School of Computing Science & Engineering (SCOPE)

Practical Record File

Operating System (CSE3003)

Slot: E11+E12+C14

Class no: 0629

Submitted By:

OM RAJ

Submitted to:

Dr. Gunjan Ansari

Associate Professor, SCSE

Indicative List of Experiments

(All the experiments need to be done in C++/Java Language)

Exp. No.	List of Experiments	Date Conducted	Remarks
1	Learn and practice basic OS commands.		
2	C program to create a clone of current process		
3	Write a C program to create a new process using fork()		
4	C program that implements a parent-child process relationship and demonstrates process termination using fork() and wait() system calls		
5	Write a C program to simulate first come first serve(fcfs) scheduling		
6	Write a C program to implement shortest job first scheduling with or without premption		
7	Write a C program to Implement priority scheduling and test with different sets of priorities		
8	Write a C program to simulate round robin scheduling and turn around and waiting tim		

Exp. No.	List of Experiments	Date Conducted	Remarks
9	**C Program to Implement Producer-Consumer Problem**		
10	**The Dining Philosophers Problem**		
11	**The Reader-Writer problem**		
12	**C program to implement Bankers algorithm of deadlock avoidance in operating system**		

Experiment - 1

Basic Linux/Unix Commands

Aim: Learn and practice basic OS commands.

1. File Management
Commands and Examples:

- **`ls`**: Lists files and directories in the current directory.
 ```bash
 ls
 ls -l
 ls -a
 ```
- **`cp`**: Copies files or directories.
 ```bash
 touch file1.txt
 cp file1.txt file2.txt
 ```
- **`mv`**: Moves or renames files or directories.
 ```bash
 mv file2.txt file3.txt
 ```
- **`rm`**: Removes files or directories.
 ```bash
 rm file3.txt
 ```
- **`touch`**: Creates an empty file.
 ```bash
 touch file4.txt

2. Process Management
Commands and Examples:

- **`ps`**: Displays information about active processes.
 ```bash
 ps
 ps aux
 ```
- **`kill`**: Terminates a process using its PID (Process ID).
 ```bash
 ps aux

kill <PID>
```
- **`top`**: Displays real-time process monitoring.
```bash
top
```

---

3. Directory Navigation
Commands and Examples:
- **`cd`**: Changes the current directory.
```bash
cd /home/user/Documents
```
- **`pwd`**: Prints the current working directory.
```bash
pwd

4. I/O Redirection and Piping
Commands and Examples:

- **`>`**: Redirects standard output to a file (overwrites the file).
```bash
echo "Hello, World!" > output.txt
```
- **`>>`**: Appends standard output to a file.
```bash
echo "This is appended text." >> output.txt
```
- **`|`**: Pipes the output of one command to another as input.
```bash
ls -l | grep "file"

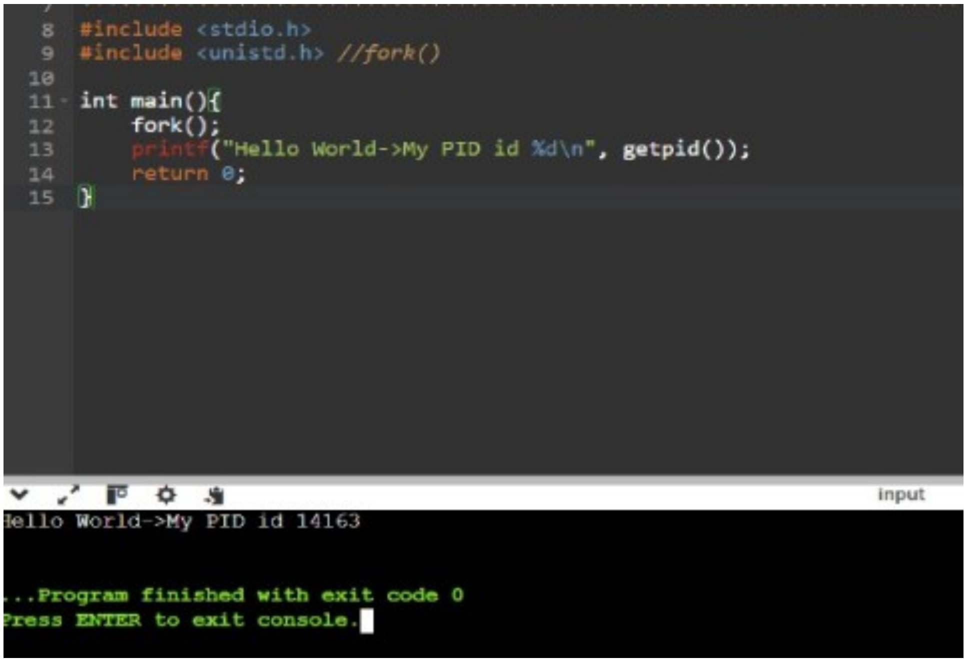
# Experiment - 2

## AIM: C program to create a clone of current process

```c
#include <stdio.h>
#include <unistd.h> //fork()

int main(){
   fork();
```
```

```
    printf("Hello World->My PID id %d\n", getpid());
    return 0;
}
```



Experiment - 3

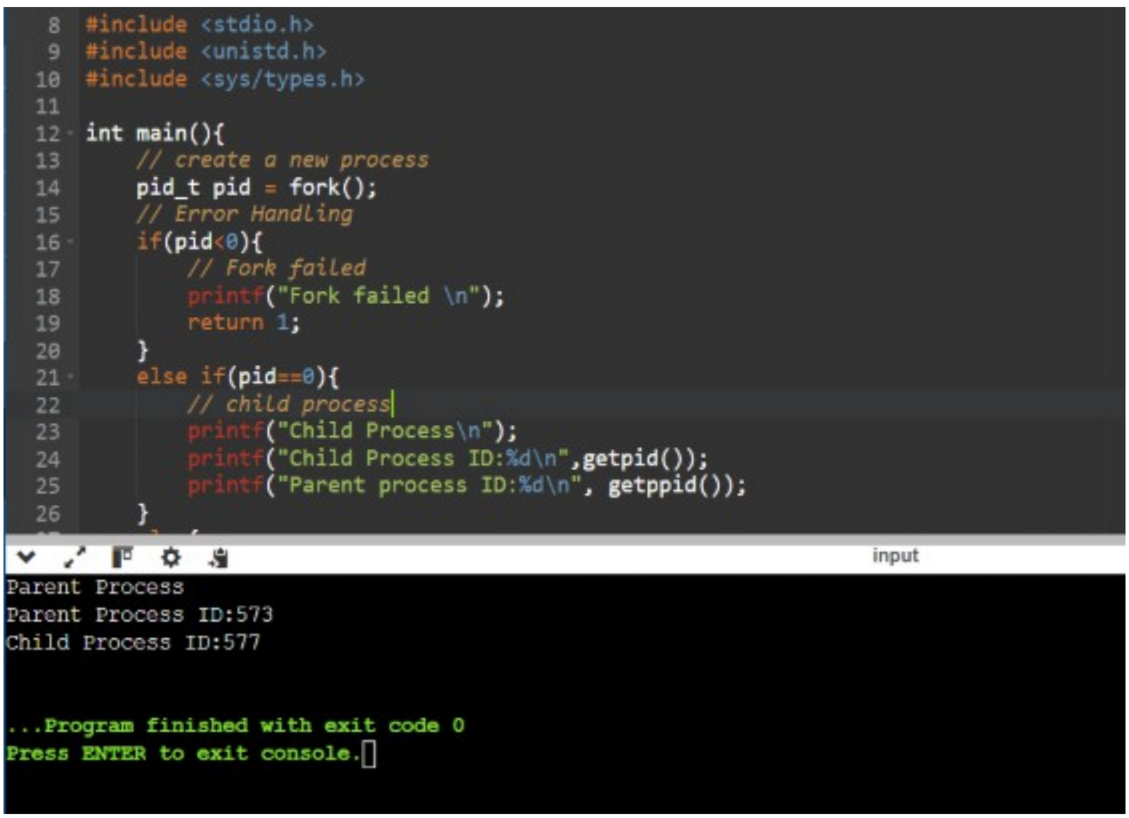
AIM: Write a C program to create a new process using fork()

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

int main(){
    // create a new process
    pid_t pid = fork();
    // Error Handling
    if(pid<0){
        // Fork failed
        printf("Fork failed \n");
        return 1;
    }
    else if(pid==0){
        // child process
        printf("Child Process\n");
```

```
        printf("Child Process ID:%d\n",getpid());
        printf("Parent process ID:%d\n", getppid());
    }
    else{
        // Parent Process
        printf("Parent Process\n");
        printf("Parent Process ID:%d\n", getpid());
        printf("Child Process ID:%d\n", pid);

    }
    return 0;
}
```



<u>Experiment - 4</u>

AIM: C program that implements a parent-child process relationship and demonstrates process termination using fork() and wait() system calls.

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

```
#include <sys/types.h>
#include <sys/wait.h>

int main(){
    pidt pid = fork();
    if(pid<0){
        // fork failed
        printf("Fork Failed\n");
        return 1;
    }
    else if(pid==0){
        // Child process
        printf("Child process:\n");
        printf("child PID:%d\n", getpid());
        printf("Parent PID:%d\n", getppid());
        printf("Child Process will terminate now\n");
        exit(0);
    }
    else{
        // Parent Process
        printf("Parent Process:\n")
        printf("Parent PID:%d\n", getpid());
        printf("Waiting for child process to terminate..\n");
        // wait for child process to terminate
        int status;
        wait(&status);
        if(WIFEXITED(status)){
            printf("Child Process terminated withexit status:%d\n", WEXITSTATUS(status));
        }
        else{
            printf("Child process did not terminate successfully\n");
        }
        printf("Parent process will terminate now\n");
    }
    return 0;
}
```

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

int main(){
    pid_t pid = fork();
    if(pid<0){
        // fork failed
        printf("Fork Failed\n");
        return 1;
    }
    else if(pid==0){
        // Child process
        printf("Child process:\n");
        printf("child PID:%d\n", getpid());
        printf("Parent PID:%d\n", getppid());
        printf("Child Process will terminate now\n");
        exit(0);
    }
```

input

```
Parent PID:21674
Waiting for child process to terminate..
Child process:
child PID:21678
Parent PID:21674
Child Process will terminate now
Child Process terminated withexit status:0
Parent process will terminate now


...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment - 5

AIM: Write a C program to simulate first come first serve(fcfs) scheduling

```
#include <stdio.h>

struct process
{
    int pid;
    int arrival_time;
    int burst_time;
    int waiting_time;
    int turnaround_time;
};

int main()
```

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

    struct process processes[n];

    for (int i = 0; i < n; i++)
    {
        printf("Enter details for 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].pid = i + 1;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }

    // Sort processes by arrival time in ascending order
    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;
            }
        }
    }

    // Calculate waiting time and turnaround time for each process
    int time = 0;
    for (int i = 0; i < n; i++)
    {
        if (time < processes[i].arrival_time)
        {
            time = processes[i].arrival_time;
        }

        processes[i].waiting_time = time - processes[i].arrival_time;
```

```
        time += processes[i].burst_time;
        processes[i].turnaround_time = time - processes[i].arrival_time;
    }

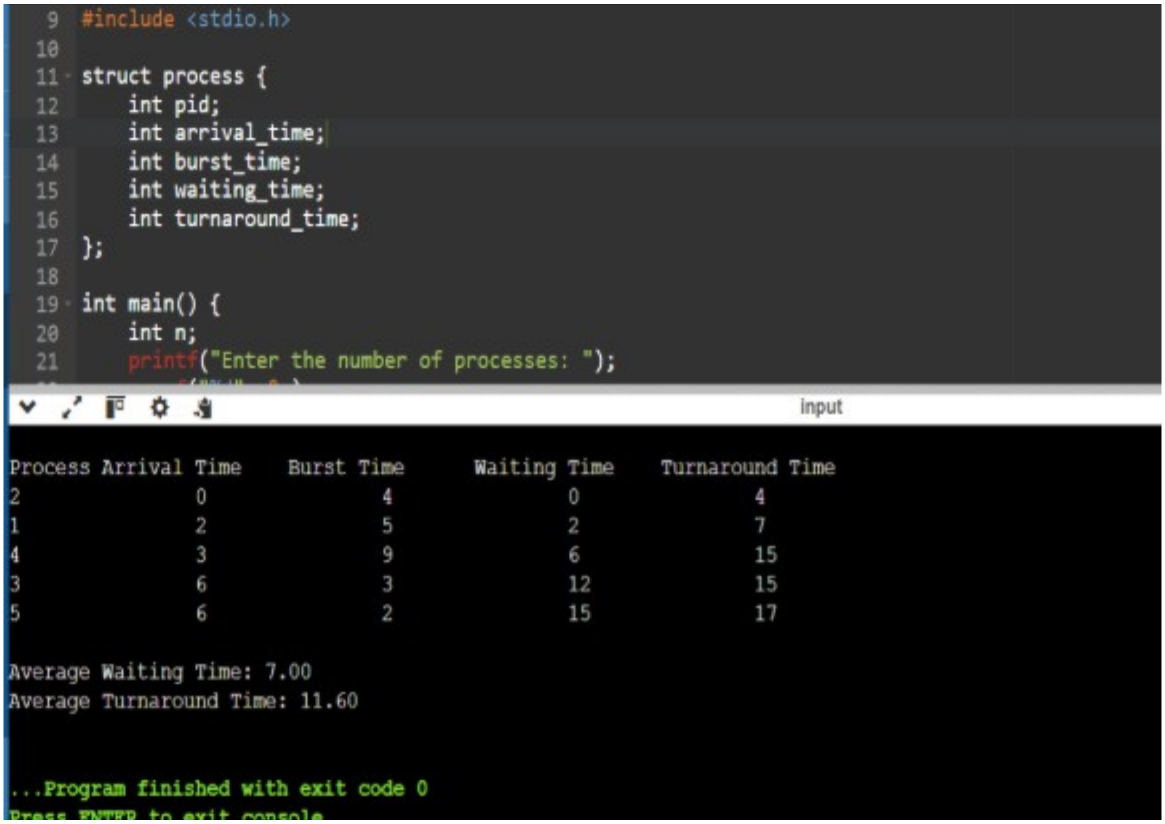
    // Calculate average waiting time and turnaround time
    float avg_waiting_time = 0.0, avg_turnaround_time = 0.0;
    for (int i = 0; i < n; i++)
    {
        avg_waiting_time += processes[i].waiting_time;
        avg_turnaround_time += processes[i].turnaround_time;
    }

    avg_waiting_time /= n;
    avg_turnaround_time /= n;

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

    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);

    return 0;
}
```



```
#include <stdio.h>

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int waiting_time;
    int turnaround_time;
};

int main() {
    int n;
    printf("Enter the number of processes: ");
```

```
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
2       0               4               0               4
1       2               5               2               7
4       3               9               6               15
3       6               3               12              15
5       6               2               15              17

Average Waiting Time: 7.00
Average Turnaround Time: 11.60

...Program finished with exit code 0
Press ENTER to exit console
```

Experiment - 6

AIM: Write a C program to implement shortest job first scheduling with or without premption

```
#include <stdio.h>

#define MAX_PROCESSES 100

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int start_time;
    int waiting_time;
    int turnaround_time;
};
```

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

    if (n > MAX_PROCESSES) {
        printf("Error: Maximum number of processes exceeded.\n");
        return 1;
    }

    struct process processes[MAX_PROCESSES];

    for (int i = 0; i < n; i++) {
        printf("Enter details for 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].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].start_time = 0;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }

    // Sort processes by arrival time in ascending order
    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;
            }
        }
    }

    // Choose preemptive or non-preemptive SJF
    int preemptive;
    printf("Enter 1 for preemptive SJF, 0 for non-preemptive SJF: ");
    scanf("%d", &preemptive);

    int time = 0;
    int completed = 0;
```

```
    while (completed < n) {
        int shortest_index = -1;
        int shortest_remaining = 10000; // A large initial value

        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= time && processes[i].remaining_time > 0) {
                if (processes[i].remaining_time < shortest_remaining) {
                    shortest_index = i;
                    shortest_remaining = processes[i].remaining_time;
                }
            }
        }

        if (shortest_index != -1) {
            if (processes[shortest_index].start_time == 0) {
                processes[shortest_index].start_time = time;
            }

            if (preemptive) {
                processes[shortest_index].remaining_time--;
                time++;

                if (processes[shortest_index].remaining_time == 0) {
                    processes[shortest_index].turnaround_time = time -
processes[shortest_index].arrival_time;
                    processes[shortest_index].waiting_time =
processes[shortest_index].turnaround_time - processes[shortest_index].burst_time;
                    completed++;
                }
            } else {
                time += processes[shortest_index].remaining_time;
                processes[shortest_index].remaining_time = 0;
                processes[shortest_index].turnaround_time = time -
processes[shortest_index].arrival_time;
                processes[shortest_index].waiting_time = processes[shortest_index].turnaround_time
- processes[shortest_index].burst_time;
                completed++;
            }
        } else {
            time++;
        }
    }
```

```
    // Calculate average waiting time and turnaround time
    float avg_waiting_time = 0.0, avg_turnaround_time = 0.0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += processes[i].waiting_time;
        avg_turnaround_time += processes[i].turnaround_time;
    }

    avg_waiting_time /= n;
    avg_turnaround_time /= n;

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

    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);

    return 0;
}
```

Experiment - 7

AIM: Write a C program to Implement priority scheduling and test with different sets of priorities.

```
#include <stdio.h>

#define MAX_PROCESSES 100

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
};
```

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

    if (n > MAX_PROCESSES) {
        printf("Error: Maximum number of processes exceeded.\n");
        return 1;
    }

    struct process processes[MAX_PROCESSES];

    for (int i = 0; i < n; i++) {
        printf("Enter details for process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Priority: ");
        scanf("%d", &processes[i].priority);

        processes[i].pid = i + 1;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }

    // Sort processes by arrival time in ascending order
    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;
            }
        }
    }

    // Calculate waiting time and turnaround time for each process
    int time = 0;
    int completed = 0;

    while (completed < n) {
        int highest_priority_index = -1;
```

```
        int highest_priority = -1;

        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= time && processes[i].priority > highest_priority) {
                highest_priority_index = i;
                highest_priority = processes[i].priority;
            }
        }

        if (highest_priority_index != -1) {
            processes[highest_priority_index].waiting_time = time -
processes[highest_priority_index].arrival_time;
            time += processes[highest_priority_index].burst_time;
            processes[highest_priority_index].turnaround_time = time -
processes[highest_priority_index].arrival_time;
            completed++;
        } else {
            time++;
        }
    }

    // Calculate average waiting time and turnaround time
    float avg_waiting_time = 0.0, avg_turnaround_time = 0.0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += processes[i].waiting_time;
        avg_turnaround_time += processes[i].turnaround_time;
    }

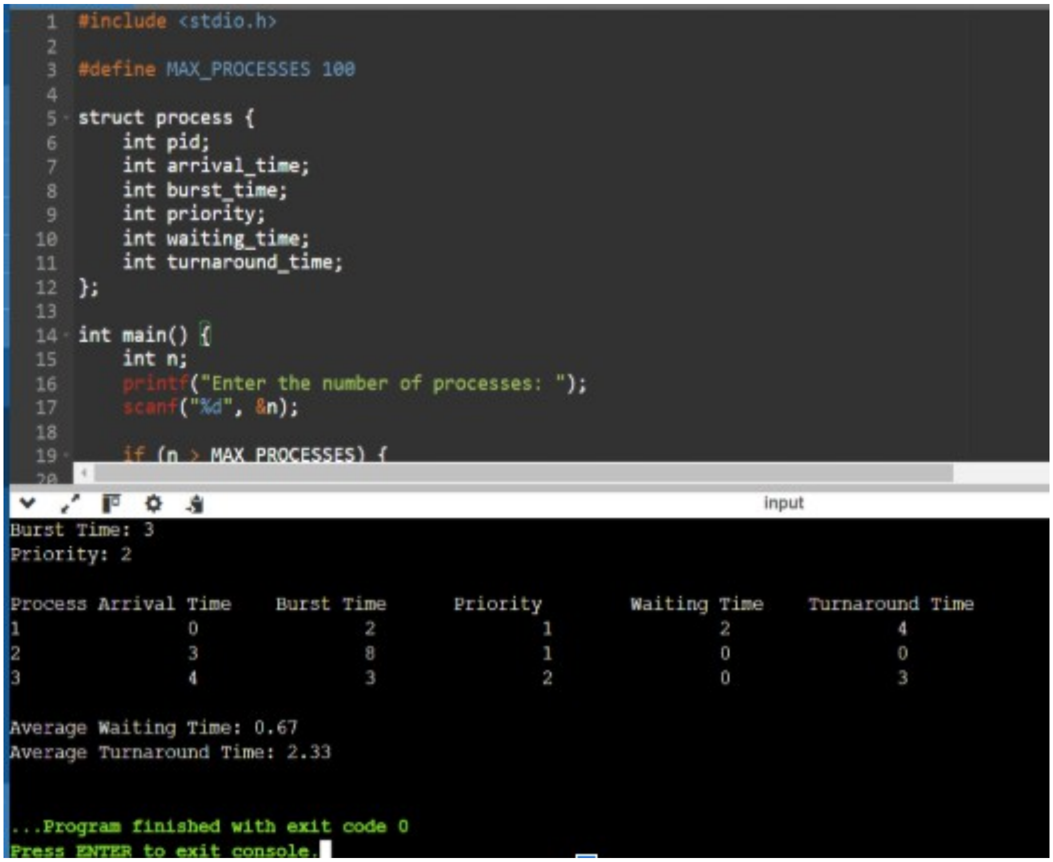
    avg_waiting_time /= n;
    avg_turnaround_time /= n;

    // Print the results
    printf("\nProcess\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
               processes[i].pid, processes[i].arrival_time,
               processes[i].burst_time, processes[i].priority,
               processes[i].waiting_time,  processes[i].turnaround_time);
    }

    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);

    return 0;
```

```
}
```



<u>Experiment - 8</u>

AIM: Write a C program to simulate round robin scheduling and turn around and waiting time.

```
#include <stdio.h>

#define MAX_PROCESSES 100

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
};
```

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

    if (n > MAX_PROCESSES) {
        printf("Error: Maximum number of processes exceeded.\n");
        return 1;
    }

    struct process processes[MAX_PROCESSES];

    for (int i = 0; i < n; i++) {
        printf("Enter details for 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].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }

    // Sort processes by arrival time in ascending order
    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;
            }
        }
    }

    int time = 0;
    int completed = 0;

    while (completed < n) {
        int i = 0;
```

```
        while (i < n) {
            if (processes[i].arrival_time <= time && processes[i].remaining_time > 0) {
                if (processes[i].remaining_time <= time_quantum) {
                    time += processes[i].remaining_time;
                    processes[i].remaining_time = 0;
                    processes[i].turnaround_time = time - processes[i].arrival_time;
                    processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
                    completed++;
                } else {
                    time += time_quantum;
                    processes[i].remaining_time -= time_quantum;
                }
            }
            i++;
        }
    }

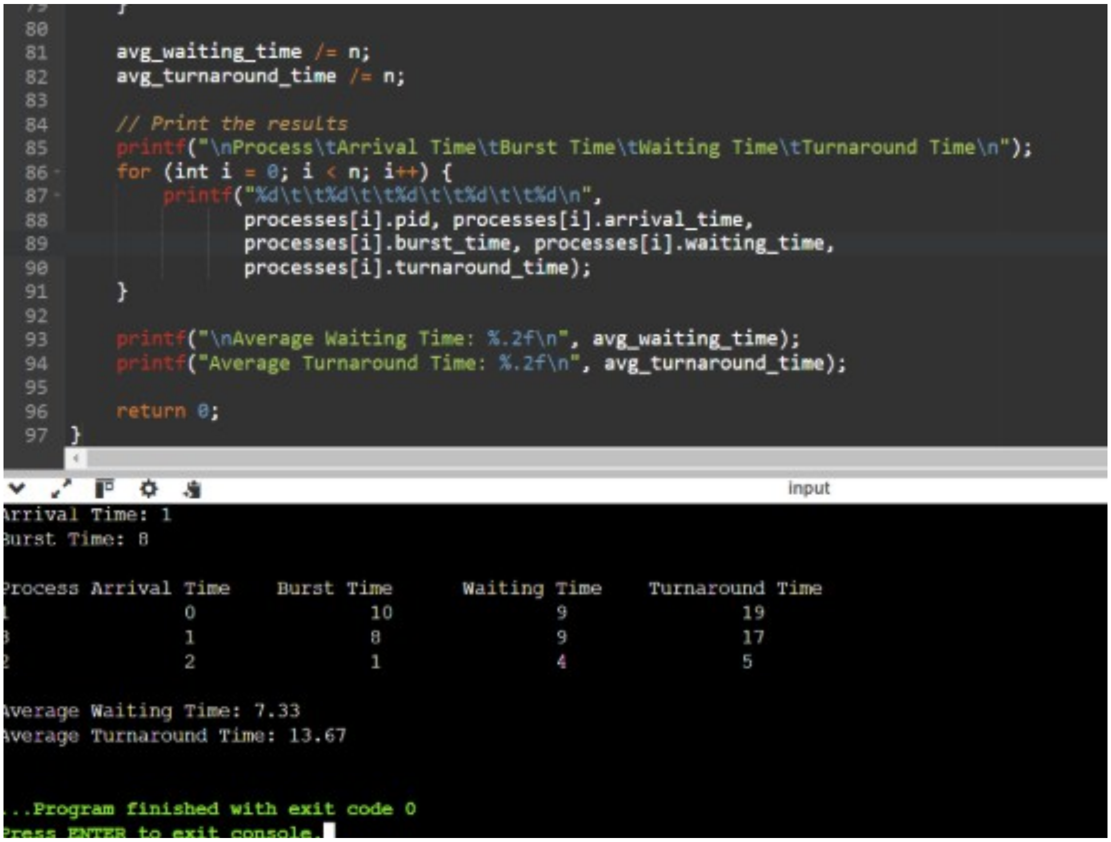
    // Calculate average waiting time and turnaround time
    float avg_waiting_time = 0.0, avg_turnaround_time = 0.0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += processes[i].waiting_time;
        avg_turnaround_time += processes[i].turnaround_time;
    }

    avg_waiting_time /= n;
    avg_turnaround_time /= n;

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

    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);

    return 0;
}
```



```
    avg_waiting_time /= n;
    avg_turnaround_time /= n;

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

    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);

    return 0;
}
```

```
Arrival Time: 1
Burst Time: 8

Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1               0               10              9               19
3               1               8               9               17
2               2               1               4               5

Average Waiting Time: 7.33
Average Turnaround Time: 13.67

...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment - 9

Aim: C Program to Implement Producer-Consumer Problem

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

#define BUFFER_SIZE 5  // Size of the buffer

int buffer[BUFFER_SIZE];  // Shared buffer
int in = 0;               // Index to place the next produced item
int out = 0;              // Index to remove the next consumed item

// Semaphores
sem_t empty;  // Counts the empty buffer slots
sem_t full;   // Counts the filled buffer slots
pthread_mutex_t mutex;  // Mutex lock for buffer
```

```
// Producer function
void* producer(void* arg) {
    int item;
    while (1) {
        item = rand() % 100;  // Produce a random item
        sem_wait(&empty);      // Wait if buffer is full
        pthread_mutex_lock(&mutex);  // Lock the buffer

        // Add item to the buffer
        buffer[in] = item;
        printf("Producer produced: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);  // Unlock the buffer
        sem_post(&full);  // Signal that the buffer has one more full slot

        sleep(1);  // Simulate production time
    }
}

// Consumer function
void* consumer(void* arg) {
    int item;
    while (1) {
        sem_wait(&full);       // Wait if buffer is empty
        pthread_mutex_lock(&mutex);  // Lock the buffer

        // Remove item from the buffer
        item = buffer[out];
        printf("Consumer consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);  // Unlock the buffer
        sem_post(&empty);  // Signal that the buffer has one more empty slot

        sleep(1);  // Simulate consumption time
    }
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize semaphores and mutex
```

```
    sem_init(&empty, 0, BUFFER_SIZE);  // Initially, buffer is empty
    sem_init(&full, 0, 0);             // Initially, no items in the buffer
    pthread_mutex_init(&mutex, NULL);

    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    // Join the threads (they will run indefinitely in this example)
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Destroy semaphores and mutex
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Experiment - 10

Aim: The Dining Philosophers Problem

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

#define NUM_PHILOSOPHERS 5

sem_t chopsticks[NUM_PHILOSOPHERS];  // Semaphores for chopsticks
pthread_t philosophers[NUM_PHILOSOPHERS];  // Philosopher threads
int philosopher_ids[NUM_PHILOSOPHERS];  // IDs for philosophers

// Function for each philosopher's behavior
void* philosopher(void* num) {
    int id = *(int*)num;

    while (1) {
        printf("Philosopher %d is thinking.\n", id);
        sleep(rand() % 3);  // Simulate thinking
```

```
        // Pick up left and right chopsticks
        printf("Philosopher %d is hungry and trying to pick up chopsticks.\n", id);
        sem_wait(&chopsticks[id]);  // Pick up left chopstick
        sem_wait(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);  // Pick up right chopstick

        // Eating
        printf("Philosopher %d is eating.\n", id);
        sleep(rand() % 3);  // Simulate eating

        // Put down chopsticks
        sem_post(&chopsticks[id]);  // Put down left chopstick
        sem_post(&chopsticks[(id + 1) % NUM_PHILOSOPHERS]);  // Put down right chopstick

        printf("Philosopher %d has finished eating and is thinking again.\n", id);
        sleep(rand() % 3);  // Simulate thinking before the next round
    }
}

int main() {
    int i;

    // Initialize semaphores for chopsticks (initial value 1, meaning available)
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&chopsticks[i], 0, 1);
    }

    // Create philosopher threads
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_ids[i]);
    }

    // Join philosopher threads (this will run indefinitely)
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Destroy semaphores (not reached in this example)
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&chopsticks[i]);
    }

    return 0;
```

}

Experiment - 11

Aim: The Reader-Writer problem

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

sem_t resource;       // Semaphore to ensure mutual exclusion on the resource (file or
database)
sem_t rmutex;         // Semaphore to ensure mutual exclusion on reader count
int reader_count = 0;  // Number of active readers

// Reader function
void* reader(void* arg) {
    int id = *(int*)arg;

    while (1) {
        // Entry section for readers
        sem_wait(&rmutex);  // Lock the reader count semaphore
        reader_count++;
        if (reader_count == 1) {
            sem_wait(&resource);  // First reader locks the resource (no writers allowed)
        }
        sem_post(&rmutex);  // Unlock the reader count semaphore

        // Reading the resource
        printf("Reader %d is reading the resource.\n", id);
        sleep(rand() % 3);  // Simulate reading time

        // Exit section for readers
        sem_wait(&rmutex);  // Lock the reader count semaphore
        reader_count--;
        if (reader_count == 0) {
            sem_post(&resource);  // Last reader unlocks the resource (writers can now write)
        }
        sem_post(&rmutex);  // Unlock the reader count semaphore
```

```
        printf("Reader %d has finished reading.\n", id);
        sleep(rand() % 3);  // Simulate the time between reading sessions
    }
}

// Writer function
void* writer(void* arg) {
    int id = *(int*)arg;

    while (1) {
        // Writer tries to access the resource
        sem_wait(&resource);  // Lock the resource (no readers or writers allowed)

        // Writing to the resource
        printf("Writer %d is writing to the resource.\n", id);
        sleep(rand() % 3);  // Simulate writing time

        sem_post(&resource);  // Unlock the resource (allow other writers/readers)
        printf("Writer %d has finished writing.\n", id);
        sleep(rand() % 3);  // Simulate the time between writing sessions
    }
}

int main() {
    pthread_t r_threads[5], w_threads[5];
    int ids[5];

    // Initialize semaphores
    sem_init(&rmutex, 0, 1);      // Reader count semaphore (mutex)
    sem_init(&resource, 0, 1);   // Resource semaphore (mutex)

    // Create reader and writer threads
    for (int i = 0; i < 5; i++) {
        ids[i] = i + 1;
        pthread_create(&r_threads[i], NULL, reader, &ids[i]);
        pthread_create(&w_threads[i], NULL, writer, &ids[i]);
    }

    // Join reader and writer threads (this will run indefinitely)
    for (int i = 0; i < 5; i++) {
        pthread_join(r_threads[i], NULL);
        pthread_join(w_threads[i], NULL);
    }
```

```
    // Destroy semaphores (not reached in this example)
    sem_destroy(&rmutex);
    sem_destroy(&resource);

    return 0;
}
```

Experiment - 12
C program to implement Bankers algorithm of deadlock avoidance in operating system.

```
#include <stdio.h>

int main() {
    int n, m, i, j, k;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter number of resources: ");
    scanf("%d", &m);

    int allocation[n][m], max[n][m], need[n][m], available[m], work[m], finish[n];
    int safeSequence[n], count = 0;

    // Input Allocation Matrix
    printf("Enter Allocation Matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &allocation[i][j]);

    // Input Max Matrix
    printf("Enter Max Matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &max[i][j]);

    // Input Available Resources
    printf("Enter Available Resources:\n");
    for (j = 0; j < m; j++)
        scanf("%d", &available[j]);

    // Calculate Need Matrix
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
```

```
            need[i][j] = max[i][j] - allocation[i][j];

    for (i = 0; i < n; i++)
        finish[i] = 0; // Initially all processes are unfinished

    // Work = Available
    for (j = 0; j < m; j++)
        work[j] = available[j];

    printf("\nSafe Sequence is: ");

    while (count < n) {
        int found = 0;
        for (i = 0; i < n; i++) {
            if (!finish[i]) { // Process not finished
                int canAllocate = 1;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > work[j]) {
                        canAllocate = 0;
                        break;
                    }
                }
                if (canAllocate) {
                    //  Allocate  resources
                    for (k = 0; k < m; k++)
                        work[k] += allocation[i][k];
                    safeSequence[count++] = i;
                    finish[i] = 1;
                    found = 1;
                }
            }
        }
        if (!found) {
            printf("System is not in a safe state.\n");
            return 0;
        }
    }

    for (i = 0; i < n; i++)
        printf("P%d ", safeSequence[i]);

    printf("\nSystem is in a safe state.\n");
    return 0;
}
```