**LABORATORY RECORD**

**OPERATING SYSTEM**

**Student Name- R MANOJ KUMAR**

**Student Regn. No.- 23BCY10293**

**Interim Semester 2024-25**

**Slot:**

**Year- 2025-2026**

**Class No:**

**Faculty Name- Dr. SUNEET JOSHI**

**School of Computing Science & Engineering**

**VIT Bhopal University**

A blue and white logo

Description automatically generated

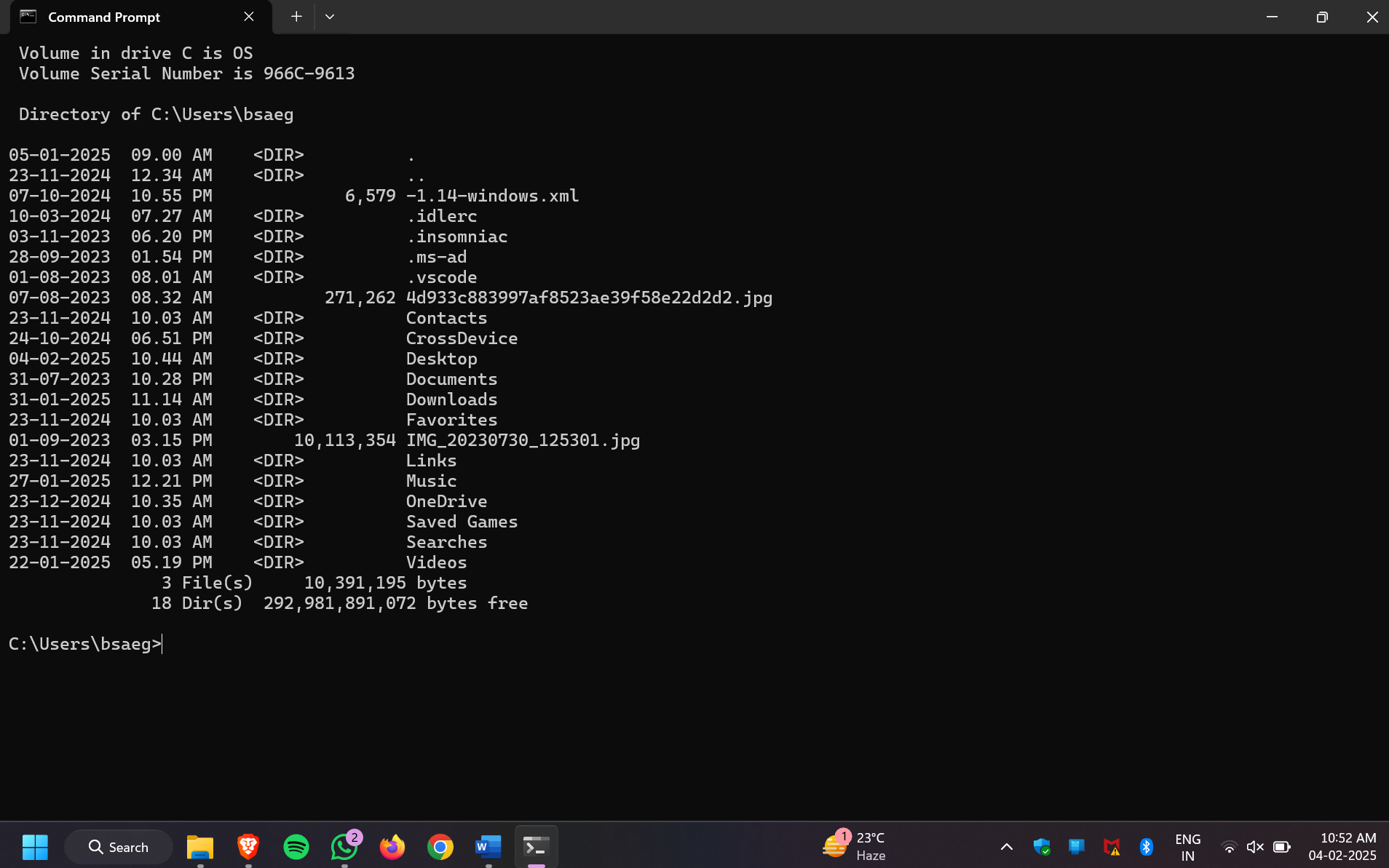
|  |  |  |
| --- | --- | --- |
| S.NO | Topic | Date |
| 1) | MS-DOS | 21.03.2025 |
| 2) | C program to create a clone of current process. | 18.03.2025 |
| 3) | write a C program to create a new process using fork(). | 18.03.2025 |
| 4) | write a C program to Implement a parent child process relationship and demonstrates process termination using fork() and wait() system calls. | 18.03.2025 |

INDEX

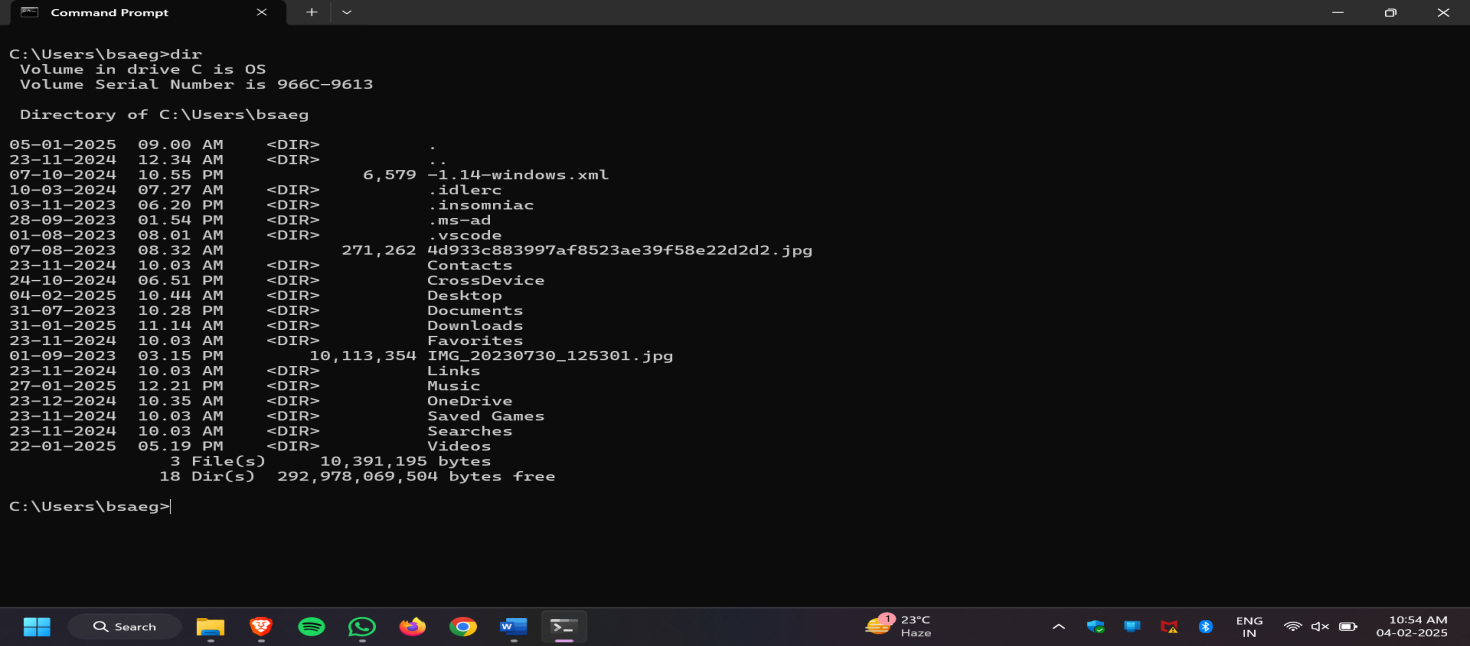
EXPERIMENT-1 :MS-DOS(i.e) Microsoft disk operating system is a command line based OS that allows user to interact with the computer through the text based commands .

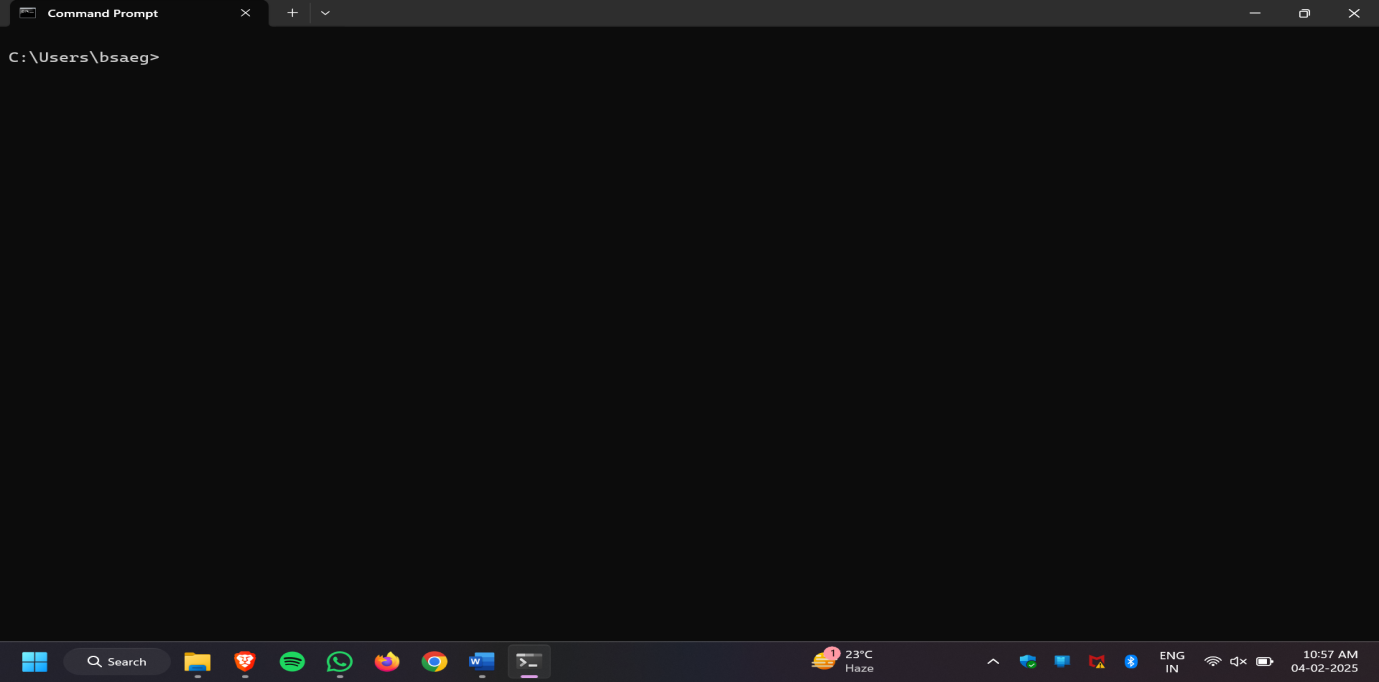
COMMANDS:

1.dir;Display the list of files and directories in a folder.

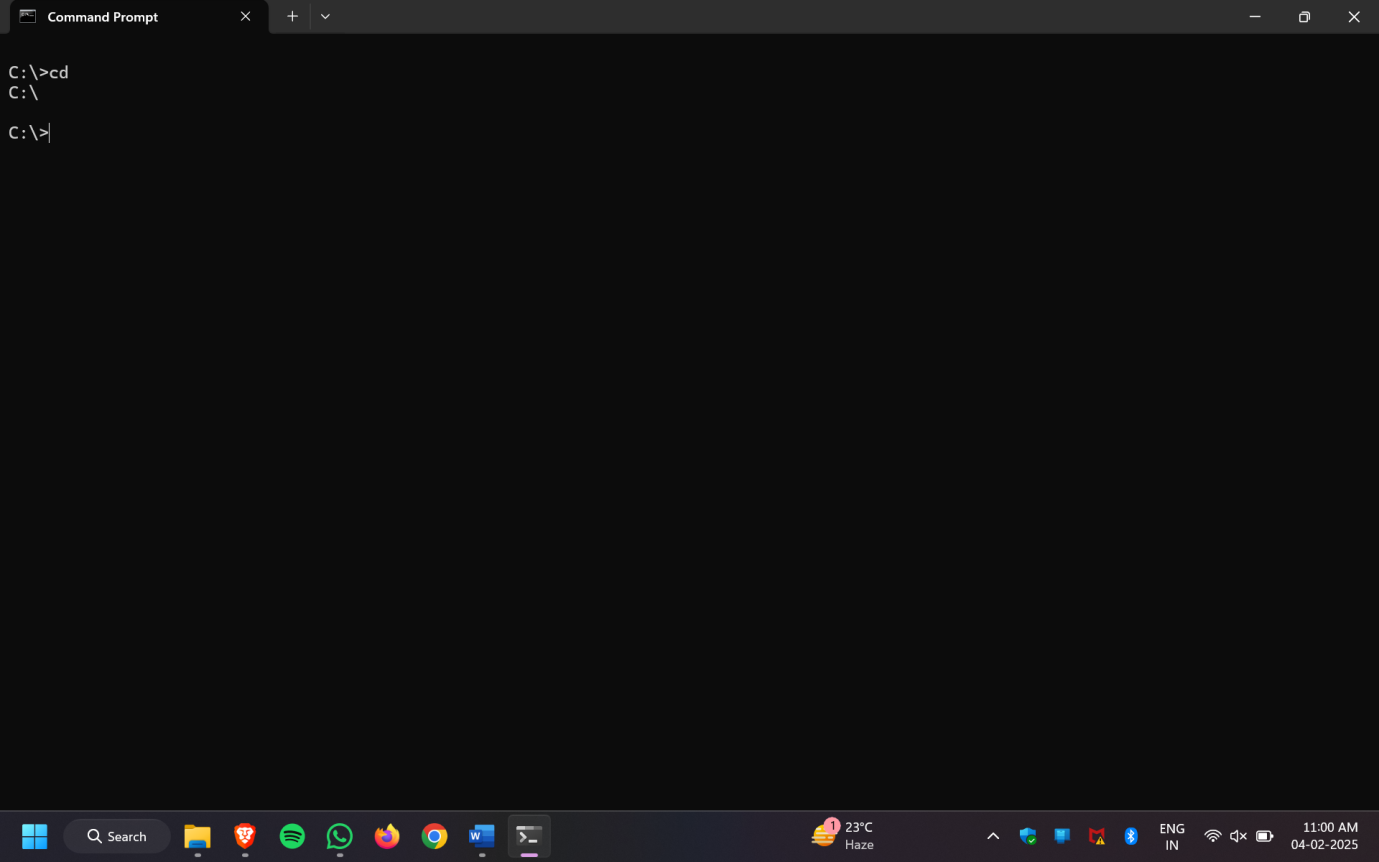


2.cls;clears the command prompt screen.

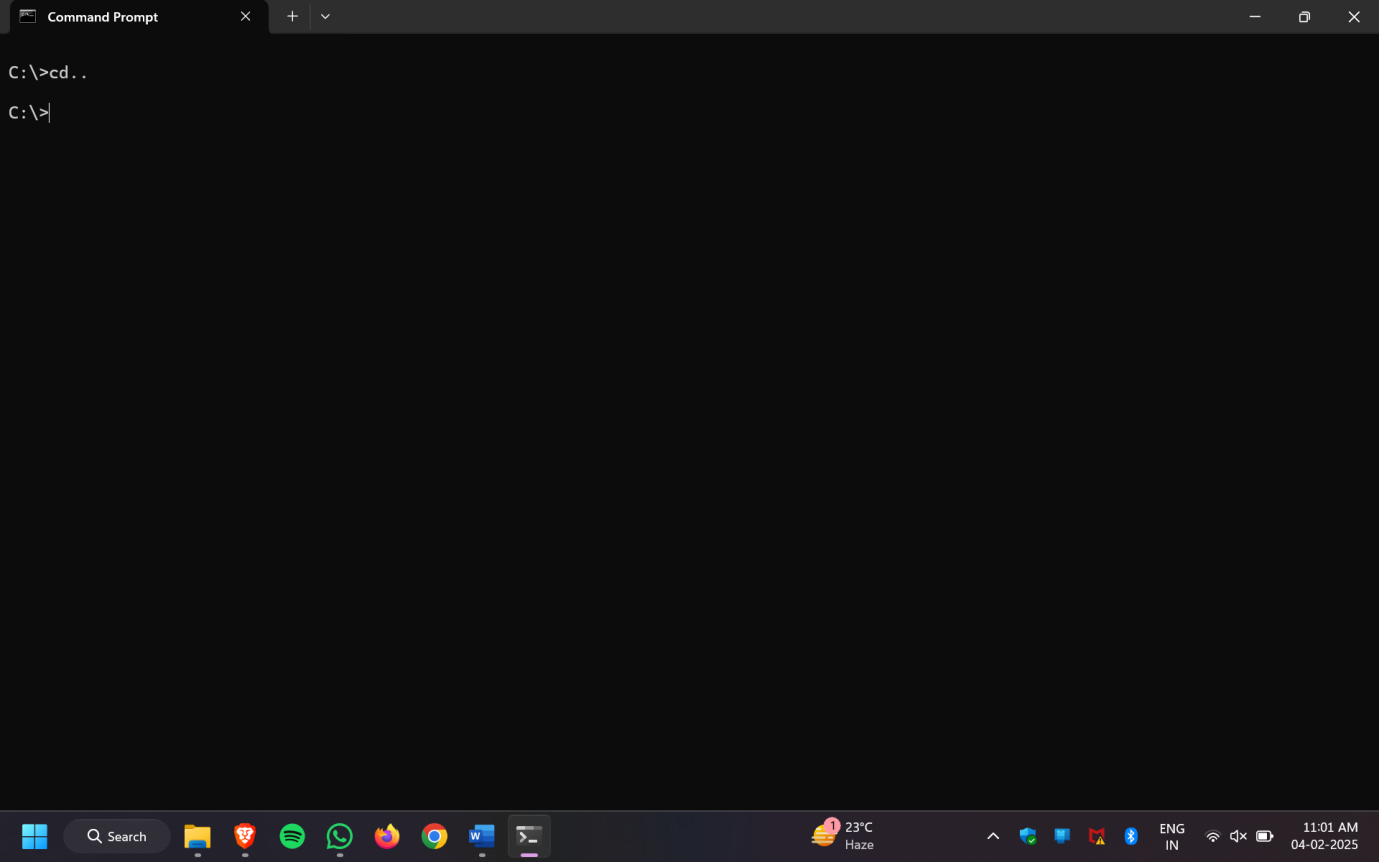




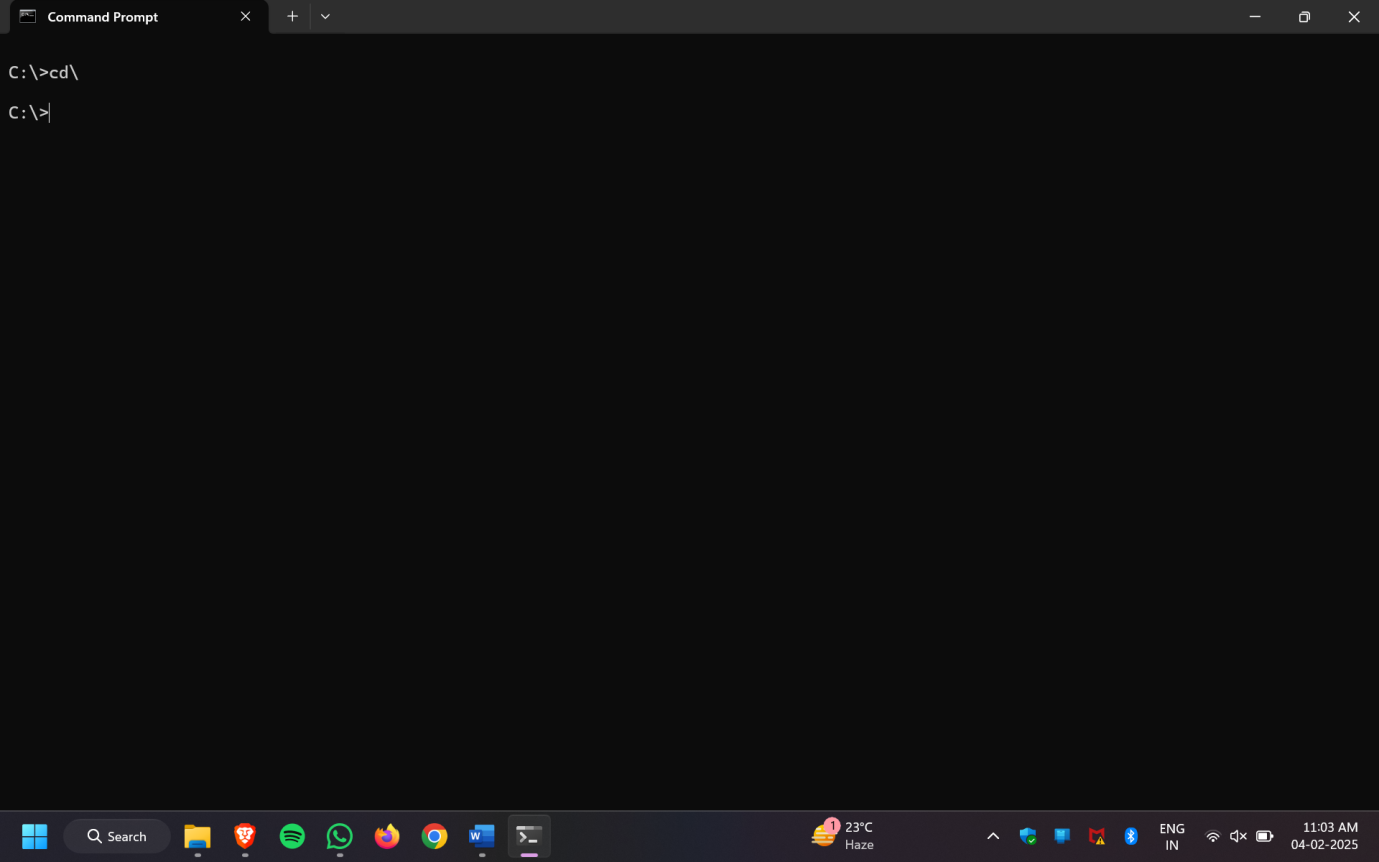
3.cd;changes the current working directory .



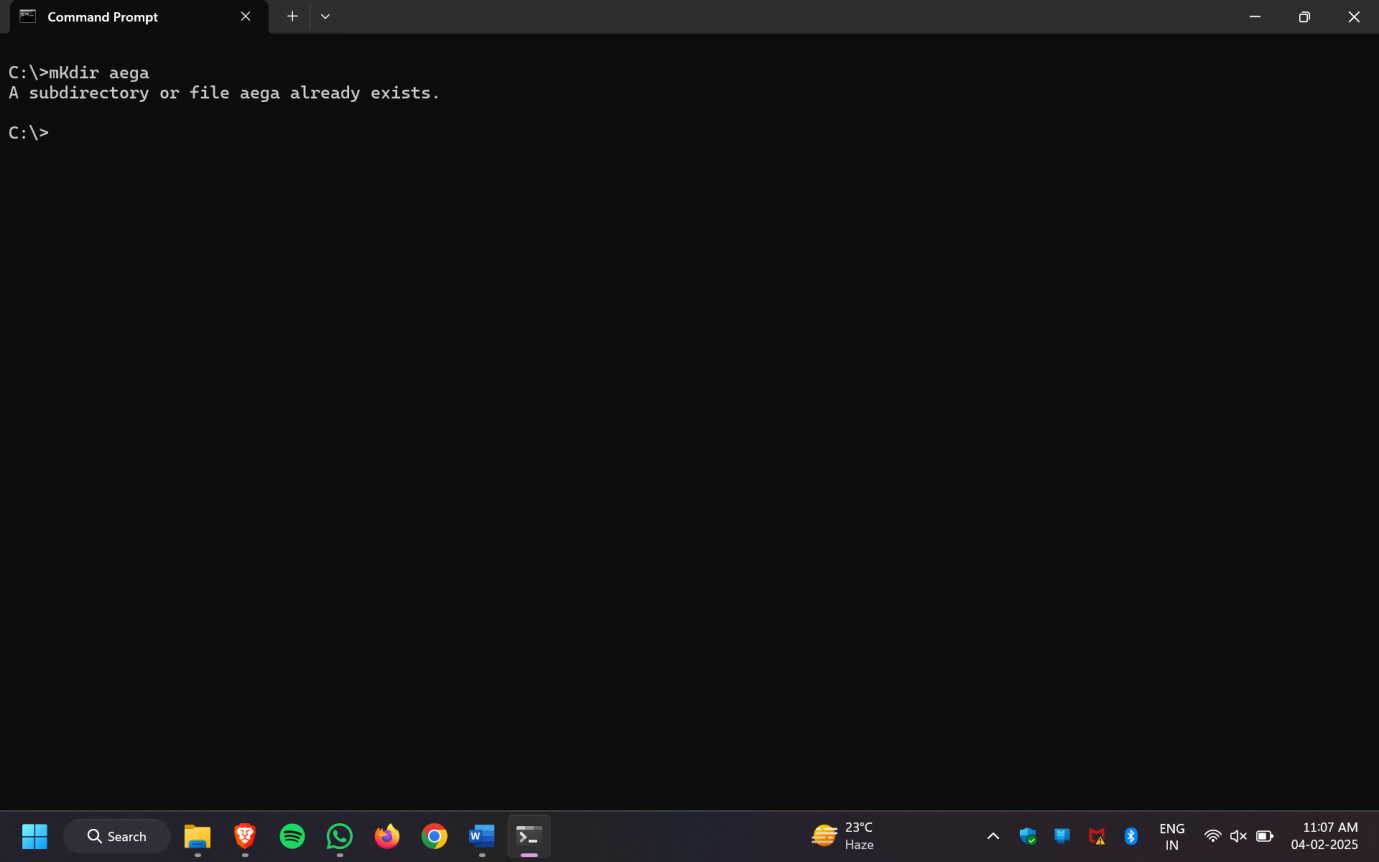
4.cd..; goes one level back .



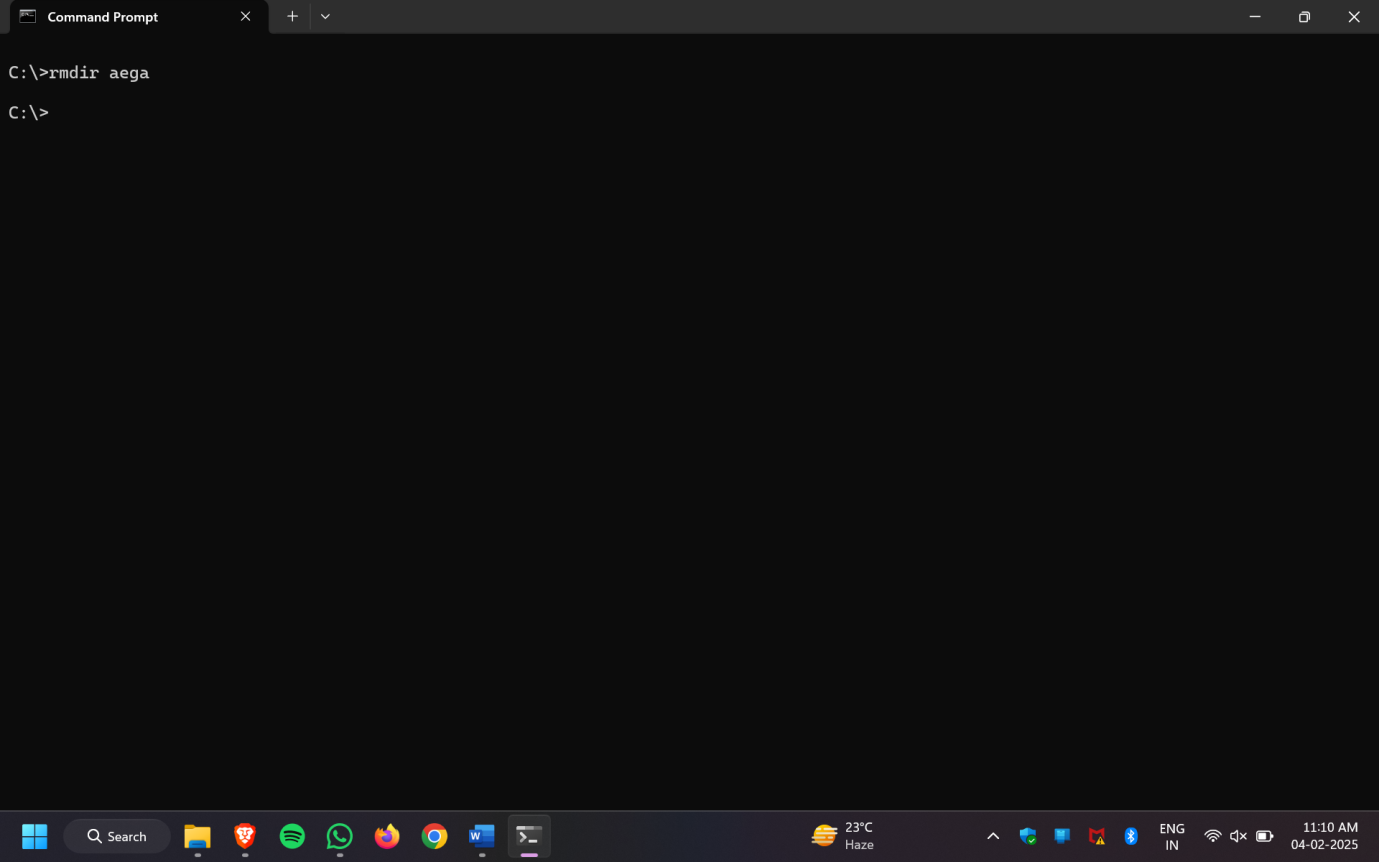
5.Cd\; To go back to the root directory.



6.Mkdir (name) ;creates new folder .



7.rmdir(name)/rd; It is used to remove the directory.

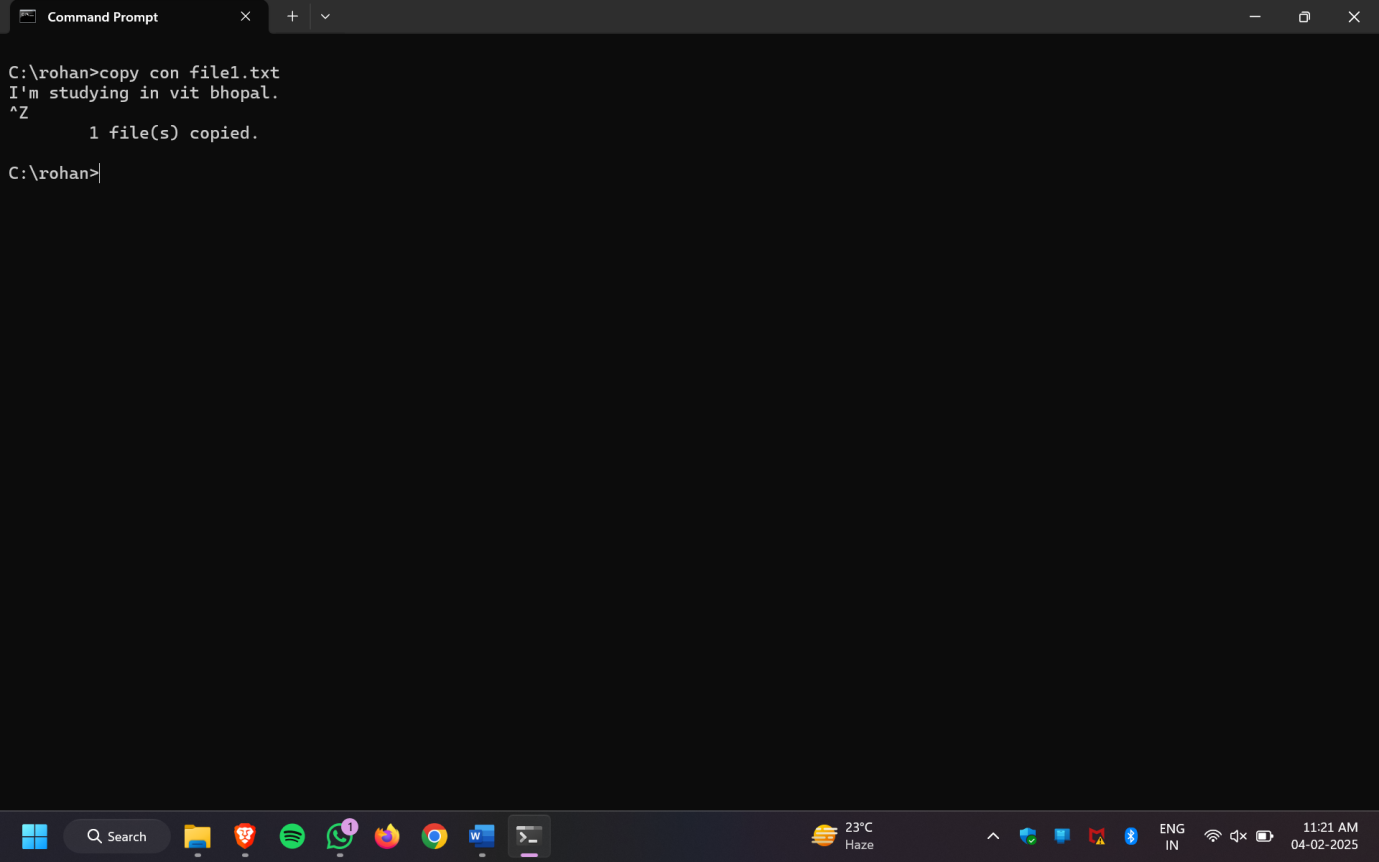


8.date/time; Returns current date .



FILE MANAGEMENT COMMANDS:

1.copy con file 1 ;



2.del;deletes a specific file .

3.ren(oldname)(new name ):used to rename a new name for folder or file in the os.

4.Type orange.txt; displays the already existing contents .

Experiment-2 C program to create a clone of current process.

PROGRAM:

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid;

// Create a new process

pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

return 1;

}

else if (pid == 0) {

// Child process

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

}

else {

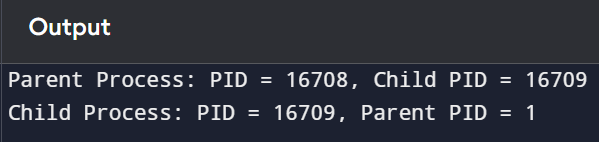
// Parent process

printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);

}

return 0;

}



EXPLANANTION:

Whenever program get a 4th-statement then at the same instance a clone of it’s is created and start executing from there only in both the process (i.e) parent and child :

1. the child process prints it’s pid and it’s parents pid then terminates.

(ii) the parent process waits for the child to finish checks it’s exit status and then continues with it’s termination.

(iii) get pid – returns calling process ID.

Experiment-3 write a C program to create a new process using fork().

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid;

// Creating a new process

pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(1);

}

else if (pid == 0) {

// Child process

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

}

else {

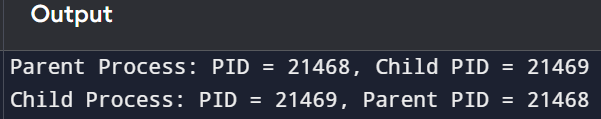
// Parent process

printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);

}

return 0;

}



EXPLANANTION:

The fork() system call creates a new process by duplicating the calling process after calling fork the chils gets the return value if 0 and the parent process receives the process id (i.e) pid of the child .

Experiment-4 write a C program to Implement a parent child process relationship and demonstrates process termination using fork() and wait() system calls.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

// Creating a new process

pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(1);

}

else if (pid == 0) {

// Child process

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

sleep(2); // Simulating work in the child process

printf("Child Process (PID = %d) terminating...\n", getpid());

exit(0); // Child terminates

}

else {

// Parent process

printf("Parent Process: PID = %d, Waiting for Child (PID = %d) to finish...\n", getpid(), pid);

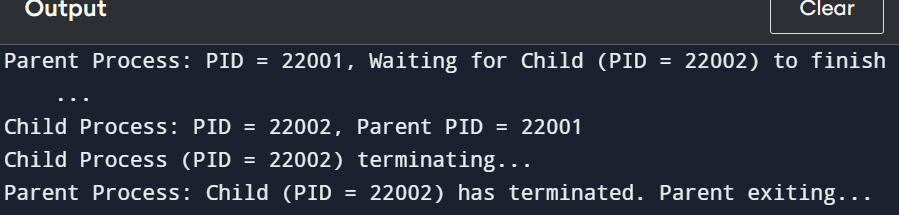
wait(NULL); // Parent waits for child to terminate

printf("Parent Process: Child (PID = %d) has terminated. Parent exiting...\n", pid);

}

return 0;

}



EXPLANANTION:

1) fork() creates a new child process.

2) Child Process: 🡪Prints its PID and Parent's PID. 🡪Simulates work using sleep (2). 🡪Terminates using exit (0).

3) Parent Process: 🡪Waits for the child to finish using wait (NULL). 🡪Once the child exits, the parent resumes execution and exits.

Experiment-5 write a C program to Simulate First Come First Serve CPU Scheduling algorithm

PROGRAM:

#include <stdio.h>

// Structure to hold process details

struct Process {

int pid; // Process ID

int at; // Arrival Time

int bt; // Burst Time

int ct; // Completion Time

int tat; // Turnaround Time

int wt; // Waiting Time

};

// Function to sort processes based on Arrival Time

void sortByArrival(struct Process p[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (p[j].at > p[j + 1].at) {

struct Process temp = p[j];

p[j] = p[j + 1];

p[j + 1] = temp;

}

}

}

}

// Function to calculate Completion Time, Turnaround Time, and Waiting Time

void calculateTimes(struct Process p[], int n) {

int currentTime = 0;

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

if (currentTime < p[i].at) {

currentTime = p[i].at; // CPU remains idle if no process arrives

}

p[i].ct = currentTime + p[i].bt; // Completion Time

p[i].tat = p[i].ct - p[i].at; // Turnaround Time

p[i].wt = p[i].tat - p[i].bt; // Waiting Time

currentTime = p[i].ct; // Update current time

}

}

// Function to display the process table

void displayProcesses(struct Process p[], int n) {

printf("\nPID\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");

printf("---------------------------------------------------------------\n");

float totalTAT = 0, totalWT = 0;

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

printf("%d\t%d\t%d\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);

totalTAT += p[i].tat;

totalWT += p[i].wt;

}

printf("\nAverage Turnaround Time: %.2f", totalTAT / n);

printf("\nAverage Waiting Time: %.2f\n", totalWT / n);

}

int main() {

int n;

// Input number of processes

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process p[n];

// Input process details

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

printf("\nEnter Arrival Time and Burst Time for Process %d: ", i + 1);

p[i].pid = i + 1; // Assign Process ID

scanf("%d %d", &p[i].at, &p[i].bt);

}

// Sort processes by arrival time

sortByArrival(p, n);

// Calculate Completion, Turnaround, and Waiting times

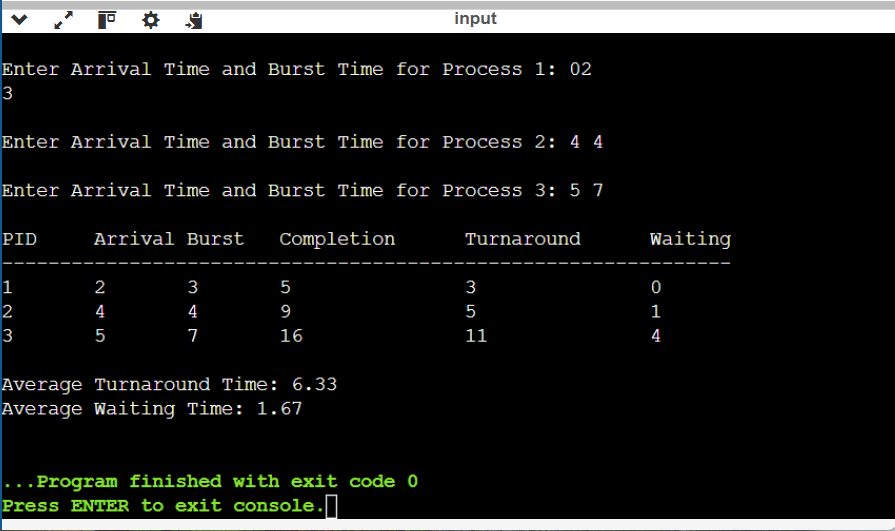
calculateTimes(p, n);

// Display the result

displayProcesses(p, n);

return 0;

}



Experiment-6 Write a C Program to implement shortest Job First CPU Schelduling algorithm (with and without preemption )in OS

C Program for Non-Preemptive SJF

PROGRAM:

#include <stdio.h>

#include <stdbool.h>

struct Process {

int pid, at, bt, ct, tat, wt, completed;

};

// Function to sort processes by arrival time

void sortByArrival(struct Process p[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (p[j].at > p[j + 1].at) {

struct Process temp = p[j];

p[j] = p[j + 1];

p[j + 1] = temp;

}

}

}

}

// Function to calculate Completion, Turnaround, and Waiting times

void sjfNonPreemptive(struct Process p[], int n) {

int completed = 0, currentTime = 0;

while (completed < n) {

int minIndex = -1;

int minBurst = 9999;

// Find process with the shortest burst time that has arrived

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

if (p[i].at <= currentTime && !p[i].completed && p[i].bt < minBurst) {

minBurst = p[i].bt;

minIndex = i;

}

}

if (minIndex == -1) {

currentTime++; // CPU is idle

} else {

p[minIndex].ct = currentTime + p[minIndex].bt;

p[minIndex].tat = p[minIndex].ct - p[minIndex].at;

p[minIndex].wt = p[minIndex].tat - p[minIndex].bt;

p[minIndex].completed = 1;

completed++;

currentTime = p[minIndex].ct;

}

}

}

// Function to display results

void displayResults(struct Process p[], int n) {

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");

printf("---------------------------------\n");

float totalTAT = 0, totalWT = 0;

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

printf("%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);

totalTAT += p[i].tat;

totalWT += p[i].wt;

}

printf("\nAverage Turnaround Time: %.2f", totalTAT / n);

printf("\nAverage Waiting Time: %.2f\n", totalWT / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process p[n];

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

printf("\nEnter Arrival Time and Burst Time for Process %d: ", i + 1);

p[i].pid = i + 1;

p[i].completed = 0;

scanf("%d %d", &p[i].at, &p[i].bt);

}

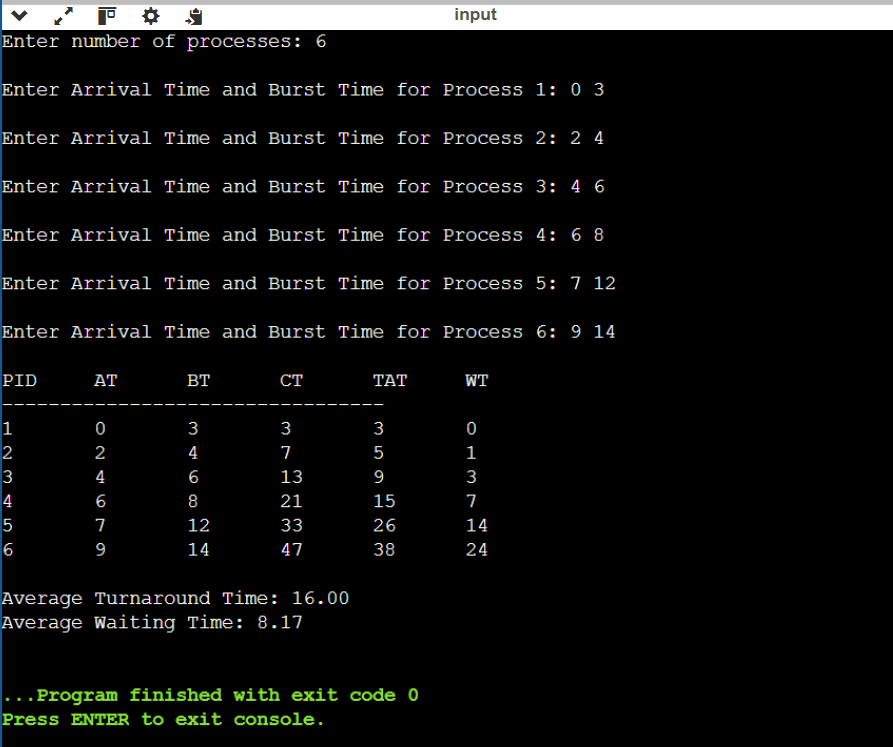
sortByArrival(p, n);

sjfNonPreemptive(p, n);

displayResults(p, n);

return 0;

}



C Program for Preemptive SJF

PROGRAM:

#include <stdio.h>

#include <stdbool.h>

struct Process {

int pid, at, bt, rt, ct, tat, wt;

};

// Function to find the process with the shortest remaining time

int getShortestRemainingProcess(struct Process p[], int n, int currentTime) {

int minIndex = -1;

int minRemainingTime = 9999;

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

if (p[i].at <= currentTime && p[i].rt > 0 && p[i].rt < minRemainingTime) {

minRemainingTime = p[i].rt;

minIndex = i;

}

}

return minIndex;

}

// Function to calculate Completion, Turnaround, and Waiting times

void sjfPreemptive(struct Process p[], int n) {

int completed = 0, currentTime = 0;

while (completed < n) {

int minIndex = getShortestRemainingProcess(p, n, currentTime);

if (minIndex == -1) {

currentTime++; // CPU is idle

} else {

p[minIndex].rt--; // Reduce remaining time

currentTime++;

if (p[minIndex].rt == 0) {

p[minIndex].ct = currentTime;

p[minIndex].tat = p[minIndex].ct - p[minIndex].at;

p[minIndex].wt = p[minIndex].tat - p[minIndex].bt;

completed++;

}

}

}

}

// Function to display results

void displayResults(struct Process p[], int n) {

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");

printf("---------------------------------\n");

float totalTAT = 0, totalWT = 0;

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

printf("%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);

totalTAT += p[i].tat;

totalWT += p[i].wt;

}

printf("\nAverage Turnaround Time: %.2f", totalTAT / n);

printf("\nAverage Waiting Time: %.2f\n", totalWT / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process p[n];

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

printf("\nEnter Arrival Time and Burst Time for Process %d: ", i + 1);

p[i].pid = i + 1;

scanf("%d %d", &p[i].at, &p[i].bt);

p[i].rt = p[i].bt; // Remaining time is initially equal to burst time

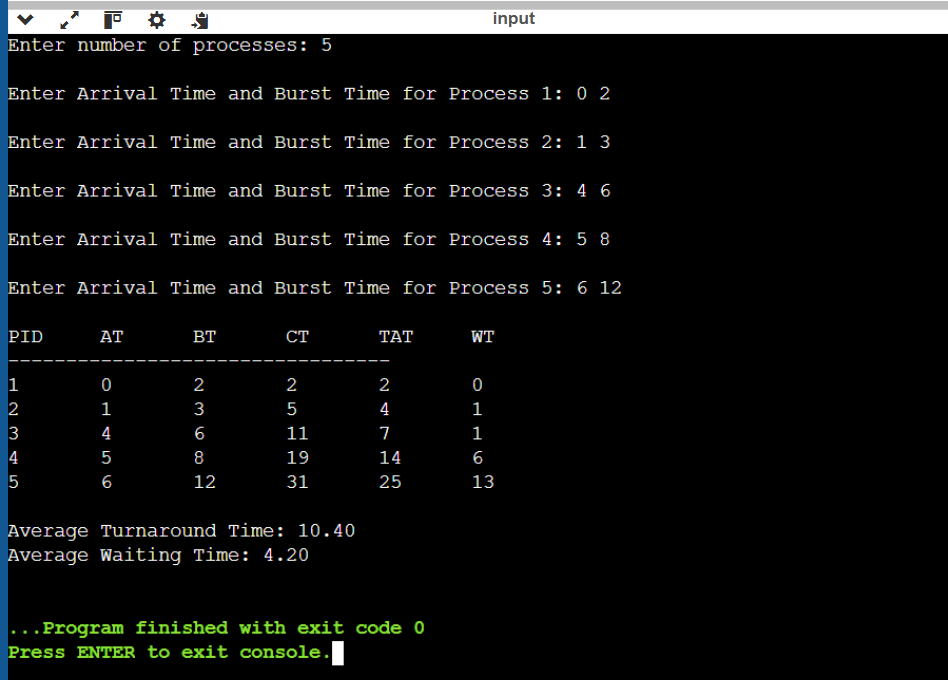
}

sjfPreemptive(p, n);

displayResults(p, n);

return 0;

}



Experiment-7 C program to implement Round Robin CPU scheduling algorithm and calculate turnaround time and waiting time

PROGRAM:

#include <stdio.h>

// Structure to store process details

struct Process {

int pid; // Process ID

int at; // Arrival Time

int bt; // Burst Time

int rt; // Remaining Time

int ct; // Completion Time

int tat; // Turnaround Time

int wt; // Waiting Time

};

// Function to perform Round Robin Scheduling

void roundRobinScheduling(struct Process p[], int n, int quantum) {

int completed = 0, currentTime = 0;

// Create a queue for Round Robin execution

int queue[n], front = 0, rear = 0;

// Add processes to queue based on arrival time

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

if (p[i].at == 0) {

queue[rear++] = i;

}

}

while (completed < n) {

if (front == rear) { // If queue is empty, move time forward

currentTime++;

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

if (p[i].at == currentTime) {

queue[rear++] = i;

}

}

continue;

}

// Get process from the queue

int idx = queue[front++];

// Execute for a time quantum or till completion

if (p[idx].rt > quantum) {

currentTime += quantum;

p[idx].rt -= quantum;

} else {

currentTime += p[idx].rt;

p[idx].rt = 0;

p[idx].ct = currentTime;

p[idx].tat = p[idx].ct - p[idx].at;

p[idx].wt = p[idx].tat - p[idx].bt;

completed++;

}

// Add new arrivals to queue

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

if (p[i].at > currentTime - quantum && p[i].at <= currentTime && p[i].rt > 0) {

queue[rear++] = i;

}

}

// If the process is not finished, add it back to the queue

if (p[idx].rt > 0) {

queue[rear++] = idx;

}

}

}

// Function to display process details

void display(struct Process p[], int n) {

printf("\nPID\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");

printf("--------------------------------------------------------------\n");

float totalTAT = 0, totalWT = 0;

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

printf("%d\t%d\t%d\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);

totalTAT += p[i].tat;

totalWT += p[i].wt;

}

printf("\nAverage Turnaround Time: %.2f", totalTAT / n);

printf("\nAverage Waiting Time: %.2f\n", totalWT / n);

}

int main() {

int n, quantum;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process p[n];

// Input process details

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

printf("Enter Arrival Time and Burst Time for Process %d: ", i + 1);

p[i].pid = i + 1;

scanf("%d %d", &p[i].at, &p[i].bt);

p[i].rt = p[i].bt; // Remaining Time = Burst Time initially

}

printf("Enter Time Quantum: ");

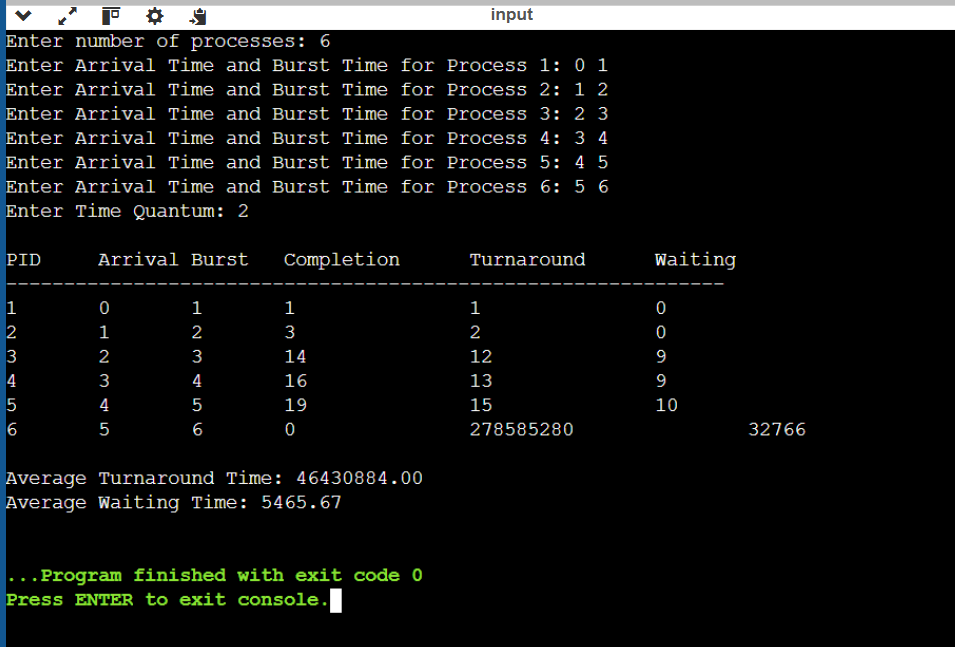
scanf("%d", &quantum);

roundRobinScheduling(p, n, quantum);

display(p, n);

return 0;

}



Experiment-8 C program to implement priority scheduling algorithm and test with different sets of priorities

PROGRAM:

#include <stdio.h>

#include <limits.h>

// Structure for process

struct Process {

int pid, at, bt, rt, priority, ct, tat, wt;

};

// Function to sort processes based on arrival time

void sortByArrival(struct Process p[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (p[j].at > p[j + 1].at) {

struct Process temp = p[j];

p[j] = p[j + 1];

p[j + 1] = temp;

}

}

}

}

// Function to perform Non-Preemptive Priority Scheduling

void nonPreemptivePriorityScheduling(struct Process p[], int n) {

int completed = 0, currentTime = 0;

while (completed < n) {

int minIndex = -1, minPriority = INT\_MAX;

// Find highest priority process that has arrived

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

if (p[i].at <= currentTime && p[i].rt > 0 && p[i].priority < minPriority) {

minPriority = p[i].priority;

minIndex = i;

}

}

if (minIndex == -1) {

currentTime++;

continue;

}

// Execute process till completion

currentTime += p[minIndex].bt;

p[minIndex].rt = 0;

p[minIndex].ct = currentTime;

p[minIndex].tat = p[minIndex].ct - p[minIndex].at;

p[minIndex].wt = p[minIndex].tat - p[minIndex].bt;

completed++;

}

}

// Function to perform Preemptive Priority Scheduling

void preemptivePriorityScheduling(struct Process p[], int n) {

int completed = 0, currentTime = 0;

while (completed < n) {

int minIndex = -1, minPriority = INT\_MAX;

// Find highest priority process that has arrived

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

if (p[i].at <= currentTime && p[i].rt > 0 && p[i].priority < minPriority) {

minPriority = p[i].priority;

minIndex = i;

}

}

if (minIndex == -1) {

currentTime++;

continue;

}

// Execute process for 1 time unit

p[minIndex].rt--;

currentTime++;

// If process is completed

if (p[minIndex].rt == 0) {

completed++;

p[minIndex].ct = currentTime;

p[minIndex].tat = p[minIndex].ct - p[minIndex].at;

p[minIndex].wt = p[minIndex].tat - p[minIndex].bt;

}

}

}

// Function to display process details

void display(struct Process p[], int n) {

printf("\nPID\tArrival\tBurst\tPriority\tCompletion\tTurnaround\tWaiting\n");

printf("----------------------------------------------------------------------\n");

float totalTAT = 0, totalWT = 0;

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

printf("%d\t%d\t%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].priority, p[i].ct, p[i].tat, p[i].wt);

totalTAT += p[i].tat;

totalWT += p[i].wt;

}

printf("\nAverage Turnaround Time: %.2f", totalTAT / n);

printf("\nAverage Waiting Time: %.2f\n", totalWT / n);

}

int main() {

int n, choice;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process p[n];

// Input process details

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

printf("Enter Arrival Time, Burst Time, and Priority for Process %d: ", i + 1);

p[i].pid = i + 1;

scanf("%d %d %d", &p[i].at, &p[i].bt, &p[i].priority);

p[i].rt = p[i].bt; // Remaining Time = Burst Time initially

}

// Sorting by Arrival Time

sortByArrival(p, n);

printf("\nChoose Scheduling Type:\n1. Non-Preemptive\n2. Preemptive\nEnter choice: ");

scanf("%d", &choice);

if (choice == 1) {

nonPreemptivePriorityScheduling(p, n);

printf("\nExecuting Non-Preemptive Priority Scheduling...\n");

} else if (choice == 2) {

preemptivePriorityScheduling(p, n);

printf("\nExecuting Preemptive Priority Scheduling...\n");

} else {

printf("Invalid choice! Exiting...\n");

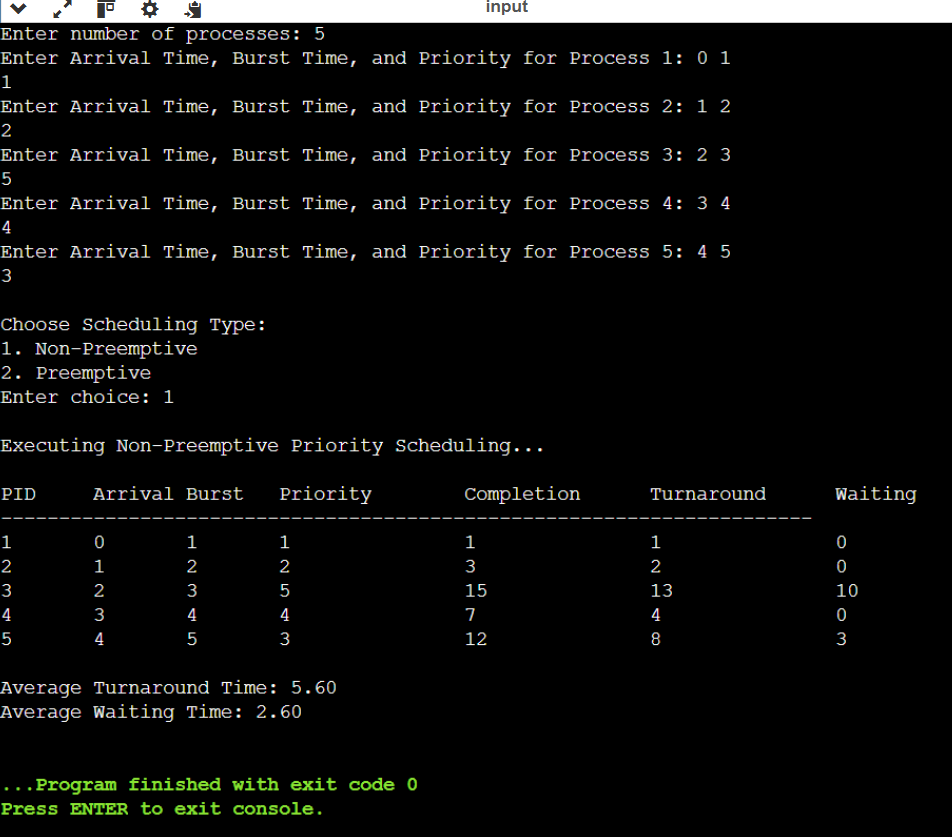
return 1;

}

display(p, n);

return 0;

}



Experiment-9 :C Program to implement producer consumers problem of process synchronization on operating system.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5 // Size of the shared buffer

#define NUM\_ITEMS 10 // Number of items to produce/consume

int buffer[BUFFER\_SIZE]; // Shared buffer

int in = 0, out = 0; // Buffer index pointers

sem\_t empty; // Semaphore for empty slots

sem\_t full; // Semaphore for filled slots

pthread\_mutex\_t mutex; // Mutex for critical section

void \*producer(void \*param) {

for (int i = 0; i < NUM\_ITEMS; i++) {

int item = rand() % 100; // Produce a random item

sem\_wait(&empty); // Wait if buffer is full

pthread\_mutex\_lock(&mutex); // Enter critical section

// Add item to buffer

buffer[in] = item;

printf("Producer produced: %d at buffer[%d]\n", item, in);

in = (in + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&full); // Signal that buffer has data

sleep(1); // Simulate time delay

}

return NULL;

}

void \*consumer(void \*param) {

for (int i = 0; i < NUM\_ITEMS; i++) {

sem\_wait(&full); // Wait if buffer is empty

pthread\_mutex\_lock(&mutex); // Enter critical section

// Consume item from buffer

int item = buffer[out];

printf("Consumer consumed: %d from buffer[%d]\n", item, out);

out = (out + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&empty); // Signal that buffer has space

sleep(2); // Simulate time delay

}

return NULL;

}

int main() {

pthread\_t prodThread, consThread;

// Initialize semaphores and mutex

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

pthread\_mutex\_init(&mutex, NULL);

// Create producer and consumer threads

pthread\_create(&prodThread, NULL, producer, NULL);

pthread\_create(&consThread, NULL, consumer, NULL);

// Wait for threads to complete

pthread\_join(prodThread, NULL);

pthread\_join(consThread, NULL);

// Destroy semaphores and mutex

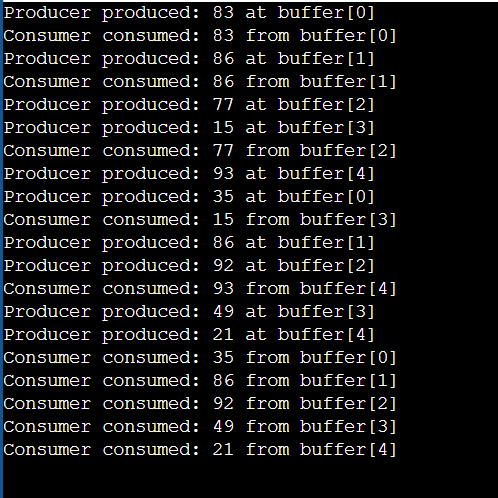
sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}



Experiment-10 C program to implement Dining Philosophers problem of process synchronization inoperating system.

PROGRAM:

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

sem\_t forks[NUM\_PHILOSOPHERS]; // Semaphores for forks

pthread\_t philosophers[NUM\_PHILOSOPHERS]; // Threads for philosophers

void think(int id) {

printf("Philosopher %d is thinking...\n", id);

sleep(1);

}

void eat(int id) {

printf("Philosopher %d is eating...\n", id);

sleep(2);

}

void\* philosopher(void\* num) {

int id = \*(int\*)num;

while (1) {

think(id);

// Pick up left fork

sem\_wait(&forks[id]);

printf("Philosopher %d picked up left fork %d.\n", id, id);

// Pick up right fork

sem\_wait(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

printf("Philosopher %d picked up right fork %d.\n", id, (id + 1) % NUM\_PHILOSOPHERS);

eat(id);

// Put down right fork

sem\_post(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

printf("Philosopher %d put down right fork %d.\n", id, (id + 1) % NUM\_PHILOSOPHERS);

// Put down left fork

sem\_post(&forks[id]);

printf("Philosopher %d put down left fork %d.\n", id, id);

}

return NULL;

}

int main() {

int i, ids[NUM\_PHILOSOPHERS];

// Initialize semaphores (1 fork each)

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_init(&forks[i], 0, 1);

ids[i] = i;

}

// Create philosopher threads

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_create(&philosophers[i], NULL, philosopher, &ids[i]);

}

// Join philosopher threads (infinite loop, will not reach)

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_join(philosophers[i], NULL);

}

// Destroy semaphores (unreachable in current code)

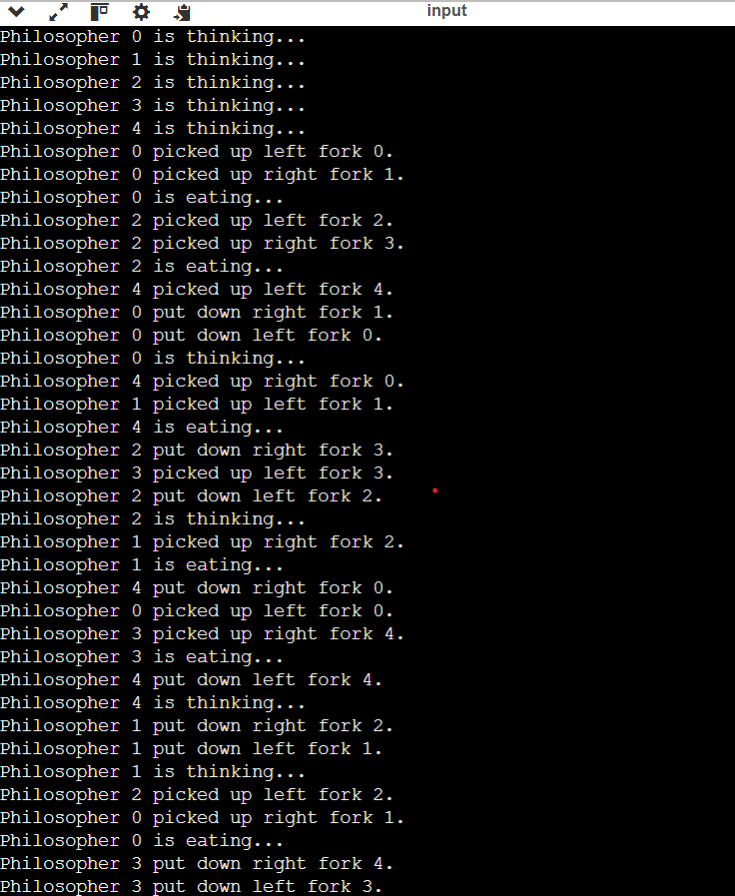
for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_destroy(&forks[i]);

}

return 0;

}



Experiment-11 C program to implement Reader Writers problem of process synchronization in Operating System

PROGRAM:

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t rw\_mutex; // Semaphore for writer

pthread\_mutex\_t mutex; // Mutex for read count

int read\_count = 0; // Number of active readers

int shared\_data = 0; // Shared resource

void \*reader(void \*arg) {

int reader\_id = \*(int \*)arg;

pthread\_mutex\_lock(&mutex);

read\_count++;

if (read\_count == 1) {

sem\_wait(&rw\_mutex); // First reader locks writers out

}

pthread\_mutex\_unlock(&mutex);

// Reading section

printf("Reader %d is reading data: %d\n", reader\_id, shared\_data);

sleep(1);

pthread\_mutex\_lock(&mutex);

read\_count--;

if (read\_count == 0) {

sem\_post(&rw\_mutex); // Last reader allows writing

}

pthread\_mutex\_unlock(&mutex);

return NULL;

}

void \*writer(void \*arg) {

int writer\_id = \*(int \*)arg;

sem\_wait(&rw\_mutex); // Lock the resource for writing

// Writing section

shared\_data++;

printf("Writer %d wrote data: %d\n", writer\_id, shared\_data);

sleep(2);

sem\_post(&rw\_mutex); // Release the resource for others

return NULL;

}

int main() {

pthread\_t readers[5], writers[3];

int reader\_ids[5], writer\_ids[3];

// Initialize semaphores and mutex

sem\_init(&rw\_mutex, 0, 1);

pthread\_mutex\_init(&mutex, NULL);

// Create reader and writer threads

for (int i = 0; i < 5; i++) {

reader\_ids[i] = i + 1;

pthread\_create(&readers[i], NULL, reader, &reader\_ids[i]);

}

for (int i = 0; i < 3; i++) {

writer\_ids[i] = i + 1;

pthread\_create(&writers[i], NULL, writer, &writer\_ids[i]);

}

// Join all threads

for (int i = 0; i < 5; i++) {

pthread\_join(readers[i], NULL);

}

for (int i = 0; i < 3; i++) {

pthread\_join(writers[i], NULL);

}

// Destroy semaphores and mutex

sem\_destroy(&rw\_mutex);

pthread\_mutex\_destroy(&mutex);

return 0;

}

