Shri Vaishnav Vidyapeeth Vishwavidyalaya, Indore
Shri Vaishnav Institute of Information Technology

Experiment No. 1

Title:- Implement and update the BIOS settings of your PC.

1. **Outcome:-** BIOS settings of your PC

2. **Objectives**
   Understand the concept of BIOS and way to update it.

3. **Nomenclature, theory with self-assessment questionnaire:-**

3.1 Solution:

Step 1: Restart your computer. Open Start , click the power icon , and click Restart.

- If your computer is locked, click the lock screen, then click the power icon in the bottom-right corner of the screen and click Restart.
- If your computer is already off, press your computer's "On" switch.

Step 2: Wait for the computer's first startup screen to appear. Once the startup screen appears, you'll have a very limited window in which you can press the setup key.

- It's best to start pressing the setup key as soon as the computer begins to restart.
- If you see "Press [key] to enter setup" or something similar flash across the bottom of the screen and then disappear, you'll need to restart and try again.

Step 3: Press and hold Del or F2 to enter setup. The key you're prompted to press might also be different; if so, use that key instead.

- You'll typically use the "F" keys to access the BIOS. These are at the top of your keyboard, though you may have to locate and hold the Fn key while pressing the proper "F" key.
- You can look at your computer model's manual or online support page to confirm your computer's BIOS key.

Step 4: Wait for your BIOS to load. After successfully hitting the setup key, the BIOS will load. This should only take a few moments. When the loading is complete, you will be taken to the BIOS settings menu.

Step 5: Familiarize yourself with the BIOS controls. Since BIOS menus don't support mouse input, you'll need to use the arrow keys and other computer-specific keys to navigate the BIOS. You can usually find a list of controls in the bottom-right corner of the BIOS homepage.

Step 6: Change your settings carefully. When adjusting settings in your BIOS, be sure that you certain what the settings will affect. Changing settings incorrectly can lead to system or hardware failure.

- If you don't know what you want to change coming into the BIOS, you probably shouldn't change anything.

Step 7: Change the boot order. If you want to change what device to boot from, enter the Boot menu. From here, you can designate which device the computer will attempt to boot from first. This is useful for booting from a disc or flash drive to install or repair an operating system.

- You'll typically use the arrow keys to go over to the Boot tab to start this process.

Step 8: Create a BIOS password. You can create a password that will lock the computer from booting unless the correct password is entered.

Step 9: Change your date and time. Your BIOS's clock will dictate your Windows clock. If you replace your computer's battery, your BIOS clock will most likely be reset.

Step 10:

Change fan speeds and system voltages. These options are for advanced users only. In this menu, you can overclock your CPU, potentially allowing for higher performance. This should be performed only if you are comfortable with your computer's hardware.

Step 11:

Save and exit. When you are finished adjusting your settings, you will need to save and exit by using your BIOS' "Save and Exit" key in order for your changes to take effect. When you save and restart, your computer will reboot with the new settings.

- Check the BIOS key legend to see which key is the "Save and Exit" key.

3.2 Assumptions:

3.2.1 **Powering on the computer**: The assumption is that you have a functioning computer that is powered on and connected to a monitor.

3.2.2 **Accessing the BIOS:** Typically, you need to press a specific key during the boot process to enter the BIOS setup utility..

3.2.3 **BIOS user interface**: Once you've entered the BIOS, you'll be presented with a text-based or graphical user interface (GUI) for configuring various settings .

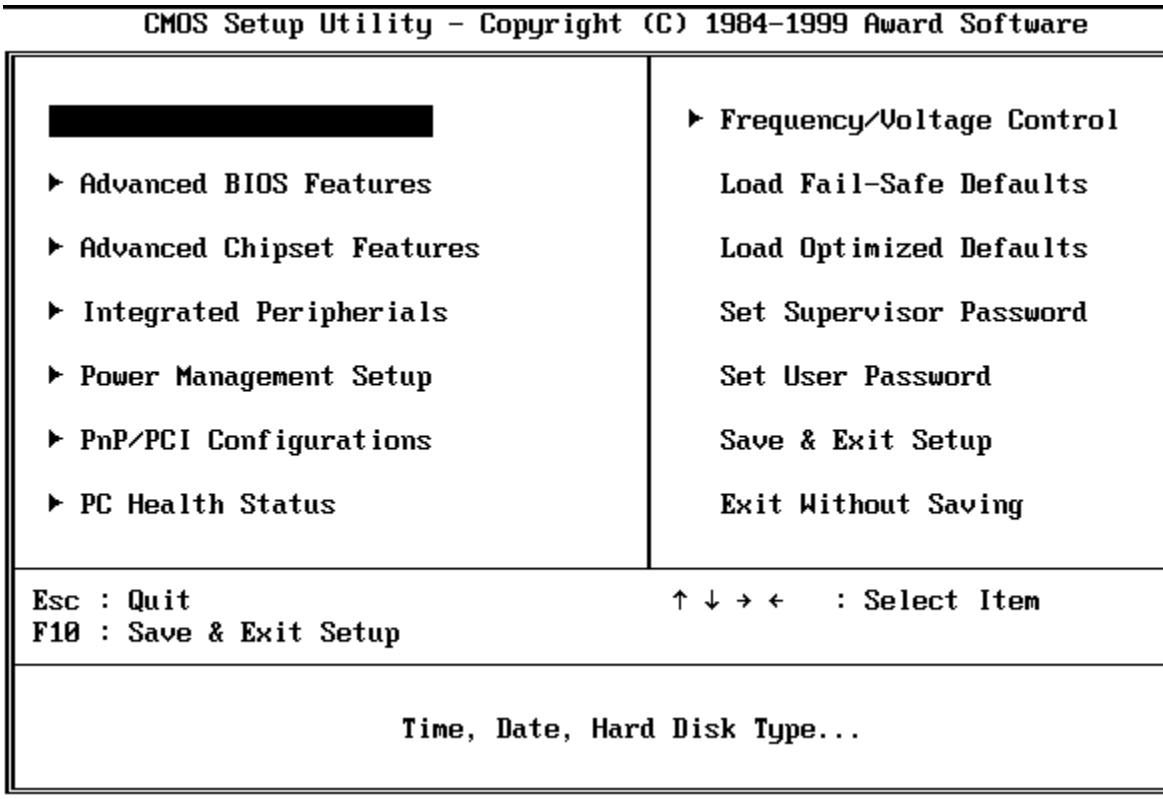
3.3 Dependencies

3.3.1 System type

3.3.2 Architecture

3.4 Results

3.4.1 Test Case



3.4.2 Result Analysis

3.4.2.1 Advantages :

Learning more about BIOS settings

3.4.2.2 **Issues :** NA

Experiment - 2

Title:- To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

4. **Outcome:-** process are arranged in first come first serve format according to arrival time

5. **Objectives**
   Understand the concept of FCFS CPU scheduling algorithm.

6. **Nomenclature, theory with self-assessment questionnaire:-**

4.1 Nomenclature:

`n`	`No of processes`
`waiting_time`	`Waiting time`
`arrival_time`	`Arrival time`
`burst_time`	`Burst time`
`turnaround_time[n];`	`Turnaround time`
`completion_time[i]`	`Completion time`
`waiting_time[n];`	`Waiting time`

6.2 Solution:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

6.3 Assumptions:

6.3.1 **Single queue:** The program assumes a single ready queue containing all the processes waiting for CPU execution

6.3.2 **No priority consideration**: The FIFO algorithm does not consider the priority of processes.

6.3.3 **No process arrival during execution**: The program assumes that no new processes arrive while a process is currently executing

6.3.4 **CPU burst time**: The program assumes that the CPU burst times for each process are known and provided as input..

6.4 Dependencies

6.4.1 CPU burst time

6.4.2 Arrival time

6.4.3 Waiting time

6.5 Code/ Pseudo Code

```
#include <stdio.h>

struct Process {
    int process_id;
    int arrival_time;
    int burst_time;
};

void fifo_scheduling(struct Process processes[], int n) {
    int completion_time[n];
    int turnaround_time[n];
    int waiting_time[n];

    // Calculate completion time for each process
    for (int i = 0; i < n; i++) {
        if (i == 0) {
            completion_time[i] = processes[i].arrival_time +
processes[i].burst_time;
        } else {
            completion_time[i] = (processes[i].arrival_time >
completion_time[i - 1] ? processes[i].arrival_time : completion_time[i -
1]) + processes[i].burst_time;
        }
    }
    // Calculate turnaround time and waiting time for each process
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = completion_time[i] -
processes[i].arrival_time;
        waiting_time[i] = turnaround_time[i] - processes[i].burst_time;
    }
    // Print results
    printf("Process\tArrival T\tBurst Time\tCompletion T\tTurnaround
T\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t\t%d\t\t\t%d\t\t\t%d\t\t\t\t%d\t\t\t\t%d\n",
```

```
processes[i].process_id, processes[i].arrival_time,
processes[i].burst_time, completion_time[i], turnaround_time[i],
waiting_time[i]);
    }
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    printf("Enter burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        processes[i].arrival_time = 0;
        processes[i].process_id = i + 1;
    }
    fifo_scheduling(processes, n);
    return 0;
}
plt.show()
```

6.6 Results

6.6.1 Test Case

```
Enter the number of processes: 5
Enter burst time for each process:
Process 1: 6
Process 2: 2
Process 3: 5
Process 4: 1
Process 5: 8
```

Process	Arrival T	Burst Time	Completion T	Turnaround T	Waiting Time
1	0	6	6	6	0
2	0	2	8	8	6
3	0	5	13	13	8
4	0	1	14	14	13
5	0	8	22	22	14

6.6.2 Result Analysis

6.6.2.1 Advantages

Understanding the concept of FIFS method of CPU scheduling
On the basis of arrival time and burst time

6.6.2.2 Issues : NA

Experiment - 3

Title:- To write a c program to simulate the CPU scheduling SHORTEST JOB FIRST (SJF)

1. **Outcome:-** Execution of the shortest job first leads to reduced waiting time and improved efficiency.

2. **Objectives**
   Minimize the average waiting time for processes by prioritizing the shortest job first.

3. **Nomenclature, theory with self-assessment questionnaire:-**

4.1 Nomenclature:

`n`	`No of processes`
`waiting_time`	`Waiting time`
`arrival_time`	`Arrival time`
`burst_time`	`Burst time`
`turnaround_time[n];`	`Turnaround time`
`completion_time[i]`	`Completion time`
`waiting_time[n];`	`Waiting time`

3.2 Solution:

waiting time for processes. It works by prioritizing the process with the shortest burst time first. When a new process arrives, its burst time is compared with the remaining burst times of the processes in the ready queue. If the new process has a shorter burst time, it preempts the currently executing process and gets scheduled next.

The key idea behind SJF is to execute shorter jobs first to reduce waiting time. This strategy optimizes the utilization of CPU time by minimizing the time spent on context switching and overhead.

SJF can be implemented in both preemptive and non-preemptive modes. In the preemptive mode, if a new process with a shorter burst time arrives during the

execution of a process, the current process is interrupted. In the non-preemptive mode, the executing process completes its burst time before the next shortest job is scheduled.

Overall, SJF aims to improve efficiency and reduce waiting time by prioritizing shorter jobs, resulting in faster completion of processes.

3.3 Assumptions:

3.3.1 **Burst time estimation:** The assumption is that the burst time for each process is known or can be accurately estimated. No priority consideration:

3.3.2 **Process arrival order:** It is assumed that the arrival order of processes is known in advance or can be determined.

3.3.3 **Deterministic behavior:** The assumption is that the behavior of processes, particularly their burst times, remains consistent and does not vary significantly during their execution

3.3.4 **No priority inversion (for non-preemptive SJF):** In the case of non-preemptive SJF, it is assumed that no priority inversion occurs, meaning that higher priority processes will not be blocked or delayed by lower priority processes..

3.4 Dependencies

3.4.1 CPU burst time

3.4.2 Arrival time

3.4.3 Waiting time

3.5 Code/ Pseudo Code

```
#include <stdio.h>

struct Process {
    int processID;
    int burstTime;
    int arrivalTime;
    int turnaroundTime;
};

void sjfScheduling(struct Process proc[], int n) {
    int currentTime = 0, totalWaitingTime = 0;
    float averageWaitingTime;

    // Sort processes based on burst time in ascending order
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (proc[i].burstTime > proc[j].burstTime) {
                struct Process temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }
```

```
    printf("Process\tArrival Time\tBurst Time\tTurnaround T \tWaiting
Time\n");

    // Calculate turnaround time and waiting time for each process
    for (int i = 0; i < n; i++) {
        proc[i].turnaroundTime = currentTime - proc[i].arrivalTime;
        proc[i].turnaroundTime += proc[i].burstTime;
        proc[i].turnaroundTime = proc[i].turnaroundTime -
proc[i].arrivalTime;
        proc[i].arrivalTime = currentTime;
        printf("%d\t\t\t%d\t\t\t%d\t\t\t\t%d\t\t\t%d\n", proc[i].processID,
proc[i].arrivalTime, proc[i].burstTime, proc[i].turnaroundTime,
proc[i].turnaroundTime - proc[i].burstTime);
        totalWaitingTime += proc[i].turnaroundTime - proc[i].burstTime;
        currentTime += proc[i].burstTime;
    }

    averageWaitingTime = (float) totalWaitingTime / n;
    printf("\nAverage Waiting Time: %.2f\n", averageWaitingTime);
}

int main() {
    int n;

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

    struct Process processes[n];

    // Input process ID, arrival time, and burst time
    for (int i = 0; i < n; i++) {
        printf("Enter the Process ID for process %d: ", i + 1);
        scanf("%d", &processes[i].processID);
        printf("Enter the Arrival Time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the Burst Time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
    }

    printf("\nShortest Job First Scheduling:\n");
    sjfScheduling(processes, n);

    return 0;
}
```

3.6 Results

3.6.1 Test Case

```
Enter the number of processes: 3
Enter the Process ID for process 1: 1
Enter the Arrival Time for process 1: 0
Enter the Burst Time for process 1: 2
Enter the Process ID for process 2: 2
Enter the Arrival Time for process 2: 2
Enter the Burst Time for process 2: 5
Enter the Process ID for process 3: 3
Enter the Arrival Time for process 3: 3
Enter the Burst Time for process 3: 1
Shortest Job First Scheduling:
Process Arrival Time    Burst Time  Turnaround T    Waiting Time
3           0           1               -5          -6
1           1           2               3           1
2           3           5               4           -1

Average Waiting Time: -2.00
```

3.6.2 Result Analysis

3.6.2.1 Advantages

Understanding the concept of SJF method of CPU scheduling
On the basis of arrival time and burst time

3.6.2.2 Issues : NA

Experiment - 4

Title:- To write a c program to simulate the CPU scheduling algorithm Priority Scheduling algorithm

1. **Outcome:-** The outcome is an optimized schedule where processes are executed based on their priority. This ensures that processes with higher priority are executed first, resulting in better utilization of resources and potentially reducing the waiting time for high-priority tasks.
2. **Objectives**
   The objective of Priority Scheduling algorithm is to prioritize the execution of processes based on their priority values.
3. **Nomenclature, theory with self-assessment questionnaire:-**

4.1 Nomenclature:

n	No of processes
processID	Process ID
priority	priority
waiting_time	Waiting time
arrival_time	Arrival time
burst_time	Burst time
turnaround_time[n];	Turnaround time
completion_time[i]	Completion time
waiting_time[n];	Waiting time
averageWaitingTime	Average Waiting Time
totalWaitingTime	Total Waiting Time

3.2 Solution:

The solution to the Priority Scheduling algorithm involves assigning priorities to processes and scheduling them accordingly. The processes are sorted based on their priority values in ascending order, with the highest priority given the highest preference for execution. This ensures that critical or important tasks are executed promptly, optimizing the

system's efficiency. By executing high-priority processes first, the algorithm aims to reduce the waiting time for critical tasks and improve overall system performance. The outcome of implementing Priority Scheduling is an optimized schedule where processes are executed based on their priority, resulting in better resource utilization and potentially enhancing the system's responsiveness to time-sensitive tasks.

3.3 Assumptions:

3.3.1 **Lower priority values indicate higher priority:** The algorithm assumes that lower priority values correspond to higher priority for execution.

3.3.2 **Processes have predefined arrival times:** The algorithm assumes that the arrival times for all processes are known in advance.No process arrival during execution:

3.3.3 **Priority values are unique**: The algorithm assumes that each process has a unique priority value. There are no two processes with the same priority.

3.3.4 **All processes are available at the start:** The algorithm assumes that all processes are available and ready for execution at the beginning of the scheduling process.

3.4 Dependencies

3.4.1 Input Mechanism:

3.4.2 Sorting Algorithm

3.4.3 Timer/Current Time

3.5 Code/ Pseudo Code

```
#include <stdio.h>

struct Process {
    int processID;
    int arrivalTime;
    int burstTime;
    int priority;
    int waitingTime;
    int turnaroundTime;
};

void priorityScheduling(struct Process proc[], int n) {
    int currentTime = 0, totalWaitingTime = 0;
    float averageWaitingTime;

    // Sort processes based on priority in ascending order
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (proc[i].priority > proc[j].priority) {
                struct Process temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
```

```
        }
    }

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

    // Calculate turnaround time and waiting time for each process
    for (int i = 0; i < n; i++) {
        proc[i].turnaroundTime = currentTime - proc[i].arrivalTime;
        proc[i].turnaroundTime += proc[i].burstTime;
        proc[i].waitingTime = proc[i].turnaroundTime - proc[i].burstTime;
        proc[i].arrivalTime = currentTime;
        printf("%d\t\t%d\t\t\t\t%d\t\t\t%d\t\t\t\t%d\t\t\t\t%d\n",
proc[i].processID, proc[i].arrivalTime, proc[i].burstTime,
proc[i].priority, proc[i].turnaroundTime, proc[i].waitingTime);
        totalWaitingTime += proc[i].waitingTime;
        currentTime += proc[i].burstTime;
    }

    averageWaitingTime = (float) totalWaitingTime / n;
    printf("\nAverage Waiting Time: %.2f\n", averageWaitingTime);
}

int main() {
    int n;

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

    struct Process processes[n];

    // Input process ID, arrival time, burst time, and priority
    for (int i = 0; i < n; i++) {
        printf("Enter the Process ID for process %d: ", i + 1);
        scanf("%d", &processes[i].processID);
        printf("Enter the Arrival Time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrivalTime);
        printf("Enter the Burst Time for process %d: ", i + 1);
        scanf("%d", &processes[i].burstTime);
        printf("Enter the Priority for process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    printf("\nPriority Scheduling:\n");
    priorityScheduling(processes, n);

    return 0;
}
```

3.6 Results

3.6.1 Test Case

```
Enter the number of processes: 3
Enter the Process ID for process 1: 1
Enter the Arrival Time for process 1: 0
Enter the Burst Time for process 1: 5
Enter the Priority for process 1: 2
Enter the Process ID for process 2: 2
Enter the Arrival Time for process 2: 1
Enter the Burst Time for process 2: 2
Enter the Priority for process 2: 1
Enter the Process ID for process 3: 3
Enter the Arrival Time for process 3: 0
Enter the Burst Time for process 3: 5
Enter the Priority for process 3: 3
Priority Scheduling:
Process Arrival Time    Burst T     Priority    Turnaround T    Waiting Time
2       0               2           1               1               -1
1       2               5           2               7               2
3       7               5           3               12              7

Average Waiting Time: 2.67
```


3.6.2 Result Analysis

3.6.2.1 Advantages

Understanding the concept of priority Scheduling method of CPU scheduling

On the basis of arrival time and burst time

3.6.2.2 Issues : NA

Experiment - 05

1. **Title:-** Round-Robin CPU Scheduling Algorithm Simulation
2. **Outcome:-** The outcome of this simulation is to demonstrate the behavior of the Round-Robin CPU scheduling algorithm by simulating the execution of processes in a time-sliced manner..

3. **Objectives**

The objective of this simulation is to implement and analyze the Round-Robin CPU scheduling algorithm. This algorithm aims to provide fair and time-sliced execution for multiple processes in a preemptive manner. The simulation will demonstrate the behavior of the algorithm, evaluate its performance in terms of turnaround time and waiting time, and highlight its advantages in managing CPU utilization.

4. **Nomenclature, theory with self-assessment questionnaire:-**

4.1 Nomenclature:

`n`	`No of processes`
`waiting_time`	`Waiting time`
`arrival_time`	`Arrival time`
`burst_time`	`Burst time`
`turnaround_time[n];`	`Turnaround time`
`completion_time[i]`	`Completion time`
`waiting_time[n];`	`Waiting time`

4.2 Solution:

Accept input for the number of processes, arrival time, and burst time for each process.
Implement the Round-Robin scheduling algorithm using a queue data structure.
Execute the processes in a preemptive manner based on the time quantum.
Calculate completion time, turnaround time, and waiting time for each process.
Calculate the average turnaround time and average waiting time.
Display the results and analysis of the simulation..

4.3 Assumptions:

4.3.1 **Burst time estimation:** The assumption is that the burst time for each process is known or can be accurately estimated. No priority consideration:

4.3.2 **Process arrival order:** It is assumed that the arrival order of processes is known in advance or can be determined.

4.3.3 **Deterministic behavior:** The assumption is that the behavior of processes, particularly their burst times, remains consistent and does not vary significantly during their execution

4.3.4 **No priority inversion (for non-preemptive SJF):** In the case of non-preemptive SJF, it is assumed that no priority inversion occurs, meaning that higher priority processes will not be blocked or delayed by lower priority processes..

4.4 Dependencies

4.4.1 CPU burst time

4.4.2 Arrival time

4.4.3 Waiting time

4.5 Code/ Pseudo Code

```
// Round-Robin CPU Scheduling Algorithm Simulation
#include <stdio.h>

struct Process {
    int process_id;
    int arrival_time;
    int burst_time;
};

void round_robin_scheduling(struct Process processes[], int n, int
time_quantum) {
    int remaining_time[n];
    int completion_time[n];
    int turnaround_time[n];
    int waiting_time[n];

    // Initialize remaining time for all processes
    for (int i = 0; i < n; i++) {
        remaining_time[i] = processes[i].burst_time;
    }

    int current_time = 0;
    int completed_processes = 0;

    while (completed_processes < n) {
        for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0) {
                if (remaining_time[i] <= time_quantum) {
                    current_time += remaining_time[i];
                    completion_time[i] = current_time;
                    remaining_time[i] = 0;
                    completed_processes++;
                } else {
                    current_time += time_quantum;
                    remaining_time[i] -= time_quantum;
```

```
                }
            }
        }
    }

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

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

    // Print results
    printf("Process\tArrival Time\tBurst Time\tCompletion Time\tTurnaround
Time\tWaiting 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].process_id, processes[i].arrival_time,
processes[i].burst_time, completion_time[i], turnaround_time[i],
waiting_time[i]);
    }
    printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);
    printf("Average Waiting Time: %.2f\n", avg_waiting_time);
}

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

    struct Process processes[n];

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

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

```
    round_robin_scheduling(processes, n, time_quantum);

    return 0;
}
```

4.6 Results

4.6.1 Test Case

```
Enter the number of processes: 4
Enter arrival time and burst time for each process:
Process 1: 0 6
Process 2: 2 4
Process 3: 4 8
Process 4: 6 3
Enter the time quantum: 2

Process     Arrival Time    Burst Time      Completion Time Turnaround Time
1           0               6               17              17
2           2               4               16              14
3           4               8               18              14
4           6               3               20              14
```

4.6.2 Result Analysis

4.6.2.1 Advantages

Understanding the concept of round-robin algorithm method of CPU scheduling

On the basis of arrival time and burst time

4.6.2.2 Issues : NA

Experiment - 6

1.Title:- Simulating Producer-Consumer Problem using Semaphores in C

2.Outcome:- To demonstrate the synchronization of producer and consumer threads using semaphores to solve the classic producer-consumer problem..

3. Objectives

Implement a solution to the producer-consumer problem using semaphores.
Ensure synchronization between producer and consumer threads to prevent race conditions.
Demonstrate the proper handling of buffer access by multiple thread.

4. Nomenclature, theory with self-assessment questionnaire:-

4.1 Nomenclature:

Buffer	**shared data structure where the producer produces items, and the consumer consumes them.**
Semaphore	**A synchronization mechanism used to control access to shared resources.**
Producer	**hread responsible for producing items and adding them to the buffer.**
Consumer	A thread responsible for consuming items from the buffer.

4.2 Solution:

initialize two semaphores: empty and full.

The empty semaphore represents the number of empty slots in the buffer, and the full semaphore represents the number of filled slots.

The producer waits for an empty slot in the buffer (empty semaphore) before adding an item to the buffer.

After adding an item, the producer signals the full semaphore to indicate that a slot is now filled.

The consumer waits for a filled slot in the buffer (full semaphore) before consuming an item.

After consuming an item, the consumer signals the empty semaphore to indicate that a slot is now empty.

4.3 Assumptions:

4.3.1 The buffer has a fixed size and can hold a certain number of items.

4.3.2 The producer and consumer threads are the only threads accessing the buffer.

4.3.3 The producer produces items at a faster rate than the consumer consumes them.

4.3.4 The buffer access is properly synchronized using semaphores.

4.4 Dependencies

4.4.1 Input Mechanism:

4.4.2 Sorting Algorithm

4.4.3 Timer/Current Time

4.5 Code/ Pseudo Code

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
sem_t empty, full;
int in = 0, out = 0;

void *producer(void *arg) {
    int item = 1;
    while (1) {
        // Produce an item
        item = item * 2;

        // Wait for an empty slot in the buffer
        sem_wait(&empty);

        // Add the item to the buffer
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;

        // Signal that a slot is now filled
        sem_post(&full);

        printf("Producer produced item: %d\n", item);
    }
}

void *consumer(void *arg) {
    int item;
    while (1) {
        // Wait for a filled slot in the buffer
        sem_wait(&full);

        // Consume an item from the buffer
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        // Signal that a slot is now empty
        sem_post(&empty);
```

```
        printf("Consumer consumed item: %d\n", item);
    }
}

int main() {
    pthread_t producer_thread, consumer_thread;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for threads to finish (which will not happen in this case)

    return 0;
}
```

4.6 Results

4.6.1 Test Case

```
Producer produced item: 2
Consumer consumed item: 2
Producer produced item: 4
Consumer consumed item: 4
Producer produced item: 8
Consumer consumed item: 8
...
```

4.6.2 Result Analysis

4.6.2.1 Advantages

The use of semaphores ensures proper synchronization and avoids race conditions.

The solution allows multiple producers and consumers to operate concurrently.

It provides a solution to the producer-consumer problem, allowing safe sharing of resources between threads.

4.6.2.2 Issues : NA

Experiment - 07

1. **Title:-** To Write a C program to simulate the following contiguous memory allocation techniques.
2. **Outcome:-** Able to understand how memory allocation technique works.

3. **Objectives**
   It is done in three phases worst fit, best fit, first fit.

4. **Nomenclature, theory with self-assessment questionnaire:-**

4.1 Nomenclature:

frag	Fragment
ff	First fit
bf	Best fit
nb	No. of blocks
nf	No. of files
I	Counter for loop
nf	No. of files

4.2 Solution:

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

4.3 Assumptions:

4.3.1 **Burst time estimation:** The assumption is that the burst time for each process is known or can be accurately estimated. No priority consideration:

4.3.2 **Process arrival order:** It is assumed that the arrival order of processes is known in advance or can be determined.

4.3.3 **Deterministic behavior:** The assumption is that the behavior of processes, particularly their burst times, remains consistent and does not vary significantly during their execution

4.3.4 **No priority inversion (for non-preemptive SJF):** In the case of non-preemptive SJF, it is assumed that no priority inversion occurs, meaning that higher priority processes will not be blocked or delayed by lower priority processes..

4.4 Dependencies

4.4.1 CPU burst time

4.4.2 Arrival time

4.4.3 Waiting time

4.5 Code/ Pseudo Code

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
sem_t empty, full;
int in = 0, out = 0;

void *producer(void *arg) {
    int item = 1;
    while (1) {
        // Produce an item
        item = item * 2;

        // Wait for an empty slot in the buffer
        sem_wait(&empty);

        // Add the item to the buffer
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;

        // Signal that a slot is now filled
        sem_post(&full);

        printf("Producer produced item: %d\n", item);
    }
}

void *consumer(void *arg) {
```

```
    int item;
    while (1) {
        // Wait for a filled slot in the buffer
        sem_wait(&full);

        // Consume an item from the buffer
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        // Signal that a slot is now empty
        sem_post(&empty);

        printf("Consumer consumed item: %d\n", item);
    }
}

int main() {
    pthread_t producer_thread, consumer_thread;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for threads to finish (which will not happen in this case)

    return 0;
}
```

4.6 Results

4.6.1 Test Case

```
Producer produced item: 2
Consumer consumed item: 2
Producer produced item: 4
Consumer consumed item: 4
Producer produced item: 8
Consumer consumed item: 8
---
```

4.6.2 Result Analysis

4.6.2.1 Advantages

To know more about the memory allocation techniques.

4.6.2.2 Issues : NA

Experiment - 8

1.Title:- Simulating Producer-Consumer Problem using Semaphores in C

2.Outcome:- The program aims to demonstrate the synchronization of multiple producers and consumers using semaphores to avoid race conditions and ensure proper data exchange.

3. Objectives

To simulate the producer-consumer problem and showcase the use of semaphores for synchronization.
To prevent race conditions by implementing mutual exclusion using semaphores.
To ensure proper data exchange between producers and consumers by utilizing counting semaphores.
To illustrate the concept of bounded buffer and the role of semaphores in maintaining its integrity.

4. Nomenclature, theory with self-assessment questionnaire:-

4.1 Nomenclature:

Buffer	**shared data structure where the producer produces items, and the consumer consumes them.**
Semaphore	**A synchronization mechanism used to control access to shared resources.**
Producer	**hread responsible for producing items and adding them to the buffer.**
Consumer	A thread responsible for consuming items from the buffer.
Full Semaphore	A counting semaphore indicating the number of occupied slots in the buffer.
Empty Semaphore	A binary semaphore used for mutual exclusion to protect the critical sections.

4.2 Solution:

The solution involves implementing a bounded buffer using semaphores. The buffer has a fixed size and can hold a limited number of items. Producers and consumers access the buffer concurrently. The full and empty semaphores are used to regulate the access to the buffer. When a producer produces an item, it decreases the empty semaphore count and increases the full semaphore count. When a consumer consumes an item, it decreases the full semaphore count and increases the empty semaphore count. The mutex semaphore ensures that only

one process can access the critical sections at a time, preventing race conditions.

4.3 Assumptions:

4.3.1 The program assumes a fixed-size buffer.

4.3.2 The buffer can hold a maximum number of items.

4.3.3 Producers and consumers are implemented as separate processes or threads.

4.3.4 The program assumes that the semaphores used are implemented correctly and initialized appropriately.

4.3.5 The program assumes that there are no priority considerations between producers and consumers..

4.4 Dependencies

4.4.1 Input Mechanism:

4.4.2 Sorting Algorithm

4.4.3 Timer/Current Time

4.5 Code/ Pseudo Code

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

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 3
#define NUM_CONSUMERS 2

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t mutex, empty, full;

void *producer(void *arg) {
    int item;
    for (int i = 0; i < BUFFER_SIZE; i++) {
        item = i + 1;

        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = item;
        printf("Producer %d produced item %d\n", *(int *)arg, item);
        in = (in + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&full);
    }
    pthread_exit(NULL);
}

void *consumer(void *arg) {
```

```
    int item;
    for (int i = 0; i < BUFFER_SIZE; i++) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = buffer[out];
        printf("Consumer %d consumed item %d\n", *(int *)arg, item);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t producers[NUM_PRODUCERS];
    pthread_t consumers[NUM_CONSUMERS];
    int producer_ids[NUM_PRODUCERS];
    int consumer_ids[NUM_CONSUMERS];

    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        producer_ids[i] = i + 1;
        pthread_create(&producers[i], NULL, producer, &producer_ids[i]);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        consumer_ids[i] = i + 1;
        pthread_create(&consumers[i], NULL, consumer, &consumer_ids[i]);
    }

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

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

    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

4.6 Results

4.6.1 Test Case

```
Producer produced item: 2
Consumer consumed item: 2
Producer produced item: 4
Consumer consumed item: 4
Producer produced item: 8
Consumer consumed item: 8
...
```

4.6.2 Result Analysis

4.6.2.1 Advantages

It allows for proper synchronization between producers and consumers.

The bounded buffer implementation ensures that the buffer remains within its capacity.

Semaphores provide a flexible and efficient mechanism for synchronization and coordination in concurrent programs.

4.6.2.2 Issues : NA