MEASI INSTITUTE OFINFORMATION TECHNOLOGY

(Approved by AICTE & Affiliated to University of Madras)

CHENNAI – 600 014

1902

MEASI
Institute *of*
Information
Technology

MASTER OF COMPUTERAPPLICATIONS

ACADEMIC YEAR 2024 - 2025

SEMESTER – I

Practical Record– II

OPERATING SYSTEMS & UNIX-LAB

REG. NO : ______________________________________

NAME : ______________________________________

BATCH : ______________________________________

MEASI INSTITUTE OF INFORMATION TECHNOLOGY

(Approved by AICTE & Affiliated to University of Madras)

CHENNAI- 600 014

MCA PRACTICAL

OPERATING SYSTEMS & UNIX-LAB

Academic Year 2024 - 2025

Semester - I

NAME	**:**	**CLASS**	**:**
REG-NO	**:**	**BATCH**	**:**

This is to certify that this is the bonafide record of work done in the Computer Science Laboratory of **MEASI Institute of Information Technology**, submitted for the **University of Madras** Practical Examination held on .... at **MEASI Institute of Information Technology**, **Chennai-600 014**.

STAFF IN-CHARGE **HEAD OF THE DEPARTMENT**

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DATE: ___________________________

INDEX

S.No.	Date	Program Name	PG No:	Signature
1.	28-08-2024	IPC Using Pipes	1	
2.	04-09-2024	Implementation of wait and signal using counting Semaphore	4	
3.	06-09-2024	Signalling processes	8	
4.	11-09-2024	Deadlock detection	11	
5.	13-09-2024	Process Scheduling: FCFS	14	
6.	18-09-2024	Process Scheduling: Least Frequency used	17	
7.	20-09-2024	Process Scheduling: Round Robin	20	
8.	20-09-2024	Two Process Mutual Exclusion	23	
9.	25-09-2024	Checking Read,write,Execute Permissions	25	
10.	25-09-2024	Cat Command	27	

11.	27-09-2024	Grep Command	32	
12.	27-09-2024	SED	34	
13.	28-09-2024	AWK	36	
14.	28-09-2024	Prime Number	38	
15.	04-10-2024	Factorial	40	
16.	04-10-2024	Palindrome	42	
17.	09-10-2024	Searching an element	44	
18.	09-10-2024	Menu	46	

C Programs

Ex.No:1

Date: 28-08-2024

1.IPC Using Pipes

AIM:

To Write A Program to Stimulate The IPC Using Pipes.

ALGORITHM:

STEP 1: Initialize an array pipefds to store the file descriptors of the pipe.

STEP 2: Call a function to create a pipe and check for any errors.

STEP 3: Use fork() to create a new process.

STEP 4: Check whether the process is the parent or the child based on the returned PID .

STEP 5: Communication in Child Process:

* Read from the read end of the pipe (pipefds[0]) into the readmessage buffer.

* Print the received message.

STEP 6: Communication in Parent Process:

*Print the message to be sent (writemessages[0]).

*Write the first message into the write end of pipe(pipefds[1]).

*Print the second message to be sent (writemessages[1]).

*Write the second message into the write end of the pipe.

STEP 7: Both processeses should closed any unused file descriptors to release System resources.

SOURCE CODE:

```
#include<stdio.h>
#include<unistd.h>
int main() {
 int pipefds[2];
 int returnstatus;
 int pid;
 char writemessages[2][20]={"Hi", "Hello"};
 char readmessage[20];
 returnstatus = pipe(pipefds);
 if (returnstatus == -1) {
 printf("Unable to create pipe\n");
 return 1;
 }
 pid = fork();
 // Child process
 if (pid == 0) {
 read(pipefds[0], readmessage, sizeof(readmessage));
 printf("Child Process - Reading from pipe - Message 1 is %s\n", readmessage);
 read(pipefds[0], readmessage, sizeof(readmessage));
 printf("Child Process - Reading from pipe - Message 2 is %s\n", readmessage);
 } else { //Parent process
 printf("Parent Process - Writing to pipe - Message 1 is %s\n", writemessages[0]);
 write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
 printf("Parent Process - Writing to pipe - Message 2 is %s\n", writemessages[1]);
 write(pipefds[1], writemessages[1], sizeof(writemessages[1]));
 }
 return 0;
}
```

Output:

```
stud@YSA:~/ysa$ ./tpipe1
Parent Process - Writing to pipe - Message 1 is Hi
Parent Process - Writing to pipe - Message 2 is Hello
Child Process - Reading from pipe - Message 1 is Hi
Child Process - Reading from pipe - Message 2 is Hello
```

RESULT:

The above program is executed successfully

Ex.No: 2
Date: 04-09-2024

2. Implementation of wait and signal using counting semaphore

AIM:

To Write A Program to make Implementations of wait and signal using counting semaphores.

ALGORITHM:

STEP 1: Create a counting semaphores with an initial value of 1.

STEP 2: Thread Function:

- Each thread executes a function (thread_function) that takes a thread ID as an argument.
- Perform a "Wait" operation (sem_wait) on the semaphore, blocking if the semaphore value is zero.
- Enter the critical section and print a message indicating that the thread is in the critical section.
- Simulate some work in the critical section.
- Perform a "Signal" operation (sem_post) on the semaphore to release it.
- Print a message indicating that the thread has left the critical section.
- Exit the thread.

STEP 3: Main Function:

- Specify the number of threads (num_threads), creating an array of thread IDs (thread_ids).
- Initialize the counting semaphore with an initial value of 1 using sem_init.
- Create threads, passing the thread ID and the thread function (thread_function) as arguments.
- Wait for all threads to finish using pthread_join.

SOURCE CODE:

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

#define MAX_RESOURCES 5

sem_t semaphore;

int available_resources = MAX_RESOURCES;

// Initialize the semaphore
void init_semaphore() {
    sem_init(&semaphore, 0, 1); // Initialize to 1, as it's a binary semaphore
}

// Implement the 'wait' operation
void wait_operation() {
    sem_wait(&semaphore);
    if (available_resources > 0) {
        available_resources--;
        printf("Resource acquired. Remaining resources: %d\n", available_resources);
    } else {
        printf("No resources available. Waiting...\n");
    }
    sem_post(&semaphore);
}

// Implement the 'signal' operation
void signal_operation() {
    sem_wait(&semaphore);
    if (available_resources < MAX_RESOURCES) {
        available_resources++;
        printf("Resource released. Remaining resources: %d\n", available_resources);
    }
    sem_post(&semaphore);
}

void *thread_function(void *thread_id) {
    int id = (int)thread_id;

    // Simulate resource allocation and release by multiple threads
    for (int i = 0; i < 5; i++) {
        wait_operation(); // Acquire a resource
        // Perform work with the resource
        printf("Thread %d is using the resource...\n", id);
        sleep(1); // Simulate work
        signal_operation(); // Release the resource
        // Continue working or release the resource for others to use
        sleep(1); // Simulate additional work or idle time
    }
    pthread_exit(NULL);
}

int main() {
    init_semaphore();

    pthread_t threads[3];
    int thread_ids[3] = {1, 2, 3};

    for (int i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void *)thread_ids[i]);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&semaphore);

    return 0;
}
```

Output :

```
stud@YSA:~/ysa$ ./tcsem
Resource acquired. Remaining resources: 4
Thread 1 is using the resource...
Resource acquired. Remaining resources: 3
Thread 2 is using the resource...
Resource acquired. Remaining resources: 2
Thread 3 is using the resource...
Resource released. Remaining resources: 3
Resource released. Remaining resources: 4
Resource released. Remaining resources: 5
Resource acquired. Remaining resources: 4
Thread 1 is using the resource...
Resource acquired. Remaining resources: 3
Thread 2 is using the resource...
Resource acquired. Remaining resources: 2
Thread 3 is using the resource...
Resource released. Remaining resources: 3
Resource released. Remaining resources: 4
Resource released. Remaining resources: 5
Resource acquired. Remaining resources: 4
Thread 1 is using the resource...
Resource acquired. Remaining resources: 3
Thread 2 is using the resource...
Resource acquired. Remaining resources: 2
Thread 3 is using the resource...
Resource released. Remaining resources: 3
Resource released. Remaining resources: 4
Resource released. Remaining resources: 5
Resource acquired. Remaining resources: 4
Thread 3 is using the resource...
Resource acquired. Remaining resources: 3
Thread 1 is using the resource...
Resource acquired. Remaining resources: 2
Thread 2 is using the resource...
Resource released. Remaining resources: 3
Resource released. Remaining resources: 4
Resource released. Remaining resources: 5
Resource acquired. Remaining resources: 4
Thread 3 is using the resource...
Resource acquired. Remaining resources: 3
Thread 1 is using the resource...
Resource acquired. Remaining resources: 2
Thread 2 is using the resource...
Resource released. Remaining resources: 3
Resource released. Remaining resources: 4
Resource released. Remaining resources: 5
```

RESULT:

The above program is completed Successfully.

Ex.No:3

3.Signalling processes

Date: 06-09-2024

AIM:

To Write A Program to stimulate the Signalling processes.

ALGORITHM:

STEP 1: Fork a child process.

STEP 2:. If fork fails, print an error and exit.

STEP 3: In the child process:

- Set a signal handler for SIGUSR1.
- Print a message indicating waiting for a signal.
- Enter an infinite loop.

STEP 4: In the parent process:

- Sleep for 1 second.
- Print a message indicating sending SIGUSR1 signal.
- Send SIGUSR1 signal to the child.
- Sleep for an additional second.
- Return 0.

SOURCE CODE

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

void signal_handler(int signum) {
   if (signum == SIGUSR1) {
      printf("Received SIGUSR1 signal.\n");
   }
}

int main() {
   pid_t pid;

   // Register the signal handler for SIGUSR1
   signal(SIGUSR1, signal_handler);

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

   if (pid == -1) {
      perror("Fork failed");
      return 1;
   }

   if (pid == 0) {
      // Child process
      sleep(2);
      // Send SIGUSR1 signal to the parent process
      kill(getppid(), SIGUSR1);
   } else {
      // Parent process
      printf("Parent process waiting for a signal...\n");
      // Wait for the child to send the signal
      pause();
      printf("Parent process exiting.\n");
   }

   return 0;
}
```

Output:

```
stud@YSA:~/ysa$ ./tsignal3
Parent process waiting for a signal...
Received SIGUSR1 signal.
Parent process exiting.
```

RESULT:

The above program is completed Successfully.

Ex.No: 4

4. Deadlock Detection

Date: 11-09-2024

AIM:

To Write A Program to stimulate the Deadlock detection
(for processes passing messages)

ALGORITHM:

STEP 1: Declare and initialize two mutexes, mutex1 and mutex2.

STEP 2: Declare and initialize resource variables (resource1 and resource2) to zero.

STEP 3: Acquire a mutex to access one resource.

STEP 4: Print acquisition message for the first resource.

STEP 5: Simulate work by sleeping for 1 second.

STEP 6: Print a message indicating waiting for the second resource.

STEP 7: Attempt to acquire the second mutex using pthread_mutex_trylock.

STEP 8: If successful, print acquisition message for the second resource and release the second mutex.

STEP 9: If unsuccessful (indicating a deadlock), print a deadlock detection message and exit the program.

STEP 10: Main Function:

- Initialize the mutexes.
- Create two threads, one for each process function.
- Wait for both threads to finish using pthread_join.
- Destroy the mutexes.

SOURCE CODE

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

  pthread_mutex_t mutex1, mutex2;
  int resource1 = 0;
  int resource2 = 0;

  void* process1(void* arg) {
      pthread_mutex_lock(&mutex1);
      printf("Process 1 acquired Resource 1.\n");
      sleep(1);
      printf("Process 1 waiting for Resource 2...\n");
      pthread_mutex_lock(&mutex2);
      printf("Process 1 acquired Resource 2.\n");
      pthread_mutex_unlock(&mutex2);
      pthread_mutex_unlock(&mutex1);
      return NULL;
  }

  void* process2(void* arg) {
      pthread_mutex_lock(&mutex2);
      printf("Process 2 acquired Resource 2.\n");
      sleep(1);
      printf("Process 2 waiting for Resource 1...\n");
      pthread_mutex_lock(&mutex1);
      printf("Process 2 acquired Resource 1.\n");
      pthread_mutex_unlock(&mutex1);
      pthread_mutex_unlock(&mutex2);
      return NULL;
  }

  int main() {
      pthread_t thread1, thread2;

      pthread_mutex_init(&mutex1, NULL);
      pthread_mutex_init(&mutex2, NULL);

      pthread_create(&thread1, NULL, process1, NULL);
      pthread_create(&thread2, NULL, process2, NULL);

      pthread_join(thread1, NULL);
      pthread_join(thread2, NULL);

      pthread_mutex_destroy(&mutex1);
      pthread_mutex_destroy(&mutex2);

      return 0;
}
```

Output :

```
stud@YSA:~/ysa$ ./tdlock
Process 1 acquired Resource 1.
Process 2 acquired Resource 2.
Process 1 waiting for Resource 2...
Process 2 waiting for Resource 1...
```

RESULT:

The above program is completed Successfully.

Ex.No:5
Date: 13-09-2024

5. Process Scheduling: FCFS

AIM:

To Write A Program to stimulate the Process Scheduling: FCFS
(First Come First Serve)

ALGORITHM:

STEP 1: Set the waiting time of the first process to 0.

STEP 2: For each subsequent process (starting from the second process)

STEP 3: Waiting time = Burst time of the previous process + Waiting time of the previous process.

STEP 4: Initialize Turnaround Time: Turnaround time = Burst time.

STEP 5: Calculate the total waiting time and total turnaround time for all processes.

STEP 6: Average waiting time = Total waiting time / Number of processes

STEP 7: Average turnaround time = Total turnaround time / Number of processes.

STEP 8: Display a table showing the process ID, burst time, waiting time, and turnaround time for each process.

STEP 9: Display the average waiting time and average turnaround time.

STEP 10: Main Function:

- Read the number of processes (n) from the user.
- Initialize arrays for process IDs and burst times.
- Read burst times for each process from the user.

SOURCE CODE

```
#include <stdio.h>
void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
 wt[0] = 0;
 for (int i = 1; i < n; i++) {
 wt[i] = bt[i - 1] + wt[i - 1];
 }
}
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
 for (int i = 0; i < n; i++) {
 tat[i] = bt[i] + wt[i];
 }
}
void findAverageTime(int processes[], int n, int bt[]) {
 int wt[n], tat[n];
 findWaitingTime(processes, n, bt, wt);
 findTurnAroundTime(processes, n, bt, wt, tat);
 float total_wt = 0, total_tat = 0;
 for (int i = 0; i < n; i++) {
 total_wt += wt[i];
 total_tat += tat[i];
 }
 printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
 for (int i = 0; i < n; i++) {
 printf("%d\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
 }
 printf("Average waiting time = %.2f\n", total_wt / n);
 printf("Average turnaround time = %.2f\n", total_tat / n);
}
int main() {
 int n;
 printf("Enter the number of processes: ");
 scanf("%d", &n);
 int processes[n];
 int burst_time[n];
 printf("Enter the burst time for each process:\n");
 for (int i = 0; i < n; i++) {
 processes[i] = i + 1;
 scanf("%d", &burst_time[i]);
 }
 findAverageTime(processes, n, burst_time);
 return 0;
}
```

Output :

```
stud@YSA:~/ysa$ ./tfcfs1
Enter the number of processes: 3
Enter the burst time for each process:
45
56
67
Process        Burst Time     Waiting Time   Turnaround Time
1                  45              0              45
2                  56              45             101
3                  67              101            168
Average waiting time = 48.67
Average turnaround time = 104.67
```

RESULT:

The above program is completed Successfully.

Ex.No:6
Date: 18-09-2024

6. Process Scheduling: Least Frequency Used

AIM:

To Write A Program to stimulate the Process Scheduling: Least Frequently Used.

ALGORITHM:

STEP 1: Create a structure Process with fields for process ID and frequency of execution.

STEP 2: Print the LFU scheduling order.

STEP 3: Run an infinite loop:

- Initialize min_frequency to -1 and selected_index to -1.
- Iterate through each process:
- If the process frequency is greater than 0 and either min_frequency is -1 or the process frequency is less than min_frequency:
- Update min_frequency and selected_index with the current process information.
- If selected_index is still -1, break the loop (all processes have run at least once).
- Print a message indicating the running process.
- Decrement the frequency of the selected process.

STEP 4: Main Function:

- Read the number of processes (n) from the user.
- Initialize an array of Process structures (processes).
- Set the process ID and read the frequency from the user for each process.

SOURCE CODE

```
#include <stdio.h>

struct Process {
    int id;
    int frequency;
};

void lfu_schedule(struct Process processes[], int n) {
    printf("LFU Scheduling Order:\n");

    while (1) {
        int min_frequency = -1;
        int selected_index = -1;

        for (int i = 0; i < n; i++) {
            if (processes[i].frequency > 0 && (min_frequency == -1 || processes[i].frequency <
min_frequency)) {
                min_frequency = processes[i].frequency;
                selected_index = i;
            }
        }

        if (selected_index == -1) {
            break; // All processes have run at least once
        }

        printf("Running Process %d\n", processes[selected_index].id);
        processes[selected_index].frequency--;
    }
}

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

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter frequency for Process %d: ", i + 1);
        scanf("%d", &processes[i].frequency);
    }

    lfu_schedule(processes, n);

    return 0;
}
```

Output :

```
stud@YSA:~/ysa$ ./tlfu
Enter the number of processes: 3
Enter frequency for Process 1: 3
Enter frequency for Process 2: 2
Enter frequency for Process 3: 4
LFU Scheduling Order:
Running Process 2
Running Process 2
Running Process 1
Running Process 1
Running Process 1
Running Process 3
Running Process 3
Running Process 3
Running Process 3
```

RESULT:

The above program is completed Successfully.

Ex.No: 7

7. Process Scheduling: Round Robin

Date: 20-09-2024

AIM:

To Write A Program to stimulate the Process Scheduling: Round Robin.

ALGORITHM:

STEP 1: Read the number of processes (n) and time quantum from the user.

STEP 2: Initialize an array of Process structures.

STEP 3: For each process:

- Set the process ID, burst time, and remaining time.
- Print the Round Robin scheduling order.Initialize remaining_processes to the total number of processes and current_time to 0.
- Run a loop until all processes have completed.Iterate through each process
- If the process has remaining time:Calculate the execution time for the process.
- Update the remaining time and current time.

STEP 4: Print a message indicating the process execution for the time slice.

STEP 5: If the process completes,decrement remaining_processes.

SOURCE CODE

```
#include <stdio.h>

struct Process {
    int id;
    int burst_time;
    int remaining_time;
};

void round_robin_schedule(struct Process processes[], int n, int quantum) {
    printf("Round Robin Scheduling Order:\n");

    int remaining_processes = n;
    int current_time = 0;

    while (remaining_processes > 0) {
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                int execute_time = (processes[i].remaining_time < quantum) ?
processes[i].remaining_time : quantum;
                processes[i].remaining_time -= execute_time;
                current_time += execute_time;

                printf("Process %d for time slice %d\n", processes[i].id, execute_time);

                if (processes[i].remaining_time == 0) {
                    remaining_processes--;
                  }             }          }      } }

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

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

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }

    round_robin_schedule(processes, n, quantum);

    return 0;
}
```

Output :

```
stud@YSA:~/ysa$ ./trr
Enter the number of processes: 3
Enter the time quantum: 10
Enter burst time for Process 1: 20
Enter burst time for Process 2: 25
Enter burst time for Process 3: 35
Round Robin Scheduling Order:
Process 1 for time slice 10
Process 2 for time slice 10
Process 3 for time slice 10
Process 1 for time slice 10
Process 2 for time slice 10
Process 3 for time slice 10
Process 2 for time slice 5
Process 3 for time slice 10
Process 3 for time slice 5
```

RESULT:

The above program is completed Successfully.

Ex.No:8

Date: 20-09-2024

8. Two Process Mutual Exclusion

AIM:

To Write A Program to stimulate the Two Process Mutual Exclusion.

ALGORITHM:

STEP 1: Declare and initialize a mutex using pthread_mutex_t mutex.

STEP 2: Create two process functions (process1 and process2).

STEP 3: In each process function, run a loop for a specified number of iterations(e.g., 5 times).

STEP 4: Lock the mutex using pthread_mutex_lock.

STEP 5: Print a message indicating entry into the critical section.

STEP 6: Simulate some work in the critical section (e.g., sleep(1)).

STEP 7: Print a message indicating exit from the critical section.

STEP 8: Unlock the mutex using pthread_mutex_unlock.

STEP 9: Simulate non-critical section work (e.g., sleep(1)).

STEP 10: Main Function:

- Declare two thread variables (thread1 and thread2).
- Initialize the mutex using pthread_mutex_init.Create two threads using pthread_create, each running one of the process functions.
- Wait for both threads to finish using pthread_join.
- Destroy the mutex using pthread_mutex_destroy.

SOURCE CODE

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

pthread_mutex_t mutex;

void* process1(void* arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutex);
        printf("Process 1: In critical section\n");
        // Simulate some work
        sleep(1);
        printf("Process 1: Exiting critical section\n");
        pthread_mutex_unlock(&mutex);
        // Simulate non-critical section work
        sleep(1);
    }
    return NULL;
}

void* process2(void* arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutex);
        printf("Process 2: In critical section\n");
        // Simulate some work
        sleep(1);
        printf("Process 2: Exiting critical section\n");
        pthread_mutex_unlock(&mutex);
        // Simulate non-critical section work
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_mutex_init(&mutex, NULL);

    pthread_create(&thread1, NULL, process1, NULL);
    pthread_create(&thread2, NULL, process2, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    pthread_mutex_destroy(&mutex);

    return 0;
```

Output :

```
stud@YSA:~/ysa$ ./tme
Process 1: In critical section
Process 1: Exiting critical section
Process 2: In critical section
Process 2: Exiting critical section
Process 1: In critical section
Process 1: Exiting critical section
Process 2: In critical section
Process 2: Exiting critical section
Process 1: In critical section
Process 1: Exiting critical section
Process 2: In critical section
Process 2: Exiting critical section
Process 1: In critical section
Process 1: Exiting critical section
Process 2: In critical section
Process 2: Exiting critical section
Process 1: In critical section
Process 1: Exiting critical section
Process 2: In critical section
Process 2: Exiting critical section
```

RESULT:

The above program is completed Successfull

SHELL PROGRAMS - OS

09. Checking Read, Write, Execute Permissions

Ex no: 09
Date: 25-09-2024

Shell script that displays a list of all the files in the current directory to which the user has read, write and execute permissions.

Aim:

A Shell Program to Perform read, write and execute permissions.

Algorithm:

Step 1: Print a header message.
Step 2: Iterate over each item in the current directory.
Step 3: Check if the item is a regular file.
Step 4: If it is a regular file, check if it has read, write, and execute permissions.
Step 5: If all permissions are present, display detailed information about the file.

Program:

```
echo "Files with read, write, and execute permissions in the current directory:"
echo "------------------------------------------------------------------------------------------" for file in *;
do    if [ -f "$file"] && [ -r "$file"] && [ -w "$file"] && [ -x "$file"]; then

#if [ -f "$file"] && [ -w "$file"] && [ -x "$file"]; then
echo "$file"    fi done
```

Output:

```
Files with read, write & execute permissions in the current directory
--------------------------------------------------------------------------------------------
Ex07.sh ~
Ex08.sh
Ex09.sh ~
Ex10.sh
Ex11.sh ~
```

Result:

The above program is executed successfully.

10. “Cat” Command & “Wc” Command

Ex no: 10
Date: 25-09-2024

Shell program to simulate cat’ command and wc command to count the number of lines and number of words in the given input file.

Aim:

A Shell Program to Perform “cat” command & “wc” command to count the number of lines & number of words

Algorithm:

```
Step 1: If the number of command-line arguments is not equal to 1,
        print a usage message and exit the script.
Step 2: Assign the first command-line argument to the variable filename.
Step 3: If the specified file does not exist, print an error message &
        exit the script.
Step 4: Print a message indicating that the script will display the contents
        of the file.
Step 5: Simulate the 'cat' command by using echo to display the contents
        of the file.
Step 6: Use the 'wc' command to count the number of lines and words in
        the file.
Step 7: Assign the line count and word count to variables line_count
        and word_count.
Step 8: Print the number of lines and words in the file.
```

Program:

```
if [ $# -ne 1]; then
echo "Usage:$0<input_file>"
exit 1
```

```
 fi
 input_file="$1"

if [! -f "$input_file"]; then
echo "Error: File not found: $input_file"
 exit 1
 fi
# Simulate 'cat' command
echo "Contents of $input_file:"
cat "$input_file"
# Simulate 'wc' command to count lines and words # Souban Aadi
echo -e " Shell program to simulate cat' command and \n wc command to
count the number of lines, \n number of words in the given input file."
echo -e "\n"
 lines=$(wc -l < "$input_file")
words=$(wc -w < "$input_file")
mxdiswd=$(wc -L < "$input_file")
charcount=$(wc -m < "$input_file")
bytecount=$(wc -c < "$input_file")
echo -e "Number of lines in $input_file: \t $lines"
 echo -e "Number of words in $input_file: \t $words"
echo -e "Number of Maximum Display Width in $input_file: \t $mxdiswd"
echo -e "Number of Character Count in $input_file: \t $charcount"
echo -e "Number of Byte Count in $input_file: \t $bytecount"
```

Output:

```
I love India
MCA is the professional course
Students are good in manner
A batch is best forever
```

```
Number lines in the input_file.txt :3
Number of words in the input_file.txt: 18
Number of max display Width in input_file.txt: 20
Number of Character Count in input_file.txt: 50
Number of Byte Count in input_file.txt: 50
```

Result:

The above program is executed successfully.

11. Grep Command

Ex no: 11
Date: 27-09-2024

Grep command that Counts the number of blank lines in the file1 and Select the lines from the file1 that have the string, "UNIX".

Aim:

A Shell Program to Perform “grep” command that Count the number of blank lines & find a String “UNIX”

Algorithm:

Step 1: If the number of command-line arguments is not equal to 1, print a usage message and exit the script.
Step 2: Assign the first command-line argument to the variable filename.
Step 3: If the specified file ($filename) does not exist, print an error message and exit the script.
Step 4: Use the grep command to count the number of blank lines in the file.
Step 5: Assign the count to the variable blank_line_count.
Step 6: Print the number of blank lines in the specified file.
Step 7: Use the grep command to select lines containing the string "UNIX" in the file.
Step 8: Display the selected lines.

Program:

```
echo " The Number of blank lines in the given file"
grep -c '^$' file1.txt #$ echo -e "\n" grep -c '^$'
swap.sh
#This command searches for lines that consist only of the beginning of the line ^
followed by the end of the line $, which represents blank lines. The -c option is used to
count the matching lines @ Souban Aadi.
```

```
echo "The Given String is: " grep "UNIX" file1.txt

#This command searches for lines containing the string "UNIX" in file1 and prints those
lines to the terminal.
```

Output:

```
>> souban@Lenovo-Ubuntu:-S . /ex11.sh
   The Number of blank lines in the given file: 2


   The Given String is:
   Souban is an UNIX User
   UNIX
   Unix
```

Result:

The above program is executed successfully.

12. SED Command

Ex no: 12
Date: 27-09-2024

Sed command that Print number of lines beginning with “O” and swap the first and second word in each line in the file

Aim:

A Shell Program to Print number of lines beginning with “O” and swap the first and second word

Algorithm:

Step 1: For each line in the text file, iterate the following steps.
Step 2: Identify the first word and the second word in the line.
Step 3: Use a regular expression or a string split operation to isolate words.
Step 4: Swap the positions of the first and second words.
Step 5: Display the modified lines with the first and second words swapped.

Program:

```
# lines that beginning with “O”,
echo "sed command that Print lines numbers of lines beginning with o" sed -n
'/^[Oo]/{=;p}' sedo.txt

# Swap the first & second word

echo "swap the first and second word in each line in the file" sed 's/\([^ ]*\)
\([^ ]*\)/ \2 \1/' sedo.txt
```

Output:

```
sed command that Print lines numbers of lines beginning with o
OS is an Elective Subject
swap the first and second word in each line in the file
is OS an Elective Subject
```

Result:

The above program is executed successfully.

13. AWK Script

Ex no: 13
Date: 28-09-2024

Awk script to Count the number of lines in a file that do not contain vowels and find the number of characters, words and lines in a file

Aim:

A Shell Program to count the num of lines in a file that do not contain vowels & find the number of characters, words and lines

Algorithm:

```
Step 1: Initialize count to 0.
Step 2: Open the file for reading.
Step 3: For each line in the file:
Step 4: If the line does not contain any vowels (A, E, I, O, U, a, e, i, o, u):
Step 5: Increment count by 1.
Step 6: Close the file.
Step 7: Print the final count.
```

- Find the number of characters, words, and lines in a file:

```
Step 1: Initialize char_count, word_count, and line_count to 0.
Step 2: Open the file for reading.
Step 3: For each line in the file:
Step 4: Increment char_count by the length of the line.
Step 5: Increment word_count by the number of words in the line.
Step 6: Increment line_count by 1.
Step 7: Close the file.
Step 8: Print the final char_count, word_count, and line_count.
```

Program:

```
echo “awk script to Count the number of lines in a file that do not contain vowels”
awk BEGIN { count = 0 }
 {
 if ($0 !~ /[AEIOUaeiou]/) {
count++
   }
   }
   END {
 print "Number of lines without vowels: " count
 }' vowoff.txt
echo “awk script to find the number of characters, words and lines in a file”
awk '{
 char_count +=
length word_count
+= NF
line_count++ }
END {
 print "Number of characters: "
char_count print "Number of words: "
word_count print "Number of lines: "
line_count
}' vowoff.txt
```

Output:

```
Number of lines without vowels: 3
Number of characters: 256 Number
of words: 12
Number of lines: 34
```

Result:

The above program is executed successfully.

14. Prime Number

Ex no: 14
Date: 28-09-2024

Shell script to find out whether the given number is prime number or not

Aim:

A Shell Program check whether the given number is prime or not

Algorithm:

Step 1: The function is_prime takes a number as a parameter.
Step 2: If the input number is less than or equal to 1, print that it is not a prime number and return.
Step 3: Use a for loop with a loop variable i starting from 2.
Step 4: Continue the loop as long as the square of i is less than or equal to the input number.
Step 5: In each iteration, check if the input number is divisible evenly by i.
Step 6: If it is divisible, print that the number is not prime and return.
Step 7: If no divisor is found in the loop, print that the number is prime.
Step 8: Read and store the input number in a variable.
Step 9: Pass the user-input number to the is_prime function for primality checking.

Program:

```
echo "Enter a number:"
read number
i=2
if [ $number -le 2 ]
then
   echo "$number is not a prime number."
   exit
fi
while [ $i -lt $number ]
do
   if [ `expr $number % $i` -eq 0 ]
  then
      echo "$number is not a prime number."
      exit
 fi
 i=`expr $i + 1`
done
echo "$number is a prime number."
```

Output:

```
>> souban@Lenovo-Ubuntu:-S . /ex14.sh
      Enter a Number:
      2 is not a prime number
      3
      is a prime number
```

Result:

The above program is executed successfully.

15. Factorial

Ex no: 15
Date: 04-10-2024

Shell program to find out factorial of the given number

Aim:
A Shell Program to find out factorial of the given number

Algorithm:

Step 1: Prompt the user to enter a number and store it in a variable num.
Step 2: Initialize a variable fact to 1.
Step 3: Use a loop to iterate from 1 to num.
Step 4: Multiply the current value of fact by the loop variable in each iteration.
Step 5: Display the calculated factorial (fact)

Program:

```
echo "Enter the No. for the FACTORIAL
Value:" read num fact=1
for((i=2;i<=num;i++))
 {
        fact=$(($fact*$i))
 }
 echo $fact
```

Output:

```
>> souban@Lenovo-Ubuntu:-S . /ex15.sh Enter
 Your Value:
 4
 24

>> souban@Lenovo-Ubuntu:-S . /ex15.sh Enter
 Your Value:
 8
 40320
```

Result:

The above program is executed successfully.

16. Palindrome

Ex no: 16
Date: 04-10-2024

Shell program to find out reverse string of the given string and check the given string is palindrome or not

Aim:

A Shell Program to check the given string is palindrome or not

Algorithm:

Step 1: Prompt the user to enter a string.
Step 2: Read and store the input string in a variable.
Step 3: Use a loop to iterate through each character of the input string in reverse order.
Step 4: Append each character to the temp variable.
Step 5: Compare the original string (a) with its reversed version (temp).
Step 6: If they are equal, the string is a palindrome.
Step 7: If not, the string is not a palindrome.
Step 8: Display whether the input string is a palindrome or not based on the comparison result.

Program:

```
echo "Enter a string: " read
original_string
# Function to reverse a string # Souban Aadi
reverse_string() { local string="$1" local
reversed="" local len=${#string} for ((
i=len-1; i>=0; i-- )); do
reversed="$reversed${string:$i:1}" done
echo "$reversed"
}
```

```
reversed=$(reverse_string "$original_string")
if [ "$original_string" == "$reversed" ]; then
echo "The string is a palindrome." else
 echo "The string is not a palindrome." Fi
```

Output:

```
>> souban@Lenovo-Ubuntu:-S . /ex16.sh
     Enter a string:
     mom
      The string is a palindrome.
```

Result:

The above program is executed successfully.

17. Searching an Element

Ex no: 17
Date: 09-10-2024

Shell script to Search an element in the list

Aim:

A Shell Program to Search an element in the given list

Algorithm:

Step 1: Initialize the list (my_list) with elements and the search element (search_element).
Step 2: Initialize a flag (element_found) to 0.
Step 3: Iterate through each item in the list using a loop.
Step 4: If the current item matches the search element, set the flag to 1 and break out of the loop.
Step 5: Check the flag value after the loop.
Step 6: If the flag is 1, display that the search element was found.
Step 7: If the flag is 0, display that the search element was not found.

Program:

```
# Define an array (list)
my_list=("apple" "banana" "cherry" "date" "fig")

# Element to search for
search_element="date"

# Flag to indicate if the element was found
 found=false

# Iterate through the array # Souban Aadi
for element in "${my_list[@]}"; do
  if [ "$element" == "$search_element" ];then
     found=true
     break
  fi
done
```

```
# Check if the element was found and display the result
if [ "$found" == true ]; then
   echo "Element '$search_element' found in the list."
else
   echo "Element '$search_element' not found in the list."
Fi
```

Output:

```
>> souban@Lenovo-Ubuntu:-S . /ex17.sh
Element 'apple' found in the list.
```

Result:

The above program is executed successfully.

18. Menu

Ex no: 18
Date: 09-10-2024

Shell script to implement menu driven program to display list of users who are currently working in the system, copying files (cp command), rename a file, list of files in the directory and quit option. (Hint: use case structure)

Aim:

A Shell Program to Perform list of operations

Algorithm:

Step 1: Begin an infinite loop using while true.
Step 2: Inside the loop, display a menu with the following options:
List users currently working o
Copy a file o Rename a file o
List files in a directory o Quit
Step 3: Prompt the user to enter their choice and read the input into a variable.
Step 4: Use a case statement to perform actions based on the user's choice.
Step 5: For each choice, execute the corresponding block of code:
Option 1: List users currently working using the who command.
Option 2: Prompt for source and destination file, then copy the source file to the destination.
Option 3: Prompt for the current and new file names, then rename the file.
Option 4: Prompt for a directory path and list files in the directory using the ls command.
Option 5: Print a goodbye message and exit the script.
Step 6: If the user enters an invalid choice, print a message indicating that and prompt the user to try again.
Step 7: After executing the chosen action or handling an invalid choice, the loop repeats to display the menu again.

Program:

```
while true; do
clear
   echo "Menu Options:"
   echo "1. Display list of users currently working"
echo "2. Copy a file"
 echo "3. Rename a file"
 echo "4. List files in the directory"
echo "5. Quit"
   read -p "Enter your choice (1/2/3/4/5): " choice
#The read -p command is used to prompt the user for input and store the input in the
variable choice. # Souban Aadi      case $choice in
 1)
         read -p "Press Enter to continue..."
          ;;
      2)
          read -p "Enter the source file: " source
read -p "Enter the destination file: " destination
cp "$source" "$destination"            echo "File copied
successfully!"
          read -p "Press Enter to continue..."
          ;;
      3)
          read -p "Enter the file to rename: " old_name
read -p "Enter the new name: " new_name
mv "$old_name" "$new_name"            echo "File
renamed successfully!"
          read -p "Press Enter to continue..."
          ;;
      4)
          read -p "Enter the directory path: " dir_path
          ls "$dir_path"
          read -p "Press Enter to continue..."
          ;;
      5)
          echo "Exiting the program. Goodbye!"
exit 0
          ;;
      *)
          echo "Invalid choice. Please select a valid option (1/2/3/4/5)."
read -p "Press Enter to continue..."
          ;;
esac done
```

Output:

```
>> souban@Lenovo-Ubuntu:-S . /ex18.sh
    Menu Options:
         1. Display list of users currently
            working
         2. Copy a file
         3. Rename a file
         4. List files in the directory
         5. Quit
    Enter your choice (1/2/3/4/5):
```

Result:

The above program is executed successfully.