OS LAB QUESTIONS

1. **Write a shell script to sort an array of numbers using any sort method.**

```
#!/bin/bash
echo "enter maximum number"
read n
# taking input from user
echo "enter Numbers in array:"
for (( i = 0; i < $n; i++ ))
do
read nos[$i]
done
#printing the number before sorting
echo "Numbers in an array are:"
for (( i = 0; i < $n; i++ ))
do
echo ${nos[$i]}
done
# Now do the Sorting of numbers
for (( i = 0; i < $n ; i++ ))
do
for (( j = $i; j < $n; j++ ))
do
if [ ${nos[$i]} -gt ${nos[$j]}  ]; then
t=${nos[$i]}
nos[$i]=${nos[$j]}
nos[$j]=$t
fi
done
done
# Printing the sorted number
echo -e "\nSorted Numbers "
for (( i=0; i < $n; i++ ))
do
echo ${nos[$i]}

done
```

2. **Write down Linux commands for following statements:**

i. Redirect the output of ls command to a file named outfile. Use this as an input file for remaining commands.

Command:

ls > outfile

wc -l < outfile
cat outfile

This will write the output of ls to a file called outfile in the current working directory.

You can then use the outfile file as an input file for remaining commands. For example, if you want to count the number of lines in outfile, you can use the following command:

ii. To select the lines in a file which has digit as one of the character in that line and redirect the output to the file named list.

You can use the grep command to select the lines in a file that have a digit as one of the characters. You can then redirect the output to a file named list using the > operator. Here is the command:

Command:
grep '[0-9]' <input_file> > list
cat list

Replace <input_file> with the name of the file that you want to search for lines with digits.

In this command, the regular expression [0-9] matches any line that contains a digit from 0 to 9. The < operator is used to redirect the input from the file to the grep command. The > operator is used to redirect the output of the grep command to the list file.

After running this command, the list file will contain all the lines from the input file that have at least one digit.

iii. Assign execute permission to owner and remove read permission from other for an ordinary file named test by relative way.

You can use the chmod command to assign execute permission to the owner and remove read permission from others for an ordinary file named test. Here is the command:

Command:chmod u+x,o-r <path_to_file>/test
(pwd to check path to file)
ls -la

Replace <path_to_file> with the path to the directory containing the test file. For example, if the file is in the current working directory, you can replace <path_to_file> with . (a dot, representing the current directory).

In this command, u+x assigns execute permission to the owner of the file, and o-r removes read permission from others. The + and - symbols are used to add or remove permissions, respectively. The u and o are short for "user" and "others", respectively.

After running this command, the owner of the test file will have execute permission, and others will not have read permission.

iv. Create an alias named rm that always deletes file interactively.

You can create an alias named rm that always deletes files interactively by adding the following line to your shell configuration file:

Command:
alias rm='rm -i'
rm
(-i is used for confirmation)

This will add an alias to the rm command that always prompts for confirmation before deleting files.

v. Count the currently login users to the system.
Command: who | wc -l

3. **Write down Linux commands for following statements:**
   i. Redirect the output of cat command to a file named outfile. Use this as an input file for remaining commands.

i. To redirect the output of cat command to a file named outfile and use it as an input file for remaining commands, use the following command:

Command : ***cat inputfile.txt > outfile***

This will output the contents of inputfile.txt to outfile, replacing any previous content in outfile. You can then use the outfile as the input file for remaining commands.

ii. List all hidden files under current directory and store in “hidden” file

ii. To list all hidden files under the current directory and store in a file named "hidden", use the following command:

Command : *ls -a | grep "^\." > hidden*

This command first lists all files in the current directory, including hidden files, using the ls command with the -a option. The output is then piped to the grep command, which searches for lines that begin with a period (which indicates a hidden file in Linux). The output of grep is then redirected to a file named "hidden".

iii. Assign write permission to owner and remove execute permission from other for an ordinary file named test by absolute way.

iii. To assign write permission to the owner and remove execute permission from others for an ordinary file named "test" using the absolute method, use the following command:

Commands : *chmod 640 /path/to/test*

This command sets the file permissions to 640, which means that the owner has read and write permissions, the group has read permission, and others have no permissions. This removes execute permission from others, while still allowing the owner to execute the file if it is executable.

iv. To create soft and hard link for given file

iv. To create a soft and hard link for a given file, use the following commands:
For Soft Link:
Command: *ln -s /path/to/original_file /path/to/soft_link*

For a hard link:
Command : *ln /path/to/original_file /path/to/hard_link*

In both cases, replace "/path/to/original_file" with the path to the original file, and "/path/to/link" with the path to the link you want to create.

v. To convert lowercase to upper case of a given file

To convert lowercase to uppercase of a given file, use the following command:

Command : *tr '[:lower:]' '[:upper:]' < inputfile.txt > outfile*

This command uses the tr command to convert all lowercase characters in inputfile.txt to uppercase, and then redirects the output to a file named "outfile".

vi. To extract 1st and 10th character of a given file

vi. To extract the 1st and 10th character of a given file, use the following command:

Command : ***cut -c 1,10 inputfile.txt***

This command uses the cut command to extract the 1st and 10th character of each line in inputfile.txt. The "-c" option tells cut to extract characters, and "1,10" specifies the character positions to extract.

vii. To display how many times lines are repeated in a given file

To display how many times lines are repeated in a given file, you can use the uniq command with the sort and uniq options.

command:sort <file_name> | uniq -c

Replace <file_name> with the name of the file you want to analyze.

In this command, the sort command is used to sort the contents of the file alphabetically. Then, the output is piped to the uniq command with the -c option, which counts the number of occurrences of each unique line. The output will display the count followed by the unique line, like this:

4. **Write a program to solve reader-writer problem using Mutex**

The reader-writer problem is a classic synchronization problem in computer science where multiple threads (or processes) access a shared resource. In this problem, readers can read the shared resource concurrently, but writers must have exclusive access to it to write. Here is a C program that uses a mutex to solve the reader-writer problem:

Code :

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

/*
This program provides a possible solution for first readers writers problem using mutex and semaphore.
I have used 10 readers and 5 producers to demonstrate the solution. You can always play with these values.
*/

sem_t wrt;
pthread_mutex_t mutex;
int cnt = 1;
int numreader = 0;

void *writer(void *wno)
{
   sem_wait(&wrt);
   cnt = cnt*2;
   printf("Writer %d modified cnt to %d\n",(*((int *)wno)),cnt);
   sem_post(&wrt);

}
void *reader(void *rno)
{
   // Reader acquire the lock before modifying numreader
   pthread_mutex_lock(&mutex);
   numreader++;
   if(numreader == 1) {
      sem_wait(&wrt); // If this id the first reader, then it will block the writer
   }
   pthread_mutex_unlock(&mutex);
   // Reading Section
   printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);
```

```
    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader--;
    if(numreader == 0) {
        sem_post(&wrt); // If this is the last reader, it will wake up the writer.
    }
    pthread_mutex_unlock(&mutex);
}

int main()
{

    pthread_t read[10],write[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt,0,1);

    int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering the producer and consumer

    for(int i = 0; i < 10; i++) {
        pthread_create(&read[i], NULL, (void *)reader, (void *)&a[i]);
    }
    for(int i = 0; i < 5; i++) {
        pthread_create(&write[i], NULL, (void *)writer, (void *)&a[i]);
    }

    for(int i = 0; i < 10; i++) {
        pthread_join(read[i], NULL);
    }
    for(int i = 0; i < 5; i++) {
        pthread_join(write[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;

}
```

The above code is a solution to the first readers-writers problem using mutex and semaphore in the C programming language.

The program uses 10 reader threads and 5 writer threads to demonstrate the solution, which can be changed by modifying the values in the main function.

The program starts by including the necessary libraries: pthread.h for threads, semaphores.h for semaphores, and stdio.h for standard input and output.

The global variables used in the program are:

- wrt: a semaphore to ensure that only one writer accesses the shared resource at a time.
- mutex: a mutex to protect the numreader variable.
- cnt: a counter variable that is shared between the reader and writer threads.
- numreader: a variable that keeps track of the number of readers accessing the shared resource.

The program then defines two functions, writer and reader, that will be called by the respective threads.

5. **Write a program to solve producer-consumer problem using semaphore**

Code :

```
// C program for the above approach

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

// Initialize a mutex to 1
int mutex = 1;

// Number of full slots as 0
int full = 0;

// Number of empty slots as size
// of buffer
int empty = 10, x = 0;

// Function to produce an item and
// add it to the buffer
void producer()
{
    // Decrease mutex value by 1
    --mutex;

    // Increase the number of full
    // slots by 1
    ++full;
```

```
    // Decrease the number of empty
    // slots by 1
    --empty;

    // Item produced
    x++;
    printf("\nProducer produces"
           "item %d",
           x);

    // Increase mutex value by 1
    ++mutex;
}

// Function to consume an item and
// remove it from buffer
void consumer()
{
    // Decrease mutex value by 1
    --mutex;

    // Decrease the number of full
    // slots by 1
    --full;

    // Increase the number of empty
    // slots by 1
    ++empty;
    printf("\nConsumer consumes "
           "item %d",
           x);
    x--;

    // Increase mutex value by 1
    ++mutex;
}

// Driver Code
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
```

```
            "\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'
// can  give wrong value due to
// synchronization issues.

// 'critical' specifies that code is
// executed by only one thread at a
// time i.e., only one thread enters
// the critical section at a given time
#pragma omp critical

    for (i = 1; i > 0; i++) {

        printf("\nEnter your choice:");
        scanf("%d", &n);

        // Switch Cases
        switch (n) {
        case 1:

            // If mutex is 1 and empty
            // is non-zero, then it is
            // possible to produce
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }

            // Otherwise, print buffer
            // is full
            else {
                printf("Buffer is full!");
            }
            break;

        case 2:

            // If mutex is 1 and full
            // is non-zero, then it is
            // possible to consume
            if ((mutex == 1)
                && (full != 0)) {
                consumer();
```

```
            }

            // Otherwise, print Buffer
            // is empty
            else {
                printf("Buffer is empty!");
            }
            break;

        // Exit Condition
        case 3:
            exit(0);
            break;
        }
    }
}
```

Explanation :

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

6. **Write a program to solve dining philosopher problem using semaphore**

```
import java.util.concurrent.Semaphore;
import java.util.concurrent.ThreadLocalRandom;

public class dining_philosopher {

    static int philosopher = 5;
    static Philosopher philosophers[] = new Philosopher[philosopher];
    static Chopstick chopsticks[] = new Chopstick[philosopher];

    static class Chopstick {

        public Semaphore mutex = new Semaphore(1);

        void grab() {
            try {

                mutex.acquire();
```

```
            } catch (Exception e) {
                e.printStackTrace(System.out);
            }
        }

        void release() {

            mutex.release();
        }

        boolean isFree() {

            return mutex.availablePermits() > 0;
        }
    }

    static class Philosopher extends Thread {
        public int number;

        public Chopstick leftchopstick;

        public Chopstick rightchopstick;

        Philosopher(int num, Chopstick left, Chopstick right) {
            number = num;
            leftchopstick = left;
            rightchopstick = right;
        }

        public void run() {
            while (true) {

                leftchopstick.grab();
                System.out.println("Philosopher " + (number + 1) + " grabs left chopstick.");
                rightchopstick.grab();
                System.out.println("Philosopher " + (number + 1) + " grabs right chopstick.");

                eat();

                leftchopstick.release();
                System.out.println("Philosopher " + (number + 1) + " releases left chopstick.");
                rightchopstick.release();
                System.out.println("Philosopher " + (number + 1) + " releases right
chopstick.");
```

```
            }
        }

        void eat() {
            try {

                int sleepTime = ThreadLocalRandom.current().nextInt(0, 1000);
                System.out.println("Philosopher " + (number + 1) + " eats for " + sleepTime +
"ms");
                Thread.sleep(sleepTime);
            } catch (Exception e) {
                e.printStackTrace(System.out);
            }
        }
    }

    public static void main(String args[]) {

        for (int i = 0; i < philosopher; i++) {
            chopsticks[i] = new Chopstick();
        }

        for (int i = 0; i < philosopher; i++) {
            philosophers[i] = new Philosopher(i, chopsticks[i], chopsticks[(i + 1) %
philosopher]);

            philosophers[i].start();
        }
        while (true) {
            try {

                Thread.sleep(1000);

                boolean deadlock = true;

                for (Chopstick cs : chopsticks) {

                    if (cs.isFree()) {
                        deadlock = false;
                        break;
                    }
                }

                if (deadlock) {
```

```
                Thread.sleep(1000);
                System.out.println("Everyone Eats");
                break;
            }
        } catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }
    System.out.println("Exit The Program!");
    System.exit(0);
  }
}
```

Explanation:

This code implements a solution to the dining philosophers problem using threads and semaphores. The dining philosophers problem is a classic synchronization problem that involves five philosophers sitting at a table with a bowl of rice and chopsticks. Each philosopher alternates between thinking and eating rice, but they can only eat rice if they have two chopsticks. The problem is to avoid deadlock and starvation.

The code initializes five philosophers and five chopsticks as threads. Each philosopher is represented as a thread that tries to grab two chopsticks to eat. Each chopstick is represented as a semaphore with an initial value of 1. When a philosopher wants to eat, they try to acquire the two chopsticks, and if they can't acquire them, they wait until they can. Once they have the chopsticks, they eat for a random amount of time and then release the chopsticks.

The code also checks for deadlock by looping through all the chopsticks and checking if they are free. If all the chopsticks are in use, there is a potential for deadlock. If deadlock is detected, the program waits for one second and then exits.

7. **Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms:**
   A. First come First serve

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

typedef struct {
    int process_id;
    int arrival_time;
    int burst_time;
    int finish_time;
    int turnaround_time;
```

```
    int waiting_time;
} Process;

void compute_metrics(Process* processes, int num_processes) {
    int total_time = 0;
    for (int i = 0; i < num_processes; i++) {
        if (total_time < processes[i].arrival_time) {
            total_time = processes[i].arrival_time;
        }
        total_time += processes[i].burst_time;
        processes[i].finish_time = total_time;
        processes[i].turnaround_time = processes[i].finish_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }
}

void print_gantt_chart(Process* processes, int num_processes) {
    int total_time = processes[num_processes - 1].finish_time;
    int* remaining_time = (int*)malloc(num_processes * sizeof(int));
    for (int i = 0; i < num_processes; i++) {
        remaining_time[i] = processes[i].burst_time;
    }
    printf("\nGantt Chart:\n");
    printf("------------\n");
    printf("|");
    int current_time = 0;
    while (current_time < total_time) {
        int next_process_id = -1;
        for (int i = 0; i < num_processes; i++) {
            if (remaining_time[i] > 0 && processes[i].arrival_time <= current_time) {
                if (next_process_id == -1 || remaining_time[i] <
remaining_time[next_process_id]) {
                    next_process_id = i;
                }
            }
        }
        if (next_process_id == -1) {
            current_time++;
            printf(" ");
        } else {
            printf("P%d", processes[next_process_id].process_id);
            remaining_time[next_process_id]--;
            current_time++;
        }
```

```
        printf("|");
    }
    printf("\n");
    free(remaining_time);
}

int main() {
    int num_processes;
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    Process* processes = (Process*)malloc(num_processes * sizeof(Process));
    printf("\nEnter the arrival time and burst time for each process:\n");
    for (int i = 0; i < num_processes; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].process_id = i + 1;
    }
    compute_metrics(processes, num_processes);
    printf("\nProcess  Arrival time   Burst time   Finish time   Turnaround time  Waiting
time\n");
    for (int i = 0; i < num_processes; i++) {
        printf("%-9d%-15d%-13d%-15d%-18d%d\n", processes[i].process_id,
processes[i].arrival_time, processes[i].burst_time, processes[i].finish_time,
processes[i].turnaround_time, processes[i].waiting_time);
    }
    print_gantt_chart(processes, num_processes);
    free(processes);
    return 0;
}
```

B. Shortest Job First (Non Preemptive)

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

struct Process {
    int pid; // process id
    int arrival_time; // arrival time
    int burst_time; // burst time
```

```
    int finish_time; // finish time
    int turnaround_time; // turnaround time
    int waiting_time; // waiting time
};

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

void sort_processes_by_burst_time(struct Process* processes, int num_processes) {
    for (int i = 0; i < num_processes - 1; i++) {
        for (int j = i + 1; j < num_processes; j++) {
            if (processes[i].burst_time > processes[j].burst_time) {
                swap(&processes[i], &processes[j]);
            }
        }
    }
}

void compute_metrics(struct Process* processes, int num_processes) {
    processes[0].finish_time = processes[0].burst_time + processes[0].arrival_time;
    processes[0].turnaround_time = processes[0].finish_time - processes[0].arrival_time;
    processes[0].waiting_time = 0;

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

void print_gantt_chart(struct Process* processes, int num_processes) {
    printf("\nGantt Chart:\n");
    printf("----------\n");
    for (int i = 0; i < num_processes; i++) {
        printf("|  P%d  ", processes[i].pid);
    }
    printf("|\n");
    printf("0");
    for (int i = 0; i < num_processes; i++) {
        printf("      %d", processes[i].finish_time);
    }
```

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

int main() {
    int num_processes;
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);

    struct Process* processes = (struct Process*) malloc(num_processes * sizeof(struct
Process));

    printf("\nEnter the arrival time and burst time for each process:\n");
    for (int i = 0; i < num_processes; i++) {
        processes[i].pid = i + 1;
        printf("Process %d:\n", processes[i].pid);
        printf("Arrival time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst time: ");
        scanf("%d", &processes[i].burst_time);
    }

    sort_processes_by_burst_time(processes, num_processes);

    compute_metrics(processes, num_processes);

    printf("\nProcess\t Arrival time\t Burst time\t Finish time\t Turnaround time\t Waiting
time\n");
    for (int i = 0; i < num_processes; i++) {
        printf("%d\t %d\t\t %d\t\t %d\t\t %d\t\t\t %d\n",
               processes[i].pid, processes[i].arrival_time, processes[i].burst_time,
processes[i].finish_time,
               processes[i].turnaround_time, processes[i.waiting_time);
}
print_gantt_chart(processes, num_processes);

free(processes);

return 0;
}
```

8. **Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms:**

A. First come First serve

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

typedef struct {
    int process_id;
    int arrival_time;
    int burst_time;
    int finish_time;
    int turnaround_time;
    int waiting_time;
} Process;

void compute_metrics(Process* processes, int num_processes) {
    int total_time = 0;
    for (int i = 0; i < num_processes; i++) {
        if (total_time < processes[i].arrival_time) {
            total_time = processes[i].arrival_time;
        }
        total_time += processes[i].burst_time;
        processes[i].finish_time = total_time;
        processes[i].turnaround_time = processes[i].finish_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }
}

void print_gantt_chart(Process* processes, int num_processes) {
    int total_time = processes[num_processes - 1].finish_time;
    int* remaining_time = (int*)malloc(num_processes * sizeof(int));
    for (int i = 0; i < num_processes; i++) {
        remaining_time[i] = processes[i].burst_time;
    }
    printf("\nGantt Chart:\n");
    printf("-----------\n");
    printf("|");
    int current_time = 0;
    while (current_time < total_time) {
        int next_process_id = -1;
        for (int i = 0; i < num_processes; i++) {
            if (remaining_time[i] > 0 && processes[i].arrival_time <= current_time) {
                if (next_process_id == -1 || remaining_time[i] <
remaining_time[next_process_id]) {
```

```
                next_process_id = i;
            }
        }
    }
    if (next_process_id == -1) {
        current_time++;
        printf(" ");
    } else {
        printf("P%d", processes[next_process_id].process_id);
        remaining_time[next_process_id]--;
        current_time++;
    }
    printf("|");
  }
  printf("\n");
  free(remaining_time);
}

int main() {
  int num_processes;
  printf("Enter the number of processes: ");
  scanf("%d", &num_processes);
  Process* processes = (Process*)malloc(num_processes * sizeof(Process));
  printf("\nEnter the arrival time and burst time for each process:\n");
  for (int i = 0; i < num_processes; i++) {
    printf("Process %d:\n", i + 1);
    printf("Arrival time: ");
    scanf("%d", &processes[i].arrival_time);
    printf("Burst time: ");
    scanf("%d", &processes[i].burst_time);
    processes[i].process_id = i + 1;
  }
  compute_metrics(processes, num_processes);
  printf("\nProcess  Arrival time   Burst time   Finish time   Turnaround time  Waiting
time\n");
  for (int i = 0; i < num_processes; i++) {
    printf("%-9d%-15d%-13d%-15d%-18d%d\n", processes[i].process_id,
processes[i].arrival_time, processes[i].burst_time, processes[i].finish_time,
processes[i].turnaround_time, processes[i].waiting_time);
  }
  print_gantt_chart(processes, num_processes);
  free(processes);
  return 0;
}
```

B. Round- Robin

```
#include <stdio.h>

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
};

int main() {
    int n, tq;
    float avg_waiting_time = 0, avg_turnaround_time = 0;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter the arrival time and burst time for process %d: ", p[i].pid);
        scanf("%d %d", &p[i].arrival_time, &p[i].burst_time);
        p[i].remaining_time = p[i].burst_time;
    }
    printf("Enter the time quantum: ");
    scanf("%d", &tq);

    int completed = 0, time = 0, i = 0, j = 0;
    int gantt[n * (p[0].burst_time / tq + 1)];
    while (completed < n) {
        if (p[i].remaining_time <= tq && p[i].remaining_time > 0) {
            time += p[i].remaining_time;
            p[i].remaining_time = 0;
        } else if (p[i].remaining_time > 0) {
            time += tq;
            p[i].remaining_time -= tq;
        }
        if (p[i].remaining_time == 0) {
            completed++;
            p[i].turnaround_time = time - p[i].arrival_time;
            p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
```

```
            avg_turnaround_time += p[i].turnaround_time;
            avg_waiting_time += p[i].waiting_time;
        }
        gantt[j++] = p[i].pid;
        i = (i + 1) % n;
    }
    printf("\nGantt Chart:\n");
    printf("0 ");
    for (int k = 0; k < j; k++) {
        printf("P%d %d ", gantt[k], time);
    }
    printf("\n\n");

    avg_turnaround_time /= n;
    avg_waiting_time /= n;
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
    printf("Average Waiting Time: %.2f\n", avg_waiting_time);
    return 0;
}
```

9. **Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms:**

A. First come First serve

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

typedef struct {
    int process_id;
    int arrival_time;
    int burst_time;
    int finish_time;
    int turnaround_time;
    int waiting_time;
} Process;

void compute_metrics(Process* processes, int num_processes) {
    int total_time = 0;
    for (int i = 0; i < num_processes; i++) {
        if (total_time < processes[i].arrival_time) {
            total_time = processes[i].arrival_time;
        }
        total_time += processes[i].burst_time;
```

```
        processes[i].finish_time = total_time;
        processes[i].turnaround_time = processes[i].finish_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }
}

void print_gantt_chart(Process* processes, int num_processes) {
    int total_time = processes[num_processes - 1].finish_time;
    int* remaining_time = (int*)malloc(num_processes * sizeof(int));
    for (int i = 0; i < num_processes; i++) {
        remaining_time[i] = processes[i].burst_time;
    }
    printf("\nGantt Chart:\n");
    printf("------------\n");
    printf("|");
    int current_time = 0;
    while (current_time < total_time) {
        int next_process_id = -1;
        for (int i = 0; i < num_processes; i++) {
            if (remaining_time[i] > 0 && processes[i].arrival_time <= current_time) {
                if (next_process_id == -1 || remaining_time[i] <
remaining_time[next_process_id]) {
                    next_process_id = i;
                }
            }
        }
        if (next_process_id == -1) {
            current_time++;
            printf(" ");
        } else {
            printf("P%d", processes[next_process_id].process_id);
            remaining_time[next_process_id]--;
            current_time++;
        }
        printf("|");
    }
    printf("\n");
    free(remaining_time);
}

int main() {
    int num_processes;
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
```

```
    Process* processes = (Process*)malloc(num_processes * sizeof(Process));
    printf("\nEnter the arrival time and burst time for each process:\n");
    for (int i = 0; i < num_processes; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].process_id = i + 1;
    }
    compute_metrics(processes, num_processes);
    printf("\nProcess  Arrival time   Burst time   Finish time   Turnaround time  Waiting
time\n");
    for (int i = 0; i < num_processes; i++) {
        printf("%-9d%-15d%-13d%-15d%-18d%d\n", processes[i].process_id,
processes[i].arrival_time, processes[i].burst_time, processes[i].finish_time,
processes[i].turnaround_time, processes[i].waiting_time);
    }
    print_gantt_chart(processes, num_processes);
    free(processes);
    return 0;
}
```

B. Priority scheduling (Non Preemptive)

```
#include <stdio.h>

// Structure to represent a process
struct process {
    int pid;          // process ID
    int burst_time;   // burst time
    int priority;     // priority
    int start_time;   // start time
    int end_time;     // end time
    int turnaround_time; // turnaround time
    int waiting_time; // waiting time
};

// Function to sort the processes by priority
void sort_by_priority(struct process *p, int n) {
    int i, j;
    struct process temp;
    for(i=0; i<n-1; i++) {
```

```
        for(j=0; j<n-i-1; j++) {
            if(p[j].priority > p[j+1].priority) {
                temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
            }
        }
    }
}

// Function to compute the waiting time and turnaround time for the processes
void compute_times(struct process *p, int n) {
    int i;
    p[0].waiting_time = 0;
    p[0].turnaround_time = p[0].burst_time;
    for(i=1; i<n; i++) {
        p[i].waiting_time = p[i-1].turnaround_time;
        p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
    }
}

// Function to draw the Gantt chart
void draw_gantt_chart(struct process *p, int n) {
    int i, j;
    printf(" ");
    for(i=0; i<n; i++) {
        for(j=0; j<p[i].burst_time; j++) printf("--");
        printf(" ");
    }
    printf("\n|");
    for(i=0; i<n; i++) {
        for(j=0; j<p[i].burst_time-1; j++) printf(" ");
        printf("P%d", p[i].pid);
        for(j=0; j<p[i].burst_time-1; j++) printf(" ");
        printf("|");
    }
    printf("\n ");
    for(i=0; i<n; i++) {
        for(j=0; j<p[i].burst_time; j++) printf("--");
        printf(" ");
    }
    printf("\n");
    printf("0");
    for(i=0; i<n; i++) {
```

```
        for(j=0; j<p[i].burst_time; j++) printf("  ");
        if(p[i].turnaround_time > 9) printf("\b"); // backspace
        printf("%d", p[i].turnaround_time);
    }
    printf("\n\n");
}

// Function to implement Priority Scheduling (Non-preemptive)
void priority_scheduling(struct process *p, int n) {
    int i;
    sort_by_priority(p, n);
    p[0].start_time = 0;
    p[0].end_time = p[0].burst_time;
    for(i=1; i<n; i++) {
        p[i].start_time = p[i-1].end_time;
        p[i].end_time = p[i].start_time + p[i].burst_time;
    }
    compute_times(p, n);
    draw_gantt_chart(p, n);
}

// Driver code
int main() {
    int n, i;
    struct process p[10];
    printf("Enter the number of processes:");
    scanf("%d", &n);
printf("Enter the burst time and priority for each process:\n");
for(i=0; i<n; i++) {
    printf("P[%d]: ", i+1);
    scanf("%d %d", &p[i].burst_time, &p[i].priority);
    p[i].pid = i+1;
}
priority_scheduling(p, n);
printf("Process\tBurst Time\tPriority\tStart Time\tEnd Time\tTurnaround Time\tWaiting
Time\n");
for(i=0; i<n; i++) {
    printf("P[%d]\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", p[i].pid, p[i].burst_time, p[i].priority,
p[i].start_time, p[i].end_time, p[i].turnaround_time, p[i].waiting_time);
}
return 0;
}
```

Assuming we have the following process information:

Process	Arrival Time	Burst Time	Priority
P1	0	5	2
P2	1	3	1
P3	2	1	3
P4	3	2	4
P5	4	4	5

A. First Come First Serve (FCFS) Scheduling Algorithm:

Gantt Chart:
| P1 | P2 | P3 | P4 | P5 |
0 5 8 9 11 15

Calculation of finish time, turnaround time, and waiting time:

Process | Arrival Time | Burst Time | Finish Time | Turnaround Time | Waiting Time
P1 | 0 | 5 | 5 | 5 | 0
P2 | 1 | 3 | 8 | 7 | 4
P3 | 2 | 1 | 9 | 7 | 6
P4 | 3 | 2 | 11 | 8 | 6
P5 | 4 | 4 | 15 | 11 | 7

B. Priority Scheduling (Non-preemptive) Algorithm:

Gantt Chart:
| P2 | P1 | P3 | P4 | P5 |
1 6 7 9 13 17

Calculation of finish time, turnaround time, and waiting time:

Process | Arrival Time | Burst Time | Priority | Finish Time | Turnaround Time | Waiting Time
P2 | 1 | 3 | 1 | 4 | 3 | 0
P1 | 0 | 5 | 2 | 9 | 9 | 4
P3 | 2 | 1 | 3 | 8 | 6 | 5
P4 | 3 | 2 | 4 | 11 | 8 | 6
P5 | 4 | 4 | 5 | 17 | 13 | 9

Note: Turnaround time is the time difference between the completion time and arrival time of the process, and waiting time is the time difference between the turnaround time and the burst time of the process.

10. **Write a Shell script to check whether given number is prime or not. Also print the reverse of the given number.**

    Code:

```
#!/bin/bash

echo -n "Enter a number: "
read num

reverse=0
original=$num

while [ $num -gt 0 ]
do
   remainder=$(( $num % 10 ))
   reverse=$(( $reverse * 10 + $remainder ))
   num=$(( $num / 10 ))
done

if [ $original -eq $reverse ]
then
   echo "$original is a palindrome"
   echo "Reverse of $original is $reverse"
else
   echo "$original is not a palindrome"
   echo "Reverse of $original is $reverse"
Fi
```

11. **Write a program to solve reader-writer problem using semaphore**

    Code:

```
import java.util.concurrent.Semaphore;

class reader_writer {

   static Semaphore readLock = new Semaphore(1);
```

```
static Semaphore writeLock = new Semaphore(1);
static int readCount = 0;

static class Read implements Runnable {
    @Override
    public void run() {
        try {

            readLock.acquire();
            readCount++;
            if (readCount == 1) {
                writeLock.acquire();
            }
            readLock.release();


            System.out.println(Thread.currentThread().getName() + " is READING");
            Thread.sleep(1500);
            System.out.println(Thread.currentThread().getName() + " has FINISHED READING");


            readLock.acquire();
            readCount--;
            if (readCount == 0) {
                writeLock.release();
            }
            readLock.release();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}

static class Write implements Runnable {
    @Override
    public void run() {
        try {
            writeLock.acquire();
            System.out.println(Thread.currentThread().getName() + " is WRITING");
            Thread.sleep(2500);
            System.out.println(Thread.currentThread().getName() + " has finished WRITING");
            writeLock.release();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
```

```
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Read read = new Read();
        Write write = new Write();
        Thread t1 = new Thread(read);
        t1.setName("Reader1");
        Thread t2 = new Thread(read);
        t2.setName("Reader2");
        Thread t3 = new Thread(write);
        t3.setName("Writer1");
        Thread t4 = new Thread(read);
        t4.setName("Reader3");
        t1.start();
        t3.start();
        t2.start();
        t4.start();
    }
}
```

Explanation:

This is a Java program that demonstrates the use of Semaphores to solve the reader-writer problem. The reader-writer problem is a classical synchronization problem, where there are multiple readers and writers accessing a shared resource. In this example, there are three readers and one writer.

The program uses two Semaphores, readLock and writeLock, to control the access of the readers and writer to the shared resource. The readCount variable keeps track of the number of readers currently accessing the shared resource.

The Read class implements the Runnable interface, and when a Read thread runs, it first acquires the readLock Semaphore. Then it increments the readCount variable and checks if it is the first reader to access the shared resource. If it is the first reader, it acquires the writeLock Semaphore to prevent any writer from accessing the shared resource. After reading the shared resource, the Read thread releases the readLock Semaphore, decrements the readCount variable, and checks if it is the last reader to leave the shared resource. If it is the last reader, it releases the writeLock Semaphore, allowing any waiting writer to access the shared resource.

The Write class also implements the Runnable interface, and when a Write thread runs, it acquires the writeLock Semaphore, preventing any other readers or writers from

accessing the shared resource. After writing to the shared resource, the Write thread releases the writeLock Semaphore.

In the main method, the program creates four threads: three Read threads and one Write thread. The threads run concurrently, accessing the shared resource in a controlled and synchronized manner using Semaphores. The program demonstrates that the readers can read the shared resource simultaneously, but writers have to wait for all readers to finish before they can write to the shared resource.

12. **Write a program to solve producer-consumer problem using Mutex**

Code:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ProducerConsumerMutex {
    private static final int MAX_QUEUE_SIZE = 10;
    private static final int NUM_PRODUCERS = 2;
    private static final int NUM_CONSUMERS = 2;

    private static Queue<Integer> buffer = new LinkedList<>();
    private static Lock lock = new ReentrantLock();

    public static void main(String[] args) {
        Thread[] producerThreads = new Thread[NUM_PRODUCERS];
        Thread[] consumerThreads = new Thread[NUM_CONSUMERS];

        for (int i = 0; i < NUM_PRODUCERS; i++) {
            producerThreads[i] = new Thread(new Producer());
            producerThreads[i].start();
        }

        for (int i = 0; i < NUM_CONSUMERS; i++) {
            consumerThreads[i] = new Thread(new Consumer());
            consumerThreads[i].start();
        }

        for (int i = 0; i < NUM_PRODUCERS; i++) {
            try {
                producerThreads[i].join();
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        for (int i = 0; i < NUM_CONSUMERS; i++) {
            try {
                consumerThreads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    static class Producer implements Runnable {
        private Random random = new Random();

        @Override
        public void run() {
            while (true) {
                try {
                    lock.lock();

                    while (buffer.size() == MAX_QUEUE_SIZE) {
                        System.out.println("Buffer is full, waiting for consumer...");
                        lock.unlock();
                        Thread.sleep(1000);
                        lock.lock();
                    }

                    int item = random.nextInt(100);
                    buffer.add(item);
                    System.out.println("Produced item: " + item);

                    lock.unlock();
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    static class Consumer implements Runnable {
```

```
        @Override
        public void run() {
            while (true) {
                try {
                    lock.lock();

                    while (buffer.isEmpty()) {
                        System.out.println("Buffer is empty, waiting for producer...");
                        lock.unlock();
                        Thread.sleep(1000);
                        lock.lock();
                    }

                    int item = buffer.remove();
                    System.out.println("Consumed item: " + item);

                    lock.unlock();
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Explanation:

This program demonstrates how to solve the producer-consumer problem using Mutex in Java. In this problem, we have one or more producers that produce items and add them to a shared buffer, and one or more consumers that consume items from the buffer.

The program uses a Queue data structure to represent the shared buffer, with a maximum size of 10 items. The program creates two Producer threads and two Consumer threads. The Producer threads randomly generate an item and add it to the buffer, while the Consumer threads remove an item from the buffer and consume it. Both the Producer and Consumer threads use a Mutex to ensure that only one thread can access the buffer at a time.

The program starts by creating the threads and starting them. The Producer threads generate a random item, check if the buffer is full, and add the item to the buffer if it is not. If the buffer is full,

the Producer threads wait for the Consumer threads to consume an item before adding a new one. The Consumer threads remove an item from the buffer, check if the buffer is empty, and wait for the Producer threads to produce an item before trying to consume another one.

The program uses the Thread.join() method to wait for all the threads to finish before terminating.

Overall, this program provides a simple and effective solution to the producer-consumer problem using Mutex in Java.

13. **Write a shell script to check and count occurrence of a sub-string in the given string using command line arguments.**

```
#!/bin/bash

# Check that two arguments were provided
if [ "$#" -ne 2 ]; then
  echo "Usage: $0 <string> <substring>"
  exit 1
fi

string="$1"
substring="$2"

# Use grep and wc commands to count the occurrence of the substring in the string
count=$(echo "$string" | grep -o "$substring" | wc -l)

echo "The substring '$substring' appears in the string '$string' $count times."
```

Command: ./substring_count.sh "Hello world" "l"
Or bash filename "Hello world" "l"

Here's how the script works:

- The `if` statement checks that two arguments were provided on the command line. If not, it prints a usage message and exits with an error code.
- The first argument is assigned to the `string` variable, and the second argument is assigned to the `substring` variable.

- The echo command pipes the string variable to the grep command with the -o option, which outputs only the matching substrings. The output is then piped to the wc command with the -l option, which counts the number of lines in the output (i.e., the number of occurrences of the substring).
- The result is assigned to the count variable.
- The script prints a message that indicates how many times the substring appears in the string.

To use the script, save it to a file (e.g., substring_count.sh), make it executable with chmod +x substring_count.sh, and then run it with two arguments, like this:

14. Write a program to implement Banker's algorithm.

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

int main() {
    int n, m; // n is the number of processes and m is the number of resources
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resources: ");
    scanf("%d", &m);

    int allocation[n][m], max[n][m], need[n][m], available[m], work[m], finish[n];

    // Get the allocation matrix
    printf("\nEnter the allocation matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    // Get the max matrix
    printf("\nEnter the max matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - allocation[i][j];
        }
        finish[i] = 0;
    }
```

```
// Get the available vector
printf("\nEnter the available vector:\n");
for (int i = 0; i < m; i++) {
    scanf("%d", &available[i]);
    work[i] = available[i];
}

// Run the algorithm
int count = 0;
bool found = true;
int safe_sequence[n], safe_count = 0;

while (count < n && found) {
    found = false;
    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            bool can_run = true;
            for (int j = 0; j < m; j++) {
                if (need[i][j] > work[j]) {
                    can_run = false;
                    break;
                }
            }
            if (can_run) {
                for (int j = 0; j < m; j++) {
                    work[j] += allocation[i][j];
                }
                safe_sequence[safe_count] = i;
                safe_count++;
                finish[i] = true;
                count++;
                found = true;
            }
        }
    }
}

// Print the result
if (count == n) {
    printf("\nThe system is in a safe state.\n");
    printf("Safe sequence: ");
    for (int i = 0; i < n; i++) {
        printf("%d", safe_sequence[i]);
```

```
            if (i != n - 1) {
                printf(" -> ");
            }
        }
        printf("\n");
    } else {
        printf("\nThe system is not in a safe state.\n");
    }

    return 0;
}
```

Explanation:

This program is an implementation of the Banker's algorithm for resource allocation in operating systems. It takes user input for the number of processes and resources, as well as allocation, max, and available matrices/vectors. It then runs the Banker's algorithm to determine if the system is in a safe state, and if so, prints the safe sequence of processes.

Here is a breakdown of the program:

1. *First, the program prompts the user to enter the number of processes and resources, and declares various arrays/matrices to store the input and intermediate results.*
2. *Next, the program prompts the user to enter the allocation matrix, which represents the number of resources currently allocated to each process.*
3. *The program then prompts the user to enter the max matrix, which represents the maximum number of resources each process may need to complete its task.*
4. *The program calculates the need matrix, which represents the difference between the max and allocation matrices for each process.*
5. *The program prompts the user to enter the available vector, which represents the number of available resources of each type.*
6. *The program initializes the work vector to be a copy of the available vector, and initializes the finish array to all 0's, since no processes have yet finished.*
7. *The program enters a loop, where it checks each process to see if it can be run safely. It does this by checking if the need for each resource is less than or equal to the number of available resources of that type. If a process can be run, its allocated resources are added to the work vector, and the process is marked as*

finished. This loop continues until either all processes have been marked as finished or no processes can be run.

8. *If all processes have been marked as finished, the program prints a message indicating that the system is in a safe state, and then prints the safe sequence of processes. If not all processes have been marked as finished, the program prints a message indicating that the system is not in a safe state.*

Note that the program assumes that the input is valid and does not perform any error checking. It also assumes that the user enters the input in the correct order, as described in the prompts.

15. Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms:

A. First come First serve

```
#include <stdio.h>

struct Process {
    int pid; // process ID
    int arrival_time; // arrival time of the process
    int burst_time; // burst time of the process
    int finish_time; // finish time of the process
    int turnaround_time; // turnaround time of the process
    int waiting_time; // waiting time of the process
};

int main() {
    int n; // n is the number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    // Get the arrival times and burst times
    printf("\nEnter the arrival times and burst times:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i;
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
    }

    // Sort the processes by arrival time
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
```

```
            if (processes[i].arrival_time > processes[j].arrival_time) {
                struct Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    // Initialize the finish time of the first process
    processes[0].finish_time = processes[0].burst_time + processes[0].arrival_time;
    // Initialize the turnaround time and waiting time of the first process
    processes[0].turnaround_time = processes[0].finish_time - processes[0].arrival_time;
    processes[0].waiting_time = processes[0].turnaround_time - processes[0].burst_time;

    // Calculate the finish time, turnaround time, and waiting time for each process
    for (int i = 1; i < n; i++) {
        processes[i].finish_time = processes[i - 1].finish_time + processes[i].burst_time;
        processes[i].turnaround_time = processes[i].finish_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }

    // Print the process details
    printf("\nProcess Details:\n");
    printf("PID\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, processes[i].finish_time,
processes[i].turnaround_time, processes[i].waiting_time);
    }

    return 0;
}
```

B. Priority (Non Preemptive)

16. **Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms:**

A. First come First serve

```
#include <stdio.h>

struct Process {
    int pid; // process ID
    int arrival_time; // arrival time of the process
```

```
    int burst_time; // burst time of the process
    int finish_time; // finish time of the process
    int turnaround_time; // turnaround time of the process
    int waiting_time; // waiting time of the process
};

int main() {
    int n; // n is the number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    // Get the arrival times and burst times
    printf("\nEnter the arrival times and burst times:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i;
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
    }

    // Sort the processes by arrival time
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].arrival_time > processes[j].arrival_time) {
                struct Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    // Initialize the finish time of the first process
    processes[0].finish_time = processes[0].burst_time + processes[0].arrival_time;
    // Initialize the turnaround time and waiting time of the first process
    processes[0].turnaround_time = processes[0].finish_time - processes[0].arrival_time;
    processes[0].waiting_time = processes[0].turnaround_time - processes[0].burst_time;

    // Calculate the finish time, turnaround time, and waiting time for each process
    for (int i = 1; i < n; i++) {
        processes[i].finish_time = processes[i - 1].finish_time + processes[i].burst_time;
        processes[i].turnaround_time = processes[i].finish_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }
```

```
    // Print the process details
    printf("\nProcess Details:\n");
    printf("PID\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, processes[i].finish_time,
processes[i].turnaround_time, processes[i].waiting_time);
    }

    return 0;
}
```

B. Shortest Job First (Non Preemptive)

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

struct Process {
    int pid; // process ID
    int arrival_time; // arrival time of the process
    int burst_time; // burst time of the process
    int finish_time; // finish time of the process
    int turnaround_time; // turnaround time of the process
    int waiting_time; // waiting time of the process
    int is_completed; // flag to check if the process is completed
};

int main() {
    int n; // n is the number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    // Get the arrival times and burst times
    printf("\nEnter the arrival times and burst times:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i;
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
        processes[i].is_completed = 0;
    }

    int current_time = 0; // current time
```

```
    int completed = 0; // number of completed processes

    while (completed != n) {
        int shortest_index = -1;
        int shortest_burst_time = INT_MAX;

        // Find the process with the shortest burst time that has arrived
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && !processes[i].is_completed &&
processes[i].burst_time < shortest_burst_time) {
                shortest_index = i;
                shortest_burst_time = processes[i].burst_time;
            }
        }

        // If no process is found, move to the next second
        if (shortest_index == -1) {
            current_time++;
        } else {
            // Update the finish time of the process
            processes[shortest_index].finish_time = current_time +
processes[shortest_index].burst_time;
            processes[shortest_index].turnaround_time =
processes[shortest_index].finish_time - processes[shortest_index].arrival_time;
            processes[shortest_index].waiting_time =
processes[shortest_index].turnaround_time - processes[shortest_index].burst_time;
            processes[shortest_index].is_completed = 1;
            current_time = processes[shortest_index].finish_time;
            completed++;
        }
    }

    // Print the process details
    printf("\nProcess Details:\n");
    printf("PID\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, processes[i].finish_time,
processes[i].turnaround_time, processes[i].waiting_time);
    }

    return 0;
}
```

17. Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms:

A. First come First serve

FIRST COME FIRST SERVE

```
#include <stdio.h>

struct Process {
    int pid; // process ID
    int arrival_time; // arrival time of the process
    int burst_time; // burst time of the process
    int finish_time; // finish time of the process
    int turnaround_time; // turnaround time of the process
    int waiting_time; // waiting time of the process
};

int main() {
    int n; // n is the number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    // Get the arrival times and burst times
    printf("\nEnter the arrival times and burst times:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i;
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
    }

    // Sort the processes by arrival time
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].arrival_time > processes[j].arrival_time) {
                struct Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    // Initialize the finish time of the first process
    processes[0].finish_time = processes[0].burst_time + processes[0].arrival_time;
```

```
    // Initialize the turnaround time and waiting time of the first process
    processes[0].turnaround_time = processes[0].finish_time - processes[0].arrival_time;
    processes[0].waiting_time = processes[0].turnaround_time - processes[0].burst_time;

    // Calculate the finish time, turnaround time, and waiting time for each process
    for (int i = 1; i < n; i++) {
        processes[i].finish_time = processes[i - 1].finish_time + processes[i].burst_time;
        processes[i].turnaround_time = processes[i].finish_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }

    // Print the process details
    printf("\nProcess Details:\n");
    printf("PID\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, processes[i].finish_time,
processes[i].turnaround_time, processes[i].waiting_time);
    }

    return 0;
}
```

B. Round- Robin

```
#include<stdio.h>

struct process {
    int arrival_time;
    int burst_time;
    int priority;
    int finish_time;
    int turn_around_time;
    int waiting_time;
    int remaining_time;
};

int main() {
    int num_processes, time_quantum;

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

    struct process processes[num_processes];
```

```
    for(int i=0; i<num_processes; i++) {
        printf("Enter arrival time, burst time, and priority for process %d: ", i+1);
        scanf("%d%d%d", &processes[i].arrival_time, &processes[i].burst_time,
&processes[i].priority);
        processes[i].remaining_time = processes[i].burst_time;
    }

    printf("Enter time quantum: ");
    scanf("%d", &time_quantum);

    int current_time = 0;
    int remaining_processes = num_processes;

    while(remaining_processes > 0) {
        for(int i=0; i<num_processes; i++) {
            if(processes[i].arrival_time <= current_time && processes[i].remaining_time > 0) {
                if(processes[i].remaining_time <= time_quantum) {
                    current_time += processes[i].remaining_time;
                    processes[i].finish_time = current_time;
                    processes[i].turn_around_time = processes[i].finish_time -
processes[i].arrival_time;
                    processes[i].waiting_time = processes[i].turn_around_time -
processes[i].burst_time;
                    processes[i].remaining_time = 0;
                    remaining_processes--;
                }
                else {
                    current_time += time_quantum;
                    processes[i].remaining_time -= time_quantum;
                }
            }
        }
    }

    printf("Process\tArrival Time\tBurst Time\tPriority\tFinish Time\tTurnaround
Time\tWaiting Time\n");

    for(int i=0; i<num_processes; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", i+1, processes[i].arrival_time,
processes[i].burst_time, processes[i].priority, processes[i].finish_time,
processes[i].turn_around_time, processes[i].waiting_time);
    }
```

```
    return 0;
}
```

18. Write down Linux commands for following statements:

i. To redirect the output of cp command to a file named outfile. Use this as an input file for remaining commands.

Command : cp [source_file] [destination_file] > outfile

Replace [source_file] with the name of the file you want to copy, and [destination_file] with the name of the new file you want to create. The > symbol redirects the output of the cp command to the outfile file.

ii. To select the lines in a file which has 4 letter words in that line and redirect the output to the file named list.

Command : grep -Eo '\b\w{4}\b' [input_file] > list

This command searches for words in [input_file] that are exactly four letters long, and uses the -o option to print only the matching words (not the entire line). The -E option is used to enable extended regular expressions, and the \b and \w symbols are regular expression patterns that match word boundaries and word characters, respectively.

iii. Assign write permission to owner and remove read permission from group user for an ordinary file named test by relative way.

Command : chmod u+w,g-r test

This command changes the permissions of the file named test in relative mode. The u+w option adds write permission for the owner of the file, and the g-r option removes read permission for the group. The test argument specifies the file to modify.

iv. Create an alias named ls that always lists all the files including hidden files.

Command : alias ls='ls -a'

This command creates an alias for the ls command, and adds the -a option, which causes ls to list all files, including hidden files.

v. Count the number of words in the list file.

Command : wc -w list

This command reads the list file and counts the number of words in it. The -w option tells wc to count words (i.e., sequences of characters separated by whitespace).

19. Write down Linux commands for following statements:

a. Redirect the output of mv command to a file named outfile. Use this as an input file for remaining commands.

Command : mv source_file destination_file > outfile

This will move the source_file to the destination_file, and then pass the output of the mv command to command1, command2, and command3, in sequence. Note that this approach may not work if the commands require the input to be a file rather than a stream of text.

b. List all hidden files under current directory and store in “hidden” file

Command: ls -a | grep "^\." > hidden

ls -a lists all files, including hidden files, in the current directory.
The grep command searches for lines that start with a dot (which is the convention for hidden files in Unix-like systems). The ^ character matches the beginning of a line, and the . character matches any character, so "^\." matches a dot at the beginning of a line.
The > operator redirects the output of the grep command to a file named "hidden". If the file already exists, its contents will be overwritten. If it doesn't exist, it will be created.

c. Assign write permission to group user and remove execute permission from owner for an ordinary file named test by absolute way

Command : chmod 640 test

The chmod command is used to change the permissions of a file or directory.
The first digit in the permission string (6 in this case) specifies the permissions for the owner of the file.
The second digit (4 in this case) specifies the permissions for the group owner of the file.
The third digit (0 in this case) specifies the permissions for all other users.
The number 6 is the sum of the values of the read (4) and write (2) permissions. This grants read and write permissions to the owner of the file.
The number 4 is the value of the read permission, which grants read permissions to the group owner of the file.
The number 0 means no permissions are granted to all other users.
This command removes execute permission from the owner, since the execute permission is not included in the permission string.

d. To create soft and hard link for given file

1. **Soft link command : ln -s /path/to/original/file /path/to/link**

   Ex. ln -s ~/original.txt ~/link.txt

2. **Hard link command :ln /path/to/original/file /path/to/link**

   Ex. ln ~/original.txt ~/link2.txt

The difference between a soft link and a hard link is that a soft link is a reference to the original file, while a hard link is a direct reference to the same data on the disk. If you delete the original file, the soft link will become broken, while the hard link will still work as long as there is at least one link to the data on the disk.

e. To convert lowercase to upper case of a given file
Command : tr '[:lower:]' '[:upper:]' < [input_file] > [output_file]

Replace [input_file] with the name of the input file, and [output_file] with the name of the file to write the uppercase text to. The tr command reads the input file and replaces all lowercase characters with their uppercase counterparts.
The [:lower:] and [:upper:] character classes are defined by the tr command to represent all lowercase and uppercase letters, respectively.
After running this command, the contents of the [output_file] will contain the same text as the input file, but with all lowercase letters converted to uppercase.

f. To extract 2 nd and 3 rd character of a given file

Command : cut -c 2-3 /path/to/file

Ex: cut -c 2-3 example.txt

g. To display how many times lines are repeated in a given file.
Command : sort [input_file] | uniq -c

Replace [input_file] with the name of the file you want to analyze. The sort command sorts the lines in the input file in alphabetical order, and the uniq -c command counts the number of times each unique line occurs in the sorted output.

After running this command, uniq will print a list of unique lines in the input file, along with a count of how many times each line occurs.

20. **Write a Shell script to check whether given number is palindrome or not. Also print the reverse of the given number.**

Code:

```
#!/bin/bash

echo -n "Enter a number: "
read num

reverse=0
original=$num

while [ $num -gt 0 ]
do
   remainder=$(( $num % 10 ))
   reverse=$(( $reverse * 10 + $remainder ))
   num=$(( $num / 10 ))
done

if [ $original -eq $reverse ]
then
   echo "$original is a palindrome"
   echo "Reverse of $original is $reverse"
else
   echo "$original is not a palindrome"
   echo "Reverse of $original is $reverse"
Fi
```

Explanation:

The script prompts the user to enter a number using the echo and read commands.
The reverse variable is initialized to 0 and the original number is stored in the original variable.
The while loop is used to reverse the number. In each iteration, the last digit of the number is extracted using the modulo operator and added to the reverse variable. The last digit is then removed from the number by dividing it by 10.

After the loop, an if statement is used to check whether the original number and the reversed number are equal. If they are equal, the script prints that the number is a palindrome, along with its reverse. If they are not equal, the script prints that the number is not a palindrome, along with its reverse.

21. **Write a Shell script to find the Factorial of given number using Recurrence Method and Without Recurrence Method (Both way)**

Function to find factorial using recurrence method

```
#!/bin/bash

echo -n "Enter a number: "
read num

factorial() {
  if [ $1 -eq 0 ]; then
    echo 1
  else
    local temp=$(factorial $(( $1 - 1 )))
    echo $(( $1 * $temp ))
  fi
}

result=$(factorial $num)
echo "Factorial of $num is $result"
```

Function to find factorial without using recurrence

```
#!/bin/bash

echo -n "Enter a number: "
read num

factorial=1
for (( i=1; i<=$num; i++ ))
do
  factorial=$(( $factorial * $i ))
done

echo "Factorial of $num is $factorial"
```

22. Write a Shell script to check whether given number is palindrome or not. Also print the reverse of the given number.

```
#!/bin/bash

# Function to check if a number is a palindrome
function is_palindrome {
    local n=$1
    local reverse=0
    local original=$n

    while [ $n -gt 0 ]; do
        local digit=$((n % 10))
        reverse=$((reverse * 10 + digit))
        n=$((n / 10))
    done

    if [ $original -eq $reverse ]; then
        echo "The number $original is a palindrome."
    else
        echo "The number $original is not a palindrome."
    fi

    echo "Reverse of $original is $reverse"
}

# Prompt user for input
echo "Enter a number:"
read n

# Check if input is a positive integer
if ! [[ "$n" =~ ^[1-9][0-9]*$ ]]; then
    echo "Invalid input. Please enter a positive integer."
    exit 1
fi

# Check if input is a palindrome
is_palindrome $n
```

23. Write a Program to Implement Page Replacement Algorithm Using: A. First Come First Serve B. Least Recently used

Code:

FCFS Java

```
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Queue;

class PageReplacementFIFO {
    static int pageFaults(int incomingStream[], int n, int frames) {
        System.out.println("Incoming \t Pages");

        HashSet s = new HashSet<>(frames);

        Queue queue = new LinkedList<>();

        int page_faults = 0;

        for (int i = 0; i < n; i++) {

            if (s.size() < frames) {

                if (!s.contains(incomingStream[i])) {
                    s.add(incomingStream[i]);
                    page_faults++;

                    queue.add(incomingStream[i]);

                }
            }

            else {

                if (!s.contains(incomingStream[i])) {

                    int val = (int) queue.peek();

                    queue.poll();

                    s.remove(val);

                    s.add(incomingStream[i]);

                    queue.add(incomingStream[i]);
                    page_faults++;
```

```
            }
        }

        System.out.print(incomingStream[i] + "\t\t");
        System.out.print(queue + " \n");
    }

    return page_faults;
  }

  public static void main(String args[]) {
    int incomingStream[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1 };
    int frames = 3;

    int len = incomingStream.length;
    int pageFaults = pageFaults(incomingStream, len, frames);
    int hit = len - pageFaults;

    System.out.println("Page faults: " + pageFaults);

  }
}
```

FCFS C

```
#include <stdio.h>

int main() {
  int pages[100], frames[10], pageFaults = 0, pointer = 0, numPages, numFrames, i, j, k;
  // pages: array to hold page references
  // frames: array to hold frames in memory
  // pageFaults: count of page faults
  // pointer: pointer to indicate oldest page in frames
  // numPages: number of page references in the pages array
  // numFrames: number of frames in memory
  // i, j, k: loop variables

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

  printf("Enter the page reference string: ");
  for (i = 0; i < numPages; i++) {
    scanf("%d", &pages[i]);
  }
```

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

    for (i = 0; i < numFrames; i++) {
        frames[i] = -1;
    }

    printf("\nPage Reference String\tPage Frames\n");

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

        int pageFound = 0;

        for (j = 0; j < numFrames; j++) {
            if (frames[j] == pages[i]) {
                pageFound = 1;
                break;
            }
        }

        if (!pageFound) {
            frames[pointer] = pages[i];
            pointer = (pointer + 1) % numFrames;
            pageFaults++;

            for (j = 0; j < numFrames; j++) {
                printf("%d ", frames[j]);
            }
        } else {
            for (j = 0; j < numFrames; j++) {
                printf("%d ", frames[j]);
            }
        }

        printf("\n");
    }

    printf("\nTotal page faults: %d\n", pageFaults);

    return 0;
}
```

LRU Java

```
import java.io.*;
import java.util.*;

public class PageReplacementLRU {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int frames, pointer = 0, hit = 0, fault = 0, ref_len;
        Boolean isFull = false;
        int buffer[];
        ArrayList<Integer> stack = new ArrayList<Integer>();
        int reference[];
        int mem_layout[][];

        System.out.println("Please enter the number of Frames: ");
        frames = Integer.parseInt(br.readLine());

        System.out.println("Please enter the length of the Reference string: ");
        ref_len = Integer.parseInt(br.readLine());

        reference = new int[ref_len];
        mem_layout = new int[ref_len][frames];
        buffer = new int[frames];
        for (int j = 0; j < frames; j++)
            buffer[j] = -1;

        System.out.println("Please enter the reference string: ");
        for (int i = 0; i < ref_len; i++) {
            reference[i] = Integer.parseInt(br.readLine());
        }
        System.out.println();
        for (int i = 0; i < ref_len; i++) {
            if (stack.contains(reference[i])) {
                stack.remove(stack.indexOf(reference[i]));
            }
            stack.add(reference[i]);
            int search = -1;
            for (int j = 0; j < frames; j++) {
                if (buffer[j] == reference[i]) {
                    search = j;
                    hit++;
                    break;
```

```
            }
        }
        if (search == -1) {
            if (isFull) {
                int min_loc = ref_len;
                for (int j = 0; j < frames; j++) {
                    if (stack.contains(buffer[j])) {
                        int temp = stack.indexOf(buffer[j]);
                        if (temp < min_loc) {
                            min_loc = temp;
                            pointer = j;
                        }
                    }
                }
            }
            buffer[pointer] = reference[i];
            fault++;
            pointer++;
            if (pointer == frames) {
                pointer = 0;
                isFull = true;
            }
        }
        for (int j = 0; j < frames; j++)
            mem_layout[i][j] = buffer[j];
    }

    for (int i = 0; i < frames; i++) {
        for (int j = 0; j < ref_len; j++)
            System.out.printf("%3d ", mem_layout[j][i]);
        System.out.println();
    }

    System.out.println("The number of Hits: " + hit);
    System.out.println("Hit Ratio: " + (float) ((float) hit / ref_len));
    System.out.println("The number of Faults: " + fault);
  }

}
```

LRU C

```
#include <stdio.h>

int main() {
```

```
int pages[100], frames[10], pageFaults = 0, numPages, numFrames, i, j, k, l, counter = 0;
// pages: array to hold page references
// frames: array to hold frames in memory
// pageFaults: count of page faults
// numPages: number of page references in the pages array
// numFrames: number of frames in memory
// i, j, k, l: loop variables
// counter: counter to keep track of when each page was last used

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

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

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

for (i = 0; i < numFrames; i++) {
    frames[i] = -1;
}

printf("\nPage Reference String\tPage Frames\n");

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

    int pageFound = 0;

    for (j = 0; j < numFrames; j++) {
        if (frames[j] == pages[i]) {
            pageFound = 1;
            counter++;
            break;
        }
    }

    if (!pageFound) {
        int minCounter = counter + 1;
        int minIndex = -1;

        for (j = 0; j < numFrames; j++) {
```

```
            for (k = i - 1; k >= 0; k--) {
                if (frames[j] == pages[k]) {
                    if (k < minCounter) {
                        minCounter = k;
                        minIndex = j;
                        break;
                    }
                }
            }
        }

        if (minIndex == -1) {
            minIndex = 0;
        }

        frames[minIndex] = pages[i];
        pageFaults++;

        for (j = 0; j < numFrames; j++) {
            printf("%d ", frames[j]);
        }

        counter++;
    } else {
        for (j = 0; j < numFrames; j++) {
            printf("%d ", frames[j]);
        }
    }

    printf("\n");
  }

  printf("\nTotal page faults: %d\n", pageFaults);

  return 0;
}
```

24. Write a Program to Implement Page Replacement Algorithm Using: A. First Come First Serve B. Not Recently Used

FCFS Java

```
    import java.util.HashSet;
import java.util.LinkedList;
```

```
import java.util.Queue;

class PageReplacementFIFO {
    static int pageFaults(int incomingStream[], int n, int frames) {
        System.out.println("Incoming \t Pages");

        HashSet s = new HashSet<>(frames);

        Queue queue = new LinkedList<>();

        int page_faults = 0;

        for (int i = 0; i < n; i++) {

            if (s.size() < frames) {

                if (!s.contains(incomingStream[i])) {
                    s.add(incomingStream[i]);
                    page_faults++;

                    queue.add(incomingStream[i]);

                }
            }

            else {

                if (!s.contains(incomingStream[i])) {

                    int val = (int) queue.peek();

                    queue.poll();

                    s.remove(val);

                    s.add(incomingStream[i]);

                    queue.add(incomingStream[i]);
                    page_faults++;

                }
            }

            System.out.print(incomingStream[i] + "\t\t");
```

```
            System.out.print(queue + " \n");
        }

        return page_faults;
    }

    public static void main(String args[]) {
        int incomingStream[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1 };
        int frames = 3;

        int len = incomingStream.length;
        int pageFaults = pageFaults(incomingStream, len, frames);
        int hit = len - pageFaults;

        System.out.println("Page faults: " + pageFaults);

    }
}
```

FCFS C

```
#include <stdio.h>

int main() {
    int pages[100], frames[10], pageFaults = 0, pointer = 0, numPages, numFrames, i, j, k;
    // pages: array to hold page references
    // frames: array to hold frames in memory
    // pageFaults: count of page faults
    // pointer: pointer to indicate oldest page in frames
    // numPages: number of page references in the pages array
    // numFrames: number of frames in memory
    // i, j, k: loop variables

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

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

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

    for (i = 0; i < numFrames; i++) {
```

```
        frames[i] = -1;
    }

    printf("\nPage Reference String\tPage Frames\n");

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

        int pageFound = 0;

        for (j = 0; j < numFrames; j++) {
            if (frames[j] == pages[i]) {
                pageFound = 1;
                break;
            }
        }

        if (!pageFound) {
            frames[pointer] = pages[i];
            pointer = (pointer + 1) % numFrames;
            pageFaults++;

            for (j = 0; j < numFrames; j++) {
                printf("%d ", frames[j]);
            }
        } else {
            for (j = 0; j < numFrames; j++) {
                printf("%d ", frames[j]);
            }
        }

        printf("\n");
    }

    printf("\nTotal page faults: %d\n", pageFaults);

    return 0;
}
```

NRU C

```
#include <stdio.h>

#define MAX_FRAMES 100
```

```
#define MAX_PAGES 100

int main() {
    int frames[MAX_FRAMES], referenceBits[MAX_FRAMES], ageBits[MAX_FRAMES];
    int pageRefs[MAX_PAGES];
    int numFrames, numPageRefs;
    int i, j, k, l, m;
    int pageFaults = 0;
    int oldestAge = 0;

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

    printf("Enter the number of page references: ");
    scanf("%d", &numPageRefs);

    printf("Enter the page reference string: ");
    for (i = 0; i < numPageRefs; i++) {
        scanf("%d", &pageRefs[i]);
    }

    for (i = 0; i < numFrames; i++) {
        frames[i] = -1;
        referenceBits[i] = 0;
        ageBits[i] = 0;
    }

    for (i = 0; i < numPageRefs; i++) {
        int page = pageRefs[i];
        int found = 0;

        for (j = 0; j < numFrames; j++) {
            if (frames[j] == page) {
                referenceBits[j] = 1;
                found = 1;
                break;
            }
        }

        if (!found) {
            int replaced = 0;

            for (j = 0; j < numFrames; j++) {
                if (frames[j] == -1) {
```

```
            frames[j] = page;
            referenceBits[j] = 1;
            replaced = 1;
            break;
        }
    }

    if (!replaced) {
        oldestAge = 0;

        for (j = 0; j < numFrames; j++) {
            if (ageBits[j] < oldestAge) {
                oldestAge = ageBits[j];
            }
        }

        for (j = 0; j < numFrames; j++) {
            if (ageBits[j] == oldestAge && referenceBits[j] == 0) {
                frames[j] = page;
                referenceBits[j] = 1;
                ageBits[j] = 0;
                replaced = 1;
                break;
            }
        }

        if (!replaced) {
            for (j = 0; j < numFrames; j++) {
                ageBits[j] >>= 1;
                ageBits[j] |= (referenceBits[j] << 7);
                referenceBits[j] = 0;
            }

            for (j = 0; j < numFrames; j++) {
                if (ageBits[j] < oldestAge) {
                    oldestAge = ageBits[j];
                }
            }

            for (j = 0; j < numFrames; j++) {
                if (ageBits[j] == oldestAge) {
                    frames[j] = page;
                    referenceBits[j] = 1;
                    ageBits[j] = 0;
```

```
                    break;
                }
            }
        }
    }

    pageFaults++;
}

printf("Page %d: ", page);

for (j = 0; j < numFrames; j++) {
    printf("%d ", frames[j]);
}

printf("\n");
}

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

return 0;
}
```

NRU Java

```
import java.util.ArrayList;
import java.util.HashMap;

public class NRUPageReplacement {
    public static void main(String[] args) {
        int[] pageRequests = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
        int numFrames = 3;
        int numPageFaults = 0;
        int currentIndex = 0;

        // Initialize the page frames and their last usage times
        ArrayList<Integer> pageFrames = new ArrayList<>(numFrames);
        HashMap<Integer, Integer> lastUsageTimes = new HashMap<>();

        // Process each page request
        for (int page : pageRequests) {
            // Check if the page is already in a page frame
            if (pageFrames.contains(page)) {
                lastUsageTimes.put(page, currentIndex);
            } else {
```

```
            // Check if there is an empty page frame
            if (pageFrames.size() < numFrames) {
                pageFrames.add(page);
                lastUsageTimes.put(page, currentIndex);
            } else {
                // Find the least recently used (LRU) page in the page frames
                int lruPage = pageFrames.get(0);
                int lruPageIndex = 0;
                for (int i = 1; i < pageFrames.size(); i++) {
                    int framePage = pageFrames.get(i);
                    if (lastUsageTimes.get(framePage) < lastUsageTimes.get(lruPage)) {
                        lruPage = framePage;
                        lruPageIndex = i;
                    }
                }

                // Replace the LRU page with the new page
                pageFrames.set(lruPageIndex, page);
                lastUsageTimes.put(page, currentIndex);
            }
            numPageFaults++;
        }

        currentIndex++;
    }

    System.out.println("Number of page faults: " + numPageFaults);
  }
}
```

25. Write a program to implement Banker's algorithm.

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

int main() {
    int n, m; // n is the number of processes and m is the number of resources
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resources: ");
    scanf("%d", &m);
```

```
int allocation[n][m], max[n][m], need[n][m], available[m], work[m], finish[n];

// Get the allocation matrix
printf("\nEnter the allocation matrix:\n");
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        scanf("%d", &allocation[i][j]);
    }
}

// Get the max matrix
printf("\nEnter the max matrix:\n");
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        scanf("%d", &max[i][j]);
        need[i][j] = max[i][j] - allocation[i][j];
    }
    finish[i] = 0;
}

// Get the available vector
printf("\nEnter the available vector:\n");
for (int i = 0; i < m; i++) {
    scanf("%d", &available[i]);
    work[i] = available[i];
}

// Run the algorithm
int count = 0;
bool found = true;
int safe_sequence[n], safe_count = 0;

while (count < n && found) {
    found = false;
    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            bool can_run = true;
            for (int j = 0; j < m; j++) {
                if (need[i][j] > work[j]) {
                    can_run = false;
                    break;
                }
            }
            if (can_run) {
```

```
                for (int j = 0; j < m; j++) {
                    work[j] += allocation[i][j];
                }
                safe_sequence[safe_count] = i;
                safe_count++;
                finish[i] = true;
                count++;
                found = true;
            }
        }
    }

    // Print the result
    if (count == n) {
        printf("\nThe system is in a safe state.\n");
        printf("Safe sequence: ");
        for (int i = 0; i < n; i++) {
            printf("%d", safe_sequence[i]);
            if (i != n - 1) {
                printf(" -> ");
            }
        }
        printf("\n");
    } else {
        printf("\nThe system is not in a safe state.\n");
    }

    return 0;
}
```

Explanation:

This program is an implementation of the Banker's algorithm for resource allocation in operating systems. It takes user input for the number of processes and resources, as well as allocation, max, and available matrices/vectors. It then runs the Banker's algorithm to determine if the system is in a safe state, and if so, prints the safe sequence of processes.

Here is a breakdown of the program:

9. First, the program prompts the user to enter the number of processes and resources, and declares various arrays/matrices to store the input and intermediate results.
10. Next, the program prompts the user to enter the allocation matrix, which represents the number of resources currently allocated to each process.
11. The program then prompts the user to enter the max matrix, which represents the maximum number of resources each process may need to complete its task.
12. The program calculates the need matrix, which represents the difference between the max and allocation matrices for each process.
13. The program prompts the user to enter the available vector, which represents the number of available resources of each type.
14. The program initializes the work vector to be a copy of the available vector, and initializes the finish array to all 0's, since no processes have yet finished.
15. The program enters a loop, where it checks each process to see if it can be run safely. It does this by checking if the need for each resource is less than or equal to the number of available resources of that type. If a process can be run, its allocated resources are added to the work vector, and the process is marked as finished. This loop continues until either all processes have been marked as finished or no processes can be run.
16. If all processes have been marked as finished, the program prints a message indicating that the system is in a safe state, and then prints the safe sequence of processes. If not all processes have been marked as finished, the program prints a message indicating that the system is not in a safe state.

Note that the program assumes that the input is valid and does not perform any error checking. It also assumes that the user enters the input in the correct order, as described in the prompts.