 4; // Number of processes
    struct Process processes[n];

    // Initialize processes (you can modify these values)
    processes[0] = (struct Process){1, 0, 6, 0, 0};
    processes[1] = (struct Process){2, 1, 8, 0, 0};
    processes[2] = (struct Process){3, 2, 7, 0, 0};
    processes[3] = (struct Process){4, 3, 3, 0, 0};

    SJF(processes, n);

    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
       printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time,
            processes[i].burst_time, processes[i].waiting_time,
processes[i].turnaround_time);
    }

    return 0;
}
```

Output:

Process	Arrival Time	Burst Time	Waiting Time	Turnaround Time
1	0	6	0	6
2	1	8	15	23
3	2	7	7	14
4	3	3	3	6

c) Round Robin Algorithm

```
#include <stdio.h>

struct Process {
   int id;
   int burst_time;
   int remaining_time;
   int turnaround_time;
   int waiting_time;
};

void roundRobin(struct Process processes[], int n, int quantum) {
   int completed_processes = 0;
   int current_time = 0;

   while (completed_processes < n) {
      for (int i = 0; i < n; i++) {
         if (processes[i].remaining_time > 0) {
            int execution_time = (processes[i].remaining_time > quantum) ? quantum :
processes[i].remaining_time;
            processes[i].remaining_time -= execution_time;
            current_time += execution_time;

            if (processes[i].remaining_time == 0) {
               completed_processes++;
               processes[i].turnaround_time = current_time;
               processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;
            }
         }
      }
   }
}

int main() {
   int n = 4; // Number of processes
   int quantum = 2; // Time quantum for Round Robin
   struct Process processes[n];

   // Initialize processes (you can modify these values)
   processes[0] = (struct Process){1, 6, 0, 0, 0};
   processes[1] = (struct Process){2, 8, 0, 0, 0};
   processes[2] = (struct Process){3, 7, 0, 0, 0};
   processes[3] = (struct Process){4, 3, 0, 0, 0};

   // Set remaining time to burst time initially
   for (int i = 0; i < n; i++) {
      processes[i].remaining_time = processes[i].burst_time;
```

```
    }

    roundRobin(processes, n, quantum);

    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
               processes[i].waiting_time, processes[i].turnaround_time);
    }

    return 0;
}
```

Output:

Process	Burst Time	Waiting Time	Turnaround Time
1	6	11	17
2	8	15	23
3	7	17	24
4	3	12	15

d) Priority Scheduling Algorithm

```
#include <stdio.h>

struct Process {
    int id;
    int burst_time;
    int priority;
    int turnaround_time;
    int waiting_time;
};

void priorityScheduling(struct Process processes[], int n) {
    int completed_processes = 0;
    int current_time = 0;

    while (completed_processes < n) {
        int highest_priority = -1;
        int highest_priority_index = -1;

        for (int i = 0; i < n; i++) {
            if (processes[i].priority > highest_priority && processes[i].burst_time > 0) {
                highest_priority = processes[i].priority;
                highest_priority_index = i;
            }
        }

        if (highest_priority_index != -1) {
            int execution_time = 1; // Execute for 1 unit
            processes[highest_priority_index].burst_time -= execution_time;
            current_time += execution_time;

            if (processes[highest_priority_index].burst_time == 0) {
                completed_processes++;
                processes[highest_priority_index].turnaround_time = current_time;
                processes[highest_priority_index].waiting_time =
processes[highest_priority_index].turnaround_time -
processes[highest_priority_index].burst_time;
            }
        } else {
            current_time++;
        }
    }
}

int main() {
    int n = 4; // Number of processes
    struct Process processes[n];
```

```
    // Initialize processes (you can modify these values)
    processes[0] = (struct Process){1, 6, 2, 0, 0};
    processes[1] = (struct Process){2, 8, 1, 0, 0};
    processes[2] = (struct Process){3, 7, 3, 0, 0};
    processes[3] = (struct Process){4, 3, 4, 0, 0};

    priorityScheduling(processes, n);

    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
               processes[i].waiting_time, processes[i].turnaround_time);
    }

    return 0;
}
```

Output:

Process	Burst Time	Waiting Time	Turnaround Time
1	0	16	16
2	0	24	24
3	0	10	10
4	0	3	3

3. Develop a C program to simulate producer consumer problem using semaphores.

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

#define BUFFER_SIZE 5

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

sem_t empty, full, mutex;

void *producer(void *arg) {
    int item = 1;
    while (1) {
        // Produce an item
        sleep(1);
        printf("Producing item %d\n", item);

        // Wait for an empty slot in the buffer
        sem_wait(&empty);
        sem_wait(&mutex);

        // Add item to the buffer
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        item++;

        sem_post(&mutex);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    while (1) {
        // Wait for a full slot in the buffer
        sem_wait(&full);
        sem_wait(&mutex);

        // Consume an item from the buffer
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("Consuming item %d\n", item);

        sem_post(&mutex);
        sem_post(&empty);
```

```
        // Consume the item (process it) outside the critical section
        sleep(2);
    }
}

int main() {
    pthread_t producer_thread, consumer_thread;

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

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for the threads to finish (which they won't since they run indefinitely)
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

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

    return 0;
}
```

Output:

```
Producing item 1
Consuming item 1
Producing item 2
Consuming item 2
Producing item 3
Producing item 4
Consuming item 3
Producing item 5
Producing item 6
Consuming item 4
Producing item 7
Producing item 8
Consuming item 5
Producing item 9
```

4. **Develop a C program which demonstrates Inter Process Communication between a reader process and a writer process. use mkfifo, open, read, write and close APIs in your program.**

Writer Process

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <fifo_name>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const char *fifo_name = argv[1];
    char message[] = "Hello, reader!";

    // Create a named pipe (FIFO)
    if (mkfifo(fifo_name, 0666) == -1) {
        perror("mkfifo");
        exit(EXIT_FAILURE);
    }

    printf("FIFO '%s' created.\n", fifo_name);

    // Open the FIFO for writing
    int fd = open(fifo_name, O_WRONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Write the message to the FIFO
    ssize_t bytes_written = write(fd, message, sizeof(message) - 1);
    if (bytes_written == -1) {
        perror("write");
        exit(EXIT_FAILURE);
    }

    printf("Message sent to the FIFO: %s\n", message);

    // Close the FIFO
    close(fd);

    // Remove the FIFO
```

```
    if (unlink(fifo_name) == -1) {
        perror("unlink");
        exit(EXIT_FAILURE);
    }

    printf("FIFO removed.\n");

    return 0;
}
```

Reader Process

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <fifo_name>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const char *fifo_name = argv[1];
    char buffer[256];

    // Open the FIFO for reading
    int fd = open(fifo_name, O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Read data from the FIFO
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
    if (bytes_read == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }

    buffer[bytes_read] = '\0'; // Null-terminate the received data

    printf("Received message from the FIFO: %s\n", buffer);

    // Close the FIFO
    close(fd);

    return 0;
}
```

5. Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

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

int n_processes, n_resources;

void inputData() {
   printf("Enter the number of processes: ");
   scanf("%d", &n_processes);
   printf("Enter the number of resource types: ");
   scanf("%d", &n_resources);

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

   printf("Enter the maximum demand of each process:\n");
   for (int i = 0; i < n_processes; i++) {
      printf("Process %d: ", i);
      for (int j = 0; j < n_resources; j++) {
         scanf("%d", &maximum[i][j]);
         need[i][j] = maximum[i][j];
      }
   }

   printf("Enter the current allocation of resources to each process:\n");
   for (int i = 0; i < n_processes; i++) {
      printf("Process %d: ", i);
      for (int j = 0; j < n_resources; j++) {
         scanf("%d", &allocation[i][j]);
         need[i][j] -= allocation[i][j];
      }
   }

   printf("\nNeed Matrix:\n");
   for (int i = 0; i < n_processes; i++) {
      printf("Process %d: ", i);
      for (int j = 0; j < n_resources; j++) {
         printf("%d ", need[i][j]);
      }
```

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

bool isSafeState(int process, int request[]) {
    int work[n_resources];
    int finish[n_processes];

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

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

    for (int i = 0; i < n_resources; i++) {
        if (request[i] > need[process][i] || request[i] > work[i]) {
            return false; // Request exceeds need or available resources
        }
    }

    for (int i = 0; i < n_resources; i++) {
        work[i] -= request[i];
        allocation[process][i] += request[i];
        need[process][i] -= request[i];
    }

    int count = 0;
    int safe_sequence[n_processes];

    while (count < n_processes) {
        bool found = false;
        for (int i = 0; i < n_processes; i++) {
            if (!finish[i]) {
                bool can_allocate = true;
                for (int j = 0; j < n_resources; j++) {
                    if (need[i][j] > work[j]) {
                        can_allocate = false;
                        break;
                    }
                }
                if (can_allocate) {
                    for (int j = 0; j < n_resources; j++) {
                        work[j] += allocation[i][j];
                    }
                    safe_sequence[count] = i;
                    finish[i] = true;
                    count++;
                    found = true;
```

```
            }
        }
    }
    if (!found) {
        break; // No safe sequence found
    }
  }

  if (count < n_processes) {
    for (int i = 0; i < n_resources; i++) {
        work[i] += request[i];
        allocation[process][i] -= request[i];
        need[process][i] += request[i];
    }
    return false;
  }

  // Found a safe sequence
  printf("Safe sequence: ");
  for (int i = 0; i < n_processes; i++) {
    printf("%d ", safe_sequence[i]);
  }
  printf("\n");

  return true;
}

void requestResources(int process) {
  int request[n_resources];

  printf("Enter the resource request for Process %d: ", process);
  for (int i = 0; i < n_resources; i++) {
    scanf("%d", &request[i]);
  }

  if (isSafeState(process, request)) {
    printf("Resource request granted to Process %d\n", process);
  } else {
    printf("Resource request denied to Process %d\n", process);
  }
}

int main() {
  inputData();

  while (1) {
    int process;
    printf("\nEnter the process to request resources (0-%d, or -1 to exit): ", n_processes - 1);
    scanf("%d", &process);
```

```
        if (process == -1) {
            break;
        }

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

        requestResources(process);
    }

    return 0;
}
```

Output:

```
Enter the number of processes: 3
Enter the number of resource types: 4
Enter the available resources (separated by spaces): 3 2 2 1
Enter the maximum demand of each process:
Process 0: 7 5 3 4
Process 1: 3 2 2 1
Process 2: 9 0 2 2
Enter the current allocation of resources to each process:
Process 0: 0 1 0 2
Process 1: 2 0 0 0
Process 2: 3 0 2 2

Need Matrix:
Process 0: 7 4 3 2
Process 1: 1 2 2 1
Process 2: 6 0 0 0

Enter the process to request resources (0-2, or -1 to exit): 1
Enter the resource request for Process 1: 1 0 2 1
Safe sequence: 1 0 2
Resource request granted to Process 1

Enter the process to request resources (0-2, or -1 to exit): 0
Enter the resource request for Process 0: 1 0 0 0
Safe sequence: 1 0 2
Resource request granted to Process 0

Enter the process to request resources (0-2, or -1 to exit): 2
Enter the resource request for Process 2: 0 0 2 0
Safe sequence: 1 0 2
Resource request denied to Process 2

Enter the process to request resources (0-2, or -1 to exit): -1
```

6. Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst Fit b) Best Fit c) First Fit

First Fit

```
#include <stdio.h>
#define max 25

void main() {
   int frag[max], b[max], f[max], i, j, nb, nf, temp;
   static int bf[max], ff[max];

   printf("\n Memory Management Scheme - First Fit");
   printf("\n Enter the number of blocks: ");
   scanf("%d", &nb);
   printf("Enter the number of files: ");
   scanf("%d", &nf);

   printf("\nEnter the size of the blocks:\n");
   for (i = 1; i <= nb; i++) {
      printf("Block %d: ", i);
      scanf("%d", &b[i]);
      bf[i] = 0; // Initializing block allocation status to 0 (unallocated)
   }

   printf("Enter the size of the files:\n");
   for (i = 1; i <= nf; i++) {
      printf("File %d: ", i);
      scanf("%d", &f[i]);
      ff[i] = 0; // Initializing file allocation status to 0 (unallocated)
   }

   for (i = 1; i <= nf; i++) {
      for (j = 1; j <= nb; j++) {
         if (bf[j] == 0 && b[j] >= f[i]) { // Checking for unallocated block with sufficient
space
            ff[i] = j; // Assigning block number to file
            bf[j] = 1; // Marking the block as allocated
            break; // Break after allocation
         }
      }
      if (ff[i] != 0) // Checking if file is allocated
         frag[i] = b[ff[i]] - f[i]; // Calculate fragmentation
   }

   printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragmentation\n");
   for (i = 1; i <= nf; i++) {
      if (ff[i] != 0)
         printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i, f[i], ff[i], b[ff[i]], frag[i]);
      else
```

```
            printf("%d\t\t%d\t\tNot Allocated\n", i, f[i]);
    }
}
```

Output:

```
Memory Management Scheme - First Fit
Enter the number of blocks: 5
Enter the number of files: 4

Enter the size of the blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter the size of the files:
File 1: 212
File 2: 412
File 3: 112
File 4: 426
```

File_no:	File_size:	Block_no:	Block_size:	Fragmentation
1	212	2	500	288
2	412	5	600	188
3	112	3	200	88
4	426	Not Allocated		

Best Fit Program

```
#include<stdio.h>

void main() {
    int fragment[20], b[20], p[20], i, j, nb, np, temp, lowest;
    static int barray[20], parray[20];

    printf("\nMemory Management Scheme - Best Fit");
    printf("\nEnter the number of blocks: ");
    scanf("%d", &nb);
    printf("Enter the number of processes: ");
    scanf("%d", &np);

    printf("\nEnter the size of the blocks:\n");
    for(i = 1; i <= nb; i++) {
        printf("Block %d: ", i);
        scanf("%d", &b[i]);
        barray[i] = 0; // Initializing block allocation status to 0 (unallocated)
    }

    printf("\nEnter the size of the processes:\n");
    for(i = 1; i <= np; i++) {
        printf("Process %d: ", i);
        scanf("%d", &p[i]);
```

```
    parray[i] = 0; // Initializing process allocation status to 0 (unallocated)
  }

  for(i = 1; i <= np; i++) {
    lowest = 9999; // Resetting lowest to a high value for each new process
    for(j = 1; j <= nb; j++) {
      if(barray[j] == 0 && b[j] >= p[i]) { // Checking for unallocated block with sufficient
space
        temp = b[j] - p[i];
        if(temp < lowest) {
          parray[i] = j; // Assigning block number to process
          lowest = temp; // Updating lowest fragmentation
        }
      }
    }
    fragment[i] = lowest; // Storing fragmentation for each process
    barray[parray[i]] = 1; // Marking the block as allocated
  }

  printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
  for(i = 1; i <= np; i++) {
    printf("\n%d\t\t%d\t\t", i, p[i]);
    if(parray[i] != 0)
      printf("%d\t\t%d\t\t%d", parray[i], b[parray[i]], fragment[i]);
    else
      printf("Not Allocated");
  }
  printf("\n");
}
```

Output:

```
Memory Management Scheme - Best Fit
Enter the number of blocks: 5
Enter the number of processes: 4

Enter the size of the blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600

Enter the size of the processes:
Process 1: 212
Process 2: 417
Process 3: 113
Process 4: 426
```

Process_no	Process_size	Block_no	Block_size	Fragment
1	212	4	300	88

2	417		2		500	83	
3	113	3		200	87		
4	426		5		600	174	

Worst Fit

```
#include<stdio.h>

void main() {
   int fragment[20], b[20], p[20], i, j, nb, np, temp, highest;
   static int barray[20], parray[20];

   printf("\nMemory Management Scheme - Worst Fit");
   printf("\nEnter the number of blocks: ");
   scanf("%d", &nb);
   printf("Enter the number of processes: ");
   scanf("%d", &np);

   printf("\nEnter the size of the blocks:\n");
   for(i = 1; i <= nb; i++) {
      printf("Block %d: ", i);
      scanf("%d", &b[i]);
      barray[i] = 0; // Initializing block allocation status to 0 (unallocated)
   }

   printf("\nEnter the size of the processes:\n");
   for(i = 1; i <= np; i++) {
      printf("Process %d: ", i);
      scanf("%d", &p[i]);
      parray[i] = 0; // Initializing process allocation status to 0 (unallocated)
   }

   for(i = 1; i <= np; i++) {
      highest = -1; // Resetting highest to a low value for each new process
      for(j = 1; j <= nb; j++) {
         if(barray[j] == 0 && b[j] >= p[i]) { // Checking for unallocated block with sufficient
space
            temp = b[j] - p[i];
            if(temp > highest) {
               parray[i] = j; // Assigning block number to process
               highest = temp; // Updating highest fragmentation
            }
         }
      }
      fragment[i] = highest; // Storing fragmentation for each process
      barray[parray[i]] = 1; // Marking the block as allocated
   }

   printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
   for(i = 1; i <= np; i++) {
      printf("\n%d\t\t%d\t\t", i, p[i]);
```

```
        if(parray[i] != 0)
            printf("%d\t\t%d\t\t%d", parray[i], b[parray[i]], fragment[i]);
        else
            printf("Not Allocated");
    }
    printf("\n");
}
```

Output:

```
Memory Management Scheme - Worst Fit
Enter the number of blocks: 5
Enter the number of processes: 4

Enter the size of the blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600

Enter the size of the processes:
Process 1: 212
Process 2: 412
Process 3: 112
Process 4: 426
```

Process_no	Process_size	Block_no	Block_size	Fragment
1	212	5	600	388
2	412	2	500	88
3	112	4	300	188
4	426	Not Allocated		

7. Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU.

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#define MAX_FRAMES 3

int frames[MAX_FRAMES];
int page_queue[MAX_FRAMES];
int frame_count = 0;
int page_fault_count = 0;
int victim = 0;

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

void displayFrames() {
        printf("Frames: ");
        for(int i=0;i<MAX_FRAMES;i++) {
                if(frames[i] != -1) printf("%d ",frames[i]);
        }
        printf("\n");
}

void FIFO(int page) {
        for(int i=0;i<MAX_FRAMES;i++) {
                        if(frames[i] == page) {
                                return;
                        }
                }
        if(frame_count<MAX_FRAMES) {
                frames[frame_count] = page;
                page_queue[frame_count] = page;
                frame_count++;
                page_fault_count++;
        } else {
                if(victim == MAX_FRAMES) {
                        victim = 0;
                }
                page_fault_count++;
                int replaced_page = page_queue[victim];
                frames[victim] = page;
                page_queue[victim++] = page;
        }
}
```

```
void LRU(int page) {
        for(int i=0;i<MAX_FRAMES;i++) {
                        if(frames[i] == page) {
                                return;
                        }
                }
        if(frame_count<MAX_FRAMES) {
                frames[frame_count] = page;
                page_queue[frame_count] = page;
                frame_count++;
                page_fault_count++;
        } else {
                page_fault_count++;
                int replaced_page = page_queue[0];
                for(int i=0;i<MAX_FRAMES-1;i++) {
                        page_queue[i] = page_queue[i+1];
                }
                frames[MAX_FRAMES-1] = page;
                page_queue[MAX_FRAMES-1] = page;

        }
}


int main() {
        int n;
        printf("Enter the number of page references: ");
        scanf("%d",&n);
        int pages[n];
        printf("Enter the page reference sequence: ");

        for(int i=0;i<n;i++) {
                scanf("%d",&pages[i]);
        }

        initialize();
        printf("\n FIFO Page replacement algorithm:\n");
        frame_count = 0;
        page_fault_count = 0;

        for(int i=0;i<n;i++) {
                FIFO(pages[i]);
                displayFrames();
        }

        printf("Total Page Faults(FIFO): %d\n",page_fault_count);
        initialize();
        printf("\nLRU Page Replacement Algorithm: \n");
        frame_count = 0;
```

```
        page_fault_count = 0;

        for(int i=0;i<n;i++) {
                LRU(pages[i]);
                displayFrames();
        }

        printf("Total Page faults(LRU): %d\n",page_fault_count);
        return 0;
}
```

Output:

```
Enter the number of page references: 10
Enter the page reference sequence: 2 3 1 4 2 1 3 7 6 1

 FIFO Page replacement algorithm:
Frames: 2
Frames: 2 3
Frames: 2 3 1
Frames: 4 3 1
Frames: 4 2 1
Frames: 4 2 1
Frames: 7 2 1
Frames: 7 6 1
Frames: 7 6 1
Frames: 7 6 1
Total Page Faults(FIFO): 8

LRU Page Replacement Algorithm:
Frames: 2
Frames: 2 3
Frames: 2 3 1
Frames: 4 3 1
Frames: 4 2 1
Frames: 4 2 1
Frames: 7 2 1
Frames: 7 6 1
Frames: 7 6 1
Frames: 7 6 1
Total Page faults(LRU): 8
```

8. Simulate following File Organization Techniques a) Single Level Directory b) Two Level Directory

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

#define MAX_FILES 100
#define MAX_FILENAME_LENGTH 50

struct File {
   char filename[MAX_FILENAME_LENGTH];
   char content[100];
};

struct Directory {
   char dirname[MAX_FILENAME_LENGTH];
   int file_count;
   struct File files[MAX_FILES];
};

struct SingleLevelFileSystem {
   int dir_count;
   struct Directory directories[MAX_FILES];
};

struct TwoLevelFileSystem {
   int user_count;
   struct Directory users[MAX_FILES];
};

// Initialize the single-level file system
void initializeSingleLevelFileSystem(struct SingleLevelFileSystem *fs) {
   fs->dir_count = 0;
}

// Initialize the two-level file system
void initializeTwoLevelFileSystem(struct TwoLevelFileSystem *fs) {
   fs->user_count = 0;
}

// Create a directory in the single-level file system
void createDirectorySingleLevel(struct SingleLevelFileSystem *fs, char
dirname[MAX_FILENAME_LENGTH]) {
   if (fs->dir_count < MAX_FILES) {
      strcpy(fs->directories[fs->dir_count].dirname, dirname);
      fs->directories[fs->dir_count].file_count = 0;
      fs->dir_count++;
      printf("Directory '%s' created.\n", dirname);
```

```
    } else {
        printf("Cannot create directory '%s'. Maximum directory count reached.\n",
dirname);
    }
}

// Create a directory in the two-level file system
void createDirectoryTwoLevel(struct TwoLevelFileSystem *fs, char
username[MAX_FILENAME_LENGTH], char dirname[MAX_FILENAME_LENGTH])
{
    int user_index = -1;

    // Find the user index
    for (int i = 0; i < fs->user_count; i++) {
        if (strcmp(fs->users[i].dirname, username) == 0) {
            user_index = i;
            break;
        }
    }

    if (user_index == -1) {
        if (fs->user_count < MAX_FILES) {
            strcpy(fs->users[fs->user_count].dirname, username);
            user_index = fs->user_count;
            fs->user_count++;
        } else {
            printf("Cannot create directory '%s' for user '%s'. Maximum user count
reached.\n", dirname, username);
            return;
        }
    }

    if (fs->users[user_index].file_count < MAX_FILES) {
        strcpy(fs->users[user_index].directories[fs->users[user_index].file_count].dirname,
dirname);
        fs->users[user_index].directories[fs->users[user_index].file_count].file_count = 0;
        fs->users[user_index].file_count++;
        printf("Directory '%s/%s' created.\n", username, dirname);
    } else {
        printf("Cannot create directory '%s/%s'. Maximum directory count reached for user
'%s'.\n", username, dirname, username);
    }
}

// Create a file in a directory in the single-level file system
void createFileSingleLevel(struct SingleLevelFileSystem *fs, char
dirname[MAX_FILENAME_LENGTH], char filename[MAX_FILENAME_LENGTH],
char content[100]) {
    int dir_index = -1;
```

```
    // Find the directory index
    for (int i = 0; i < fs->dir_count; i++) {
        if (strcmp(fs->directories[i].dirname, dirname) == 0) {
            dir_index = i;
            break;
        }
    }

    if (dir_index == -1) {
        printf("Directory '%s' not found. Cannot create file '%s'.\n", dirname, filename);
        return;
    }

    if (fs->directories[dir_index].file_count < MAX_FILES) {
        strcpy(fs->directories[dir_index].files[fs-
>directories[dir_index].file_count].filename, filename);
        strcpy(fs->directories[dir_index].files[fs->directories[dir_index].file_count].content,
content);
        fs->directories[dir_index].file_count++;
        printf("File '%s/%s' created.\n", dirname, filename);
    } else {
        printf("Cannot create file '%s/%s'. Maximum file count reached for directory
'%s'.\n", dirname, filename, dirname);
    }
}

// Create a file in a directory in the two-level file system
void createFileTwoLevel(struct TwoLevelFileSystem *fs, char
username[MAX_FILENAME_LENGTH], char dirname[MAX_FILENAME_LENGTH],
char filename[MAX_FILENAME_LENGTH], char content[100]) {
    int user_index = -1;

    // Find the user index
    for (int i = 0; i < fs->user_count; i++) {
        if (strcmp(fs->users[i].dirname, username) == 0) {
            user_index = i;
            break;
        }
    }

    if (user_index == -1) {
        printf("User '%s' not found. Cannot create file '%s/%s'.\n", username, dirname,
filename);
        return;
    }

    int dir_index = -1;

    // Find the directory index
    for (int i = 0; i < fs->users[user_index].file_count; i++) {
```

```
        if (strcmp(fs->users[user_index].directories[i].dirname, dirname) == 0) {
            dir_index = i;
            break;
        }
    }

    if (dir_index == -1) {
        printf("Directory '%s/%s' not found. Cannot create file '%s/%s'.\n", username,
dirname, dirname, filename);
        return;
    }

    if (fs->users[user_index].directories[dir_index].file_count < MAX_FILES) {
        strcpy(fs->users[user_index].directories[dir_index].files[fs-
>users[user_index].directories[dir_index].file_count].filename, filename);
        strcpy(fs->users[user_index].directories[dir_index].files[fs-
>users[user_index].directories[dir_index].file_count].content, content);
        fs->users[user_index].directories[dir_index].file_count++;
        printf("File '%s/%s/%s' created.\n", username, dirname, filename);
    } else {
        printf("Cannot create file '%s/%s/%s'. Maximum file count reached for directory
'%s/%s'.\n", username, dirname, filename, username, dirname);
    }
}

// List files in a directory in the single-level file system
void listFilesSingleLevel(struct SingleLevelFileSystem *fs, char
dirname[MAX_FILENAME_LENGTH]) {
    int dir_index = -1;

    // Find the directory index
    for (int i = 0; i < fs->dir_count; i++) {
        if (strcmp(fs->directories[i].dirname, dirname) == 0) {
            dir_index = i;
            break;
        }
    }

    if (dir_index == -1) {
        printf("Directory '%s' not found.\n", dirname);
        return;
    }

    printf("Files in directory '%s':\n", dirname);
    for (int i = 0; i < fs->directories[dir_index].file_count; i++) {
        printf("- %s\n", fs->directories[dir_index].files[i].filename);
    }
}

// List files in a directory in the two-level file system
```

```
void listFilesTwoLevel(struct TwoLevelFileSystem *fs, char
username[MAX_FILENAME_LENGTH], char dirname[MAX_FILENAME_LENGTH])
{
    int user_index = -1;

    // Find the user index
    for (int i = 0; i < fs->user_count; i++) {
        if (strcmp(fs->users[i].dirname, username) == 0) {
            user_index = i;
            break;
        }
    }

    if (user_index == -1) {
        printf("User '%s' not found.\n", username);
        return;
    }

    int dir_index = -1;

    // Find the directory index
    for (int i = 0; i < fs->users[user_index].file_count; i++) {
        if (strcmp(fs->users[user_index].directories[i].dirname, dirname) == 0) {
            dir_index = i;
            break;
        }
    }

    if (dir_index == -1) {
        printf("Directory '%s/%s' not found.\n", username, dirname);
        return;
    }

    printf("Files in directory '%s/%s':\n", username, dirname);
    for (int i = 0; i < fs->users[user_index].directories[dir_index].file_count; i++) {
        printf("- %s\n", fs->users[user_index].directories[dir_index].files[i].filename);
    }
}

int main() {
    int choice;
    struct SingleLevelFileSystem singleLevelFS;
    struct TwoLevelFileSystem twoLevelFS;

    initializeSingleLevelFileSystem(&singleLevelFS);
    initializeTwoLevelFileSystem(&twoLevelFS);

    while (1) {
        printf("\nFile Organization Techniques:\n");
        printf("1. Single-Level Directory\n");
```

```
printf("2. Two-Level Directory\n");
printf("3. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
   case 1: {
      int singleLevelChoice;
      printf("\nSingle-Level Directory:\n");
      printf("1. Create Directory\n");
      printf("2. Create File\n");
      printf("3. List Files\n");
      printf("4. Back to Main Menu\n");
      printf("Enter your choice: ");
      scanf("%d", &singleLevelChoice);

      switch (singleLevelChoice) {
         case 1: {
            char dirname[MAX_FILENAME_LENGTH];
            printf("Enter the directory name: ");
            scanf("%s", dirname);
            createDirectorySingleLevel(&singleLevelFS, dirname);
            break;
         }
         case 2: {
            char  dirname[MAX_FILENAME_LENGTH];
            char filename[MAX_FILENAME_LENGTH];
            char content[100];
            printf("Enter the directory name: ");
            scanf("%s", dirname);
            printf("Enter the file name: ");
            scanf("%s", filename);
            printf("Enter the file content: ");
            scanf("%s", content);
            createFileSingleLevel(&singleLevelFS, dirname, filename, content);
            break;
         }
         case 3: {
            char dirname[MAX_FILENAME_LENGTH];
            printf("Enter the directory name: ");
            scanf("%s", dirname);
            listFilesSingleLevel(&singleLevelFS, dirname);
            break;
         }
         case 4:
            break;
         default:
            printf("Invalid choice.\n");
      }
      break;
```

```
            }
            case 2: {
                int twoLevelChoice;
                printf("\nTwo-Level Directory:\n");
                printf("1. Create User\n");
                printf("2. Create Directory\n");
                printf("3. Create File\n");
                printf("4. List Files\n");
                printf("5. Back to Main Menu\n");
                printf("Enter your choice: ");
                scanf("%d", &twoLevelChoice);

                switch (twoLevelChoice) {
                    case 1: {
                        char username[MAX_FILENAME_LENGTH];
                        printf("Enter the username: ");
                        scanf("%s", username);
                        createDirectoryTwoLevel(&twoLevelFS, username, username);
                        break;
                    }
                    case 2: {
                        char username[MAX_FILENAME_LENGTH];
                        char dirname[MAX_FILENAME_LENGTH];
                        printf("Enter the username: ");
                        scanf("%s", username);
                        printf("Enter the directory name: ");
                        scanf("%s", dirname);
                        createDirectoryTwoLevel(&twoLevelFS, username, dirname);
                        break;
                    }
                    case 3: {
                        char username[MAX_FILENAME_LENGTH];
                        char dirname[MAX_FILENAME_LENGTH];
                        char filename[MAX_FILENAME_LENGTH];
                        char content[100];
                        printf("Enter the username: ");
                        scanf("%s", username);
                        printf("Enter the directory name: ");
                        scanf("%s", dirname);
                        printf("Enter the file name: ");
                        scanf("%s", filename);
                        printf("Enter the file content: ");
                        scanf("%s", content);
                        createFileTwoLevel(&twoLevelFS, username, dirname, filename,
    content);
                        break;
                    }
                    case 4: {
                        char username[MAX_FILENAME_LENGTH];
                        char dirname[MAX_FILENAME_LENGTH];
```

```
                printf("Enter the username: ");
                scanf("%s", username);
                printf("Enter the directory name: ");
                scanf("%s", dirname);
                listFilesTwoLevel(&twoLevelFS, username, dirname);
                break;
            }
            case 5:
                break;
            default:
                printf("Invalid choice.\n");
        }
        break;
    }
    case 3:
        printf("Exiting...\n");
        exit(0);
    default:
        printf("Invalid choice.\n");
  }
}

  return 0;
}
```

Output:

File Organization Techniques:
1. Single-Level Directory
2. Two-Level Directory
3. Exit
Enter your choice: 1

Single-Level Directory:
1. Create Directory
2. Create File
3. List Files
4. Back to Main Menu
Enter your choice: 1
Enter the directory name: documents
Directory 'documents' created.

Single-Level Directory:
1. Create Directory
2. Create File
3. List Files
4. Back to Main Menu
Enter your choice: 2
Enter the directory name: documents

```
Enter the file name: report.txt
Enter the file content: This is a sample report.
File 'documents/report.txt' created.

Single-Level Directory:
1. Create Directory
2. Create File
3. List Files
4. Back to Main Menu
Enter your choice: 3
Enter the directory name: documents
Files in directory 'documents':
- report.txt

Single-Level Directory:
1. Create Directory
2. Create File
3. List Files
4. Back to Main Menu
Enter your choice: 4

File Organization Techniques:
1. Single-Level Directory
2. Two-Level Directory
3. Exit
Enter your choice: 2

Two-Level Directory:
1. Create User
2. Create Directory
3. Create File
4. List Files
5. Back to Main Menu
Enter your choice: 1
Enter the username: john
Directory 'john' created.

Two-Level Directory:
1. Create User
2. Create Directory
3. Create File
4. List Files
5. Back to Main Menu
Enter your choice: 2
Enter the username: john
Enter the directory name: photos
Directory 'john/photos' created.

Two-Level Directory:
1. Create User
```

```
2. Create Directory
3. Create File
4. List Files
5. Back to Main Menu
Enter your choice: 3
Enter the username: john
Enter the directory name: photos
Enter the file name: beach.jpg
Enter the file content: [Binary content]
File 'john/photos/beach.jpg' created.

Two-Level Directory:
1. Create User
2. Create Directory
3. Create File
4. List Files
5. Back to Main Menu
Enter your choice: 4
Enter the username: john
Enter the directory name: photos
Files in directory 'john/photos':
- beach.jpg

Two-Level Directory:
1. Create User
2. Create Directory
3. Create File
4. List Files
5. Back to Main Menu
Enter your choice: 5

File Organization Techniques:
1. Single-Level Directory
2. Two-Level Directory
3. Exit
Enter your choice: 3
Exiting...
```

9. Develop a C program to simulate the Linked File Allocation strategies.

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

#define MAX_FILENAME_LENGTH 50

struct File {
   char filename[MAX_FILENAME_LENGTH];
   int start_block; // Starting block index of the file
   int file_size;  // Size of the file in blocks
   struct File* next; // Pointer to the next file in the linked list
};

struct File* file_system = NULL; // Linked list representing the file system

// Function to create a new file and add it to the file system
void createFile(char filename[MAX_FILENAME_LENGTH], int file_size) {
   struct File* new_file = (struct File*)malloc(sizeof(struct File));

   if (new_file == NULL) {
      printf("Error: Memory allocation failed.\n");
      return;
   }

   strcpy(new_file->filename, filename);
   new_file->file_size = file_size;

   // Find an empty block for the new file
   int block_index = 0;
   struct File* current_file = file_system;
   while (current_file != NULL) {
      if (block_index + file_size <= current_file->start_block) {
         break; // Found an empty block
      }
      block_index = current_file->start_block + current_file->file_size;
      current_file = current_file->next;
   }

   new_file->start_block = block_index;
   new_file->next = NULL;

   // Add the new file to the file system linked list
   if (file_system == NULL) {
      file_system = new_file;
   } else {
```

```
        current_file = file_system;
        while (current_file->next != NULL) {
            current_file = current_file->next;
        }
        current_file->next = new_file;
    }

    printf("File '%s' created with size %d blocks starting from block %d.\n", filename,
file_size, block_index);
}

void deleteFile(char filename[MAX_FILENAME_LENGTH]) {
    struct File* current_file = file_system;
    struct File* prev_file = NULL;

    while (current_file != NULL) {
        if (strcmp(current_file->filename, filename) == 0) {
            if (prev_file == NULL) {
                file_system = current_file->next;
            } else {
                prev_file->next = current_file->next;
            }
            free(current_file);
            printf("File '%s' deleted.\n", filename);
            return;
        }
        prev_file = current_file;
        current_file = current_file->next;
    }

    printf("File '%s' not found.\n", filename);
}

// Function to list all files in the file system
void listFiles() {
    struct File* current_file = file_system;

    if (current_file == NULL) {
        printf("No files in the file system.\n");
        return;
    }

    printf("Files in the file system:\n");
    while (current_file != NULL) {
        printf("- File: %s, Size: %d blocks, Start Block: %d\n", current_file->filename,
current_file->file_size, current_file->start_block);
        current_file = current_file->next;
    }
}
```

```
int main() {
    int choice;

    while (1) {
        printf("\nLinked File Allocation:\n");
        printf("1. Create File\n");
        printf("2. Delete File\n");
        printf("3. List Files\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                char filename[MAX_FILENAME_LENGTH];
                int file_size;
                printf("Enter the filename: ");
                scanf("%s", filename);
                printf("Enter the file size in blocks: ");
                scanf("%d", &file_size);
                createFile(filename, file_size);
                break;
            }
            case 2: {
                char filename[MAX_FILENAME_LENGTH];
                printf("Enter the filename to delete: ");
                scanf("%s", filename);
                deleteFile(filename);
                break;
            }
            case 3:
                listFiles();
                break;
            case 4:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }

    return 0;
}
```

Output:

Linked File Allocation:
1. Create File
2. Delete File
3. List Files
4. Exit

```
Enter your choice: 1
Enter the filename: document.txt
Enter the file size in blocks: 3
File 'document.txt' created with size 3 blocks starting from block 0.

Linked File Allocation:
1. Create File
2. Delete File
3. List Files
4. Exit
Enter your choice: 1
Enter the filename: image.jpg
Enter the file size in blocks: 2
File 'image.jpg' created with size 2 blocks starting from block 3.

Linked File Allocation:
1. Create File
2. Delete File
3. List Files
4. Exit
Enter your choice: 3
Files in the file system:
- File: document.txt, Size: 3 blocks, Start Block: 0
- File: image.jpg, Size: 2 blocks, Start Block: 3

Linked File Allocation:
1. Create File
2. Delete File
3. List Files
4. Exit
Enter your choice: 2
Enter the filename to delete: document.txt
File 'document.txt' deleted.

Linked File Allocation:
1. Create File
2. Delete File
3. List Files
4. Exit
Enter your choice: 3
Files in the file system:
- File: image.jpg, Size: 2 blocks, Start Block: 3

Linked File Allocation:
1. Create File
2. Delete File
3. List Files
4. Exit
Enter your choice: 4
Exiting...
```

10. Develop a C program to simulate SCAN disk scheduling algorithm.

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

#define MAX_CYLINDERS 200
#define MIN_CYLINDERS 0

// Function to sort an array in ascending order
void sort(int arr[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to simulate the SCAN disk scheduling algorithm
int SCAN(int request[], int n, int start) {
    int total_seek_time = 0;

    // Sort the request array in ascending order
    sort(request, n);

    int current = start;
    int direction = 1; // 1 for right, -1 for left

    printf("Sequence of disk head movement:\n");

    // Find the position of the current request in the sorted array
    int i;
    for (i = 0; i < n; i++) {
        if (request[i] >= start) {
            break;
        }
    }

    while (i < n) {
        if (direction == 1) {
            for (; i < n; i++) {
                if (request[i] > current) {
                    int seek = request[i] - current;
                    printf("Move from %d to %d (seek: %d)\n", current, request[i], seek);
                    total_seek_time += seek;
```

```
                current = request[i];
            }
        }
        direction = -1; // Change direction to left
    } else {
        for (i = i - 1; i >= 0; i--) {
            if (request[i] < current) {
                int seek = current - request[i];
                printf("Move from %d to %d (seek: %d)\n", current, request[i], seek);
                total_seek_time += seek;
                current = request[i];
            }
        }
        direction = 1; // Change direction to right
    }
}

return total_seek_time;
}

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

    if (n <= 0 || n > MAX_CYLINDERS) {
        printf("Invalid number of disk requests. Please enter a value between 1 and
%d.\n", MAX_CYLINDERS);
        return 1;
    }

    int request[n];
    printf("Enter the disk requests: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &request[i]);
        if (request[i] < MIN_CYLINDERS || request[i] >= MAX_CYLINDERS) {
            printf("Invalid disk request. Please enter a value between %d and %d.\n",
MIN_CYLINDERS, MAX_CYLINDERS - 1);
            return 1;
        }
    }

    printf("Enter the starting position of the disk head: ");
    scanf("%d", &start);

    if (start < MIN_CYLINDERS || start >= MAX_CYLINDERS) {
        printf("Invalid starting position. Please enter a value between %d and %d.\n",
MIN_CYLINDERS, MAX_CYLINDERS - 1);
        return 1;
    }
```

```
    int total_seek_time = SCAN(request, n, start);
    printf("Total seek time: %d\n", total_seek_time);

    return 0;
}
```

Output:

```
Enter the number of disk requests: 6
Enter the disk requests: 82 170 43 140 24 16
Enter the starting position of the disk head: 50
Sequence of disk head movement:
Move from 50 to 82 (seek: 32)
Move from 82 to 140 (seek: 58)
Move from 140 to 170 (seek: 30)
Move from 170 to 24 (seek: 146)
Move from 24 to 16 (seek: 8)
Total seek time: 274
```