YENEPOYA INSTITUTE OF TECHNOLOGY

NBA-Accredited : B.E (CSE & ME)

(Affiliated to Visvesvaraya Technological University, Belagavi)

N.H. 13, Thodar, Moodbidri, Mangaluru 574225 (D.K), Karnataka

DEPARTMENT OF CSE (IoT, Cyber Security Including Block Chain Technology)

LAB MANUAL

Operating Systems Laboratory

BCS303

[As per Choice Based Credit System (CBCS) scheme]

Vision

To empower students such that they will be technologically adept, innovation driven, self-motivated and responsible global citizens possessing human values by imparting quality technical education in the field of Computer Science.

Mission

1. To Facilitating and exposing the students to various learning opportunities through dedicated academic guidance and monitoring.
2. To Provide a learning ambience to encourage innovations, problem solving skills, leadership qualities, team-spirit, entrepreneurship skills and ethical responsibilities.
3. To Encourage faculty and students to actively participate in innovation, industry solutions,research and lifelong learning so that their contribution makes a substantial difference to the society.

Program Educational Objectives (PEO's):

- Graduates will be equipped to be employed in IT industries and be engaged in learning understanding and applying new ideas.
- Graduates through academic training will be able to take up higher studies and industry specific research programs.
- Graduates will be responsible computing professional with social obligations in their own area of interest.

Program Specific Outcomes (PSO's):

- Graduates will be able to use the knowledge and ability to write programs and integrate them with the hardware/software products in the domains of embedded systems, databases/data analytics, network/web systems and mobile products.
- Graduate will be able to use knowledge of information science in various domains to identify research gaps and hence to provide solution to new ideas and innovations.

Program Outcomes

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 analyse 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.

BCS303 –Operating Systems Laboratory Course Outcomes	
BCS303 .1	Explain the structure and functionality of operating system.
BCS303 .2	Apply appropriate CPU scheduling algorithms for the given problem.
BCS303.3	Analyze the various techniques for process synchronization and deadlock handling.
BCS303.4	Apply the various techniques for memory management.
BCS303 .5	Explain file and secondary storage management strategies.
BCS303.6	Describe the need for information protection mechanisms.

TABLE OF CONTENT

OPERATING SYSTEMS LABORATORY

(Effective from the academic year

2023-2024)SEMESTER – III

Course Code	BCS303	CIE Marks	50
Number of Contact Hours/Week	0:0:2	SEE Marks	50
Total Number of Lab Contact Hours	20	Theory Exam Hours	03

Course objectives:

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

Descriptions (if any):

- Simulation packages preferred: Multisim, Xilinx _ISE9.2, PSpice or any other relevant.
- For Analog Electronic Circuits students must trace the wave form on Tracing sheet /Graph sheetand label trace.
- Continuous evaluation by the faculty must be carried by including performance of a studentinboth hardware implementation and simulation (if any) for the given circuit.
- A batch not exceeding 4 must be formed for conducting the experiment. For simulationindividual student must execute the program.

Laboratory Component:

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 anda 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) Worstfit 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.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
 int status;
 pid_t child_pid;

 // Fork a child process
 child_pid = fork();

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

 if (child_pid == 0)
   { // This is the child process
     printf("Child process (PID %d) created.\n", getpid());

     // Execute a different program using exec
     char *program = "ls";
     char *args[] = {"ls", "-l", NULL};
     execvp(program, args);

     // If execvp returns, it means there was an error
     perror("Exec failed");
     exit(EXIT_FAILURE);
     }
else
  { // This is the parent process
     printf("Parent process (PID %d) waiting for the child (PID %d) to finish...\n", getpid(), child_pid);

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

     if (WIFEXITED(status))
     {
        int exit_status = WEXITSTATUS(status);
        printf("Child process (PID %d) has terminated with status: %d\n", child_pid, exit_status);
     }
  }

 // Create another child process
 child_pid = fork();

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

```
        exit(EXIT_FAILURE);
    }

    if (child_pid == 0)
       { // This is the second child process
        printf("Second child process (PID %d) created.\n", getpid());

        // Simulate some work and then exit
        sleep(2);


        printf("Second child process (PID %d) is done.\n", getpid());
        exit(EXIT_SUCCESS);
      }
     else
       { // This is the parent process
        printf("Parent process (PID %d) waiting for the second child (PID %d) to finish...\n", getpid(), child_pid);

        // Wait for the second child process to complete
        wait(&status);

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);
            printf("Second child process (PID %d) has terminated with status: %d\n", child_pid, exit_status);
        }
    }

    return 0;
}
```

OUTPUT:

```
Parent process (PID 2261) waiting for the child (PID 2262) to finish...
Child process (PID 2262) created.
Exec failed: No such file or directory
Child process (PID 2262) has terminated with status: 1
Parent process (PID 2261) waiting for the second child (PID 2263) to finish...
Second child process (PID 2263) created.
Second child process (PID 2263) is done.
Second child process (PID 2263) has terminated with status: 0
```

OR

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

int main()
{
 pid_t child_pid;
 int status;
```

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

if (child_pid < 0)
{
   perror("Fork failed");
   exit(1);
}
else if (child_pid == 0)
{
   // This code is executed by the child process
   printf("Child process (PID: %d) is running.\n", getpid());

   // Replace the child process with a new program using exec()
   char *args[] = {"C:\Windows\System32", NULL}; // Replace " " with the path to your child program
   execvp(args[0], args);

   // If exec() fails, this code will be executed
   perror("Exec failed");
   exit(1);
}
else
{
   // This code is executed by the parent process
   printf("Parent process (PID: %d) is waiting for the child to complete.\n", getpid());

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

   if (WIFEXITED(status))
   {
      printf("Child process (PID: %d) has completed with status %d.\n", child_pid, WEXITSTATUS(status));
   }
}

return 0;
}
```

OUTPUT:

```
Parent process (PID: 13885) is waiting for the child to complete.
Child process (PID: 13886) is running.
Exec failed: No such file or directory
Child process (PID: 13886) has completed with status 1.
```

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:

AIM:

To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting time the average waiting time is calculated as the average of all the waiting times. FCFS mainly says firstcome first serve the algorithm which came first will be served first.

ALGORITHM:

Step 1: Start the process.
Step 2: Accept the number of processes in the ready Queue.
Step 3: For each process in the ready Q, assign the process name and the burst time.
Step 4: Set the waiting of the first process as _0'and its burst time as its turnaround time.
Step 5: for each process in the Ready Q calculate.
a). Waiting time (n) = waiting time (n-1) + Burst time (n-1)
b). Turnaround time (n) = waiting time(n)+Burst time(n)
Step 6: Calculate
a) Average waiting time = Total waiting Time / Number of process \
b) Average Turnaround time = Total Turnaround Time / Number of process
Step 7: Stop the process.

SOURCE CODE:

```
#include<stdio.h>
void main()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
```

```
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
}
```

INPUT

Enter the number of processes -- 3
Enter Burst Time for Process 0 -- 24
Enter Burst Time for Process 1 -- 3
Enter Burst Time for Process 2 – 3

OUTPUT:

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30

Average Waiting Time -- 17.000000
Average Turnaround Time -- 27.000000

b) SJF SHORTEST JOB FIRST:

AIM:
To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Pre-emption)

DESCRIPTION:
To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum ofthe bursting times of all the process previous or before to that process.

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue.
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according to the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as 0‘and its turnaround time as its burst time.
Step 6: Sort the processes names based on their Burt time
Step 7: For each process in the ready queue,
calculate

i. Waiting time(n)= waiting time (n-1) + Burst time (n-1)
ii. Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate

iii. Average waiting time = Total waiting Time / Number of process
iv. Average Turnaround time = Total Turnaround Time / Number of process

Step 9: Stop the process.

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg,tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
return 0;
}
```

INPUT:

```
Enter the number of processes -- 4.
Enter Burst Time for Process 0 -- 6
Enter Burst Time for Process 1 -- 8
Enter Burst Time for Process 2 -- 7
Enter Burst Time for Process 3 – 3
```

OUTPUT:

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24

Average Waiting Time -- 7.000000
Average Turnaround Time -- 13.000000

b) ROUND ROBIN:

AIM:
To simulate the CPU scheduling algorithm round - robin.

DESCRIPTION:
To aim is to calculate the average waiting time. There will be a time slice, each process shouldbe executed within that time-slice and if not it will go to the waiting state so first check whether the burst time isless than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time - slot and the loop continues until all the processes are completed.

ALGORITHM:

Step 1: Start the process.
Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice.
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time.
Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time
process (n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.
Step 6: Consider the ready queue is a circular Q, Calculate

a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)
b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process.

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]);
```

```
 ct[i]=bu[i];
 }
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t) {
tat[i]=temp+bu[i];
temp=temp+bu[i];
bu[i]=0;
}
else {
bu[i]=bu[i]-t;
temp=temp+t;
}
for(i=0;i<n;i++){
wa[i]=tat[i]-
ct[i]; att+=tat[i];
awt+=wa[i];}
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
return 0;
}
```

INPUT:

```
Enter the no of processes -- 6
Enter Burst Time for process 1 -- 5.
Enter Burst Time for process 2 -- 6.
Enter Burst Time for process 3 -- 7.
Enter Burst Time for process 4 -- 9.
Enter Burst Time for process 5 -- 2.
Enter Burst Time for process 6 -- 3.
Enter the size of time slice – 3.
```

OUTPUT:

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
1	5	14	19
2	6	16	22
3	7	22	29
4	9	23	32
5	2	12	14
6	3	14	17

```
The Average Turnaround time is -- 22.166666.
The Average Waiting time is -- 16.833334.
```

D).PRIORITY:

AIM: To write a c program to simulate the CPU scheduling priority algorithm.

DESCRIPTION: To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

Step 1: Start the process.
Step 2: Accept the number of processes in the ready Queue.
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time.
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as _0'and its burst time as its turnaround time.
Step 6: Arrange the processes based on process priority.
Step 7: For each process in the Ready Queue calculate.
Step 8: For each process in the Ready Queue calculate.
a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process
e) Print the results in an order.
Step 9: Stop

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg,tatavg;
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i); scanf("%d%d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(pri[i] > pri[k])
{
temp=p[i];
p[i]=p[k];
p[k]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
```

```
 tat[i] = tat[i-1] + bt[i];
 wtavg = wtavg + wt[i];
 tatavg = tatavg + tat[i];
}
 printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
 for(i=0;i<n;i++)
 printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
 printf("\nAverage Waiting Time is --- %f",wtavg/n);
 printf("\nAverage Turnaround Time is --- %f",tatavg/n);
 return 0;
}
```

INPUT:

```
Enter the number of processes --- 5
Enter the Burst Time & Priority of Process 0 ---      1       3
                                                      0
 Enter the Burst Time & Priority of Process 1 --      1       1
                        -
 Enter the Burst Time & Priority of Process 2 --      2       4
                        -
 Enter the Burst Time & Priority of Process 3 --      1       5
                        -
 Enter the Burst Time & Priority of Process 4 --      5       2
                        -
```

OUTPUT:

PROCESS	PRIORITY	BURST TIME	WAITING TIME	TURNAROUND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19

Average Waiting Time is --- 8.200000.
Average Turnaround Time is --- 12.000000.

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

AIM: To Write a C program to simulate producer-consumer problem using semaphores.

DESCRIPTION: Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#define BUFFER_SIZE 10
typedef struct
{
  int buffer[BUFFER_SIZE];
  int in;
  int  out;
  sem_t empty;
  sem_t full;
  pthread_mutex_t mutex;
}
shared_data_t;
void producer(shared_data_t *shared_data)
{
  int item;
  while (1)
   {
    item = rand() % 100;
    sem_wait(&shared_data->empty);
    pthread_mutex_lock(&shared_data->mutex);
    shared_data->buffer[shared_data->in] = item;
    shared_data->in = (shared_data->in + 1) % BUFFER_SIZE;
    printf("Produced item %d\n", item);
    fflush(stdout);
    pthread_mutex_unlock(&shared_data->mutex);
    sem_post(&shared_data->full);
   }
}
void consumer(shared_data_t *shared_data)
{
  int item;
  while (1)
  {
    sem_wait(&shared_data->full);
    pthread_mutex_lock(&shared_data->mutex);
    item = shared_data->buffer[shared_data->out];
    shared_data->out = (shared_data->out + 1) % BUFFER_SIZE;
    printf("Consumed item %d\n", item);
```

```
    fflush(stdout);
    pthread_mutex_unlock(&shared_data->mutex);
    sem_post(&shared_data->empty);
   }
 }
 int main()
 {
  shared_data_t shared_data;
  pthread_t producer_thread;
  pthread_t consumer_thread;
  shared_data.in = 0;
  shared_data.out = 0;
  sem_init(&shared_data.empty, 0, BUFFER_SIZE);
  sem_init(&shared_data.full, 0, 0);
  pthread_mutex_init(&shared_data.mutex, NULL);
  pthread_create(&producer_thread, NULL, (void *)producer, &shared_data);
  pthread_create(&consumer_thread, NULL, (void *)consumer, &shared_data);
  pthread_join(producer_thread, NULL);
  pthread_join(consumer_thread, NULL);
  sem_destroy(&shared_data.empty);
  sem_destroy(&shared_data.full);
  pthread_mutex_destroy(&shared_data.mutex);
  return 0;
 }
```

OUTPUT:

```
Produced item 83
Produced item 86
Produced item 77
Produced item 15
Produced item 93
Consumed item 83
Consumed item 86
Consumed item 77
Consumed item 15
Consumed item 93
```

4. **Develop a C program which demonstrates interprocess communication between a reader processand a writer process. Use mkfifo, open, read, write and close APIs in your program**.

Reader Process:

```
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#define MAX_BUF 1024
int main()
{
  int fd;
  /* A temporary FIFO file is not created in reader */
  char *myfifo = "/tmp/myfifo";
  char buf[MAX_BUF];
  /* Open the named pipe for reading */
  fd = open(myfifo, O_RDONLY);
  /* Read data from the FIFO */
  read(fd, buf, MAX_BUF);
  printf("Writer: %s\n", buf);
  /* Close the FIFO */
  close(fd);
  return 0;
}
```

Writer Process:

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
  int fd;
  char buf[1024];
  /* Create the named pipe (FIFO) */
  char *myfifo = "/tmp/myfifo";
  mkfifo(myfifo, 0666);
  printf("Run Reader process to read the FIFO File\n");
  /* Open the named pipe for writing */
  fd = open(myfifo, O_WRONLY);
  /* Write data to the FIFO */
  strcpy(buf, "Hello from Writer Process");
  write(fd, buf, sizeof(buf));
  /* Close the FIFO */
  close(fd);
  /* Remove the FIFO */
  unlink(myfifo);
  return 0;
        }
```

To execute the program, first compile the writer process and the reader process separately:

```
gcc -o writer writer.c
gcc -o reader reader.c
```

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

Source Code:

```
#include <stdio.h>
 // Function to check if the system is in a safe state
 int isSafe(int processes, int resources, int max[][resources], int allocated[][resources], int available[]) {
   int need[processes][resources];
   int finish[processes];
   for (int i = 0; i < processes; i++)
 {
     finish[i] = 0;
     for (int j = 0; j < resources; j++)
 {
       need[i][j] = max[i][j] - allocated[i][j];
     }
   }
   int work[resources];
   for (int i = 0; i < resources; i++)
 {
     work[i] = available[i];
   }
   int safe = 0;
   while (1)
 {
     int found = 0;
     for (int i = 0; i < processes; i++)
 {
       if (!finish[i])
 {
         int canAllocate = 1;
         for (int j = 0; j < resources; j++)
 {
           if (need[i][j] > work[j])
 {
             canAllocate = 0;
             break;
           }
         }
         if (canAllocate)
 {
           for (int j = 0; j < resources; j++)
 {
             work[j] += allocated[i][j];
           }
           finish[i] = 1;
           found = 1;
           break;
         }
       }
     }
     if (!found)
 {
```

```
      break;
    }
  }
  for (int i = 0; i < processes; i++)
{
    if (!finish[i])
{
      return 0;
    }
  }
  return 1;
}
int main()
{
  int processes, resources;
  printf("Enter the number of processes: ");
  scanf("%d", &processes);
  printf("Enter the number of resources: ");
  scanf("%d", &resources);
  int max[processes][resources];
  int allocated[processes][resources];
  int available[resources];
  printf("\nEnter the maximum resource matrix:\n");
  for (int i = 0; i < processes; i++)
{
    for (int j = 0; j < resources; j++)
{
      scanf("%d", &max[i][j]);
    }
  }
  printf("\nEnter the allocated resource matrix:\n");
  for (int i = 0; i < processes; i++)
{
    for (int j = 0; j < resources; j++)
{
      scanf("%d", &allocated[i][j]);
    }
  }
  printf("\nEnter the available resources:\n");
  for (int i = 0; i < resources; i++)
{
    scanf("%d", &available[i]);
  }

  if (isSafe(processes, resources, max, allocated, available))
{
    printf("\nThe system is in a safe state.\n");
  }
else
{
    printf("\nThe system is not in a safe state.\n");
  }

  return 0;
```

OUTPUT:

```
Enter the number of processes: 2
Enter the number of resources: 2
Enter the maximum resource matrix:
1
2
34
5
Enter the allocated resource matrix:
8
9
5
6
Enter the available resources:
4
8
```

The system is not in a safe state.

```
Enter the number of processes: 2
Enter the number of resources: 2

Enter the maximum resource matrix:
1
2
3
4
Enter the allocated resource matrix:
9
8
7
6
Enter the available resources:
10
20
```

The system is in a safe state.

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

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
typedef struct {
    int id;
    int size;
    int allocated;
} Block;
void initialize(Block blocks[], int size) {
    for (int i = 0; i < size; ++i) {
        blocks[i].id = i + 1;
        blocks[i].size = 0;
        blocks[i].allocated = 0;
    }
}
void displayBlocks(Block blocks[], int size) {
    printf("Block\tSize\tAllocated\n");
    for (int i = 0; i < size; ++i) {
        printf("%d\t%d\t", blocks[i].id, blocks[i].size);
        if (blocks[i].allocated)
            printf("Yes\n");
        else
            printf("No\n");
    }
    printf("\n");
}
int worstFit(Block blocks[], int size, int processSize) {
    int index = -1;
    int maxBlockSize = -1;
    for (int i = 0; i < size; ++i) {
        if (!blocks[i].allocated && blocks[i].size >= processSize) {
            if (blocks[i].size > maxBlockSize) {
                maxBlockSize = blocks[i].size;
                index = i;
            }
        }
```

```
    }
    return index;
}
int bestFit(Block blocks[], int size, int processSize) {
    int index = -1;
    int minBlockSize = MAX_SIZE + 1;
    for (int i = 0; i < size; ++i) {
        if (!blocks[i].allocated && blocks[i].size >= processSize) {
            if (blocks[i].size < minBlockSize) {
                minBlockSize = blocks[i].size;
                index = i;
            }
        }
    }
    return index;
}
int firstFit(Block blocks[], int size, int processSize) {
    for (int i = 0; i < size; ++i) {
        if (!blocks[i].allocated && blocks[i].size >= processSize) {
            return i;
        }
    }
    return -1;
}
void allocateMemory(Block blocks[], int size, int processSize, int (*allocationTechnique)(Block[], int, int), char
*allocationName) {
    int index = allocationTechnique(blocks, size, processSize);
    if (index != -1) {
        blocks[index].size = processSize;
        blocks[index].allocated = 1;
        printf("Memory allocated for process of size %d using %s at block %d\n", processSize, allocationName,
blocks[index].id);
    } else {
        printf("Cannot allocate memory for process of size %d using %s\n", processSize, allocationName);
    }
}
int main() {
    Block blocks[MAX_SIZE];
    int choice, processSize;
```

```
    initialize(blocks, MAX_SIZE);
    do {
        printf("Memory Allocation Techniques:\n");
        printf("1. Worst Fit\n");
        printf("2. Best Fit\n");
        printf("3. First Fit\n");
        printf("4. Display Memory Blocks\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter process size to allocate: ");
                scanf("%d", &processSize);
                allocateMemory(blocks, MAX_SIZE, processSize, worstFit, "Worst Fit");
                break;
            case 2:
                printf("Enter process size to allocate: ");
                scanf("%d", &processSize);
                allocateMemory(blocks, MAX_SIZE, processSize, bestFit, "Best Fit");
                break;
            case 3:
                printf("Enter process size to allocate: ");
                scanf("%d", &processSize);
                allocateMemory(blocks, MAX_SIZE, processSize, firstFit, "First Fit");
                break;
            case 4:
                displayBlocks(blocks, MAX_SIZE);
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }
while (choice != 5);
    return 0;
}
```

OUTPUT:

Memory Allocation Techniques:

1. Worst Fit
2. Best Fit
3. First Fit
4. Display Memory Blocks
5. Exit

Enter your choice: 1

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

SOURCE CODE:

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

#define MAX_FRAMES 3
#define MAX_PAGES 20

void displayPages(int pages[], int n) {
    printf("Current Pages in Memory: ");
    for (int i = 0; i < n; ++i) {
        printf("%d ", pages[i]);
    }
    printf("\n");
}

int findPage(int pages[], int n, int target) {
    for (int i = 0; i < n; ++i) {
        if (pages[i] == target) {
            return i;
        }
    }
    return -1;
}

void FIFO(int pageRequests[], int numRequests) {
    int frames[MAX_FRAMES];
    int frameIndex = 0;
    int pageFaults = 0;

    for (int i = 0; i < numRequests; ++i) {
        printf("Referencing Page %d\n", pageRequests[i]);
        displayPages(frames, frameIndex);

        if (findPage(frames, frameIndex, pageRequests[i]) == -1) {
            if (frameIndex < MAX_FRAMES) {
                frames[frameIndex++] = pageRequests[i];
            } else {
```

```
            for (int j = 0; j < MAX_FRAMES - 1; ++j) {
                frames[j] = frames[j + 1];
            }
            frames[MAX_FRAMES - 1] = pageRequests[i];
        }
        printf("Page %d is not in memory. Page fault!\n", pageRequests[i]);
        pageFaults++;
    } else {
        printf("Page %d is already in memory. No page fault.\n", pageRequests[i]);
    }
}

printf("Total page faults with FIFO: %d\n", pageFaults);
}

void LRU(int pageRequests[], int numRequests) {
    int frames[MAX_FRAMES];
    int pageFaults = 0;
    int used[MAX_FRAMES];

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

    for (int i = 0; i < numRequests; ++i) {
        printf("Referencing Page %d\n", pageRequests[i]);
        displayPages(frames, MAX_FRAMES);

        int page_index = findPage(frames, MAX_FRAMES, pageRequests[i]);

        if (page_index == -1) {
            int replace_index = 0;
            for (int j = 1; j < MAX_FRAMES; ++j) {
                if (used[j] < used[replace_index]) {
                    replace_index = j;
                }
            }
```

```
            frames[replace_index] = pageRequests[i];
            used[replace_index] = i;
            printf("Page %d is not in memory. Page fault!\n", pageRequests[i]);
            pageFaults++;
        } else {
            used[page_index] = i;
            printf("Page %d is already in memory. No page fault.\n", pageRequests[i]);
        }
    }

    printf("Total page faults with LRU: %d\n", pageFaults);
}

int main() {
    int pageRequests[MAX_PAGES] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
    int numRequests = 12;

    printf("Simulating FIFO Page Replacement Algorithm:\n");
    FIFO(pageRequests, numRequests);

    printf("\nSimulating LRU Page Replacement Algorithm:\n");
    LRU(pageRequests, numRequests);

    return 0;
}
```

OUTPUT:

Simulating FIFO Page Replacement Algorithm:
Referencing Page 1
Current Pages in Memory:
Page 1 is not in memory. Page fault!
Referencing Page 2
Current Pages in Memory: 1
Page 2 is not in memory. Page fault!
Referencing Page 3
Current Pages in Memory: 1 2
Page 3 is not in memory. Page fault!
Referencing Page 4
Current Pages in Memory: 1 2 3
Page 4 is not in memory. Page fault!
Referencing Page 1
Current Pages in Memory: 2 3 4
Page 1 is not in memory. Page fault!
Referencing Page 2
Current Pages in Memory: 3 4 1
Page 2 is not in memory. Page fault!
Referencing Page 5
Current Pages in Memory: 4 1 2
Page 5 is not in memory. Page fault!

Referencing Page 1
Current Pages in Memory: 1 2 5
Page 1 is already in memory. No page fault.
Referencing Page 2
Current Pages in Memory: 1 2 5
Page 2 is already in memory. No page fault.
Referencing Page 3
Current Pages in Memory: 1 2 5
Page 3 is not in memory. Page fault!
Referencing Page 4
Current Pages in Memory: 2 5 3
Page 4 is not in memory. Page fault!
Referencing Page 5
Current Pages in Memory: 5 3 4
Page 5 is already in memory. No page fault.
Total page faults with FIFO: 9

Simulating LRU Page Replacement Algorithm:
Referencing Page 1
Current Pages in Memory: -1 -1 -1
Page 1 is not in memory. Page fault!
Referencing Page 2
Current Pages in Memory: 1 -1 -1
Page 2 is not in memory. Page fault!
Referencing Page 3
Current Pages in Memory: 2 -1 -1
Page 3 is not in memory. Page fault!
Referencing Page 4
Current Pages in Memory: 3 -1 -1
Page 4 is not in memory. Page fault!
Referencing Page 1
Current Pages in Memory: 4 -1 -1
Page 1 is not in memory. Page fault!
Referencing Page 2
Current Pages in Memory: 1 -1 -1
Page 2 is not in memory. Page fault!
Referencing Page 5
Current Pages in Memory: 2 -1 -1
Page 5 is not in memory. Page fault!
Referencing Page 1
Current Pages in Memory: 5 -1 -1
Page 1 is not in memory. Page fault!
Referencing Page 2
Current Pages in Memory: 1 -1 -1
Page 2 is not in memory. Page fault!
Referencing Page 3
Current Pages in Memory: 2 -1 -1
Page 3 is not in memory. Page fault!
Referencing Page 4
Current Pages in Memory: 3 -1 -1
Page 4 is not in memory. Page fault!
Referencing Page 5
Current Pages in Memory: 4 -1 -1
Page 5 is not in memory. Page fault!
Total page faults with LRU: 12

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

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_FILES 5
#define MAX_NAME_LENGTH 20
typedef struct {
    char name[MAX_NAME_LENGTH];
    int size;
} File;

typedef struct {
    char name[MAX_NAME_LENGTH];
    int numFiles;
    File files[MAX_FILES];
} Directory;

Directory singleLevelDirectory;
Directory twoLevelDirectory[MAX_FILES];

void initializeSingleLevelDirectory() {
    strcpy(singleLevelDirectory.name, "Root");
    singleLevelDirectory.numFiles = 0;
}

void initializeTwoLevelDirectory()  {
    for (int i = 0; i < MAX_FILES; ++i) {
        sprintf(twoLevelDirectory[i].name, "Directory%d", i + 1);
        twoLevelDirectory[i].numFiles = 0;
    }
}


void displaySingleLevelDirectory() {
    printf("Single Level Directory:\n");
    printf("Directory Name: %s\n", singleLevelDirectory.name);
    printf("Number of Files: %d\n", singleLevelDirectory.numFiles);
```

```
    printf("Files:\n");
    for (int i = 0; i < singleLevelDirectory.numFiles; ++i) {
        printf("File Name: %s, Size: %d KB\n", singleLevelDirectory.files[i].name, singleLevelDirectory.files[i].size);
    }
    printf("\n");
}

void displayTwoLevelDirectory() {
    printf("Two Level Directory:\n");
    for (int i = 0; i < MAX_FILES; ++i) {
        printf("Directory Name: %s\n", twoLevelDirectory[i].name);
        printf("Number of Files: %d\n", twoLevelDirectory[i].numFiles);
        printf("Files:\n");
        for (int j = 0; j < twoLevelDirectory[i].numFiles; ++j) {
            printf("File Name: %s, Size: %d KB\n", twoLevelDirectory[i].files[j].name, twoLevelDirectory[i].files[j].size);
        }
        printf("\n");
    }
}
void addFileSingleLevelDirectory(char name[], int size) {
    if (singleLevelDirectory.numFiles < MAX_FILES) {
        strcpy(singleLevelDirectory.files[singleLevelDirectory.numFiles].name, name);
        singleLevelDirectory.files[singleLevelDirectory.numFiles].size = size;
        singleLevelDirectory.numFiles++;
        printf("File '%s' added to Single Level Directory\n", name);
    } else {
        printf("Single Level Directory is full, cannot add file '%s'\n", name);
    }
}

void addFileTwoLevelDirectory(char name[], int size, int directoryIndex) {
    if (directoryIndex >= 0 && directoryIndex < MAX_FILES) {
        if (twoLevelDirectory[directoryIndex].numFiles < MAX_FILES) {
            strcpy(twoLevelDirectory[directoryIndex].files[twoLevelDirectory[directoryIndex].numFiles].name, name);
            twoLevelDirectory[directoryIndex].files[twoLevelDirectory[directoryIndex].numFiles].size = size;
            twoLevelDirectory[directoryIndex].numFiles++;
            printf("File '%s' added to Directory '%s'\n", name, twoLevelDirectory[directoryIndex].name);
        } else {
            printf("Directory '%s' is full, cannot add file '%s'\n", twoLevelDirectory[directoryIndex].name, name);
```

```
        }
    } else {
        printf("Invalid Directory Index for Two Level Directory\n");
    }
}

int main() {
    initializeSingleLevelDirectory();
    initializeTwoLevelDirectory();

    addFileSingleLevelDirectory("file1.txt", 1024);
    addFileSingleLevelDirectory("file2.txt", 2048);
    addFileSingleLevelDirectory("file3.txt", 3072);

    displaySingleLevelDirectory();

    addFileTwoLevelDirectory("file4.txt", 4096, 0);
    addFileTwoLevelDirectory("file5.txt", 5120, 1);
    addFileTwoLevelDirectory("file6.txt", 6144, 2);

    displayTwoLevelDirectory();

    return 0;
}
```

OUTPUT:

```
File 'file1.txt' added to Single Level Directory
File 'file2.txt' added to Single Level Directory
File 'file3.txt' added to Single Level Directory
Single Level Directory:
Directory Name: Root
Number of Files: 3
Files:
File Name: file1.txt, Size: 1024 KB
File Name: file2.txt, Size: 2048 KB
File Name: file3.txt, Size: 3072 KB

File 'file4.txt' added to Directory 'Directory1'
File 'file5.txt' added to Directory 'Directory2'
File 'file6.txt' added to Directory 'Directory3'
Two Level Directory:
Directory Name: Directory1
Number of Files: 1
Files:
File Name: file4.txt, Size: 4096 KB
```

Directory Name: Directory2
Number of Files: 1
Files:
File Name: file5.txt, Size: 5120 KB

Directory Name: Directory3
Number of Files: 1
Files:
File Name: file6.txt, Size: 6144 KB

Directory Name: Directory4
Number of Files: 0
Files:

Directory Name: Directory5
Number of Files: 0
Files:

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

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_FILES 5
#define MAX_NAME_LENGTH 10
typedef struct {
    char name[MAX_NAME_LENGTH];
    int size;
    int blocks[100]; // Assuming a maximum of 100 blocks per file for simplicity
    int numBlocks;
} File;
File files[MAX_FILES];
int disk[1000]; // Simulating disk blocks
void initializeDisk() {
    for (int i = 0; i < 1000; ++i) {
        disk[i] = 0;
    }
}
void displayDiskStatus() {
    printf("Disk Blocks Status:\n");
    for (int i = 0; i < 100; ++i) {
        if (i % 10 == 0 && i != 0) {
            printf("\n");
        }
        printf("%d ", disk[i]);
    }
    printf("\n\n");
}
void createFile(char name[], int size) {
    int blockCounter = 0;
    for (int i = 0; i < MAX_FILES; ++i) {
        if (files[i].size == 0) {
            strcpy(files[i].name, name);
            files[i].size = size;

            for (int j = 0; j < size; ++j) {
                files[i].blocks[blockCounter++] = i * size + j + 1;
                disk[i * size + j + 1] = 1; // Allocating blocks on disk
            }
            files[i].numBlocks = blockCounter;
            printf("File '%s' created with size %d KB\n", name, size);
            return;
        }
    }
    printf("Cannot create file '%s'. File limit reached.\n", name);
}
void deleteFile(char name[]) {
    for (int i = 0; i < MAX_FILES; ++i) {
        if (strcmp(files[i].name, name) == 0) {
            for (int j = 0; j < files[i].numBlocks; ++j) {
                disk[files[i].blocks[j]] = 0; // Deallocating blocks on disk
            }
            files[i].size = 0;
```

```
            printf("File '%s' deleted\n", name);
            return;
        }
    }
    printf("File '%s' not found\n", name);
}
void displayFiles() {
    printf("Files on Disk:\n");
    for (int i = 0; i < MAX_FILES; ++i) {
        if (files[i].size != 0) {
            printf("File Name: %s, Size: %d KB, Blocks: ", files[i].name, files[i].size);
            for (int j = 0; j < files[i].numBlocks; ++j) {
                printf("%d ", files[i].blocks[j]);
            }
            printf("\n");
        }
    }
    printf("\n");
}

int main() {
    initializeDisk();
    createFile("file1.txt", 4);
    createFile("file2.txt", 3);
    createFile("file3.txt", 5);
    displayDiskStatus();
    displayFiles();
    deleteFile("file2.txt");
    displayDiskStatus();
    displayFiles();
    return 0;
}
```

OUTPUT:

```
File 'file1.txt' created with size 4 KB
File 'file2.txt' created with size 3 KB
File 'file3.txt' created with size 5 KB
Disk Blocks Status:
0 1 1 1 1 1 1 0 0 0
0 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

Files on Disk:
File Name: file1.txt, Size: 4 KB, Blocks: 1 2 3 4
File Name: file2.txt, Size: 3 KB, Blocks: 4 5 6
File Name: file3.txt, Size: 5 KB, Blocks: 11 12 13 14 15

File 'file2.txt' deleted
Disk Blocks Status:
0 1 1 1 0 0 0 0 0 0
0 1 1 1 1 1 0 0 0 0
```

0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

Files on Disk:
File Name: file1.txt, Size: 4 KB, Blocks: 1 2 3 4
File Name: file3.txt, Size: 5 KB, Blocks: 11 12 13 14 15

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

SOURCE CODE:

```
#include  <stdio.h>
#include <stdlib.h>
typedef struct {
  int track;
} request_t;
void scan(request_t requests[], int head, int n) {
  int direction = 1; // 1 for moving towards larger tracks, -1 for moving towards smaller tracks
  // Sort requests in ascending order
  for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - 1 - i; j++) {
      if (requests[j].track > requests[j + 1].track) {
        request_t temp = requests[j];
        requests[j] = requests[j + 1];
        requests[j + 1] = temp;
      }
    }
  }
  // Service requests in the specified direction
  int seekTime = 0;
  for (int i = 0; i < n; i++) {
    if (direction == 1 && requests[i].track >= head) {
      seekTime += requests[i].track - head;
      head = requests[i].track;
    } else if (direction == -1 && requests[i].track <= head) {
      seekTime += head - requests[i].track;
      head = requests[i].track;
    }
  }
  // Reverse the direction and service requests in the opposite direction
  direction *= -1;
  for (int i = n - 1; i >= 0; i--) {
    if (direction == 1 && requests[i].track >= head) {
      seekTime += requests[i].track - head;
      head = requests[i].track;
    } else if (direction == -1 && requests[i].track <= head) {
      seekTime += head - requests[i].track;
      head = requests[i].track;
    }
  }
  printf("Total seek time: %d\n", seekTime);
}

int main() {
  int diskSize = 200; // Disk size in tracks
  int headPosition = 50; // Initial head position

  int n = 4; // Number of requests
  request_t requests[] = {{85}, {150}, {30}, {90}};

  scan(requests, headPosition, n);

  return 0;
}
```

OUTPUT:

Total seek time: 220

OR

Source Code:

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

#define MAX_REQUESTS 100

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

void sortRequests(int requests[], int numRequests) {
    for (int i = 0; i < numRequests - 1; ++i) {
        for (int j = 0; j < numRequests - i - 1; ++j) {
            if (requests[j] > requests[j + 1]) {
                swap(&requests[j], &requests[j + 1]);
            }
        }
    }
}

void SCAN(int requests[], int head, int numRequests, char direction) {
    int totalSeekTime = 0;
    int currHead = head;
    int endIndex, startIndex;

    sortRequests(requests, numRequests);

    int i;
    for (i = 0; i < numRequests; ++i) {
        if (requests[i] >= head) {
            break;
        }
    }

    if (direction == 'L') {
        endIndex = i - 1;
        startIndex = i;
    } else if (direction == 'R') {
        endIndex = numRequests - 1;
        startIndex = i - 1;
    } else {
        printf("Invalid direction\n");
        return;
    }

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

```
    if (direction == 'L') {
        for (int i = startIndex; i >= 0; --i) {
            printf("%d -> ", requests[i]);
            totalSeekTime += abs(requests[i] - currHead);
            currHead = requests[i];
        }

        totalSeekTime += abs(0 - currHead);
        currHead = 0;

        for (int i = startIndex + 1; i <= endIndex; ++i) {
            printf("%d -> ", requests[i]);
            totalSeekTime += abs(requests[i] - currHead);
            currHead = requests[i];
        }
    } else if (direction == 'R') {
        for (int i = startIndex; i <= endIndex; ++i) {
            printf("%d -> ", requests[i]);
            totalSeekTime += abs(requests[i] - currHead);
            currHead = requests[i];
        }

        totalSeekTime += abs(199 - currHead);
        currHead = 199;

        for (int i = startIndex - 1; i >= 0; --i) {
            printf("%d -> ", requests[i]);
            totalSeekTime += abs(requests[i] - currHead);
            currHead = requests[i];
        }
    }

    printf("End\n");
    printf("Total seek time: %d\n", totalSeekTime);
}

int main() {
    int requests[MAX_REQUESTS];
    int numRequests, head;
    char direction;

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

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

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

    printf("Enter the direction (L/R): ");
    scanf(" %c", &direction);
```

```
    SCAN(requests, head, numRequests, direction);

    return 0;
}
```

OUTPUT:

```
Enter the number of requests: 2
Enter the requests (cylinder numbers):
4
6
Enter the initial position of the head: Right
Enter the direction (L/R): Sequence of head
movement:2 -> 4 -> 6 -> End
Total seek time: 199
```