Experiment-1:Develop a c program to implement the Process system calls *fork*, exec, wait, create process, terminate process

```c
//1. Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid == -1) {
        // Error handling for fork failure
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process\n");
        // Use exec() to execute another program
        execl("/bin/ls", "ls", "-l", NULL);
    } else {
        // Parent process
        printf("Parent process\n");
        // Wait for the child process to finish
        wait(NULL);
        printf("Child process finished\n");
    }

    return 0;
}
```

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

```
//2. Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority
#include<stdio.h>

// Function to find the waiting time for all processes
void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
    wt[0] = 0;

    for (int i = 1; i < n ; i++ )
        wt[i] = bt[i-1] + wt[i-1];
}

// Function to find the turnaround time for all processes
void findTurnaroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++)
        tat[i] = bt[i] + wt[i];
}

// Function to calculate average time
void findAverageTime(int processes[], int n, int bt[]) {
    int wt[n], tat[n];

    // Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt);

    // Function to find turnaround time for all processes
    findTurnaroundTime(processes, n, bt, wt, tat);

    // Display processes along with their waiting time and turnaround time
    printf("\nProcess Burst Time Waiting"
           " Time Turn-Around Time\n");
    int total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        printf(" %d ", (i + 1));
        printf(" %d ", bt[i]);
        printf(" %d", wt[i]);
        printf(" %d\n", tat[i]);
    }
```

```
40
41      printf("\nAverage waiting time = %.2f", (float)total_wt / (float)n);
42      printf("\nAverage turn-around time = %.2f", (float)total_tat / (float)n);
43  }
44
45  // Function to implement FCFS scheduling algorithm
46  void FCFS(int processes[], int n, int bt[]) {
47      findAverageTime(processes, n, bt);
48  }
49
50  // Function to implement SJF scheduling algorithm
51  void SJF(int processes[], int n, int bt[]) {
52      // Sort processes based on burst time
53      for (int i = 0; i < n-1; i++)
54          for (int j = 0; j < n-i-1; j++)
55              if (bt[j] > bt[j+1]) {
56                  // Swap the processes
57                  int temp = bt[j];
58                  bt[j] = bt[j+1];
59                  bt[j+1] = temp;
60
61                  temp = processes[j];
62                  processes[j] = processes[j+1];
63                  processes[j+1] = temp;
64              }
65
66      findAverageTime(processes, n, bt);
67  }
68
69  // Function to implement Round Robin scheduling algorithm
70  void RoundRobin(int processes[], int n, int bt[], int quantum) {
71      int wt[n], tat[n];
72
73      // Initialize waiting time and turnaround time arrays
74      for (int i = 0; i < n; i++) {
75          wt[i] = 0;
76          tat[i] = 0;
77      }
78
79      int time = 0; // Current time
80
81      // Run until all processes are completed
82      while (1) {
83          int done = 1;
```

```
84
85          // Traverse all processes
86          for (int i = 0; i < n; i++) {
87              // If burst time is greater than 0, process i is not done
88              if (bt[i] > 0) {
89                  done = 0;
90
91                  // If remaining burst time is less than or equal to quantum, finish the process
92                  if (bt[i] <= quantum) {
93                      time += bt[i];
94                      wt[i] = time - bt[i];
95                      bt[i] = 0;
96                  } else {
97                      // Reduce the burst time by the quantum time
98                      time += quantum;
99                      bt[i] -= quantum;
100                 }
101             }
102         }
103
104         // If all processes are done, exit the loop
105         if (done == 1)
106             break;
107     }
108
109     // Calculate turnaround time
110     for (int i = 0; i < n; i++)
111         tat[i] = wt[i] + bt[i];
112
113     // Display processes along with their waiting time and turnaround time
114     printf("\nProcess Burst Time Waiting"
115            " Time Turn-Around Time\n");
116     int total_wt = 0, total_tat = 0;
117     for (int i = 0; i < n; i++) {
118         total_wt = total_wt + wt[i];
119         total_tat = total_tat + tat[i];
120         printf(" %d ", (i + 1));
121         printf(" %d ", bt[i]);
122         printf(" %d", wt[i]);
123         printf(" %d\n", tat[i]);
124     }
125
126     printf("\nAverage waiting time = %.2f", (float)total_wt / (float)n);
127     printf("\nAverage turn-around time = %.2f", (float)total_tat / (float)n);
```

```
}

// Function to implement Priority scheduling algorithm
void Priority(int processes[], int n, int bt[], int priority[]) {

        printf("Enter the priority for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Priority for Process %d: ", i + 1);
        scanf("%d", &priority[i]);
    }
    // Sort processes based on priority
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (priority[j] > priority[j+1]) {
                // Swap the processes
                int temp = priority[j];
                priority[j] = priority[j+1];
                priority[j+1] = temp;

                temp = bt[j];
                bt[j] = bt[j+1];
                bt[j+1] = temp;

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

    findAverageTime(processes, n, bt);
}

int main() {
    int n; // Number of processes

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

    int processes[n]; // Array to store process IDs
    int bt[n]; // Array to store burst times
    int priority[n]; // Array to store priority values

    printf("Enter the burst times for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
```

```
        printf("Burst time for Process %d: ", i + 1);
        scanf("%d", &bt[i]);
    }



     // FCFS Scheduling
     // RoundRobin(processes, n, bt,3);
      FCFS(processes, n, bt);
      //SJF(processes, n, bt);

}
/* FCFS Scheduling - Expected Output for FCFS
Enter the number of processes: 3
Enter the burst times for each process:
Burst time for Process 1: 5
Burst time for Process 2: 1
Burst time for Process 3: 2

Process Burst Time Waiting Time Turn-Around Time
 1 5 0 5
 2 1 5 6
 3 2 6 8

Average waiting time = 3.67
Average turn-around time = 6.33
```

Experiment-3:Develop a C program to simulate producer-consumer problem using semaphores

```c
// 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, out = 0;

sem_t mutex, empty, full;

void *producer(void *arg) {
    int item = 1;

    while (1) {
        sem_wait(&empty);
        sem_wait(&mutex);

        // Produce item
        buffer[in] = item;
        printf("Produced item %d\n", item);
        in = (in + 1) % BUFFER_SIZE;
        item++;

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

        sleep(1); // Simulate production time
    }

    pthread_exit(NULL);
}

void *consumer(void *arg) {
    int item;

    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);

```

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

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

        sleep(2); // Simulate consumption time
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t producer_thread, consumer_thread;

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

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

    // Join threads
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

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

    return 0;
}

In this program:

mutex semaphore is used to provide mutual exclusion while accessing the buffer.
empty semaphore is used to track the number of empty slots in the buffer.
full semaphore is used to track the number of filled slots in the buffer.
The producer waits on empty semaphore before producing an item and signals full
```

```
    ↪ semaphore after producing an item.
87 The consumer waits on full semaphore before consuming an item and signals empty
    ↪ semaphore after consuming an item.
88 sleep() functions are used to simulate production and consumption time.
```

Experiment-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.

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

#define FIFO_NAME "myfifo"
#define BUFFER_SIZE 256

int main() {
    int fd;
    char message[BUFFER_SIZE];

    // Create the FIFO (named pipe)
    mkfifo(FIFO_NAME, 0666);

    // Writer process
    if (fork() == 0) {
        // Open the FIFO for writing
        fd = open(FIFO_NAME, O_WRONLY);
        if (fd == -1) {
            perror("open");
            exit(EXIT_FAILURE);
        }

        // Write messages to the FIFO
        while (1) {
            printf("Enter message to send: ");
            fgets(message, BUFFER_SIZE, stdin);
            write(fd, message, strlen(message) + 1);
        }

        // Close the FIFO
        close(fd);
    }
    // Reader process
```

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

        // Read messages from the FIFO
        while (1) {
            if (read(fd, message, BUFFER_SIZE) > 0) {
                printf("Received message: %s", message);
            }
        }

        // Close the FIFO
        close(fd);
    }

    return 0;
}
```

In this program:

We create a named pipe (FIFO) using mkfifo.
The writer process opens the FIFO for writing (O_WRONLY), writes messages entered by the user to the FIFO using write, and then closes the FIFO.
The reader process opens the FIFO for reading (O_RDONLY), reads messages from the FIFO using read, and prints them to the console.
The communication between the writer and reader processes is achieved through the named pipe. Each process can read from or write to the pipe independently.
The program continues running indefinitely until manually terminated.
Compile and run the program, and you can enter messages in the writer process, which will be read and displayed by the reader process.

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

```c
//Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance
#include <stdio.h>
#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

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

int nProcesses, nResources;

// Function declarations
void initialize();
void input();
void display();
int isSafe();
void requestResource(int processNumber);
void releaseResource(int processNumber);

int main() {
    initialize();
    input();
    display();

    if (isSafe()) {
        printf("\nSystem is in safe state.\n");

        // Example: Request and release resources
        int processNumber = 0;
        requestResource(processNumber);
        display();

        releaseResource(processNumber);
        display();
    } else {
        printf("\nSystem is in unsafe state.\n");
    }

    return 0;
```

```
}

void initialize() {
    printf("Enter the number of processes: ");
    scanf("%d", &nProcesses);

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

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

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

void input() {
    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < nProcesses; ++i) {
        for (int j = 0; j < nResources; ++j) {
            scanf("%d", &allocation[i][j]);
            need[i][j] = maximum[i][j] - allocation[i][j];
        }
        finish[i] = 0;
    }
}

void display() {
    printf("\nAvailable Resources:\n");
    for (int j = 0; j < nResources; ++j) {
        printf("%d ", available[j]);
    }

    printf("\n\nMaximum Resources:\n");
    for (int i = 0; i < nProcesses; ++i) {
        for (int j = 0; j < nResources; ++j) {
            printf("%d ", maximum[i][j]);
        }
```

```
87          printf("\n");
88      }
89
90      printf("\nAllocation Matrix:\n");
91      for (int i = 0; i < nProcesses; ++i) {
92          for (int j = 0; j < nResources; ++j) {
93              printf("%d ", allocation[i][j]);
94          }
95          printf("\n");
96      }
97  }
98
99  int isSafe() {
100     for (int j = 0; j < nResources; ++j) {
101         work[j] = available[j];
102     }
103
104     int count = 0;
105     while (count < nProcesses) {
106         int found = 0;
107         for (int i = 0; i < nProcesses; ++i) {
108             if (finish[i] == 0) {
109                 int j;
110                 for (j = 0; j < nResources; ++j) {
111                     if (need[i][j] > work[j]) {
112                         break;
113                     }
114                 }
115                 if (j == nResources) {
116                     for (int k = 0; k < nResources; ++k) {
117                         work[k] += allocation[i][k];
118                     }
119                     finish[i] = 1;
120                     found = 1;
121                     ++count;
122                 }
123             }
124         }
125         if (found == 0) {
126             break;
127         }
128     }
129
130     return (count == nProcesses);
```

```
}

void requestResource(int processNumber) {
    printf("\nEnter the resource request for process %d:\n", processNumber);
    for (int j = 0; j < nResources; ++j) {
        int request;
        scanf("%d", &request);

        if (request <= need[processNumber][j] && request <= available[j]) {
            allocation[processNumber][j] += request;
            need[processNumber][j] -= request;
            available[j] -= request;
        } else {
            printf("\nInvalid request. Process must wait.\n");
            break;
        }
    }
}

void releaseResource(int processNumber) {
    printf("\nEnter the resource release for process %d:\n", processNumber);
    for (int j = 0; j < nResources; ++j) {
        int release;
        scanf("%d", &release);

        if (release <= allocation[processNumber][j]) {
            allocation[processNumber][j] -= release;
            need[processNumber][j] += release;
            available[j] += release;
        } else {
            printf("\nInvalid release. Process must wait.\n");
            break;
        }
    }
}

//Expected Output
Enter the number of processes: 5
Enter the number of resources: 4
Enter the available resources:
1
2
1
1
```

```
Enter the maximum resources for process 0:
1
1
1
1
Enter the maximum resources for process 1:
2
2
2
2
Enter the maximum resources for process 2:
4
5
5
3
Enter the maximum resources for process 3:
4
1
1
1
Enter the maximum resources for process 4:
1
1
1
1
Enter the allocation matrix:
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
```

```
1
1

Available Resources:
1 2 1 1

Maximum Resources:
1 1 1 1
2 2 2 2
4 5 5 3
4 1 1 1
1 1 1 1

Allocation Matrix:
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1

System is in safe state.
```

Experiment-6:Develop a C program to simulate the following contiguous memory allocation Techniques: a Worst fit b Best fit c First fit

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

#include <stdio.h>
#define MAX_MEMORY_SIZE 1000
#define MAX_PROCESSES 10

typedef struct {
    int pid; // Process ID
    int size; // Process size
    int allocated; // Allocation status (1 for allocated, 0 for unallocated)
} Process;

void worstFit(Process processes[], int numProcesses, int memory[], int numMemoryBlocks) {
    for (int i = 0; i < numProcesses; i++) {
        int worstFitIndex = -1;

        for (int j = 0; j < numMemoryBlocks; j++) {
            if (memory[j] >= processes[i].size) {
                if (worstFitIndex == -1 || memory[j] > memory[worstFitIndex]) {
                    worstFitIndex = j;
                }
            }
        }

        if (worstFitIndex != -1) {
            // Allocate memory to the process
            processes[i].allocated = 1;
            memory[worstFitIndex] -= processes[i].size;
            printf("Process %d allocated to memory block %d using Worst Fit\n", processes[i].pid, worstFitIndex);
        } else {
            printf("Process %d cannot be allocated using Worst Fit\n", processes[i].pid);
        }
    }
}

void bestFit(Process processes[], int numProcesses, int memory[], int
```

```
    ↪ numMemoryBlocks) {
    for (int i = 0; i < numProcesses; i++) {
        int bestFitIndex = -1;

        for (int j = 0; j < numMemoryBlocks; j++) {
            if (memory[j] >= processes[i].size) {
                if (bestFitIndex == -1 || memory[j] < memory[bestFitIndex]) {
                    bestFitIndex = j;
                }
            }
        }

        if (bestFitIndex != -1) {
            // Allocate memory to the process
            processes[i].allocated = 1;
            memory[bestFitIndex] -= processes[i].size;
            printf("Process %d allocated to memory block %d using Best Fit\n",
                ↪ processes[i].pid, bestFitIndex);
        } else {
            printf("Process %d cannot be allocated using Best Fit\n",
                ↪ processes[i].pid);
        }
    }
}

void firstFit(Process processes[], int numProcesses, int memory[], int
    ↪ numMemoryBlocks) {
    for (int i = 0; i < numProcesses; i++) {
        for (int j = 0; j < numMemoryBlocks; j++) {
            if (memory[j] >= processes[i].size) {
                // Allocate memory to the process
                processes[i].allocated = 1;
                memory[j] -= processes[i].size;
                printf("Process %d allocated to memory block %d using First Fit\n",
                    ↪ processes[i].pid, j);
                break;
            }
        }

        if (!processes[i].allocated) {
            printf("Process %d cannot be allocated using First Fit\n",
                ↪ processes[i].pid);
        }
    }
```

```
}

int main() {
    int memory[MAX_MEMORY_SIZE];
    Process processes[MAX_PROCESSES];

    int numProcesses, numMemoryBlocks;

    printf("Enter the number of memory blocks: ");
    scanf("%d", &numMemoryBlocks);

    printf("Enter the sizes of memory blocks:\n");
    for (int i = 0; i < numMemoryBlocks; i++) {
        scanf("%d", &memory[i]);
    }

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

    printf("Enter the sizes of processes:\n");
    for (int i = 0; i < numProcesses; i++) {
        scanf("%d", &processes[i].size);
        processes[i].pid = i;
        processes[i].allocated = 0;
    }

    // Worst Fit
    worstFit(processes, numProcesses, memory, numMemoryBlocks);

    // Reset memory and processes allocation status
    for (int i = 0; i < numMemoryBlocks; i++) {
        memory[i] = MAX_MEMORY_SIZE;
    }
    for (int i = 0; i < numProcesses; i++) {
        processes[i].allocated = 0;
    }


    // Best Fit
    bestFit(processes, numProcesses, memory, numMemoryBlocks);

    // Reset memory and processes allocation status
    for (int i = 0; i < numMemoryBlocks; i++) {
        memory[i] = MAX_MEMORY_SIZE;
```

```
    }
    for (int i = 0; i < numProcesses; i++) {
        processes[i].allocated = 0;
    }

    // First Fit
    firstFit(processes, numProcesses, memory, numMemoryBlocks);

    return 0;
}

//Expected Output
Enter the number of memory blocks: 3
Enter the sizes of memory blocks:
10
25
1
Enter the number of processes: 2
Enter the sizes of processes:
1
20
Process 0 allocated to memory block 2 using Best Fit
Process 1 allocated to memory block 1 using Best Fit
```

Experiment-7:Develop a C program to simulate page replacement algorithms: a FIFO b LRU

```c
//Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU
#include <stdio.h>
#include <stdbool.h>

#define MAX_PAGES 100
#define MAX_FRAMES 10

void fifo(int pages[], int n, int frames) {
    int pageFaults = 0;
    int frame[MAX_FRAMES];
    bool isPresent[MAX_PAGES] = {false};
    int rear = -1;

    printf("Page\tFrame\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t", pages[i]);

        if (!isPresent[pages[i]]) {
            pageFaults++;
            if (rear < frames - 1) {
                rear++;
            } else {
                rear = 0;
            }
            frame[rear] = pages[i];
            isPresent[pages[i]] = true;
        }

        for (int j = 0; j < frames; j++) {
            if (j <= rear) {
                printf("%d ", frame[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }
    printf("Total Page Faults: %d\n", pageFaults);
}

void lru(int pages[], int n, int frames) {
```

```
    int pageFaults = 0;
    int frame[MAX_FRAMES];
    bool isPresent[MAX_PAGES] = {false};
    int leastUsed[MAX_FRAMES] = {0};
    int count = 0;

    printf("Page\tFrame\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t", pages[i]);

        if (!isPresent[pages[i]]) {
            pageFaults++;
            if (count < frames) {
                frame[count] = pages[i];
                isPresent[pages[i]] = true;
                leastUsed[count] = i;
                count++;
            } else {
                int index = 0;
                for (int j = 1; j < frames; j++) {
                    if (leastUsed[j] < leastUsed[index]) {
                        index = j;
                    }
                }
                isPresent[frame[index]] = false;
                frame[index] = pages[i];
                isPresent[pages[i]] = true;
                leastUsed[index] = i;
            }
        } else {
            for (int j = 0; j < frames; j++) {
                if (frame[j] == pages[i]) {
                    leastUsed[j] = i;
                    break;
                }
            }
        }

        for (int j = 0; j < frames; j++) {
            if (isPresent[pages[i]] && j < count) {
                printf("%d ", frame[j]);
            } else {
                printf("- ");
```

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

int main() {
    int pages[MAX_PAGES];
    int n, frames;
    char choice;

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

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

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

    printf("Choose a page replacement algorithm:\n");
    printf("a) FIFO\n");
    printf("b) LRU\n");
    printf("Enter your choice: ");
    scanf(" %c", &choice);

    switch (choice) {
        case 'a':
            fifo(pages, n, frames);
            break;
        case 'b':
            lru(pages, n, frames);
            break;
        default:
            printf("Invalid choice\n");
    }

    return 0;
}

//Expected Output
```

```
Enter the number of pages: 5
Enter the pages: 1
1
2
4
5
Enter the number of frames: 2
Choose a page replacement algorithm:
a) FIFO
b) LRU
Enter your choice: a
Page Frame
1 1 -
1 1 -
2 1 2
4 4 -
5 4 5
Total Page Faults: 4
```

Experiment-8:Develop a C program to simulate SCAN disk scheduling algorithm

```c
//Develop a C program to simulate SCAN disk scheduling algorithm.
#include <stdio.h>
#include <stdlib.h>
void SCAN(int *requests, int n, int head, char direction) {
    int total_movement = 0;
    int i, j;
    int min_index = 0;
    int max_index = 0;

    // Finding the min and max indexes in the request array
    for (i = 0; i < n; i++) {
        if (requests[i] < requests[min_index])
            min_index = i;
        if (requests[i] > requests[max_index])
            max_index = i;
    }

    if (direction == 'l') { // Left direction
        // Move the head to the minimum index first
        total_movement += abs(head - requests[min_index]);
        printf("%d -> ", requests[min_index]);
        head = requests[min_index];

        // Move towards the beginning of the disk
        for (i = min_index - 1; i >= 0; i--) {
            total_movement += abs(head - requests[i]);
            printf("%d -> ", requests[i]);
            head = requests[i];
        }

        // Move towards the end of the disk
        for (i = min_index + 1; i < n; i++) {
            total_movement += abs(head - requests[i]);
            printf("%d -> ", requests[i]);
            head = requests[i];
        }
    } else if (direction == 'r') { // Right direction
        // Move the head to the maximum index first
        total_movement += abs(head - requests[max_index]);
        printf("%d -> ", requests[max_index]);
        head = requests[max_index];

```

```
        // Move towards the end of the disk
        for (i = max_index + 1; i < n; i++) {
            total_movement += abs(head - requests[i]);
            printf("%d -> ", requests[i]);
            head = requests[i];
        }

        // Move towards the beginning of the disk
        for (i = max_index - 1; i >= 0; i--) {
            total_movement += abs(head - requests[i]);
            printf("%d -> ", requests[i]);
            head = requests[i];
        }
    }

    printf("\nTotal head movement: %d\n", total_movement);
}

int main() {
    int n, head, i;
    char direction;
    int *requests;

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

    requests = (int*)malloc(n * sizeof(int));

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

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

    printf("Enter the direction (l for left, r for right): ");
    scanf(" %c", &direction);

    printf("\nSCAN Schedule:\n");
    SCAN(requests, n, head, direction);

    free(requests);

```

```
    return 0;
}

//Expected Output
Enter the number of requests: 10
Enter the requests: 10
20
30
40
50
60
70
80
90
1
Enter the initial position of the head: 0
Enter the direction (l for left, r for right): l

SCAN Schedule:
1 -> 90 -> 80 -> 70 -> 60 -> 50 -> 40 -> 30 -> 20 -> 10 ->
Total head movement: 170
```

Experiment-9:Simulate following File Organization Techniques a Single level directory b Two level directory

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

#define MAX_BLOCKS 100

// Structure to represent a block in the file system
typedef struct Block {
    int blockNumber;
    struct Block* next;
} Block;

// Structure to represent a file
typedef struct File {
    int fileId;
    Block* blocks;
    struct File* next;
} File;

// Global variables
File* files[MAX_BLOCKS] = {NULL}; // Array to hold pointers to file blocks
int lastBlock = 0; // Variable to track the last allocated block

// Function to allocate a block for a file
Block* allocateBlock(int blockNumber) {
    Block* newBlock = (Block*)malloc(sizeof(Block));
    if (newBlock == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newBlock->blockNumber = blockNumber;
    newBlock->next = NULL;
    return newBlock;
}

// Function to create a new file
void createFile(int fileId) {
    File* newFile = (File*)malloc(sizeof(File));
    if (newFile == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newFile->fileId = fileId;
```

```
    newFile->blocks = NULL;
    newFile->next = NULL;

    // Insert the file into the files array
    files[fileId] = newFile;
    printf("File created with ID %d\n", fileId);
}

// Function to allocate blocks for a file
void allocateBlocks(int fileId, int numBlocks) {
    File* file = files[fileId];
    if (file == NULL) {
        printf("File with ID %d does not exist\n", fileId);
        return;
    }

    // Allocate blocks for the file
    int i;
    for (i = 0; i < numBlocks; i++) {
        Block* newBlock = allocateBlock(++lastBlock);
        newBlock->next = file->blocks;
        file->blocks = newBlock;
    }
    printf("%d blocks allocated for file %d\n", numBlocks, fileId);
}

// Function to delete a file
void deleteFile(int fileId) {
    File* file = files[fileId];
    if (file == NULL) {
        printf("File with ID %d does not exist\n", fileId);
        return;
    }

    // Free blocks allocated to the file
    Block* current = file->blocks;
    while (current != NULL) {
        Block* temp = current;
        current = current->next;
        free(temp);
    }

    // Free file structure
    free(file);
```

```
87      files[fileId] = NULL;
88      printf("File with ID %d deleted\n", fileId);
89  }
90
91  // Function to display allocated blocks for a file
92  void displayFile(int fileId) {
93      File* file = files[fileId];
94      if (file == NULL) {
95          printf("File with ID %d does not exist\n", fileId);
96          return;
97      }
98
99      printf("Blocks allocated for file %d: ", fileId);
100     Block* current = file->blocks;
101     while (current != NULL) {
102         printf("%d ", current->blockNumber);
103         current = current->next;
104     }
105     printf("\n");
106 }
107
108 int main() {
109     int choice, fileId, numBlocks;
110
111     while (1) {
112         printf("\nLinked File Allocation Strategies\n");
113         printf("1. Create File\n");
114         printf("2. Allocate Blocks\n");
115         printf("3. Delete File\n");
116         printf("4. Display File\n");
117         printf("5. Exit\n");
118         printf("Enter your choice: ");
119         scanf("%d", &choice);
120
121         switch (choice) {
122             case 1:
123                 printf("Enter file ID: ");
124                 scanf("%d", &fileId);
125                 createFile(fileId);
126                 break;
127             case 2:
128                 printf("Enter file ID: ");
129                 scanf("%d", &fileId);
130                 printf("Enter number of blocks: ");
```

```
                scanf("%d", &numBlocks);
                allocateBlocks(fileId, numBlocks);
                break;
            case 3:
                printf("Enter file ID to delete: ");
                scanf("%d", &fileId);
                deleteFile(fileId);
                break;
            case 4:
                printf("Enter file ID to display: ");
                scanf("%d", &fileId);
                displayFile(fileId);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}

/*Expected Output
Linked File Allocation Strategies
1. Create File
2. Allocate Blocks
3. Delete File
4. Display File
5. Exit
Enter your choice: 2
Enter file ID: 1
Enter number of blocks: 10
10 blocks allocated for file 1

Linked File Allocation Strategies
1. Create File
2. Allocate Blocks
3. Delete File
4. Display File
5. Exit
Enter your choice: 4
Enter file ID to display: 1
Blocks allocated for file 1: 10 9 8 7 6 5 4 3 2 1
```

```
Linked File Allocation Strategies
1. Create File
2. Allocate Blocks
3. Delete File
4. Display File
5. Exit
Enter your choice: 3
Enter file ID to delete: 1
File with ID 1 deleted
```

Experiment-10:Develop a C program to simulate the Linked file allocation strategies.

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

#define MAX_FILES 100
#define MAX_NAME_LENGTH 50

// Structure to represent a file
typedef struct File {
    char name[MAX_NAME_LENGTH];
    int size;
} File;

// Structure to represent a directory
typedef struct Directory {
    char name[MAX_NAME_LENGTH];
    int numFiles;
    File files[MAX_FILES];
} Directory;

// Single level directory
Directory singleLevelDirectory;

// Two level directory
Directory twoLevelDirectory[MAX_FILES];

// Function to create a file in a single-level directory
void createFileSingleLevel(char* filename, int size) {
    if (singleLevelDirectory.numFiles < MAX_FILES) {
        strcpy(singleLevelDirectory.files[singleLevelDirectory.numFiles].name,
            ↪ filename);
```

```
31          singleLevelDirectory.files[singleLevelDirectory.numFiles].size = size;
32          singleLevelDirectory.numFiles++;
33          printf("File '%s' created successfully in single-level directory\n",
                ↪ filename);
34      } else {
35          printf("Cannot create file. Single-level directory is full.\n");
36      }
37  }
38
39  // Function to delete a file from a single-level directory
40  void deleteFileSingleLevel(char* filename) {
41      int i;
42      for (i = 0; i < singleLevelDirectory.numFiles; i++) {
43          if (strcmp(singleLevelDirectory.files[i].name, filename) == 0) {
44              int j;
45              for (j = i; j < singleLevelDirectory.numFiles - 1; j++) {
46                  singleLevelDirectory.files[j] = singleLevelDirectory.files[j + 1];
47              }
48              singleLevelDirectory.numFiles--;
49              printf("File '%s' deleted successfully from single-level directory\n",
                    ↪ filename);
50              return;
51          }
52      }
53      printf("File '%s' not found in single-level directory\n", filename);
54  }
55
56  // Function to display files in a single-level directory
57  void displayFilesSingleLevel() {
58      printf("Files in single-level directory:\n");
59      int i;
60      for (i = 0; i < singleLevelDirectory.numFiles; i++) {
61          printf("%s\t%d bytes\n", singleLevelDirectory.files[i].name,
                ↪ singleLevelDirectory.files[i].size);
62      }
63  }
64
65  // Function to create a file in a two-level directory
66  void createFileTwoLevel(char* dirname, char* filename, int size) {
67      int i;
68      for (i = 0; i < MAX_FILES; i++) {
69          if (strcmp(twoLevelDirectory[i].name, dirname) == 0) {
70              if (twoLevelDirectory[i].numFiles < MAX_FILES) {
71                  strcpy(twoLevelDirectory[i].files[twoLevelDirectory[i].numFiles].name,
```

```
                ↪ filename);
72              twoLevelDirectory[i].files[twoLevelDirectory[i].numFiles].size =
                ↪ size;
73              twoLevelDirectory[i].numFiles++;
74              printf("File '%s' created successfully in directory '%s' of
                ↪ two-level directory\n", filename, dirname);
75              return;
76          } else {
77              printf("Cannot create file. Directory '%s' is full.\n", dirname);
78              return;
79          }
80      }
81  }
82  printf("Directory '%s' not found in two-level directory\n", dirname);
83 }
84
85 // Function to delete a file from a two-level directory
86 void deleteFileTwoLevel(char* dirname, char* filename) {
87      int i;
88      for (i = 0; i < MAX_FILES; i++) {
89          if (strcmp(twoLevelDirectory[i].name, dirname) == 0) {
90              int j;
91              for (j = 0; j < twoLevelDirectory[i].numFiles; j++) {
92                  if (strcmp(twoLevelDirectory[i].files[j].name, filename) == 0) {
93                      int k;
94                      for (k = j; k < twoLevelDirectory[i].numFiles - 1; k++) {
95                          twoLevelDirectory[i].files[k] = twoLevelDirectory[i].files[k
                            ↪ + 1];
96                      }
97                      twoLevelDirectory[i].numFiles--;
98                      printf("File '%s' deleted successfully from directory '%s' of
                        ↪ two-level directory\n", filename, dirname);
99                      return;
100                 }
101             }
102             printf("File '%s' not found in directory '%s' of two-level
                ↪ directory\n", filename, dirname);
103             return;
104         }
105     }
106     printf("Directory '%s' not found in two-level directory\n", dirname);
107 }
108
109 // Function to display files in a two-level directory
```

```
110 void displayFilesTwoLevel(char* dirname) {
111     int i;
112     for (i = 0; i < MAX_FILES; i++) {
113         if (strcmp(twoLevelDirectory[i].name, dirname) == 0) {
114             printf("Files in directory '%s' of two-level directory:\n", dirname);
115             int j;
116             for (j = 0; j < twoLevelDirectory[i].numFiles; j++) {
117                 printf("%s\t%d bytes\n", twoLevelDirectory[i].files[j].name,
                        ↪ twoLevelDirectory[i].files[j].size);
118             }
119             return;
120         }
121     }
122     printf("Directory '%s' not found in two-level directory\n", dirname);
123 }
124
125 int main() {
126     singleLevelDirectory.numFiles = 0;
127
128     int choice;
129     while (1) {
130         printf("\nDirectory Simulation\n");
131         printf("1. Single-level Directory - Create File\n");
132         printf("2. Single-level Directory - Delete File\n");
133         printf("3. Single-level Directory - Display Files\n");
134         printf("4. Two-level Directory - Create File\n");
135         printf("5. Two-level Directory - Delete File\n");
136         printf("6. Two-level Directory - Display Files\n");
137         printf("7. Exit\n");
138         printf("Enter your choice: ");
139         scanf("%d", &choice);
140
141         char filename[MAX_NAME_LENGTH], dirname[MAX_NAME_LENGTH];
142         int size;
143
144         switch (choice) {
145             case 1:
146                 printf("Enter file name: ");
147                 scanf("%s", filename);
148                 printf("Enter file size (in bytes): ");
149                 scanf("%d", &size);
150                 createFileSingleLevel(filename, size);
151                 break;
152             case 2:
```

```
153                 printf("Enter file name to delete: ");
154                 scanf("%s", filename);
155                 deleteFileSingleLevel(filename);
156                 break;
157             case 3:
158                 displayFilesSingleLevel();
159                 break;
160             case 4:
161                 printf("Enter directory name: ");
162                 scanf("%s", dirname);
163                 printf("Enter file name: ");
164                 scanf("%s", filename);
165                 printf("Enter file size (in bytes): ");
166                 scanf("%d", &size);
167                 createFileTwoLevel(dirname, filename, size);
168                 break;
169             case 5:
170                 printf("Enter directory name: ");
171                 scanf("%s", dirname);
172                 printf("Enter file name to delete: ");
173                 scanf("%s", filename);
174                 deleteFileTwoLevel(dirname, filename);
175                 break;
176             case 6:
177                 printf("Enter directory name: ");
178                 scanf("%s", dirname);
179                 displayFilesTwoLevel(dirname);
180                 break;
181             case 7:
182                 break;
183             }
184         }
185 }
186
187 /*Expected Output
188 Directory Simulation
189 1. Single-level Directory - Create File
190 2. Single-level Directory - Delete File
191 3. Single-level Directory - Display Files
192 4. Two-level Directory - Create File
193 5. Two-level Directory - Delete File
194 6. Two-level Directory - Display Files
195 7. Exit
196 Enter your choice: 1
```

```
Enter file name: aiml
Enter file size (in bytes): 120
File 'aiml' created successfully in single-level directory

Directory Simulation
1. Single-level Directory - Create File
2. Single-level Directory - Delete File
3. Single-level Directory - Display Files
4. Two-level Directory - Create File
5. Two-level Directory - Delete File
6. Two-level Directory - Display Files
7. Exit
Enter your choice: 3
Files in single-level directory:
aiml 120 bytes
```