National Institute of Technology, Hamirpur

Computer Science & Engineering

Operating System Lab File
CS-225

Submitted By:

Name: Tej Pratap Yadav
Roll No: 195104

INDEX

Assignment 1

Unix Commands

1. bc: bc command is used for command line calculator.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
2
2
2 + 4
6
```

2. cal: Displays a calendar.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ cal
    January 2021
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

3. cd: It is used to change the directory.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ cd ..
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem$ cd os
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

4. clear: It clears the terminal screen.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ clear
```

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

5. cp: cp command copies file from one location to another.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os/lab1$ cp bc.jpg ..
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os/lab1$ ls
bc.JPG   cd.JPG       clearout.JPG  ls.JPG     manout.JPG  pwd.JPG    stty.JPG  uname.JPG  whoami.JPG
cal.JPG  clearin.JPG  echo.JPG      manin.JPG  mkdir.JPG   rmdir.JPG  tty.JPG   who.JPG
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os/lab1$ cd ..
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ ls
Lab1  bc.jpg
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

6. echo: It prints the given input string to standard output.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ echo Hello
Hello
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

7. exit: It is used to terminate program, hell or log you out of a network normally.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os/lab1$ exit
```

8. ls: Lists the content of a directory.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ ls
Lab1
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

9. mkdir: This command is used to create a new directory.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ mkdir lab2
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ ls
Lab1  lab2
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

10. pwd: Displays path from root to current directory.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ pwd
/mnt/d/Nit Hamirpur/4th Sem/os
```

11. man: provides in depth information about a requested command or allows user to search for commands related to a particular keyword.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ man
What manual page do you want?
For example, try 'man man'.
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ man cp
```

```
CP(1)                                                   User Commands

NAME
       cp - copy files and directories

SYNOPSIS
       cp [OPTION]... [-T] SOURCE DEST
       cp [OPTION]... SOURCE... DIRECTORY
       cp [OPTION]... -t DIRECTORY SOURCE...

DESCRIPTION
       Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --archive
              same as -dR --preserve=all

       --attributes-only
              don't copy the file data, just the attributes

       --backup[=CONTROL]
              make a backup of each existing destination file

       -b     like --backup but does not accept an argument

       --copy-contents
              copy contents of special files when recursive

       -d     same as --no-dereference --preserve=links

       -f, --force
              if an existing destination file cannot be opened, remove it and try again (this option is ignored when the -n option is also used)

       -i, --interactive
              prompt before overwrite (overrides a previous -n option)

       -H     follow command-line symbolic links in SOURCE

       -l, --link
              hard link files instead of copying

 Manual page cp(1) line 1 (press h for help or q to quit)
```

12. mv: It is used to rename/ move file from one directory to other.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ mv bc.jpg bca.jpg
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ ls
Lab1   bca.jpg
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

13. tty: Print the file name of the terminal connected to the standard input.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ tty
/dev/pts/0
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

14. who: who command can list the name of the users currently logged in, their terminal, the time.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ who
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

15. rm: It is used to remove/delete the file from a directory.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ ls
Lab1  bca.jpg
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ rm bca.jpg
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ ls
Lab1
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

16. rmdir: It is used to delete/ remove a directory and its subdirectories.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ rmdir lab2
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ ls
Lab1
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

17. stty: Change and print terminal line settings.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ stty
speed 38400 baud; line = 0;
-brkint -imaxbel
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

18. uname: It is used to print system information.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ uname
Linux
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

19. whoami: Print effective user id.

```
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$ whoami
tej
tej@ASUS-VivoBook14:/mnt/d/Nit Hamirpur/4th Sem/os$
```

- chmod

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ chmod u=rw,g=r,o=r hello.txt
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- cmp

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ cmp numbers.txt alpha.txt
numbers.txt alpha.txt differ: byte 1, line 1
```

- cut

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ cut -d " " -f 1 hello.txt
hello
this
this
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- diff

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ diff alpha.txt alpha2.txt
1d0
< a
3,4d1
< r
< V
6d2
< d
7a4,6
> V
> a
> d
8a8
> r
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- file

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ file numbers.txt
numbers.txt: ASCII text
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ mkdir trial
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ file trial
trial: directory
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- find

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ find trial/tempfile.txt
trial/tempfile.txt
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- head

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ head -n 2 numbers.txt
1
2
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- ln

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ ln trial/tempfile.txt linktotempfile.txt
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ ls
alpha.txt  alpha2.txt  cmp.JPG  diff.JPG  file.JPG  find.JPG  head.JPG  hello.txt  linktotempfile.txt  nl.JPG  numbers.txt
```

- tail

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ tail -n 3 alpha.txt
d
E
f
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- more

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ more Pride\ and\ Prejudice\,\ by\ Jane\ Austen.txt

The Project Gutenberg EBook of Pride and Prejudice, by Jane Austen

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.org


Title: Pride and Prejudice

Author: Jane Austen

Release Date: August 26, 2008 [EBook #1342]
Last Updated: November 12, 2019


Language: English

Character set encoding: UTF-8

*** START OF THIS PROJECT GUTENBERG EBOOK PRIDE AND PREJUDICE ***




Produced by Anonymous Volunteers, and David Widger

THERE IS AN ILLUSTRATED EDITION OF THIS TITLE WHICH MAY VIEWED AT EBOOK
[# 42671 ]

cover




      Pride and Prejudice

      By Jane Austen

        CONTENTS

--More--(0%)
```

- paste

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ paste -d'_' alpha.txt numbers.txt
a_1
A_2
r_3
V_7
C_6
d_5
E_
f_
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- ps

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ ps
  PID TTY          TIME CMD
    9 pts/0    00:00:00 bash
   53 pts/0    00:00:00 ps
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- sleep

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ sleep 3
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- sort

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ sort alpha.txt
A
C
E
V
a
d
f
r
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$
```

- nl

```
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ cat alpha.txt
a
A
r
V
C
d
E
f
tej@ASUS-Vivobook14:/mnt/d/Nit Hamirpur/4th Sem/OS/Lab2/Linux Commands$ nl alpha.txt
     1  a
     2  A
     3  r
     4  V
     5  C
     6  d
     7  E
     8  f
```

Assignment 2

Write program in C/C++ program to implement following CPU scheduling Algorithms:

[1. FCFS]

[2. SJF]

1. First Come First Serve (FCFS):

```
#include <iostream>
#include <queue>
using namespace std;

class Process{
	int arrival_time;
	int burst_time;

	public:

	Process(int at, int bt){
		arrival_time = at;
		burst_time = bt;
	}

	void print_process(){
		cout<<"Arrival Time "<<arrival_time<<"\t";
		cout<<"Burst Time "<<burst_time<<endl;
	}

	int get_arrival_time(){
		return arrival_time;
	}
	int get_burst_time(){
		return burst_time;
	}

};

void print_desc(queue<Process> process_queue, int num_of_processes){
	int sno = 0;
	cout<<"=================================="<<endl;
	cout<<"\tDescription of Processes"<<endl;
	while(sno < num_of_processes){
		cout<<"Process Number: "<<sno++<<"\t";
		Process process = process_queue.front();
		process.print_process();
		process_queue.pop();
		cout<<endl;
	}
	cout<<"================================="<<endl;
}
```

```cpp
void prepare_gantt_chart(queue<Process> process_queue, int num_of_processes){
	cout<<"\n\tGantt Chart\n";
	float total_turnaround_time = 0;;
	float total_waiting_time = 0;
	int process_start_time = 0;
	int process_finish_time = 0;
	for(int i = 0; i < num_of_processes; i++){
		Process process = process_queue.front();
		cout<<"Process "<<i<<endl;
		cout<<"Started at: "<<process_start_time<<endl;
		process = process_queue.front();
		process_finish_time += process.get_burst_time();
		cout<<"finished at: "<<process_finish_time<<endl;
		total_turnaround_time += process_finish_time - process.get_arrival_time();
		total_waiting_time += (process_finish_time - process.get_arrival_time()) -
process.get_burst_time();
		process_start_time = process_finish_time;
		process_queue.pop();
		cout<<endl;
	}
	cout<<"\n\nAvg Turnaround Time = "<<total_turnaround_time/num_of_processes;
	cout<<"\nAvg Waiting Time = "<<total_waiting_time/num_of_processes<<endl;
}

void first_come_first_served(){
	cout<<"Enter Number of Processes: ";
	int num_of_processes;
	cin>>num_of_processes;
	queue<Process> process_queue;
	int num_of_process = 0;
	while(num_of_process < num_of_processes){
		cout<<"\nEnter Arrival Time of Process "<<num_of_process<<": ";
		int at;
		cin>>at;
		cout<<"Enter Burst Time of Process "<<num_of_process<<": ";
		int bt;
		cin>>bt;
		Process process(at, bt);
		process_queue.push(process);
		num_of_process++;
		cout<<"\n";
	}
	print_desc(process_queue, num_of_processes);
	prepare_gantt_chart(process_queue, num_of_processes);
}


int main(){
	cout<<"First Come First Served\n";
	cout<<"---------------------------"<<endl;
	first_come_first_served();
}
```

2. Shortest Job First (SJF):

```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>

using namespace std;

class Process{
	int arrival_time;
	int burst_time;
	int id;

	public:
	Process(int at, int bt, int id_){
		arrival_time = at;
		burst_time = bt;
		id = id_;
	}

	void print_process(){
		cout<<"Process Id: "<<id<<"\t";
		cout<<"Arrival Time: "<<arrival_time<<"\t";
		cout<<"Burst Time: "<<burst_time<<endl;
	}

	int get_arrival_time(){
		return arrival_time;
	}
	int get_burst_time(){
		return burst_time;
	}
	int get_id(){
		return id;
	}

	friend bool compare_processes(Process &a, Process &b);

};

bool compare_processes(Process &a, Process &b){
	return (a.get_burst_time() < b.get_burst_time());
}

void print_desc(queue<Process> process_queue);
void prepare_gantt_chart(queue<Process> process_queue);

void sort_process_queue(queue<Process> &process_queue){
	vector<Process> process_vector;
	while(!process_queue.empty()){
		process_vector.push_back(process_queue.front());
		process_queue.pop();
	}
	sort(process_vector.begin() + 1, process_vector.end(), compare_processes);
	for(int i = 0; i < process_vector.size(); i++){
		process_queue.push(process_vector[i]);
	}
	prepare_gantt_chart(process_queue);
}
```

```
void print_desc(queue<Process> process_queue){
	cout<<"=================================================="<<endl;
	cout<<"\tDescription of Processes"<<endl;
	int num_of_processes = process_queue.size();
	int sno = 1;
	while(num_of_processes--){
		Process process = process_queue.front();
		cout<<"\n"<<sno<<"\t";
		sno++;
		process.print_process();
		process_queue.pop();
		cout<<endl;
	}
	cout<<"================================================="<<endl;
}

void prepare_gantt_chart(queue<Process> process_queue){
	cout<<"\nAfter Job Scheduling"<<endl;
	print_desc(process_queue);
	cout<<"\n\tGantt Chart\n";
	float total_turnaround_time = 0;;
	float total_waiting_time = 0;
	int process_start_time = 0;
	int process_finish_time = 0;
	int num_of_processes = process_queue.size();
	for(int i = 0; i < num_of_processes; i++){
		Process process = process_queue.front();
		cout<<"Process Id: "<<process.get_id()<<endl;
		cout<<"Started at: "<<process_start_time<<endl;
		process_finish_time += process.get_burst_time();
		cout<<"finished at: "<<process_finish_time<<endl;
		total_turnaround_time += process_finish_time - process.get_arrival_time();
		total_waiting_time += (process_finish_time - process.get_arrival_time()) - 
process.get_burst_time();
		process_start_time = process_finish_time;
		process_queue.pop();
		cout<<endl;
	}
	cout<<"\nAvg Turnaround Time = "<<total_turnaround_time/num_of_processes;
	cout<<"\nAvg Waiting Time = "<<total_waiting_time/num_of_processes<<endl;
}
```

```cpp
void shortest_job_first(){
	cout<<"Enter Number of Processes: ";
	int num_of_processes;
	cin>>num_of_processes;
	queue<Process> process_queue;
	int num_of_process = 0;
	while(num_of_process < num_of_processes){
		cout<<"\nEnter Arrival Time of Process "<<num_of_process + 1<<": ";
		int at;
		cin>>at;
		cout<<"Enter Burst Time of Process "<<num_of_process + 1<<": ";
		int bt;
		cin>>bt;
		cout<<endl;
		int id = num_of_process;
		Process process(at, bt, id);
		process_queue.push(process);
		num_of_process++;
	}
	print_desc(process_queue);
	sort_process_queue(process_queue);

}


int main(){
	cout<<"Shortest Job First\n";
	cout<<"---------------------------"<<endl;
	shortest_job_first();
}
```

Assignment 3

Write program in C/C++ program to implement following CPU scheduling Algorithms:

1. Shortest Remaining Time Next (SRTN) Scheduling

2. Round Robin (RR) Scheduling

3. Priority Scheduling (Pre-emptive Priority and Non-Pre-emptive Priority)

1. Shortest Remaining Time Next (SRTN) Scheduling:

```
#include <stdio.h>

// sturcture to store a process
struct Process{
    int id;
    int arrival_time;
    int burst_time;
    int finish_time;
    int initial_burst_time;
};

int isEmpty(struct Process *queue, int size);
int getProcessIndex(struct Process *queue, int size, int cpu_time);
void sort_process_queue(struct Process *queue, int size);
void prepare_gantt_chart(struct Process *queue, int size);

int main(){
    printf("\tShortest Remaining Time Next\n");
    int SIZE;       // number of processes
    printf("Enter number of processes: ");
    scanf("%d", &SIZE);

    struct Process job_queue[SIZE];

    // taking input of processes and their details:
    for(int i = 0; i < SIZE; i++){
        printf("\nProcess Id: %d\n", i);
        job_queue[i].id = i;
        printf("Enter Arrival Time: ");
        scanf("%d", &job_queue[i].arrival_time);
        printf("Enter Burst Time: ");
        scanf("%d", &job_queue[i].burst_time);
        // initially we mark finish_time to be -1
        // we will change it when it leaves job_queue
        job_queue[i].finish_time = -1;
        job_queue[i].initial_burst_time = job_queue[i].burst_time;
    }
```

```c
    // Printing Process Details before execution
    printf("\nBefore Execution Starts: \n");
    for(int i = 0; i < SIZE; i++){
        printf(
                "\nProcess ID: %d\n Arrival Time: %d\n Burst Time: %d\n Finish Time:
%d\n",
                 job_queue[i].id,
                 job_queue[i].arrival_time,
                 job_queue[i].burst_time,
                 job_queue[i].finish_time
             );
    }

    // Job Scheduling
    int cpu_time = 0;
    int waste_time = 0;
    int pid = -1;
    while(!isEmpty(job_queue, SIZE)){
        pid = getProcessIndex(job_queue, SIZE, cpu_time);
        if(pid == -1){
            // printf("\nCPU is waiting at time: %d\n", cpu_time);
            cpu_time++;
            waste_time++;
        }
        else{
            // printf("\nProcess Id %d is being executed at time %d\n", pid,
cpu_time);
            cpu_time++;
            job_queue[pid].burst_time--;
            if(job_queue[pid].burst_time == 0){
                job_queue[pid].finish_time = cpu_time;
            }
        }
    }

    prepare_gantt_chart(job_queue, SIZE);
    return 0;
}

// initially we marked finish_time to be -1
// we modify it when the process actually finishes
// so if it is not -1 it has finished
int isEmpty(struct Process *queue, int size){
    for(int i = 0; i < size; i++){
        if(queue[i].finish_time == -1){
            return 0;
        }
    }
    return 1;
}

// returns the index of the process to be executed
// as per the Shortest Remaining Time Next algorithm
int getProcessIndex(struct Process *queue, int size, int cpu_time){
    sort_process_queue(queue, size);
    for(int i = 0; i < size; i++){
        if(queue[i].finish_time == -1){
            if(queue[i].arrival_time <= cpu_time){
                return i;
            }
        }
    }
```

```c
    return -1;
}

// sorts the process queue as per the shortest
// remaining cpu burst time
void sort_process_queue(struct Process *queue, int size){
    for(int i = 0; i < size - 1; i++){
        for(int j = 0; j < size - 1 - i; j++){
            if(queue[j].burst_time > queue[j + 1].burst_time){
                struct Process temp = queue[j];
                queue[j] = queue[j+1];
                queue[j+1] = temp;
            }
            // if two process have the same burst time
            // we give priority to the arrival time
            if(queue[j].burst_time == queue[j + 1].burst_time){
                if(queue[j].arrival_time > queue[j + 1].arrival_time){
                    struct Process temp = queue[j];
                    queue[j] = queue[j+1];
                    queue[j+1] = temp;
                }
            }
        }
    }
}

void prepare_gantt_chart(struct Process *queue, int size){
    float total_waiting_time = 0;
    float total_turnaround_time = 0;
    for(int i = 0; i < size; i++){
        int waiting_time = 0;
        int turnaround_time = 0;
        turnaround_time = queue[i].finish_time - queue[i].arrival_time;
        waiting_time = turnaround_time - queue[i].initial_burst_time;
        total_waiting_time += waiting_time;
        total_turnaround_time += turnaround_time;
    }

    // Printing after completing all process execution
    // Gantt chart can be created using this easily
    printf("\nAfter Execution of all Processes: \n");
    for(int i = 0; i < size; i++){
        printf(
            "\nProcess ID: %d\nArrival Time: %d\nFinish Time:  %d\n",
                 queue[i].id,
                 queue[i].arrival_time,
                 queue[i].finish_time
             );
    }

    printf("\nAverage Turnaround Time: %.2f\n", total_turnaround_time / size);
    printf("\nAverage Waiting Time: %.2f\n", total_waiting_time / size);
}
```

2. Round Robin (RR) Scheduling

```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>
using namespace std;

struct Process{
    int id;
    int arrival_time;
    int burst_time;
    int finish_time;
    int valid_bit;
    int initial_burst_time;
    int initial_arrival_time;
};

Process fake_process;
Process current_process;
void getProcess(int &cpu_time, queue<Process> &process_queue, int context_switch_time);
void calculation(vector<Process> process_queue);
void sort_process_queue(queue<Process> &process_queue);

int main(){
    // SIZE stores total number of processes
    // time_quanta stores the value of Time Quanta
    int SIZE, time_quanta;
    cout<<"Enter Number of Processes: ";
    cin>>SIZE;
    cout<<"Enter value of Time Quanta: ";
    cin>>time_quanta;
    cout<<"Enter Context Switch Time: ";
    int context_switch_time;
    cin>>context_switch_time;

    cout<<"\nEnter Details of Processes: \n";
    // queue to store the Processes
    queue<Process> process_queue;
    for(int i = 0; i < SIZE; i++){
        Process process;
        process.id = i;
        cout<<"\nProcess Id: "<<i<<endl;
        cout<<"Enter Arrival Time: ";
        cin>>process.arrival_time;
        process.initial_arrival_time = process.arrival_time;
        cout<<"Enter Burst Time: ";
        cin>>process.burst_time;
        process.initial_burst_time = process.burst_time;
        process.finish_time = -1;
        process.valid_bit = 1;
        process_queue.push(process);
    }

    // Scheduling as per the Round Robin Algorithm
    // finished_vector stores the processes those have
    // finished Execution
    vector<Process> finished_vector;
    int cpu_time = 0, waste_time = 0, remaining_time = time_quanta;
    cout<<"\n\tStarting Execution\t CPU Time: "<<cpu_time<<endl;
```

```
    getProcess(cpu_time, process_queue, context_switch_time);
    while(finished_vector.size() != SIZE){

        if(current_process.valid_bit == -1){
            // cout<<"\nNo Process \n";
            cpu_time++;
            waste_time++;
            getProcess(cpu_time, process_queue, context_switch_time);
        }
        else{
            cout<<"\nServing process: "<<current_process.id<<endl;
            cpu_time++;
            remaining_time--;
            current_process.burst_time--;
            cout<<"\nCPU Time: "<<cpu_time<<"\tTime Quanta Remaining: "<<remain-
ing_time<<endl;
            // cout<<"\nburst time of process "<<current_process.id<<" "<<current_pro-
cess.burst_time<<endl;
            if(current_process.burst_time == 0){
                cout<<"\n===Process "<<current_process.id<<" Finished====="<<endl;
                current_process.finish_time = cpu_time;
                finished_vector.push_back(current_process);
                getProcess(cpu_time, process_queue, context_switch_time);
                remaining_time = time_quanta;
            }
            else if(remaining_time == 0){
                cout<<"\n!!Time Quanta is over!!\n";
                Process previous_process = current_process;
                previous_process.arrival_time = cpu_time;
                getProcess(cpu_time, process_queue, context_switch_time);
                process_queue.push(previous_process);
                remaining_time = time_quanta;
            }
        }
    }

    // Printing the details of processes after Execution
    cout<<"\nProcesses after execution:\n";
    for(int i = 0; i < SIZE; i++){
        Process process = finished_vector[i];
        cout<<"\nProcess Id: "<<process.id<<endl;
        cout<<"Process Arrival Time: "<<process.arrival_time<<endl;
        cout<<"Process Finish Time: "<<process.finish_time<<endl;
    }
    calculation(finished_vector);
    return 0;
}

void getProcess(int &cpu_time, queue<Process>& process_queue, int context_switch_time){
    // cout<<"\nIN getProcess: "<<"cpu_time: \n"<<cpu_time;
    fake_process.valid_bit = -1;
    // checking if any process has arrived or not
    sort_process_queue(process_queue);
    if(process_queue.front().arrival_time <= cpu_time){
        if(cpu_time > 0){
            cpu_time += context_switch_time;
            cout<<"\nSwitching to new Process.......\n";
```

```
            cout<<"\nAdding context switch time:"<<context_switch_time<<"\nCurrent CPU
Time: "<<cpu_time<<endl;
        }

        current_process = process_queue.front();
        process_queue.pop();
    }
    else{
        current_process = fake_process;
    }
}

void sort_process_queue(queue<Process> &process_queue){
       vector<Process> process_vector;
       while(!process_queue.empty()){
              process_vector.push_back(process_queue.front());
              process_queue.pop();
       }
       sort(process_vector.begin(), process_vector.end(), [](Process &a, Process &b){
       return (a.arrival_time < b.arrival_time);
    });
       for(int i = 0; i < process_vector.size(); i++){
              process_queue.push(process_vector[i]);
       }
}

void calculation(vector<Process> process_queue){
    float total_waiting_time = 0;
    float total_turnaround_time = 0;
    int size = process_queue.size();
    for(int i = 0; i < size; i++){
        int waiting_time = 0;
        int turnaround_time = 0;
        turnaround_time = process_queue[i].finish_time - process_queue[i].initial_ar-
rival_time;
        waiting_time = turnaround_time - process_queue[i].initial_burst_time;
        total_waiting_time += waiting_time;
        total_turnaround_time += turnaround_time;
    }
    cout<<"\nAverage Turnaround Time: " << total_turnaround_time / size;
    cout<<"\nAverage Waiting Time: " << total_waiting_time / size;
}
```

3. Priority Scheduling

a. Pre-emptive Priority

```c
#include <stdio.h>

// sturcture to store a process
struct Process{
    int id;
    int arrival_time;
    int burst_time;
    int finish_time;
    int initial_burst_time;
    // small value for priority means higher priority of the process
    int priority;
};

int isEmpty(struct Process *queue, int size);
int getProcessIndex(struct Process *queue, int size, int cpu_time);
void sort_process_queue(struct Process *queue, int size);
void prepare_gantt_chart(struct Process *queue, int size);

int main(){
    printf("\tPriority Pre-Emptive Scheduling Algorithm\n");
    int SIZE;       // number of processes
    printf("Enter number of processes: ");
    scanf("%d", &SIZE);

    struct Process job_queue[SIZE];

    // taking input of processes and their details:
    for(int i = 0; i < SIZE; i++){
        printf("\nProcess Id: %d\n", i);
        job_queue[i].id = i;
        printf("Enter Arrival Time: ");
        scanf("%d", &job_queue[i].arrival_time);
        printf("Enter Burst Time: ");
        scanf("%d", &job_queue[i].burst_time);
        printf("Enter Priority: ");
        scanf("%d", &job_queue[i].priority);
        // initially we mark finish_time to be -1
        // we will change it when it leaves job_queue
        job_queue[i].finish_time = -1;
        job_queue[i].initial_burst_time = job_queue[i].burst_time;
    }


    // Printing Process Details before execution
    printf("\nBefore Execution Starts: \n");
    for(int i = 0; i < SIZE; i++){
        printf(
                "\nProcess ID: %d\nPriority: %d\nArrival Time: %d\nBurst Time: %d\nFinish 
Time:  %d\n",
                 job_queue[i].id,
                 job_queue[i].priority,
                 job_queue[i].arrival_time,
                 job_queue[i].burst_time,
                 job_queue[i].finish_time
             );
    }
```

```c
    // Job Scheduling
    int cpu_time = 0;
    int waste_time = 0;
    int pid = -1;
    while(!isEmpty(job_queue, SIZE)){
        pid = getProcessIndex(job_queue, SIZE, cpu_time);
        if(pid == -1){
            // printf("\nCPU is waiting at time: %d\n", cpu_time);
            cpu_time++;
            waste_time++;
        }
        else{
            // printf("\nProcess Id %d is being executed at time %d\n", pid,
cpu_time);
            cpu_time++;
            job_queue[pid].burst_time--;
            if(job_queue[pid].burst_time == 0){
                job_queue[pid].finish_time = cpu_time;
            }
        }
    }

    prepare_gantt_chart(job_queue, SIZE);
    return 0;
}

// initially we marked finish_time to be -1
// we modify it when the process actually finishes
// so if it is not -1 it has finished
int isEmpty(struct Process *queue, int size){
    for(int i = 0; i < size; i++){
        if(queue[i].finish_time == -1){
            return 0;
        }
    }
    return 1;
}

// returns the index of the process to be executed
// as per the Priority of the processes
int getProcessIndex(struct Process *queue, int size, int cpu_time){
    sort_process_queue(queue, size);
    for(int i = 0; i < size; i++){
        if(queue[i].finish_time == -1){
            if(queue[i].arrival_time <= cpu_time){
                return i;
            }
        }
    }
    return -1;
}

// sorts the process queue as per the priority
void sort_process_queue(struct Process *queue, int size){
    for(int i = 0; i < size - 1; i++){
        for(int j = 0; j < size - 1 - i; j++){
            if(queue[j].priority > queue[j + 1].priority){
                struct Process temp = queue[j];
                queue[j] = queue[j+1];
                queue[j+1] = temp;
            }
        }
```

```c
    }
}

void prepare_gantt_chart(struct Process *queue, int size){
    float total_waiting_time = 0;
    float total_turnaround_time = 0;
    for(int i = 0; i < size; i++){
        int waiting_time = 0;
        int turnaround_time = 0;
        turnaround_time = queue[i].finish_time - queue[i].arrival_time;
        waiting_time = turnaround_time - queue[i].initial_burst_time;
        total_waiting_time += waiting_time;
        total_turnaround_time += turnaround_time;
    }

    // Printing after completing all process execution
    // Gantt chart can be created using this easily
    printf("\nAfter Execution of all Processes: \n");
    for(int i = 0; i < size; i++){
        printf(
            "\nProcess ID: %d\nPriority: %d\nArrival Time: %d\nFinish Time:  %d\n",
                 queue[i].id,
                 queue[i].priority,
                 queue[i].arrival_time,
                 queue[i].finish_time
             );
    }

    printf("\nAverage Turnaround Time: %.2f\n", total_turnaround_time / size);
    printf("\nAverage Waiting Time: %.2f\n", total_waiting_time / size);
}
```

b. Non-Pre-emptive Priority

```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>

using namespace std;

class Process{
	int arrival_time;
	int burst_time;
	int id;
	// small value for priority means higher priority of the process
	int priority;

	public:

	Process(int at, int bt, int id_, int priority_){
		arrival_time = at;
		burst_time = bt;
		id = id_;
		priority = priority_;
	}

	void print_process(){
		cout<<"Process Id: "<<id<<"\t";
		cout<<"Process Priority: "<<priority<<"\t";
		cout<<"Arrival Time: "<<arrival_time<<"\t";
		cout<<"Burst Time: "<<burst_time<<endl;
	}

	int get_arrival_time(){
		return arrival_time;
	}
	int get_burst_time(){
		return burst_time;
	}
	int get_id(){
		return id;
	}
	int get_priority(){
		return priority;
	}

	friend bool compare_processes(Process &a, Process &b);

};

// print_desc - Prints the description of all the processes
void print_desc(queue<Process> process_queue);
// priorityNonPreemptive function does the Scheduling
// and calculation of average turnaround_time and average
// waiting_time.
void priorityNonPreemptive(queue<Process> process_queue);
// sort_process_queue -> sorts the processes as per
// their priority (low numerical value means high priority)
void sort_process_queue(queue<Process> &process_queue);
// takeInput -> takes input of the details of the processes
queue<Process> takeInput();
```

```
int main(){
	cout<<"Priority Non Pre-emptive Scheduling Algorithm\n";
	cout<<"--------------------------"<<endl;
	queue<Process> process_queue = takeInput();
	print_desc(process_queue);
	sort_process_queue(process_queue);
	priorityNonPreemptive(process_queue);
}

queue<Process> takeInput(){
	cout<<"Enter Number of Processes: ";
	int num_of_processes;
	cin>>num_of_processes;
	queue<Process> process_queue;
	int num_of_process = 0;
	while(num_of_process < num_of_processes){
		cout<<"\nEnter Arrival Time of Process "<<num_of_process + 1<<": ";
		int at;
		cin>>at;
		cout<<"Enter Burst Time of Process "<<num_of_process + 1<<": ";
		int bt;
		cin>>bt;
		cout<<"Enter Priority of the Process "<<num_of_process + 1<<": ";
		int priorityIn;
		cin>>priorityIn;
		cout<<endl;
		int id = num_of_process;
		Process process(at, bt, id, priorityIn);
		process_queue.push(process);
		num_of_process++;
	}
	return process_queue;
}

void print_desc(queue<Process> process_queue){
	cout<<"=================================================="<<endl;
	cout<<"\tDescription of Processes"<<endl;
	int num_of_processes = process_queue.size();
	int sno = 1;
	while(num_of_processes--){
		Process process = process_queue.front();
		cout<<"\n"<<sno<<"\t";
		sno++;
		process.print_process();
		process_queue.pop();
		cout<<endl;
	}
	cout<<"================================================="<<endl;
}

void sort_process_queue(queue<Process> &process_queue){
	vector<Process> process_vector;
	while(!process_queue.empty()){
		process_vector.push_back(process_queue.front());
		process_queue.pop();
	}
	sort(process_vector.begin() + 1, process_vector.end(), [](Process &a, Process
&b){
		return (a.get_priority() < b.get_priority());
	});
	for(int i = 0; i < process_vector.size(); i++){
		process_queue.push(process_vector[i]);
```

```
        }
}

void priorityNonPreemptive(queue<Process> process_queue){
        cout<<"\n\tGantt Chart\n";
        float total_turnaround_time = 0;;
        float total_waiting_time = 0;
        int process_start_time = 0;
        int process_finish_time = 0;
        int num_of_processes = process_queue.size();
        for(int i = 0; i < num_of_processes; i++){
                Process process = process_queue.front();
                cout<<"Process Id: "<<process.get_id()<<endl;
                cout<<"Started at: "<<process_start_time<<endl;
                process_finish_time += process.get_burst_time();
                cout<<"finished at: "<<process_finish_time<<endl;
                total_turnaround_time += process_finish_time -
process.get_arrival_time();
                total_waiting_time += (process_finish_time - process.get_arrival_time())
- process.get_burst_time();
                process_start_time = process_finish_time;
                process_queue.pop();
                cout<<endl;
        }
        cout<<"\nAvg Turnaround Time = "<<total_turnaround_time / num_of_processes;
        cout<<"\nAvg Waiting Time = "<<total_waiting_time / num_of_processes<<endl;
}
```

Assignment 4

Solution:

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

int X = 0;      //shared variable

//function A increments X by one 10 times
void *A()
{
    int a;
    a = X; //accessing X through a
    for(int i = 1; i <= 10; i++)
        a++;
    sleep(1);   //sleep slows down the process to simulate race condition
    X = a;
    sleep(1);

}

//function B decrements X by one 10 times
void *B()
{
    int b;
    for(int i = 1; i <= 10; i++)
        b--;
    sleep(1);
    X = b;
}

int main()
{
    //threads can behave like child processes
    pthread_t ta, tb;
    pthread_create(&ta, NULL, A, NULL);
    pthread_create(&tb, NULL, B, NULL);
    pthread_join(ta, NULL);
    pthread_join(tb, NULL);
    printf("Final value of shared value X is %d\n", X);
}
```

```c
//using semaphores for process synchronization to avoid race condition

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

 int X = 0;
 sem_t s; //semaphore

//process A increments X by one 10 times
 void *A()
{
    int a;
    sem_wait(&s);
    a = X;
    for (int i = 1; i <= 10; i++)
        a++;
    sleep(1);
    X = a;
    sem_post(&s);
}

//process B decrements X by one 10 times
void *B()
{
    int b;
    sem_wait(&s);
    b = X;
    for(int i = 1; i <= 10; i++)
        b--;
    sleep(1);
    X = b;
    sem_post(&s);
}

 int main()
 {
    sem_init(&s,0,1);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, A, NULL);
    pthread_create(&t2, NULL, B, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2,NULL);
    printf("Final value of shared variable X is %d\n", X);
 }
```

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

int main()
{
	int shmid,status;
	int *a,*b;
	int i,j,k;
	int m,n;

	printf("Enter number of producer and the number of consumers\n");
	scanf("%d%d",&m,&n);
	shmid=shmget(IPC_PRIVATE, 21*sizeof(int), 0777|IPC_CREAT);
	for(k=0;k<m;k++)
	{
		if(!fork())
		{

			b=(int *)shmat(shmid,0,0);
			for(i=0;i<50;)
			{
				sleep(2);
				for(j=0;j<20 && i<50;j++)
				{
					b[20] = b[20] + b[j];
					i++;
					printf(" %d ",b[j]);
				}
				printf("\n");
			}

			shmdt(b);
			exit(0);
		}


		a=(int *)shmat(shmid,0,0);
		for(i=0;i<50;)
		{
			sleep(2);
			for(j=0;j<20 && i<50;j++)
			{
				a[j]=(i+1);
				i++;
			}
		}
		wait(&status);
		printf(" sum in parent is = %d \n",a[20]);
		shmdt(a);
	}
	shmctl( shmid, IPC_RMID , 0 );
	return 0;
}
```

Assignment 5

Implement Banker's Algorithm:

```
#include <stdio.h>

#define N 5 // number of processes
#define M 3 // number of types of resources

void input_from_file();
void print_data();
int check_safe_state();
int available[M], max[N][M], allocation[N][M], need[N][M];

int main(){
    printf("%s\n", "\t\tBanker's Algorithm");
    input_from_file(); // takes input from a file in the format mentioned in the function
definition
    print_data();   // prints the available resources, allocation matrix, max matrix and
also calculates and prints need matrix

    int is_in_safe_state = check_safe_state();
    // printf("\n%d\n", is_in_safe_state);
    if (is_in_safe_state){
        printf("\nCongratulations! The system is in safe state as per the Banker's
Algorithm.\n");
    }
    else{
        printf("\nThe system is not in safe state as per the Banker's Algorithm. \nSo
this can lead to DeadLock!\n\t\t Be Careful!!!\n");
    }

    return 0;
}
```

```c
void input_from_file(){
    FILE* fin;
    fin = fopen("input.txt", "r");
    /*contents of input.txt:
    1st line available resources of each type
    2nd line onwards allocation Matrix
    followed by max Matrix.
    eg.
    1 5 2 0
    0 0 1 2
    1 0 0 0
    1 3 5 4
    0 6 3 2
    0 0 1 4
    0 0 1 2
    1 7 5 0
    2 3 5 6
    0 6 5 2
    0 6 5 6
    */

    if(fin == NULL){
        printf("%s\n", "Error in opening output file");
        // return -1;
    }

    for(int i = 0; i < M; i++){
        fscanf(fin, "%d", &available[i]);
    }

    for(int i = 0; i < N; i++){
        for (int j = 0; j < M; j++){
            fscanf(fin, "%d", &allocation[i][j]);
        }
    }

    for(int i = 0; i < N; i++){
        for (int j = 0; j < M; j++){
            fscanf(fin, "%d", &max[i][j]);
        }
    }

    fclose(fin);

}
```

```c
void print_data(){
    // FILE* fout;
    // fout = fopen("output.txt", "w");

    // if(fout == NULL){
    //     printf("%s\n", "Error in opening output file");
    //     return -1;
    // }
    printf("%s\n", "Available resources:");
    for(int j = 0; j < M; j++){
        // fprintf(fout, "%d ", available[j]);
        printf("%d ", available[j]);
    }
    // fprintf(fout, "\n");
    printf("\n\n");
    printf("%s\n", "Allocation Matrix:");
    for(int i = 0; i < N; i++){
        for (int j = 0; j < M; j++){
            // fprintf(fout, "%d ", max[i][j]);
            printf("%d ", allocation[i][j]);
        }
        // fprintf(fout, "\n");
        printf("\n");
    }
    printf("\n%s\n", "Max Requirement Matrix:");
    for(int i = 0; i < N; i++){
        for (int j = 0; j < M; j++){
            // fprintf(fout, "%d ", max[i][j]);
            printf("%d ", max[i][j]);
        }
        // fprintf(fout, "\n");
        printf("\n");
    }
    printf("\nNeed Matrix:\n");
    for(int i = 0; i < N; i++){
        for (int j = 0; j < M; j++){
            // fprintf(fout, "%d ", max[i][j]);
            need[i][j] = max[i][j] - allocation[i][j];
            printf("%d ", need[i][j]);
        }
        // fprintf(fout, "\n");
        printf("\n");
    }

    // fclose(fout);


}

int check_safe_state(){
    int Work[M], Finish[N];
    for (int i = 0; i < M; i++){
        Work[i] = available[i];     // Initially work has the value of available
resources.
    }

    for (int i = 0; i < N; i++){
        Finish[i] = 0;  // Initially all the processes are running
```

```
    }
    while(1){
        int flag = -1; // This is used the store the index of process
        for (int i = 0; i < N; i++){
            // First checking the process is running or has finished
            if (Finish[i] == 0){
                int flag1 = 0;
                // Then we check whether the available resources can fulfill the 
requirement of the process
                for (int j = 0; j < M; j++){
                    if (need[i][j] > Work[j]){
                        // requirement cannot be finished
                        flag1 = 1;
                        break;
                    }
                }
                // selecting the process as it is running as well as its requirements can 
be finished by the available set of resources
                if (flag1 == 0){
                    flag = i; // stores the index of the process which has satisfied the 
above two conditions
                    break;
                }
            }
        }

        // If none of the processes fulfills above two condition
        if (flag == -1){
            int flag2 = 0;
            for (int i = 0; i < N; i++){
                // Counting the processes those have finished
                if (Finish[i] == 1){
                    flag2 += 1;
                }
            }
            // If all the processes have finished returns 1 i.e is in safe state
            if (flag2 == N){
                return 1;
            }
            // else returns 0 i.e is not in safe state
            else{
                return 0;
            }
        }
        else{
            // If any process fulfills the two conditions:
            printf("\nProcess[%d] can be finished...\nReleasing resources...\n", flag+1);
            printf("%s", "New set of available Resources: ");
            for (int i = 0; i < M; i++){
                Work[i] = Work[i] + allocation[flag][i];    // Adding the released 
process to the available set
                printf("%d ", Work[i]);
            }
            printf("\n");
            Finish[flag] = 1;   // Marking that process to be finished
        }
    }
}
```

Assignment 6

Implement following page replacement algorithms:

1. Optimal Page Replacement Algorithm:

```
#include <stdio.h>
#define SIZE 3  // Number of Frames

char frames[SIZE];  // Frame
// ref_str is declared as global as the Optimal Page Replacement Algorithm
// requires future knowledge of the reference string.
char ref_str[] = "70120304230321201701";

// A function which returns an index for the insertion of next reference string
// based on the fact: 'replace the page that will not be used for the longest period of
time.'
int priority_index(int curr_index){
    int priority_array[SIZE] = {4, 4, 4};
    // printf("priority_array: %d %d %d\n", priority_array[0], priority_array[1],
priority_array[2]);
    // printf("curr_index: %d\n", curr_index);
    for(int k = 0; k < SIZE; k++){
        for(int j = curr_index; ref_str[j] != '\0'; j++){
            if(ref_str[j] == frames[k]){
                if(priority_array[k] == 4){
                    priority_array[k] = j - curr_index;
                    break;
                }
            }
        }
    }

    // printf("priority_array: %d %d %d\n", priority_array[0], priority_array[1],
priority_array[2]);
    int index = 0;
    for(int i = 1; i < SIZE; i++){
        if(priority_array[i] >= priority_array[index]){
            index = i;
        }
    }
    return index;
}
```

```c
// A function to insert the reference string at a particular index or
// to be more precise it replaces a reference to a page with the one
// which would not be used for the longest period of time

void insert(int curr_index, char ref_str){
    int index = priority_index(curr_index);
    // printf("Index to be inserted at: %d\n", index);
    frames[index] = ref_str;
}

// Checks whether a page is already present in the frames
int isPresent(char ref_str){
    for(int i = 0; i < SIZE; i++){
        if(frames[i] == ref_str){
            return 1;
        }
    }
    return 0;
}

// Prints the content of all the frames
void print_frames(){
    for(int i = 0; i < SIZE; i++){
        printf("%c\n", frames[i]);
    }
}

int main(){
    printf("\tOptimal Page Replacement\n");

    int i = 0, number_of_page_fault = 0;
    printf("Reference String: %s\n", ref_str);
    printf("Frame Size: %d\n\n", SIZE);
    printf("\tAlgorithm Starts\n");

    // Algorithm for Optimal Page Replacement

    /*For our example reference string, our three frames are initially empty. The
    first three references (7, 0, 1) cause page faults and are brought into these empty
    frames.*/
    for(int j = 0; j < SIZE; j++){
        printf("Reference: %c\n", ref_str[i]);
        i++;
        frames[j] = ref_str[j];
        number_of_page_fault += 1;
        printf("Page Frame:\n");
        print_frames();
        printf("Number of Page Faults: %d\n\n", number_of_page_fault);
    }
```

```
    // Page Replacement for other references
    while(ref_str[i] != '\0'){
        printf("Reference: %c\n", ref_str[i]);
        if(!isPresent(ref_str[i])){
            insert(i, ref_str[i]);
            // printf("Just for Ref: %d\n", i);
            number_of_page_fault += 1;
            printf("Page Frame:\n");
            print_frames();
            printf("Number of Page Faults: %d\n\n", number_of_page_fault);
        }
        else{
            printf("Reference %c is already in memory. So there is no fault for this 
reference.\n\n", ref_str[i]);
        }
        i++;
    }
    printf("Total Number of Page Faults: %d\n", number_of_page_fault);
    return 0;
}
```

2. FIFO Page Replacement Algorithm:

```c
#include <stdio.h>
#define SIZE 3          // Number of frames

char frames[SIZE];          // Frames
int front = -1;
int rear = -1;

// Inserts a page in the Frames
void enqueue(char ref_str){
    if(front == -1){
        front = 0;
        rear = 0;
        frames[rear] = ref_str;
    }
    else if(front == (rear + 1) % SIZE){
        printf("Queue is full!\n");
    }
    else if(rear == SIZE - 1){
        rear = 0;
        frames[rear] = ref_str;
    }
    else{
        rear += 1;
        frames[rear] = ref_str;
    }
}

// Removes a page from the frames
void pop(){
    if(front == -1){
        printf("Empty\n");
    }
    else if(front == rear){
        front = -1;
        rear = -1;
    }
    else if(front == SIZE - 1){
        front = 0;
    }
    else{
        front++;
    }
}

// Prints the content of all the Frames
void print_frames(){
    for(int i = 0; i < SIZE; i++){
        printf("%c\n", frames[i]);
    }
}
```

```c
// Checks whether a page is already present in the frame
int isPresent(char ref_str){
    for(int i = 0; i < SIZE; i++){
        if(frames[i] == ref_str){
            return 1;
        }
    }
    return 0;
}

int main(){
    printf("\tFIFO Page Replacement\n\n");

    char ref_str[] = "70120304230321201701";    // Reference String

    int i = 0, number_of_page_fault = 0;
    printf("Reference String: %s\n", ref_str);
    printf("Frame Size: %d\n\n", SIZE);
    printf("\tAlgorithm Starts\n");

    // Algorithm for FIFO Page Replacement
    /*For our example reference string, our three frames are initially empty. The
    first three references (7, 0, 1) cause page faults and are brought into these
empty
    frames.*/
    for(int j = 0; j < SIZE; j++){
        printf("Reference: %c\n", ref_str[i]);
        i++;
        enqueue(ref_str[j]);
        number_of_page_fault += 1;
        print_frames();
        printf("Number of Page Faults: %d\n\n", number_of_page_fault);
    }

    // Page Replacement for other references
    while(ref_str[i] != '\0'){
        printf("Reference: %c\n", ref_str[i]);
        if(!isPresent(ref_str[i])){
            pop();
            enqueue(ref_str[i]);
            number_of_page_fault += 1;
            printf("Page Frame:\n");
            print_frames();
            printf("Number of Page Faults: %d\n\n", number_of_page_fault);
        }
        else{
            printf("Reference %c is already in memory. So there is no fault for this
reference.\n\n", ref_str[i]);
        }
        i++;
    }
    printf("Total Number of Page Faults: %d\n", number_of_page_fault);
    return 0;
}
```

3. The Second Chance Page Replacement Algorithm:

```c
#include <stdio.h>
#define SIZE 3  // Number of frames

// Each page has an extra field known as reference bit which is checked before replacing
it
struct page{
        char reference;
        unsigned int reference_bits;
        unsigned int second_chance;
};

struct page frames[SIZE];   //Frames


// finds a location to insert new reference
int find_index(){
    int index = 0;
    for(int i = 0; i < SIZE; i++){
        if(frames[i].reference_bits == 0 && frames[i].second_chance == 0){
            return i;
        }
        else if(frames[i].reference_bits == 1){
            frames[i].reference_bits = 0;
            frames[i].second_chance = 1;
        }
    }
}

// inserts a new reference by replacing an appropriate page from the frame
void insert(char ref_str){

    int index = find_index();
    frames[index].reference = ref_str;
    frames[index].reference_bits = 0;
    frames[index].second_chance = 0;
}

// prints the content of the frames
void print_frames(){
    printf("Page Frame: \n");
    for(int i = 0; i < SIZE; i++){
        printf("%c\n", frames[i].reference);
    }
}

// checks whether a page is already in the memory
int isPresent(char ref_str){
    for(int i = 0; i < SIZE; i++){
        if(frames[i].reference == ref_str){
            frames[i].reference_bits = 1;
            return 1;
        }
    }
    return 0;
}
```

```
int main(){
    printf("\tSecond Chance Page Replacement\n");

    char ref_str[] = "70120304230321201701";
    printf("Reference String: %s\n", ref_str);
    printf("Frame Size: %d\n\n", SIZE);
    printf("\tAlgorithm Starts\n");
    // char ref_str[] =  "041424342404142434";
    int index_of_curr_ref = 0, number_of_page_fault = 0;
    for(int i = 0; i < SIZE; i++){
        index_of_curr_ref++;
        number_of_page_fault++;
        printf("\nReference: %c\n", ref_str[i]);
        frames[i].reference = ref_str[i];
        frames[i].reference_bits = 0;
        frames[i].second_chance = 0;
        print_frames();
        printf("Number of Page Fault(s): %d\n", number_of_page_fault);
    }

    while(ref_str[index_of_curr_ref] != '\0'){
        if(!isPresent(ref_str[index_of_curr_ref])){
            printf("\nReference: %c\n", ref_str[index_of_curr_ref]);
            insert(ref_str[index_of_curr_ref]);
            number_of_page_fault += 1;
            print_frames();
            printf("Number of Page Fault(s): %d\n", number_of_page_fault);
        }
        else{
            printf("Reference %c is already in memory. So there is no fault for this
reference.\n\n", ref_str[index_of_curr_ref]);
        }
        index_of_curr_ref++;
    }

    printf("Total Number of Page Faults: %d\n", number_of_page_fault);
    return 0;
}
```

4. The Clock Page Replacement Algorithm:

```
#include <stdio.h>
#define SIZE 3

struct page{
    char reference;
    int reference_bit;
};

struct page frames[SIZE];

int curr_page_frame = -1;

void insert(char ref_str){
    while(1){
        // checks the frame currently pointed by the clock hand
        // if its reference_bit is one it is set to 0 and
        // the hand points to the next frame
        if(frames[curr_page_frame].reference_bit == 1){
            frames[curr_page_frame].reference_bit = 0;
            curr_page_frame = (curr_page_frame + 1) % SIZE;
        }
        // else if the reference_bit is zero the new page is
        // inserted at this place and the hand advances
        else if(frames[curr_page_frame].reference_bit == 0){
            frames[curr_page_frame].reference = ref_str;
            frames[curr_page_frame].reference_bit = 0;
            curr_page_frame = (curr_page_frame + 1) % SIZE;
            break;
        }
    }

}

// check whether the page is already present in it or not
int isPresent(char ref_str){
    for(int i = 0; i < SIZE; i++){
        if(frames[i].reference == ref_str){
            frames[i].reference_bit = 1;
            return 1;
        }
    }
    return 0;
}
```

```c
// prints the content of the frames
void print_frames(){
    printf("Page Frame: \n");
    for(int i = 0; i < SIZE; i++){
        printf("%c %d\n", frames[i].reference, frames[i].reference_bit);
    }
}

int main(){
    printf("\tClock Page Replacement\n");
    char ref_str[] = "70120304230321201701";
    printf("Reference String: %s\n", ref_str);
    printf("Frame Size: %d\n\n", SIZE);
    printf("\tAlgorithm Starts\n");

    // char ref_str[] =  "0414243424041424 34";
    int index_of_curr_ref = 0, number_of_page_fault = 0;

    // for the initial pages when the frame is empty
    for(int i = 0; i < SIZE; i++){
        index_of_curr_ref++;
        number_of_page_fault++;
        printf("\nReference: %c\n", ref_str[i]);
        frames[i].reference = ref_str[i];
        frames[i].reference_bit = 0;
        curr_page_frame++;                              // the hand points to the oldest page
        print_frames();
        printf("Number of Page Fault(s): %d\n", number_of_page_fault);
    }

    // for the rest of the pages
    while(ref_str[index_of_curr_ref] != '\0'){

        // if the page is not present there is a page fault
        // and the new page is inserted at a appropriate place
        if(!isPresent(ref_str[index_of_curr_ref])){
            printf("\nReference: %c\n", ref_str[index_of_curr_ref]);
            insert(ref_str[index_of_curr_ref]);
            number_of_page_fault += 1;
            print_frames();
            printf("Number of Page Fault(s): %d\n", number_of_page_fault);
        }
        else{
            printf("Reference %c is already in memory. So there is no fault for this 
reference.\n\n", ref_str[index_of_curr_ref]);
        }
        index_of_curr_ref++;
    }

    printf("Total Number of Page Faults: %d\n", number_of_page_fault);
    return 0;
}
```

5. LRU (Least Recently Used) Page Replacement Algorithm:

```
#include <stdio.h>
#define SIZE 3  // Number of Frames

char frames[SIZE];  // Frame

// When a page must be replaced, LRU chooses the page that has not been used
// for the longest period of time so we need to look backward.

char ref_str[] = "70120304230321201701";

// A function which returns an index for the insertion of next reference string
// based on the fact: 'we can replace the page that has not been used for the longest
period of time'

int priority_index(int curr_index){
    int priority_array[SIZE] = {4, 4, 4};
    // printf("priority_array: %d %d %d\n", priority_array[0], priority_array[1],
priority_array[2]);
    // printf("curr_index: %d\n", curr_index);
    for(int k = 0; k < SIZE; k++){
        for(int j = curr_index; j >= 0; j--){
            if(ref_str[j] == frames[k]){
                if(priority_array[k] == 4){
                    priority_array[k] = curr_index - j;
                    break;
                }
            }
        }
    }

    // printf("priority_array: %d %d %d\n", priority_array[0], priority_array[1],
priority_array[2]);

    int index = 0;
    for(int i = 1; i < SIZE; i++){
        if(priority_array[i] >= priority_array[index]){
            index = i;
        }
    }
    return index;
}


// A function to insert the reference string at a particular index or
// to be more precise it replaces a reference to a page with the one
// that has not been used for the longest period of time

void insert(int curr_index, char ref_str){
    int index = priority_index(curr_index);
    // printf("Index to be inserted at: %d\n", index);
    frames[index] = ref_str;
}

// Checks whether a page is already present in the frames

int isPresent(char ref_str){
    for(int i = 0; i < SIZE; i++){
        if(frames[i] == ref_str){
```

```
            return 1;
        }
    }
    return 0;
}

// Prints the content of all the frames
void print_frames(){
    for(int i = 0; i < SIZE; i++){
        printf("%c\n", frames[i]);
    }
}

int main(){
    printf("\tLeast Recently Used Page Replacement\n");

    int i = 0, number_of_page_fault = 0;
    printf("Reference String: %s\n", ref_str);
    printf("Frame Size: %d\n\n", SIZE);
    printf("\tAlgorithm Starts\n");

    // Algorithm for Least Recently Used Page Replacement

    /*For our example reference string, our three frames are initially empty. The
    first three references (7, 0, 1) cause page faults and are brought into these empty
    frames.*/
    for(int j = 0; j < SIZE; j++){
        printf("Reference: %c\n", ref_str[i]);
        i++;
        frames[j] = ref_str[j];
        number_of_page_fault += 1;
        printf("Page Frame:\n");
        print_frames();
        printf("Number of Page Faults: %d\n\n", number_of_page_fault);
    }

    // Page Replacement for other references
    while(ref_str[i] != '\0'){
        printf("Reference: %c\n", ref_str[i]);
        if(!isPresent(ref_str[i])){
            insert(i, ref_str[i]);
            number_of_page_fault += 1;
            printf("Page Frame:\n");
            print_frames();
            printf("Number of Page Faults: %d\n\n", number_of_page_fault);
        }
        else{
            printf("Reference %c is already in memory. So there is no fault for this
reference.\n\n", ref_str[i]);
        }
        i++;
    }
    printf("Total Number of Page Faults: %d\n", number_of_page_fault);
    return 0;
}
```

6. NRU (Not Recently Used):

```c
// NOTE: Modified Bit is not changed in this program ever. In actual Algorithm it is supposed to be changed.
// So this program may not be a perfect implementation of the Algorithm.

#include <stdio.h>
#define SIZE 3

struct page{
    char reference;
    int referenced_bit;
    int modified_bit;
};

struct page frames[SIZE];

// returns an appropriate index to insert a new page
int find_index(){
    for(int i = 0; i < SIZE; i++){
        // Case 0 : not referenced, not modified
        if((frames[i].referenced_bit == 0 && frames[i].modified_bit == 0)){
            return i;
        }
        // Case 1 : not referenced, modified
        if((frames[i].referenced_bit == 0 && frames[i].modified_bit == 1)){
            return i;
        }
        // Case 2 : referenced, not modified
        if((frames[i].referenced_bit == 1 && frames[i].modified_bit == 0)){
            return i;
        }
        // Case 3 : referenced, modified
        if((frames[i].referenced_bit == 1 && frames[i].modified_bit == 1)){
            return i;
        }

    }
}

// inserts a new page in the frames
void insert(char ref_str){
    int i = find_index();
    frames[i].reference = ref_str;
    frames[i].referenced_bit = 0;
    frames[i].modified_bit = 0;
}

// prints the content of the frames
void print_frames(){
    printf("Page Frames: \n");
    for(int i = 0; i < SIZE; i++){
        printf("%c\n", frames[i]);
    }
}
```

```c
// check whether the page is already present in it or not
int isPresent(char ref_str){
    for(int i = 0; i < SIZE; i++){
        if(frames[i].reference == ref_str){
            frames[i].referenced_bit = 1;
            frames[i].modified_bit = 1;
            return 1;
        }
    }
    return 0;
}

int main(){
    printf("\tNot Recently Used Page Replacement\n");
    // Reference String
    char ref_str[] = "70120304230321201701";
    printf("Reference String: %s\n", ref_str);
    printf("Frame Size: %d\n\n", SIZE);
    printf("\tAlgorithm Starts\n");
    int index_of_curr_ref = 0, number_of_page_fault = 0;

    // for the initial pages when the frame is empty
    for(int i = 0; i < SIZE; i++){
        index_of_curr_ref++;
        number_of_page_fault++;
        printf("\nReference: %c\n", ref_str[i]);
        frames[i].reference = ref_str[i];
        frames[i].referenced_bit = 0;
        frames[i].modified_bit = 0;
        print_frames();
        printf("Number of Page Fault(s): %d\n", number_of_page_fault);
    }

    while(ref_str[index_of_curr_ref] != '\0'){
        if(!isPresent(ref_str[index_of_curr_ref])){
            printf("\nReference: %c\n", ref_str[index_of_curr_ref]);
            insert(ref_str[index_of_curr_ref]);
            number_of_page_fault += 1;
            print_frames();
            printf("Number of Page Fault(s): %d\n", number_of_page_fault);
        }
        else{
            printf("Reference %c is already in memory. So there is no fault for this 
reference.\n\n", ref_str[index_of_curr_ref]);
        }

        index_of_curr_ref++;
    }
    printf("Total Number of Page Faults: %d\n", number_of_page_fault);

    return 0;
}
```

Assignment 7

Implement following Disk Scheduling Algorithms

1. FCSC (First Come First Serve):

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 8

// Request array represents an array storing indexes
// of tracks that have been requested in ascending
// order of their time of arrival.
int request_array[SIZE];
// head_pos is the position of disk head
int head_pos;
// input_from_file function takes input
// for request_array and head_pos
void input_from_file();
// fcfs_disk_scheduling function implements the First Come
//  First Serve (FCFS) disk scheduling algorithm
// and calculates the total number of seek operations
// done to access all the requested tracks
void fcfs_disk_scheduling();

int main(){
    printf("\tFCFS Disk Scheduling\n");
    input_from_file();
    printf("Request array: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", request_array[i]);
    }
    printf("\nPosition of disk head: %d\n", head_pos);
    fcfs_disk_scheduling();
    return 0;
}
```

```c
//  this algorithm services requests in the order
//  they arrive in the disk queue
void fcfs_disk_scheduling(){
    // distance_from_head is the distance of the track from
    // the current head position
    // total_seek_count is the sum of all the seek operations
    int distance_from_head = 0, total_seek_count = 0;
    for(int i = 0; i < SIZE; i++){
        printf("\nServing Request Number: %d \nCurrent disk head: %d\nDisk Track
Number: %d\n", i+1, head_pos, request_array[i]);
        distance_from_head = abs(request_array[i] - head_pos);
        printf("Distance from the head: %d\n", distance_from_head);
        total_seek_count += distance_from_head;
        head_pos = request_array[i];
    }
    printf("\nTotal Seek Count: %d\n", total_seek_count);
}

void input_from_file(){
    /*
    format of the input file should be:
    1st line is the head position
    2nd line onwards is the content of request_array
    eg:
    50
    176
    79
    34
    60
    92
    11
    41
    114
    */
    FILE *fin;
    fin = fopen("input.txt", "r");
    if(fin == NULL){
        printf("Error in opening file!\n");
    }
    else{
        fscanf(fin, "%d", &head_pos);
        for(int i = 0; i < SIZE; i++){
            fscanf(fin, "%d", &request_array[i]);
        }
    }
}
```

2. SSTF (Shorted Seek Time First):

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define SIZE 8

// Request array represents an array storing indexes of tracks
// that have been requested in ascending order of their time
// of arrival.
int request_array[SIZE];
// head_pos is the position of disk head
int head_pos;
// input_from_file function takes input for request_array
// and head_pos
void input_from_file();
void sstf_disk_scheduling();

// find_appropriate_request function helps in finding the
// appropriate disk track number to be serviced
// as per the algorithm which states that
// the tracks which are closer to current disk head position
// should be serviced first in order to minimise the seek
// operations
int find_appropriate_request();

int main(){
    printf("\tSSTF Disk Scheduling\n");
    input_from_file();
    printf("Request array: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", request_array[i]);
    }
    printf("\nPosition of disk head: %d\n", head_pos);
    sstf_disk_scheduling();
    return 0;
}

void sstf_disk_scheduling(){
    int disk_sequence[SIZE];
    int chosen_request_index;
    int distance_from_head = 0, total_seek_count = 0;
    for(int i = 0; i < SIZE; i++){
        chosen_request_index = find_appropriate_request();
        disk_sequence[i] = request_array[chosen_request_index];
        printf("\nCurrent disk head: %d\nChosen Disk Track Number: %d\n", head_pos,
request_array[chosen_request_index]);
        distance_from_head = abs(request_array[chosen_request_index] - head_pos);
        printf("Distance from the head: %d\n", distance_from_head);
        total_seek_count += distance_from_head;
        head_pos = request_array[chosen_request_index];
        request_array[chosen_request_index] = -1;
    }
    printf("\nTotal Seek Count: %d\n", total_seek_count);
    printf("Seek Sequence: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", disk_sequence[i]);
    }
}
```

```c
int find_appropriate_request(){
    int min_distance = INT_MAX;
    int index = -1;
    for(int i = 0; i < SIZE; i++){
        if(request_array[i] == -1) continue;
        if(abs(request_array[i] - head_pos) < min_distance){
            index = i;
            min_distance = abs(request_array[i] - head_pos);
        }
    }
    return index;
}

void input_from_file(){
    /*
    format of the input file should be:
    1st line is the head position
    2nd line onwards is the content of request_array
    eg:
    50
    176
    79
    34
    60
    92
    11
    41
    114
    */
    FILE *fin;
    fin = fopen("input.txt", "r");
    if(fin == NULL){
        printf("Error in opening file!\n");
    }
    else{
        fscanf(fin, "%d", &head_pos);
        for(int i = 0; i < SIZE; i++){
            fscanf(fin, "%d", &request_array[i]);
        }
    }
}
```

3. SCAN:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define SIZE 8

// Request array represents an array storing indexes of tracks
// that have been requested in ascending order of their time
// of arrival.
int request_array[SIZE];
// head_pos is the position of disk head
int head_pos;
// inputFromFile function takes input for request_array
// and head_pos
void inputFromFile();
int inRequestArray(int head_ptr);
int maxOfRequestArray();
void deleteFromRequestArray(int head_ptr);
void scan_disk_scheduling();

int main(){
    printf("\tSCAN Disk Scheduling\n");
    inputFromFile();
    printf("Request array: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", request_array[i]);
    }
    printf("\nPosition of disk head: %d\n", head_pos);
    scan_disk_scheduling();
    return 0;
}


void scan_disk_scheduling(){
    int disk_sequence[SIZE];
    int j = 0;
    int total_seek_count = 0, distance_from_head = 0;
    int head_ptr = head_pos;
    int max_value = maxOfRequestArray();
    // As we have implemented this algorithm with an
    // assumption that the head is initially moving in the left
    // so it will go till zero but while moving to right
    // it would only go upto the max disk track number
    printf("\nMax value: %d\n", max_value);
    while(head_ptr >= 0){
        if(inRequestArray(head_ptr)){
            distance_from_head = abs(head_pos - head_ptr);
            printf("Head is moving from %d to %d\n", head_pos, head_ptr);
            disk_sequence[j++] = head_ptr;
            head_pos = head_ptr;
            // mark track number to
            // denote that it has been serviced
            // so while returning we do not serve it again
            deleteFromRequestArray(head_ptr);
            total_seek_count += distance_from_head;
        }
        head_ptr--;
    }
```

```c
    head_ptr = 0;
    printf("Head is moving from %d to %d\n", head_pos, head_ptr);
    total_seek_count += abs(head_pos - head_ptr);
    head_pos = 0;

    while(head_ptr <= max_value){
        if(inRequestArray(head_ptr)){
            distance_from_head = abs(head_pos - head_ptr);
            printf("Head is moving from %d to %d\n", head_pos, head_ptr);
            disk_sequence[j++] = head_ptr;
            head_pos = head_ptr;
            // mark track number to
            // denote that it has been serviced
            // so while returning we do not serve it again
            deleteFromRequestArray(head_ptr);
            total_seek_count += distance_from_head;
        }
        head_ptr++;
    }

    printf("\nTotal Seek Count: %d\n", total_seek_count);
    printf("\nSeek Sequence: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", disk_sequence[i]);
    }

}

//inRequestArray() function returns true if the
// value head_pos is in the request_array
int inRequestArray(int head_ptr){
    for(int i = 0; i < SIZE; i++){
        if(head_ptr == request_array[i])
            return 1;
    }
    return 0;
}
// maxof() function returns the max of all the
// track numbers in request_array
int maxOfRequestArray(){
    int max_value = -1;
    for(int i = 0; i < SIZE; i++){
        if(max_value < request_array[i]){
            max_value = request_array[i];
        }
    }
    return max_value;
}

// marks the track number to denote it has been serviced
void deleteFromRequestArray(int head_ptr){
    for(int i = 0; i < SIZE; i++){
        if(head_ptr == request_array[i]){
            request_array[i] = -1;
        }
    }
}
```

```c
void inputFromFile(){
    /*
    format of the input file should be:
    1st line is the head position
    2nd line onwards is the content of request_array
    eg:
    50
    176
    79
    34
    60
    92
    11
    41
    114
    */
    FILE *fin;
    fin = fopen("input.txt", "r");
    if(fin == NULL){
        printf("Error in opening file!\n");
    }
    else{
        fscanf(fin, "%d", &head_pos);
        for(int i = 0; i < SIZE; i++){
            fscanf(fin, "%d", &request_array[i]);
        }
    }
}
```

4. C-SCAN:

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define SIZE 8

// Request array represents an array storing indexes of tracks
// that have been requested in ascending order of their time
// of arrival.
int request_array[SIZE];
// head_pos is the position of disk head
int head_pos;
// inputFromFile function takes input for request_array
// and head_pos
void inputFromFile();
int inRequestArray(int head_ptr);
int maxOfRequestArray();
void deleteFromRequestArray(int head_ptr);
int isEmpty();
void cscan_disk_scheduling();

int main(){
    printf("\tC-SCAN Disk Scheduling\n");
    inputFromFile();
    printf("Request array: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", request_array[i]);
    }
    printf("\nPosition of disk head: %d\n", head_pos);
    cscan_disk_scheduling();
    return 0;
}


void cscan_disk_scheduling(){
    int disk_sequence[SIZE];
    int j = 0;
    int total_seek_count = 0, distance_from_head = 0;
    int head_ptr = head_pos;
    // Suppose the head goes till 199 in the right
    // serving the requests, once it reaches 199
    // then it reverse and come to 0 without
    // serving any of the requests and start moving
    // to the right or towards the end serving the requests again

    while(!isEmpty() && head_ptr <= 199){
        if(inRequestArray(head_ptr)){
            distance_from_head = abs(head_pos - head_ptr);
            printf("Head is moving from %d to %d\n", head_pos, head_ptr);
            disk_sequence[j++] = head_ptr;
            head_pos = head_ptr;
            // mark track number to
            // denote that it has been serviced
            // so while returning we do not serve it again
            deleteFromRequestArray(head_ptr);
            total_seek_count += distance_from_head;
        }
```

```c
        // when it reaches the end of the track number
        // it comes back to zero and starts again
        if(head_ptr == 199){
            printf("Head is moving from %d to %d\n", head_pos, head_ptr);
            total_seek_count += abs(head_pos - head_ptr);
            head_pos = head_ptr;
            head_ptr = 0;
            printf("Head is moving from %d to %d, but this time without serving any
requests!\n", head_pos, head_ptr);
            head_pos = 0;
            total_seek_count += 199;
        }
        head_ptr++;
    }

    printf("\nTotal Seek Count: %d\n", total_seek_count);
    printf("\nSeek Sequence: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", disk_sequence[i]);
    }

}

int isEmpty(){
    for(int i = 0; i < SIZE; i++){
        if(request_array[i] != -1){
            return 0;
        }
    }
    return 1;
}

//inRequestArray() function returns true if the
// value head_pos is in the request_array
int inRequestArray(int head_ptr){
    for(int i = 0; i < SIZE; i++){
        if(head_ptr == request_array[i])
            return 1;
    }
    return 0;
}


// marks the track number to denote it has been serviced
void deleteFromRequestArray(int head_ptr){
    for(int i = 0; i < SIZE; i++){
        if(head_ptr == request_array[i]){
            request_array[i] = -1;
        }
    }
}
```

```c
void inputFromFile(){
    /*
    format of the input file should be:
    1st line is the head position
    2nd line onwards is the content of request_array
    eg:
    50
    176
    79
    34
    60
    92
    11
    41
    114
    */
    FILE *fin;
    fin = fopen("input.txt", "r");
    if(fin == NULL){
        printf("Error in opening file!\n");
    }
    else{
        fscanf(fin, "%d", &head_pos);
        for(int i = 0; i < SIZE; i++){
            fscanf(fin, "%d", &request_array[i]);
        }
    }
}
```

5. LOOK (Elevator):

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define SIZE 8

// Request array represents an array storing indexes of tracks
// that have been requested in ascending order of their time
// of arrival.
int request_array[SIZE];
// head_pos is the position of disk head
int head_pos;
// inputFromFile function takes input for request_array
// and head_pos
void inputFromFile();
int inRequestArray(int head_ptr);
int maxOfRequestArray();
void deleteFromRequestArray(int head_ptr);
int isEmptyForward(int head, int direction);
void look_disk_scheduling();

int main(){
    printf("\tLOOK Disk Scheduling\n");
    inputFromFile();
    printf("Request array: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", request_array[i]);
    }
    printf("\nPosition of disk head: %d\n\n", head_pos);
    look_disk_scheduling();
    return 0;
}


void look_disk_scheduling(){
    int disk_sequence[SIZE];
    int j = 0;
    int total_seek_count = 0, distance_from_head = 0;
    int head_ptr = head_pos;
    int right = 0, left = 1;

    while(head_ptr <= 199){
        if(inRequestArray(head_ptr)){
            distance_from_head = abs(head_pos - head_ptr);
            printf("Head is moving from %d to %d\n", head_pos, head_ptr);
            disk_sequence[j++] = head_ptr;
            head_pos = head_ptr;
            // mark track number to
            // denote that it has been serviced
            // so while returning we do not serve it again
            deleteFromRequestArray(head_ptr);
            total_seek_count += distance_from_head;
        }
        // LOOK condition
        if(isEmptyForward(head_pos, right)){
            break;
        }
        head_ptr++;
    }
```

```c
    while(head_ptr >= 0){

        if(inRequestArray(head_ptr)){
            distance_from_head = abs(head_pos - head_ptr);
            printf("Head is moving from %d to %d\n", head_pos, head_ptr);
            disk_sequence[j++] = head_ptr;
            head_pos = head_ptr;
            // mark track number to
            // denote that it has been serviced
            // so while returning we do not serve it again
            deleteFromRequestArray(head_ptr);
            total_seek_count += distance_from_head;
        }
        // LOOK condition
        if(isEmptyForward(head_pos, left)){
            break;
        }
        head_ptr--;
    }

    printf("\nTotal Seek Count: %d\n", total_seek_count);
    printf("\nSeek Sequence: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", disk_sequence[i]);
    }

}

int isEmptyForward(int head, int direction){
    if (direction == 0){
        for(int i = 0; i < SIZE; i++){
            if(request_array[i] > head){
                return 0;
            }
        }
    }
    else{
        for(int i = 0; i < SIZE; i++){
            if(request_array[i] < head){
                return 0;
            }
        }
    }

    return 1;
}

//inRequestArray() function returns true if the
// value head_pos is in the request_array
int inRequestArray(int head_ptr){
    for(int i = 0; i < SIZE; i++){
        if(head_ptr == request_array[i])
            return 1;
    }
    return 0;
}
```

```c
// marks the track number to denote it has been serviced
void deleteFromRequestArray(int head_ptr){
    for(int i = 0; i < SIZE; i++){
        if(head_ptr == request_array[i]){
            request_array[i] = -1;
        }
    }
}

void inputFromFile(){
    /*
    format of the input file should be:
    1st line is the head position
    2nd line onwards is the content of request_array
    eg:
    50
    176
    79
    34
    60
    92
    11
    41
    114
    */
    FILE *fin;
    fin = fopen("input.txt", "r");
    if(fin == NULL){
        printf("Error in opening file!\n");
    }
    else{
        fscanf(fin, "%d", &head_pos);
        for(int i = 0; i < SIZE; i++){
            fscanf(fin, "%d", &request_array[i]);
        }
    }
}
```

6. C-LOOK:

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define SIZE 8

// Request array represents an array storing indexes of tracks
// that have been requested in ascending order of their time
// of arrival.
int request_array[SIZE];
// head_pos is the position of disk head
int head_pos;
// inputFromFile function takes input for request_array
// and head_pos
void inputFromFile();
int inRequestArray(int head_ptr);
int minOfRequestArray();
void deleteFromRequestArray(int head_ptr);
int isEmptyForward(int head);
void clook_disk_scheduling();
int isEmpty();

int main(){
    printf("\tC-LOOK Disk Scheduling\n");
    inputFromFile();
    printf("Request array: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", request_array[i]);
    }
    printf("\nPosition of disk head: %d\n\n", head_pos);
    clook_disk_scheduling();
    return 0;
}


void clook_disk_scheduling(){
    int disk_sequence[SIZE];
    int j = 0;
    int total_seek_count = 0, distance_from_head = 0;
    int head_ptr = head_pos;
    int right = 0, left = 1;

    while(!isEmpty() && head_ptr <= 199){
        if(inRequestArray(head_ptr)){
            distance_from_head = abs(head_pos - head_ptr);
            printf("Head is moving from %d to %d\n", head_pos, head_ptr);
            disk_sequence[j++] = head_ptr;
            head_pos = head_ptr;
            // mark track number to
            // denote that it has been serviced
            // so while returning we do not serve it again
            deleteFromRequestArray(head_ptr);
            total_seek_count += distance_from_head;
        }
        // LOOK condition

        if(!isEmpty() && isEmptyForward(head_pos)){
            head_ptr = minOfRequestArray();
```

```c
            printf("Head is moving from %d to %d without serving any requests\n",
head_pos, head_ptr);
            head_pos = head_ptr;
        }
        else
            head_ptr++;
    }


    printf("\nTotal Seek Count: %d\n", total_seek_count);
    printf("\nSeek Sequence: ");
    for(int i = 0; i < SIZE; i++){
        printf("%d ", disk_sequence[i]);
    }

}

int minOfRequestArray(){
    int min_value = 200;
    for(int i = 0; i < SIZE; i++){
        if(request_array[i] != -1 && min_value > request_array[i]){
            min_value = request_array[i];
        }
    }
    return min_value;
}

int isEmptyForward(int head){

    for(int i = 0; i < SIZE; i++){
        if(request_array[i] > head){
            return 0;
        }
    }

    return 1;
}

//inRequestArray() function returns true if the
// value head_pos is in the request_array
int inRequestArray(int head_ptr){
    for(int i = 0; i < SIZE; i++){
        if(head_ptr == request_array[i])
            return 1;
    }
    return 0;
}


// marks the track number to denote it has been serviced
void deleteFromRequestArray(int head_ptr){
    for(int i = 0; i < SIZE; i++){
        if(head_ptr == request_array[i]){
            request_array[i] = -1;
        }
    }
}
```

```c
int isEmpty(){
    for(int i = 0; i < SIZE; i++){
        if(request_array[i] != -1){
            return 0;
        }
    }
    return 1;
}

void inputFromFile(){
    /*
    format of the input file should be:
    1st line is the head position
    2nd line onwards is the content of request_array
    eg:
    50
    176
    79
    34
    60
    92
    11
    41
    114
    */
    FILE *fin;
    fin = fopen("input.txt", "r");
    if(fin == NULL){
        printf("Error in opening file!\n");
    }
    else{
        fscanf(fin, "%d", &head_pos);
        for(int i = 0; i < SIZE; i++){
            fscanf(fin, "%d", &request_array[i]);
        }
    }
}
```