VISVESVARAYA TECHNOLOGICAL UNIVERSITY

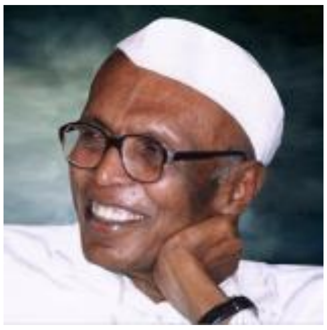
Jnana Sangama, Belagavi-590018

LABORATORY MANUAL

OPERATING SYSTEMS LABORATORY- BCS303

Prepared by,

Mrs. Archana B K



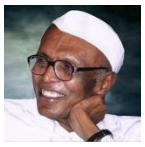
DEPARTMENT OF CSE, ISE, AI & DS

DR. H N NATIONAL COLLEGE OF ENGINEERING

Bengaluru-560070

2025-2026

Dr. H N National College of Engineering
Approved by All India Council for Technical Education (AICTE), Govt. of India and affiliated to Visvesvaraya Technological University (VTU)
36BCross, Jayanagar 7th block, Bengaluru–560070



DEPARTMENT OF CSE, ISE, AI & DS

OPERATING SYSTEMS LABORATORY

BCS303 – 3 SEMESTER

[AS PER OUTCOME BASED EDUCATION (OBE) AND CHOICE BASED CREDIT SYSTEM (CBCS) 2022 SCHEME]

Academic Year–2024-2025 LAB

MANUAL

Prepared by:
Mrs. Archana B K
Department of ECE

Institute Vision and Mission

Our Vision

To become a premier education institute in the country, for providing futuristic knowledge and profound skill in Engineering and Management, to produce global active brains that will provide smart solutions in engineering for sustainable society.

Our Mission

• To impart the technical and managerial knowledge components, over an adorable academic ambiance and enhanced learning over benchmarked syllabus through outcome-based education systems.

• To develop professional skills in the research over creative conducive atmosphere of inter disciplinary research and innovations. To enhance collaborative and IPR skill, through an active MOU with global organizations and clustering in emerging areas.

Program Outcomes (POs)

Engineering Graduates will be able to:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. Design/development of solutions: 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.

4. Conduct investigations of complex problems: 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.

5. Modern tool usage: 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.

6. The engineer and society: 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.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: 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.

11. Project management and finance: 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.

12. Life-long learning: 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.

Course Objectives:

1. To Demonstrate the need for OS and different types of OS
2. To discuss suitable techniques for management of different resources
3. To demonstrate different APIs/Commands related to processor, memory, storage and file system management

Course Outcomes:

At the end of the course student will be able to:

1. Explain the structure and functionality of operating system.
2. Apply appropriate CPU scheduling algorithms for the given problem.
3. Analyse the various techniques for process synchronization and deadlock handling.
4. Apply the various techniques for memory management.
5. Explain file and secondary storage management strategies.
6. Describe the need for information protection mechanisms.

List of Experiments

Sl.No	Name of the Experiment
1	Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process).
2	Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.
3	Develop a C program to simulate producer-consumer problem using semaphores.
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.
5	Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.
6	Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.
7	Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU.
8	Simulate following File Organization Techniques a) Single level directory b) Two level directory.
9	Develop a C program to simulate the Linked file allocation strategies.
10	Develop a C program to simulate SCAN disk scheduling algorithm.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50). The minimum passing mark for the SEE is 35% of the maximum marks (18 marks out of 50). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures not less than 35% (18 Marks out of 50) in the semester-end examination (SEE), and a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

Continuous Internal Evaluation (CIE):

The CIE marks for the theory component of the IC shall be 30 marks and for the laboratory component 20 Marks.

CIE for the practical component of the IC

• On completion of every experiment/program in the laboratory, the students shall be evaluated and marks shall be awarded on the same day. The 15 marks are for conducting the experiment and preparation of the laboratory record, the other 05 marks shall be for the test conducted at the end of the semester.

• 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 03 hours) at the end of the 15th week of the semester /after completion of all the experiments (whichever is early) shall be conducted for 50 marks and scaled down to 05 marks. Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IC/IPCC for 20 marks.

• The minimum marks to be secured in CIE to appear for SEE shall be 12 (40% of maximum marks) in the theory component and 08 (40% of maximum marks) in the practical component. The laboratory component of the IC/IPCC shall be for CIE only. However, in SEE, the questions from the laboratory component shall be included. The maximum of 05 questions is to be set from the practical component of IC/IPCC, the total marks of all questions should not be more than 25 marks. The theory component of the IC shall be for both CIE and SEE.

Semester End Examination (SEE):

SEE for IC

Theory SEE will be conducted by University as per the scheduled time table, with common question papers for the course (duration 03 hours)
1. The question paper will have ten questions. Each question is set for 20 marks.

2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), should have a mix of topics under that module.

3. The students have to answer 5 full questions, selecting one full question from each module. The theory portion of the Integrated Course shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper shall include questions from the practical component).

Passing standard:

• The minimum marks to be secured in CIE to appear for SEE shall be 12 (40% of maximum marks-30) in the theory component and 08 (40% of maximum marks -20) in the practical component. The laboratory component of the IPCC shall be for CIE only. However, in SEE, the questions from the laboratory component shall be included. The maximum of 04/05 questions to be set from the practical component of IPCC, the total marks of all questions should not be more than 30 marks.

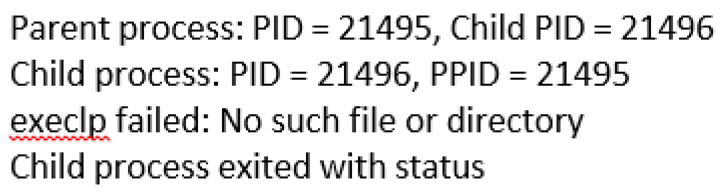
• SEE will be conducted for 100 marks and students shall secure 35% of the maximum marks to qualify for the SEE. Marks secured will be scaled down to 50.

Table of Contents

Program1: Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process).

```
   /* process_syscalls.c
      Demonstrates: fork(), execvp(), wait(), exit()
      Behavior:
        - Parent creates a child with fork()
        - Child executes `ls -l` using execvp()
        - Parent waits for child to terminate and prints child's exit status
      Compile: gcc process_syscalls.c -o process_syscalls
      Run: ./process_syscalls
   */

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

  int main() {
pid_t pid, child_pid;
int status;

pid = fork();

if (pid < 0) {

            perror("fork failed");
            exit(EXIT_FAILU);
        }

if (pid == 0) {

            printf("Child process: PID = %d, PPID = %d\n", getpid(), getppid());

            execlp("ls", "ls", NULL);

            perror("execlp failed");
            exit(EXIT_FAILURE);
} else {
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);

        child_pid = wait(&status);

        if (child_pid == -1) {
            perror("wait failed");
            exit(EXIT_FAILUR
            E);
        }

        if (WIFEXITED(status)) {
            printf("Child process exited with status %d\n", WEXITSTATUS(status));
        } else {
            printf("Child process did not exit normally\n");
        }
    }

return 0;
  }
```

Output:



```
Parent process: PID = 21495, Child PID = 21496
Child process: PID = 21496, PPID = 21495
execlp failed: No such file or directory
Child process exited with status
```

Viva Questions

1. What is the role of fork() in process creation?
2. How does exec() differ from fork()?
3. What happens if we don't call wait() in the parent?

Exercise Programs

1. Create **two child processes** using multiple fork() calls.
2. Demonstrate **zombie** and **orphan** processes.
3. Modify to use execvp() to execute user-input commands.

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

```
/* cpu_scheduling.c
   Simulates FCFS, SJF (non-preemptive), Round Robin, Priority (non-preemptive)
   Compile: gcc cpu_scheduling.c -o cpu_scheduling
   Run: ./cpu_scheduling
   The program uses a sample set of processes; edit arrays to test other inputs.
*/

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

typedef struct { int id;
   int burst_time; int
   priority;
      } Process;

void fcfs_scheduling(int n, int burst_times[]) { int
   waiting_time[n], turnaround_time[n];
   waiting_time[0] = 0;
   turnaround_time[0] = burst_times[0];

   for (int i = 1; i < n; i++) {
            waiting_time[i] = waiting_time[i - 1] + burst_times[i - 1];
            turnaround_time[i] = waiting_time[i] + burst_times[i];
         }

   printf("FCFS Scheduling\n");
   printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n"); for (int
   i = 0; i < n; i++) {
            printf("P%d\t\t%d\t\t%d\t\t%d\n", i + 1, burst_times[i], waiting_time[i],
      turnaround_time[i]);
         }
      }
      int compare_sjf(const void *a, const void *b) {
   return ((Process *)a)->burst_time - ((Process *)b)->burst_time;
      }

void sjf_scheduling(int n, Process processes[]) { int
   waiting_time[n], turnaround_time[n];

   qsort(processes, n, sizeof(Process), compare_sjf);

   waiting_time[0] = 0;
   turnaround_time[0] = processes[0].burst_time;

   for (int i = 1; i < n; i++) {
            waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
            turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
         }

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

void round_robin_scheduling(int n, int burst_times[], int quantum) { int
   remaining_times[n], waiting_time[n], turnaround_time[n];
   int t = 0;

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

   while (1) {
            int done = 1;
            for (int i = 0; i < n; i++) {
               if (remaining_times[i] > 0) {
                    done = 0;
                    if (remaining_times[i] > quantum)
                        { t += quantum;
                        remaining_times[i] -= quantum;
                    } else {
                        t += remaining_times[i];
                        waiting_time[i] = t - burst_times[i];
                        remaining_times[i] = 0;
                    }
                }
            }
            if (done) {
                break;
            }
        }

    for (int i = 0; i < n; i++) {
            turnaround_time[i] = burst_times[i] + waiting_time[i];
        }

    printf("Round Robin Scheduling\n");
    printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n"); for (int
    i = 0; i < n; i++) {
            printf("P%d\t\t%d\t\t%d\t\t%d\n", i + 1, burst_times[i], waiting_time[i],
        turnaround_time[i]);
        }
    }

int compare_priority(const void *a, const void *b) { return
    ((Process *)a)->priority - ((Process *)b)->priority;
    }

void priority_scheduling(int n, Process processes[]) { int
    waiting_time[n], turnaround_time[n];

    qsort(processes, n, sizeof(Process), compare_priority);
```

```
waiting_time[0] = 0;
turnaround_time[0] = processes[0].burst_time;

for (int i = 1; i < n; i++) {
            waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
            turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
        }

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

     int main() {
int n, quantum;

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

int burst_times[n]; Process
processes[n];

printf("Enter burst times for each process:\n"); for (int i
= 0; i < n; i++) {
            printf("Burst Time for P%d: ", i + 1);
            scanf("%d", &burst_times[i]);
            processes[i].id = i + 1;
            processes[i].burst_time = burst_times[i];
        }

printf("Enter the quantum time for Round Robin (0 to skip): "); scanf("%d",
&quantum);
```

```
if (quantum > 0) {
            round_robin_scheduling(n, burst_times, quantum);
        }

printf("Enter priorities for each process:\n"); for
(int i = 0; i < n; i++) {
            printf("Priority for P%d: ", i + 1);
            scanf("%d", &processes[i].priority);
        }

fcfs_scheduling(n, burst_times);
sjf_scheduling(n, processes);
priority_scheduling(n, processes);

return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter burst times for each process:
Burst Time for P1: 10
Burst Time for P2: 5
Burst Time for P3: 8
Burst Time for P4: 12
Enter the quantum time for Round Robin (0 to skip): 4
Round Robin Scheduling
Process ID   Burst Time   Waiting Time        Turnaround Time
   P1            10           21                  31
   P2             5           16                  21
   P3             8           17                  25
   P4            12           23
Enter priorities for each process:
Priority for P1: 2
Priority for P2: 1
Priority for P3: 4
Priority for P4: 3
FCFS Scheduling
Process ID   Burst Time   Waiting Time        Turnaround Time
   P1           10           0          10
   P2           5            10         15
   P3           8            15         23
   P4           12           23         35
SJF Scheduling
Process ID   Burst Time   Waiting Time        Turnaround Time
     P2            5           0                          5
     P3            8           5                          13
     P1            10          13                         23
     P4            12          23                         35
                             Priority Scheduling
 Process ID   Burst Time    Priority     Waiting    Time      Turnaround Time
     P2            5           1           0          5
     P1            10          2           5          15
     P4            12          3           15         27
     P3            8           4           27         35
```

Viva Questions

1. Define turnaround time and waiting time.
2. Which algorithm can cause starvation and why?
3. How does Round Robin handle fairness?

Exercise Programs

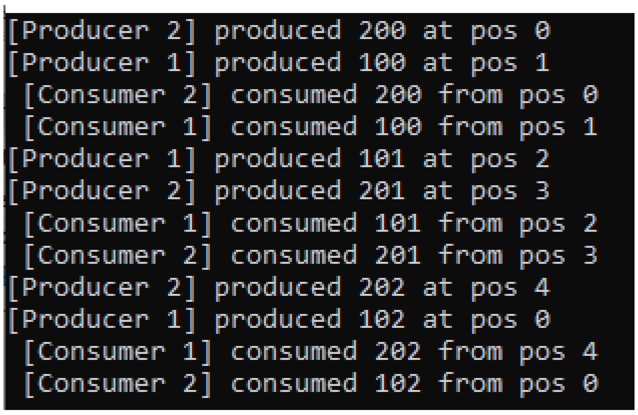
1. Include **arrival time** for each process.
2. Compare **preemptive vs non-preemptive SJF**.
3. Plot Gantt chart for all algorithms.

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

```
/* prod_cons_sem.c
   Producer-Consumer using pthreads + POSIX unnamed semaphores
   Compile: gcc prod_cons_sem.c -o prod_cons_sem -pthread
   Run: ./prod_cons_sem
*/

#include <stdio.h>
#include <stdlib.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 empty; // counts empty slots
sem_t full;  // counts full slots
pthread_mutex_t mutex;

void *producer(void *arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 10; i++) {
        int item = id*100 + i;
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        printf("[Producer %d] produced %d at pos %d\n", id, item, in);
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
        usleep(100000); // simulate work
    }
    return NULL;
```

```
}

void *consumer(void *arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("    [Consumer %d] consumed %d from pos %d\n", id, item, out);
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        usleep(150000); // simulate work
    }
    return NULL;
}

int main() {
    pthread_t prod1, prod2, cons1, cons2;
    int p1 = 1, p2 = 2, c1 = 1, c2 = 2;

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

    pthread_create(&prod1, NULL, producer, &p1);
    pthread_create(&prod2, NULL, producer, &p2);
    pthread_create(&cons1, NULL, consumer, &c1);
    pthread_create(&cons2, NULL, consumer, &c2);

    pthread_join(prod1, NULL);
    pthread_join(prod2, NULL);
    pthread_join(cons1, NULL);
    pthread_join(cons2, NULL);

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

    return 0;
}
```

Output:



```
[Producer 2] produced 200 at pos 0
[Producer 1] produced 100 at pos 1
 [Consumer 2] consumed 200 from pos 0
 [Consumer 1] consumed 100 from pos 1
[Producer 1] produced 101 at pos 2
[Producer 2] produced 201 at pos 3
 [Consumer 1] consumed 101 from pos 2
 [Consumer 2] consumed 201 from pos 3
[Producer 2] produced 202 at pos 4
[Producer 1] produced 102 at pos 0
 [Consumer 1] consumed 202 from pos 4
 [Consumer 2] consumed 102 from pos 0
```

Viva Questions

1. What are semaphores and how do they prevent race conditions?
2. What is the purpose of the mutex in this problem?
3. What happens if semaphore order (wait/post) is reversed?

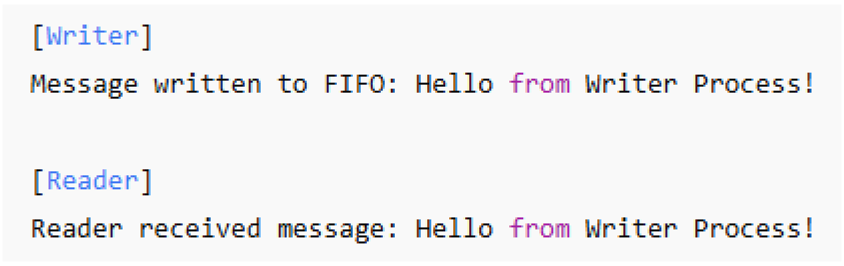
Exercise Programs

1. Add **two producers and two consumers** threads.
2. Use **sleep()** to simulate production and consumption delays.
3. Replace semaphore with **mutex only** and observe issues.

Program 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.

```
/* fifo_ipc.c
   Demonstrates a writer and a reader using a named FIFO.
   Program forks: child acts as writer, parent acts as reader (or vice versa).
   Compile: gcc fifo_ipc.c -o fifo_ipc
   Run: ./fifo_ipc
   It creates /tmp/myfifo and cleans up after.
*/

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

#define FIFO_PATH "/tmp/myfifo_demo"

int main() {
    if (mkfifo(FIFO_PATH, 0666) == -1) {
        // Might already exist; ignore error
    }

    pid_t pid = fork();
    if (pid < 0) { perror("fork"); return 1; }
    if (pid == 0) { // child -> writer
        int fd = open(FIFO_PATH, O_WRONLY);
        if (fd < 0) { perror("open writer"); exit(1); }
        char *messages[] = {"Hello from child", "This is a demo", "END"};
        for (int i=0;i<3;i++) {
            write(fd, messages[i], strlen(messages[i]) + 1);
            printf("[writer] sent: %s\n", messages[i]);
            sleep(1);
        }
        close(fd);
        exit(0);
    } else { // parent -> reader
        int fd = open(FIFO_PATH, O_RDONLY);
        if (fd < 0) { perror("open reader"); return 1; }
        char buf[128];
        while (1) {
            ssize_t r = read(fd, buf, sizeof(buf));
            if (r <= 0) break;
            printf("[reader] got: %s\n", buf);
            if (strcmp(buf, "END") == 0) break;
        }
        close(fd);
        wait(NULL);
        unlink(FIFO_PATH); // clean up
    }
    return 0;
}
```

Output:



```
[Writer]
Message written to FIFO: Hello from Writer Process!

[Reader]
Reader received message: Hello from Writer Process!
```

Viva Questions

1. What is the difference between a pipe and FIFO?
2. Can FIFO be used for two-way communication?
3. What is the role of mkfifo()?

Exercise Programs

1. Implement **bidirectional communication** using two FIFOs.
2. Modify to **send numbers** and compute their sum.
3. Extend to **client-server chat** using FIFO.

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

```
/* bankers.c
   Simulates Banker's Algorithm
   Compile: gcc bankers.c -o bankers
   Run: ./bankers
   This uses a small hardcoded example—modify matrices to test more cases.
*/

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

  int main() {
      int n, m;
      printf("Enter Processes and Resources: ");
      scanf("%d %d", &n, &m);

      int alloc[n][m], max[n][m], avail[m];
      int need[n][m], finish[n], safeSeq[n];

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

      printf("Enter Max Matrix:\n");
      for (int i = 0; i < n; i++)
          for (int j = 0; j < m; j++)
              scanf("%d", &max[i][j]);

      printf("Enter Available Resources:\n");
      for (int j = 0; j < m; j++)
          scanf("%d", &avail[j]);

      // Calculate Need matrix
      for (int i = 0; i < n; i++)
          for (int j = 0; j < m; j++)
              need[i][j] = max[i][j] - alloc[i][j];

      // Initialize finish[] = false
      for (int i = 0; i < n; i++)
          finish[i] = 0;
```

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

                if (j == m) {
                    for (int k = 0; k < m; k++)
                        avail[k] += alloc[p][k];
                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = true;
                }
            }
        }
        if (!found) {
            printf("System is NOT in a SAFE STATE.\n");
            return 0;
        }
    }

    printf("\nSafe Sequence: ");
    for (int i = 0; i < n; i++)
        printf("P%d%s", safeSeq[i], (i == n - 1) ? "" : " -> ");
    printf("\nSystem is in a SAFE STATE.\n");

    return 0;
}
```

Output:

```
Enter Processes and Resources: 5 3
Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Available Resources:
3 3 2

Safe Sequence: P1 -> P3 -> P4 -> P0 -> P2
System is in a SAFE STATE.
```

Viva Questions

1. What is the meaning of a safe state?
2. What does the Need matrix represent?
3. How does the Banker's Algorithm prevent deadlock?

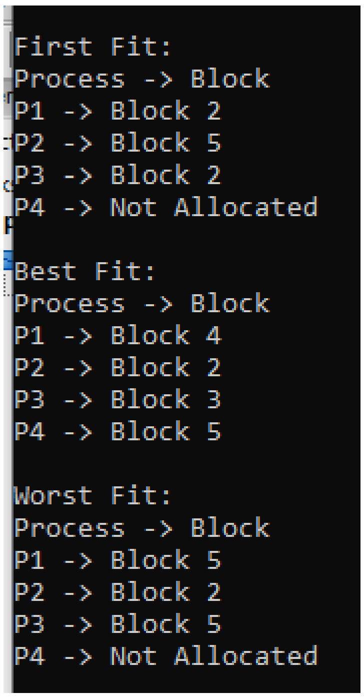
Exercise Programs

1. Simulate a **deadlock detection** algorithm.
2. Allow user to make a **resource request** dynamically.
3. Print step-by-step **safe sequence evolution**.

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

```
/* contiguous_alloc.c
   Simulates worst-fit, best-fit, first-fit memory allocation.
   Compile: gcc contiguous_alloc.c -o contiguous_alloc
   Run: ./contiguous_alloc
   Modify holes[] and processes[] arrays to test different scenarios.
*/

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

void print_alloc(int n, int alloc[]) {
    printf("Process -> Block\n");
    for (int i=0;i<n;i++) {
        if (alloc[i] == -1) printf("P%d -> Not Allocated\n", i+1);
        else printf("P%d -> Block %d\n", i+1, alloc[i]+1);
    }
}

void first_fit(int holes[], int m, int procs[], int n) {
    int alloc[50]; for (int i=0;i<n;i++) alloc[i]=-1;
    int rem[m]; for (int i=0;i<m;i++) rem[i]=holes[i];

    for (int i=0;i<n;i++) {
        for (int j=0;j<m;j++) {
            if (rem[j] >= procs[i]) {
                alloc[i] = j;
                rem[j] -= procs[i];
                break;
            }
        }
    }
    printf("\nFirst Fit:\n");
    print_alloc(n, alloc);
}

void best_fit(int holes[], int m, int procs[], int n) {
    int alloc[50]; for (int i=0;i<n;i++) alloc[i]=-1;
    int rem[m]; for (int i=0;i<m;i++) rem[i]=holes[i];

    for (int i=0;i<n;i++) {
        int best_idx = -1, best_diff = INT_MAX;
        for (int j=0;j<m;j++) {
            if (rem[j] >= procs[i] && rem[j] - procs[i] < best_diff) {
                best_diff = rem[j] - procs[i];
                best_idx = j;
            }
        }
        if (best_idx != -1) {
            alloc[i] = best_idx;
            rem[best_idx] -= procs[i];
        }
    }
    printf("\nBest Fit:\n");
    print_alloc(n, alloc);
}

void worst_fit(int holes[], int m, int procs[], int n) {
    int alloc[50]; for (int i=0;i<n;i++) alloc[i]=-1;
    int rem[m]; for (int i=0;i<m;i++) rem[i]=holes[i];

    for (int i=0;i<n;i++) {
        int worst_idx = -1, worst_diff = -1;
        for (int j=0;j<m;j++) {
            if (rem[j] >= procs[i] && rem[j] - procs[i] > worst_diff) {
                worst_diff = rem[j] - procs[i];
                worst_idx = j;
            }
        }
        if (worst_idx != -1) {
            alloc[i] = worst_idx;
            rem[worst_idx] -= procs[i];
        }
    }
    printf("\nWorst Fit:\n");
    print_alloc(n, alloc);
}

int main() {
    int holes[] = {100, 500, 200, 300, 600}; // block sizes
    int m = sizeof(holes)/sizeof(holes[0]);
    int procs[] = {212, 417, 112, 426};
    int n = sizeof(procs)/sizeof(procs[0]);

    first_fit(holes,  m,  procs,  n);
    best_fit(holes,  m,  procs,  n);
    worst_fit(holes, m, procs, n);
    return 0;
}
```

Output:



```
First Fit:
Process -> Block
P1 -> Block 2
P2 -> Block 5
P3 -> Block 2
P4 -> Not Allocated

Best Fit:
Process -> Block
P1 -> Block 4
P2 -> Block 2
P3 -> Block 3
P4 -> Block 5

Worst Fit:
Process -> Block
P1 -> Block 5
P2 -> Block 2
P3 -> Block 5
P4 -> Not Allocated
```

Viva Questions

1. Differentiate internal and external fragmentation.
2. Which allocation strategy minimizes waste space?
3. Why is compaction sometimes necessary?

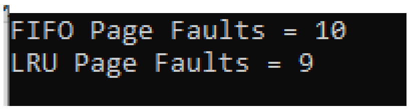
Exercise Programs

1. Add **process deallocation** and reallocation.
2. Implement **compaction** feature.
3. Compare **average memory utilization** of each method.

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

```
/* page_replacement.c
   Simulates FIFO and LRU page replacement.
   Compile: gcc page_replacement.c -o page_replacement
   Run: ./page_replacement
   Edit ref[] and frames to test.
*/

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

int find_page(int frames[], int frames_count, int page) {
    for (int i=0;i<frames_count;i++) if (frames[i]==page) return i;
    return -1;
}

void fifo(int ref[], int ref_len, int frames_count) {
    int frames[50];
    for (int i=0;i<frames_count;i++) frames[i]=-1;
    int pointer=0, page_faults=0;

    for (int i=0;i<ref_len;i++) {
        int page = ref[i];
        if (find_page(frames, frames_count, page) == -1) {
            frames[pointer] = page;
            pointer = (pointer+1)%frames_count;
            page_faults++;
        }
    }
    printf("FIFO Page Faults = %d\n", page_faults);
}

void lru(int ref[], int ref_len, int frames_count) {
    int frames[50];
    int timeStamp[50];
    for (int i=0;i<frames_count;i++) frames[i]=-1, timeStamp[i]=0;
    int page_faults=0, time=0;

    for (int i=0;i<ref_len;i++) {
        int page = ref[i];
        time++;
        int pos = find_page(frames, frames_count, page);
        if (pos != -1) {
            timeStamp[pos] = time;
        } else {
            // find empty
            int emptyPos = -1;
            for (int j=0;j<frames_count;j++) if (frames[j]==-1) { emptyPos=j; break; }
            if (emptyPos != -1) {
                frames[emptyPos] = page;
                timeStamp[emptyPos] = time;
            } else {
                // replace least recently used
                int lru_idx = 0, minTime = timeStamp[0];
                for (int j=1;j<frames_count;j++)
                    if (timeStamp[j] < minTime) { minTime = timeStamp[j]; lru_idx=j; }
                frames[lru_idx] = page;
                timeStamp[lru_idx] = time;
            }
            page_faults++;
        }
    }
    printf("LRU Page Faults = %d\n", page_faults);
}

int main() {
    int ref[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int len = sizeof(ref)/sizeof(ref[0]);
    int frames = 3;
    fifo(ref, len, frames);
    lru(ref, len, frames);
    return 0;
}
```

Output:



```
FIFO Page Faults = 10
LRU Page Faults = 9
```

Viva Questions

1. What is a page fault?
2. Why does LRU perform better than FIFO in most cases?
3. Define locality of reference.

Exercise Programs

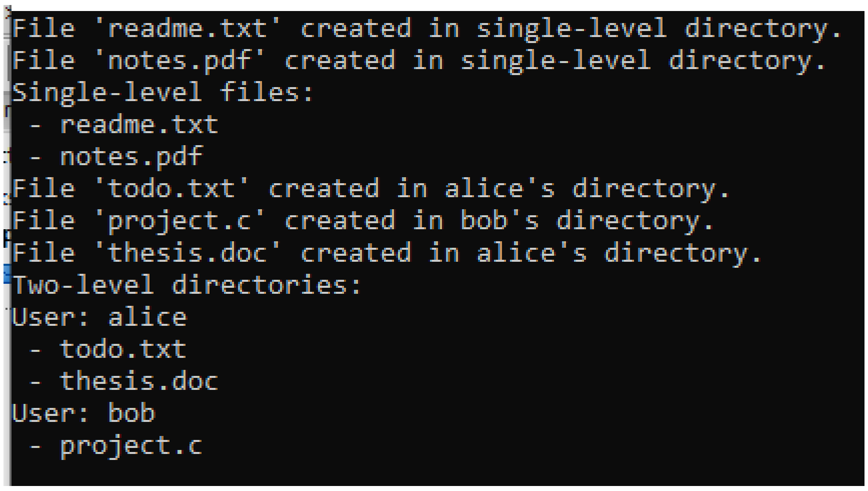
1. Implement **Optimal Page Replacement**.
2. Compare **page faults** for all three algorithms.
3. Draw **page replacement table** after each reference.

Program 8: Simulate following File Organization Techniques a) Single level directory b) Two level directory.

```
/* file_organization.c
   Simulates single-level and two-level directory structures with simple menus.
   Compile: gcc file_organization.c -o file_organization
   Run: ./file_organization
*/

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

#define MAX_FILES 100
#define MAX_NAME 50
#define MAX_USERS 10

/* Single-level directory: all files in one namespace */
typedef struct {
    char name[MAX_NAME];
} FileEntry;

FileEntry singleFiles[MAX_FILES];
int singleCount = 0;

/* Two-level directory: each user has own directory */
typedef struct {
    char username[MAX_NAME];
    char files[MAX_FILES][MAX_NAME];
    int count;
} UserDir;

UserDir users[MAX_USERS];
int userCount = 0;

void create_file_single(char *name) {
    if (singleCount >= MAX_FILES) { printf("Single dir full.\n"); return; }
    strcpy(singleFiles[singleCount++].name, name);
    printf("File '%s' created in single-level directory.\n", name);
}

void list_single() {
    printf("Single-level files:\n");
    for (int i=0;i<singleCount;i++) printf(" - %s\n", singleFiles[i].name);
}
```

```
int find_user(char *uname) {
    for (int i=0;i<userCount;i++) if (strcmp(users[i].username, uname)==0) return i;
    return -1;
}

void create_user_if_needed(char *uname) {
    if (find_user(uname) == -1 && userCount < MAX_USERS) {
        strcpy(users[userCount].username, uname);
        users[userCount].count = 0;
        userCount++;
    }
}

void create_file_two(char *uname, char *fname) {
    create_user_if_needed(uname);
    int idx = find_user(uname);
    if (idx == -1) { printf("Cannot create user.\n"); return; }
    if (users[idx].count >= MAX_FILES) { printf("User's directory full.\n"); return; }
    strcpy(users[idx].files[users[idx].count++], fname);
    printf("File '%s' created in %s's directory.\n", fname, uname);
}

void list_two() {
    printf("Two-level directories:\n");
    for (int i=0;i<userCount;i++) {
        printf("User: %s\n", users[i].username);
        for (int j=0;j<users[i].count;j++) printf("    - %s\n", users[i].files[j]);
    }
}

int main() {
    // demo operations:
    create_file_single("readme.txt");
    create_file_single("notes.pdf");
    list_single();

    create_file_two("alice", "todo.txt");
    create_file_two("bob", "project.c");
    create_file_two("alice", "thesis.doc");
    list_two();

    return 0;
}
```

Output:



```
File 'readme.txt' created in single-level directory.
File 'notes.pdf' created in single-level directory.
Single-level files:
 - readme.txt
 - notes.pdf
File 'todo.txt' created in alice's directory.
File 'project.c' created in bob's directory.
File 'thesis.doc' created in alice's directory.
Two-level directories:
User: alice
 - todo.txt
 - thesis.doc
User: bob
 - project.c
```

Viva Questions

1. What is the main limitation of a single-level directory?
2. How does two-level directory remove naming conflicts?
3. What data structure is used to maintain directory info?

Exercise Programs

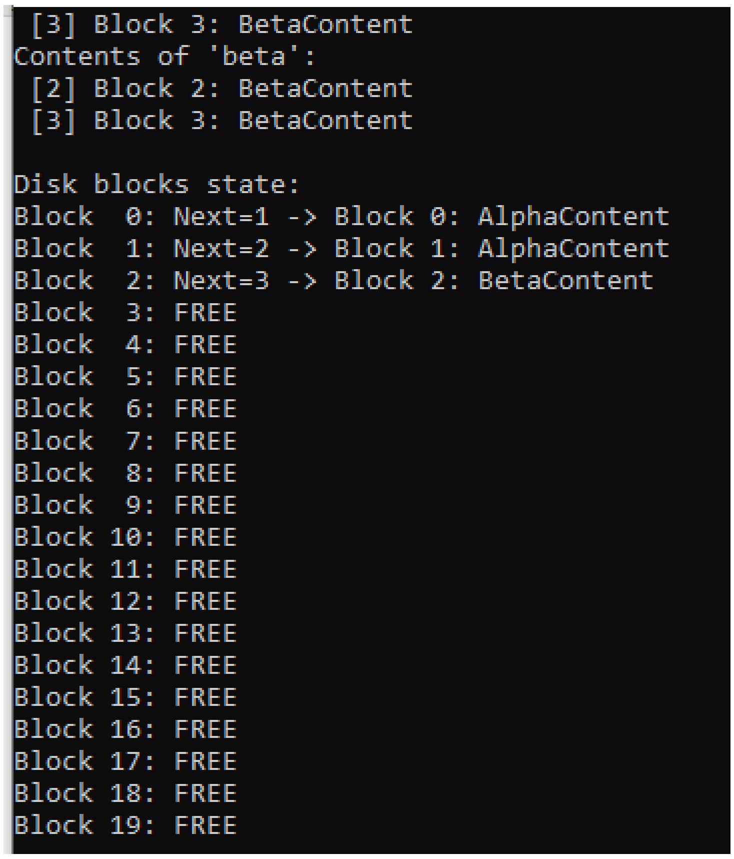
1. Add **file search** and **delete** operations.
2. Display total **file count per user**.
3. Extend to **tree-structured directories**.

Program 9: Develop a C program to simulate the Linked file allocation strategies.

```
/* linked_file_alloc.c
   Simulates linked file allocation: disk blocks form a linked list; each file stores start
block.
   Compile: gcc linked_file_alloc.c -o linked_file_alloc
   Run: ./linked_file_alloc
*/

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

#define BLOCKS 20
#define MAX_FILES 10
#define BLOCK_FREE -1
#define MAX_FILENAME 50

int next_block[BLOCKS]; // -1 means end, -2 means free
char block_data[BLOCKS][64];

typedef struct {
    char name[MAX_FILENAME];
    int start;
} FileEntry;

FileEntry files[MAX_FILES];
int fileCount = 0;

void init_disk() {
    for (int i=0;i<BLOCKS;i++) next_block[i] = BLOCK_FREE;
}

int allocate_blocks(int blocks_needed, const char *content) {
    int allocated[BLOCKS], k=0;
    for (int i=0;i<BLOCKS && k<blocks_needed;i++) {
        if (next_block[i] == BLOCK_FREE) {
            allocated[k++] = i;
            next_block[i] = -2; // temporarily mark used
        }
    }
    if (k < blocks_needed) {
        // free temp marks
        for (int i=0;i<BLOCKS;i++) if (next_block[i] == -2) next_block[i] =
BLOCK_FREE;
```

```
        return -1;
    }
    // link them
    for (int i=0;i<blocks_needed;i++) {
        int b = allocated[i];
        if (i == blocks_needed-1) next_block[b] = -1;
        else next_block[b] = allocated[i+1];
        // store some content fragment
        snprintf(block_data[b], sizeof(block_data[b]), "Block %d: %s", b, content);
    }
    return allocated[0]; // return start
}

void create_file(const char *name, int size_in_blocks, const char *content) {
    if (fileCount >= MAX_FILES) { printf("Too many files.\n"); return; }
    int start = allocate_blocks(size_in_blocks, content);
    if (start == -1) { printf("Not enough free blocks to create file '%s'.\n", name); return;
}
    strcpy(files[fileCount].name, name);
    files[fileCount].start = start;
    fileCount++;
    printf("File '%s' created starting at block %d\n", name, start);
}

void display_file(const char *name) {
    for (int i=0;i<fileCount;i++) {
        if (strcmp(files[i].name, name)==0) {
            int b = files[i].start;
            printf("Contents of '%s':\n", name);
            while (b != -1) {
                printf("  [%d] %s\n", b, block_data[b]);
                b = next_block[b];
            }
            return;
        }
    }
    printf("File not found.\n");
}

void display_disk() {
    printf("\nDisk blocks state:\n");
    for (int i=0;i<BLOCKS;i++) {
        printf("Block %2d: ", i);
        if (next_block[i] == BLOCK_FREE) printf("FREE\n");
        else if (next_block[i] == -1) printf("END -> %s\n", block_data[i]);
        else printf("Next=%d -> %s\n", next_block[i], block_data[i]);
    }
}

int main() {
    init_disk();
    create_file("alpha", 3, "AlphaContent");
    create_file("beta", 2, "BetaContent");
    display_file("alpha");
    display_file("beta");
    display_disk();
    return 0;
}
```

Output:



```
 [3] Block 3: BetaContent
Contents of 'beta':
 [2] Block 2: BetaContent
 [3] Block 3: BetaContent

Disk blocks state:
Block  0: Next=1 -> Block 0: AlphaContent
Block  1: Next=2 -> Block 1: AlphaContent
Block  2: Next=3 -> Block 2: BetaContent
Block  3: FREE
Block  4: FREE
Block  5: FREE
Block  6: FREE
Block  7: FREE
Block  8: FREE
Block  9: FREE
Block 10: FREE
Block 11: FREE
Block 12: FREE
Block 13: FREE
Block 14: FREE
Block 15: FREE
Block 16: FREE
Block 17: FREE
Block 18: FREE
Block 19: FREE
```

Viva Questions

1. How is linked allocation different from contiguous allocation?
2. What is the drawback of linked allocation?
3. How is end-of-file represented?

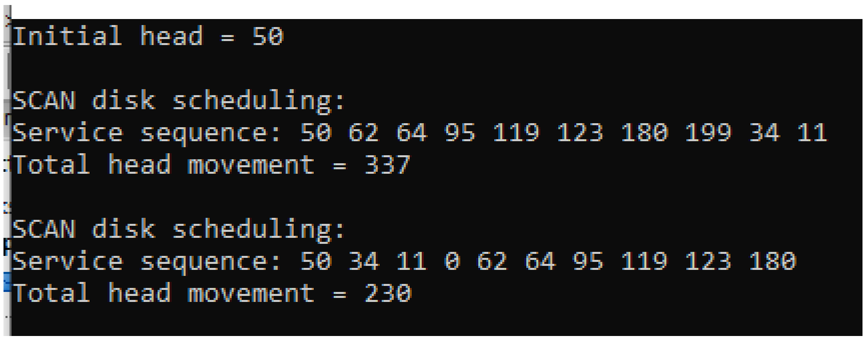
Exercise Programs

1. Simulate **indexed file allocation**.
2. Add **file deletion** feature to free blocks.
3. Display **block allocation table** visually.

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

```
/* scan_disk.c
   Simulates SCAN disk scheduling algorithm (elevator algorithm)
   Compile: gcc scan_disk.c -o scan_disk
   Run: ./scan_disk
*/

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

int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

void scan(int requests[], int n, int head, int disk_size, int direction /*0=left,1=right*/) {
    // sort requests
    int arr[n];
    for (int i=0;i<n;i++) arr[i]=requests[i];
    qsort(arr, n, sizeof(int), compare);

    // find split point
    int idx = 0;
    while (idx < n && arr[idx] < head) idx++;

    printf("\nSCAN disk scheduling:\n");
    int seq[1000], seq_len=0;
    int distance = 0;
    int cur = head;
    seq[seq_len++] = cur;

    if (direction == 1) { // moving right/upwards (increasing)
        // service all requests to the right
        for (int i=idx;i<n;i++) {
            distance += abs(arr[i] - cur);
            cur = arr[i];
            seq[seq_len++] = cur;
        }
        // go to end
        distance += abs((disk_size-1) - cur);
        cur = disk_size-1;
        seq[seq_len++] = cur;
        // reverse: service left side
        for (int i=idx-1;i>=0;i--) {
            distance += abs(arr[i] - cur);
            cur = arr[i];
            seq[seq_len++] = cur;
        }
    } else { // moving left/downwards (decreasing)
        for (int i=idx-1;i>=0;i--) {
            distance += abs(arr[i] - cur);
            cur = arr[i];
            seq[seq_len++] = cur;
        }
        distance += abs(cur - 0);
        cur = 0;
        seq[seq_len++] = cur;
        for (int i=idx;i<n;i++) {
            distance += abs(arr[i] - cur);
            cur = arr[i];
            seq[seq_len++] = cur;
        }
    }

    printf("Service sequence: ");
    for (int i=0;i<seq_len;i++) printf("%d ", seq[i]);
    printf("\nTotal head movement = %d\n", distance);
}

int main() {
    int requests[] = { 95, 180, 34, 119, 11, 123, 62, 64 };
    int n = sizeof(requests)/sizeof(requests[0]);
    int head = 50;
    int disk_size = 200; // 0..199
    printf("Initial head = %d\n", head);
    scan(requests, n, head, disk_size, 1); // move right first
    scan(requests, n, head, disk_size, 0); // move left first
    return 0;
}
```

Output:



```
Initial head = 50

SCAN disk scheduling:
Service sequence: 50 62 64 95 119 123 180 199 34 11
Total head movement = 337

SCAN disk scheduling:
Service sequence: 50 34 11 0 62 64 95 119 123 180
Total head movement = 230
```

Viva Questions

1. What is seek time in disk scheduling?
2. How does SCAN differ from C-SCAN?
3. Why is SCAN called the elevator algorithm?

Exercise Programs

1. Implement **SSTF and C-SCAN** algorithms.
2. Compare **total head movement** for all.
3. Plot **request vs. head position graph**.