VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

B.Tech Programme: Computer Science & Engineering

Course Title: Operating Systems Lab

Course Code: CIC-359

Semester: 5th

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

A++

VISION OF INSTITUTE

MISSION OF INSTITUTE

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

A++

INDEX

S.No	EXP.	Date	Marks			Remark	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

A++

INDEX

S.No	EXP.	Date	Marks			Remark	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			

PROGRAM 1

PROBLEM STATEMENT: Write a program to implement CPU scheduling for first come first serve.

THEORY:

First-Come, First-Served (FCFS) is one of the simplest and most straightforward CPU scheduling algorithms used in operating systems. It operates on a queue-based mechanism where processes are executed in the order they arrive in the ready queue. This means that the process that arrives first will be the first to get executed, and it will run to completion before the next process starts.

Basic Operation

- **Queue Management**: In FCFS scheduling, processes are maintained in a queue, typically a FIFO (First In, First Out) queue.
- **Execution Order**: When the CPU becomes available, the process at the head of the queue is selected for execution.
- **Completion**: A process runs to completion before the next process in the queue starts execution.

Waiting Time (WT): The average waiting time in FCFS is calculated as the sum of the waiting times for all processes divided by the number of processes. It tends to be high, especially if there is a large disparity in process burst times.

Turnaround Time (TAT): The turnaround time is the total time taken from the arrival of a process to its completion. In FCFS, turnaround time can also be high due to the sequential nature of execution.

ALGORITHM:

PROBLEM:

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

PROBLEM SOLUTION:

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	0	4	4	4	0
2	1	3	7	6	3
3	2	1	8	6	5
4	3	2	10	7	5
5	4	5	15	11	6

CODE:

```
#include <iostream>

using namespace std;

struct process

{ int PID;

int arrival_time;

int burst_time;

int completion_time;

int turn_around_time;

int waiting_time;

};

void fcfs_scheduling(process processes[],int n)

{ int current_time=0;

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

{ processes[i].completion_time = current_time + processes[i].burst_time;

processes[i].turn_around_time = processes[i].completion_time - processes[i].arrival_time;

processes[i].waiting_time = processes[i].turn_around_time - processes[i].burst_time ;

current_time+= processes[i].burst_time;

}

}

void print_processes_data(process processes[],int n)

{ cout<<"Process ID \t Arrival Time \t Burst Time \t Completion Time \t Turn Around Time \t
Waiting Time"<<endl;

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

{ cout<< processes[i].PID<<"\t \t \t \t"<< processes[i].arrival_time<<"\t \t \t
\t"<<processes[i].burst_time<<"\t \t \t \t"<<processes[i].completion_time<<"\t \t \t \t \t
\t"<<processes[i].turn_around_time<<"\t \t \t \t"<<processes[i].waiting_time <<endl;

}

}

int main()

{int n;

cout<< "Enter no of processes:" ;
```

```
cin >> n;
process processes[n];
for(int i=0;i<n;i++)
{ cout << "Enter Process ID:" <<endl;
cin >> processes[i].PID;
cout << "Enter Process Arrival Time:" <<endl;
cin >> processes[i].arrival_time;
cout << "Enter Process Burst Time:" <<endl;
cin >> processes[i].burst_time;
}
fcfs_scheduling(processes,n);
print_processes_data(processes,n);
return 0;
}
```

OUTPUT:

```
Enter no of processes:5
Enter Process ID:
1
Enter Process Arrival Time:
0
Enter Process Burst Time:
4
Enter Process ID:
2
Enter Process Arrival Time:
1
Enter Process Burst Time:
3
Enter Process ID:
3
Enter Process Arrival Time:
2
Enter Process Burst Time:
1
Enter Process ID:
4
Enter Process Arrival Time:
3
Enter Process Burst Time:
2
Enter Process ID:
5
Enter Process Arrival Time:
4
Enter Process Burst Time:
5
Process ID   Arrival Time    Burst Time      Completion Time     Turn Around Time    Waiting Time
1               0               4               4                   4               0
2               1               3               7                   6               3
3               2               1               8                   6               5
4               3               2               10                  7               5
5               4               5               15                  11              6
```

LEARNING OUTCOMES:

PROGRAM 2

PROBLEM STATEMENT: Write a program to implement CPU scheduling for shortest job first.

THEORY: Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest execution (burst) time from the ready queue. It is designed to minimize the average waiting time of processes by prioritizing shorter jobs, which typically leads to better overall system performance. SJF can be implemented in either non-preemptive or preemptive (Shortest Remaining Time First, SRTF) forms.

Basic Operation

- **Queue Management**: In SJF scheduling, processes are maintained in a ready queue. The key distinction is that the scheduler selects the process with the shortest burst time, not necessarily the one that arrived first.
- **Execution Order**: When the CPU becomes available, the scheduler scans the ready queue and picks the process with the smallest CPU burst. If the system is using preemptive SJF (SRTF), it will interrupt the current process if a new process arrives with a shorter remaining time.
- **Completion**: In non-preemptive SJF, a selected process runs to completion before the next process with the shortest burst time is chosen. In preemptive SJF (SRTF), the running process may be preempted if a shorter job arrives, and the CPU is assigned to the new process.

ALGORITHM:

PROBLEM:

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

PROBLEM SOLUTION:

P	AT	BT	CT	TAT	WT
P1	3	1	7	4	3
P2	1	4	16	15	11
P3	4	2	9	5	3
P4	0	6	6	6	0
P5	2	3	12	10	7

CODE:

```
#include <iostream>
using namespace std;

typedef struct{
    int id;
    int at;  // Arrival Time
    int bt;  // Burst Time
    int ct;  // Completion Time
    int tat; // Turnaround Time
    int wt;  // Waiting Time
}process;

int main()
{
    int n = 5;
    process p[n], q[n];

    // Input process details
    for(int i = 0; i < n; i++)
    {
        p[i].id = i + 1;
        printf("Enter arrival time and burst time for process P%d: ", i + 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
    }

    // Sort according to arrival time
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if(p[j].at > p[j + 1].at)
            {
                process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    int front = 0, rear = 1;
    int time = p[0].at;
    q[front] = p[0];
    int i = 1;
```

```
// Process scheduling simulation
while(i < n || front != rear)
{
    time += q[front].bt;
    for(int k = 0; k < n; k++)
    {
        if(p[k].id == q[front].id)
        {
            p[k].ct = time;  // Calculate completion time
            break;
        }
    }

    while(i < n && p[i].at <= time)
    {
        q[rear] = p[i];
        rear++;
        i++;
    }

    front++;

    // Sort ready queue by burst time for SJF
    for(int j = front + 1; j < rear; j++)
    {
        if(q[front].bt > q[j].bt)
        {
            process temp = q[front];
            q[front] = q[j];
            q[j] = temp;
        }
    }
}

// Calculate Turnaround Time (TAT) and Waiting Time (WT)
for (int i = 0; i < n; i++)
{
    p[i].tat = p[i].ct - p[i].at;  // TAT = CT - AT
    p[i].wt = p[i].tat - p[i].bt;  // WT = TAT - BT
}
// Arrange processes according to their original IDs
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n - i - 1; j++)
    {
```

```
            if(p[j].id > p[j + 1].id)
            {
                process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
    // Print the results
    printf("P\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++)
    {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);
    }
    return 0;
}
```

OUTPUT:

```
Enter arrival time and burst time for process P1: 3 1
Enter arrival time and burst time for process P2: 1 4
Enter arrival time and burst time for process P3: 4 2
Enter arrival time and burst time for process P4: 0 6
Enter arrival time and burst time for process P5: 2 3
P       AT      BT      CT      TAT     WT
P1      3       1       7       4       3
P2      1       4       16      15      11
P3      4       2       9       5       3
P4      0       6       6       6       0
P5      2       3       12      10      7

--------------------------------
Process exited after 14.41 seconds with return value 0
Press any key to continue . . .
```

LEARNING OUTCOMES:

PROGRAM 3

PROBLEM STATEMENT: Write a program to implement CPU scheduling for Round Robin.

THEORY:

Round Robin (RR) is a CPU scheduling algorithm designed for time-sharing systems. It is one of the simplest and most widely used scheduling techniques. In this algorithm, each process is assigned a fixed **time quantum** or **time slice** during which it can execute. If a process doesn't finish within its assigned time quantum, it is preempted and moved to the back of the ready queue, allowing the next process to execute.

Basic Operation

- **Queue Management**: Processes are maintained in a **FIFO (First In, First Out)** queue. When a process arrives, it is added to the end of the ready queue.
- **Execution Order**: The CPU scheduler picks the process at the front of the queue for execution. The process is allowed to run for a time period equal to the **time quantum**. If the process finishes its execution before the time quantum expires, it is removed from the queue. Otherwise, the process is preempted and placed at the back of the queue, and the next process in line is selected for execution.
- **Completion**: A process continues this cycle of execution and requeueing until it completes. Once a process finishes, it is removed from the system and no longer re-enters the queue.

ALGORITHM:

PROBLEM:

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

PROBLEM SOLUTION:

P	AT	BT	CT	TAT	WT
P1	0	5	13	13	8
P2	1	3	12	11	8
P3	2	1	5	3	2
P4	3	2	9	6	4
P5	4	3	14	10	7

CODE:

```
#include <iostream>
using namespace std;

typedef struct{
    int id;
    int at;  // Arrival Time
    int bt;  // Burst Time
    int ct;  // Completion Time
    int tat; // Turnaround Time
    int wt;  // Waiting Time
}process;

int main()
{
    printf("Enter number of processes: ");
    int n;
    scanf("%d", &n);
    process p[n], rq[64];
    int tq = 2; // Time Quantum
    int front = 0, rear = 1;

    for (int i = 0; i < n; i++)
    {
        p[i].id = i + 1;
        printf("Enter arrival time and burst time for P%d: ", i + 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
    }

    // Sort processes according to arrival time
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (p[j].at > p[j + 1].at)
            {
                process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    rq[front] = p[0];
    int time = p[0].at;
```

```
int i = 1;

while (i < n || front != rear)
{
    if (rq[front].bt > tq)
    {
        time += tq;
        rq[front].bt -= tq;
        while (i < n && p[i].at <= time)
        {
            rq[rear] = p[i];
            rear++;
            i++;
        }
        rq[rear++] = rq[front];  // Place back in the queue
    }
    else
    {
        time += rq[front].bt;
        for (int k = 0; k < n; k++)
        {
            if (p[k].id == rq[front].id)
            {
                p[k].ct = time;  // Set completion time
                break;
            }
        }
    }
    front++;
}

// Calculate Turnaround Time (TAT) and Waiting Time (WT)
for (int i = 0; i < n; i++)
{
    p[i].tat = p[i].ct - p[i].at;  // TAT = CT - AT
    p[i].wt = p[i].tat - p[i].bt;  // WT = TAT - BT
}

// Arrange processes according to their IDs
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n - i - 1; j++)
    {
        if (p[j].id > p[j + 1].id)
        {
```

```
            process temp = p[j];
            p[j] = p[j + 1];
            p[j + 1] = temp;
        }
    }
}

printf("P\tAT\tBT\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++)
{
    printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);
}
return 0;
}
```

OUTPUT:

```
Enter number of processes: 5
Enter arrival time and burst time for P1: 0 5
Enter arrival time and burst time for P2: 1 3
Enter arrival time and burst time for P3: 2 1
Enter arrival time and burst time for P4: 3 2
Enter arrival time and burst time for P5: 4 3
P       AT      BT      CT      TAT     WT
P1      0       5       13      13      8
P2      1       3       12      11      8
P3      2       1       5       3       2
P4      3       2       9       6       4
P5      4       3       14      10      7

--------------------------------
Process exited after 17.21 seconds with return value 0
Press any key to continue . . .
```

LEARNING OUTCOMES:

REQUEST	4	7	6	1	7	6	1	2	7	2
FRAME 2										
HIT/MISS										

REQUEST	4	7	6	1	7	6	1	2	7	2
FRAME 2										
HIT/MISS										

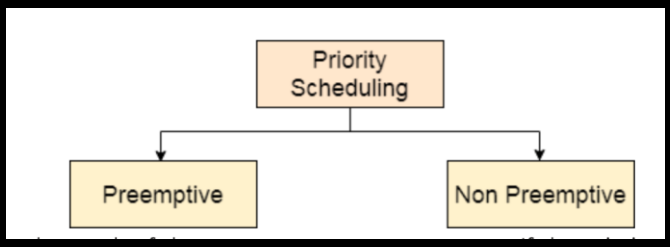
REQUEST	4	7	6	1	7	6	1	2	7	2
FRAME 2										
HIT/MISS										

```
}
```

```
Enter size of reference String: 9
Enter reference string: 6 5 3 5 6 2 3 7 5
Enter frame size: 3
Hits: 3
Page Faults: 6
Hit Ratio: 0.333333
Page Fault Ratio: 0.666667
```

```
Enter size of reference String: 9
Enter reference string: 4 5 3 2 7 5 6 1 4
Enter frame size: 3
Hits: 2
Page Faults: 7
Hit Ratio: 0.222222
Page Fault Ratio: 0.777778
Final frame state: 4 1 7
```

```
Enter size of reference String: 9
Enter reference string: 4 5 6 1 5 4 7 8 6
Enter frame size: 3
Hits: 1
Page Faults: 8
Hit Ratio: 0.111111
Page Fault Ratio: 0.888889
Final frame state: 7 8 6
```


Priority Scheduling
Preemptive
Non Preemptive

PROCESS ID	PRIORITY	ARRIVAL TIME	BURST TIME
P2	3	1	3
P4	5	3	5

PROCESS ID	PRIORITY	ARRIVAL TIME	BURST TIME	COMPLETION TIME	TURN-AROUND TIME	WAITING TIME
P2	3	1	3			
P4	5	3	5			

PROCESS ID	PRIORITY	ARRIVAL TIME	BURST TIME	COMPLETION TIME	TURN-AROUND TIME	WAITING TIME
P2	3	1	3			
P4	5	3	5			

```
Enter number of processes: 4
Enter the arrival time of process 1: 2
Enter the burst time of process 1: 5
Enter the priority of process 1: 7
Enter the arrival time of process 2: 0
Enter the burst time of process 2: 4
Enter the priority of process 2: 6
Enter the arrival time of process 3: 1
Enter the burst time of process 3: 3
Enter the priority of process 3: 5
Enter the arrival time of process 4: 3
Enter the burst time of process 4: 1
Enter the priority of process 4: 2
Switching to process 2 at time 0
Switching to process 3 at time 1
Switching to process 4 at time 3
Switching to process 3 at time 4
Switching to process 2 at time 5
Switching to process 1 at time 8

--------------------
PID     AT      BT      Priority        CT      TAT     WT
1       2       5       7               13      11      6
2       0       4       6               8       8       4
3       1       3       5               5       4       1
4       3       1       2               4       1       0
```

```
Enter number of processes: 5
Enter the arrival time of process 1: 0
Enter the burst time of process 1: 3
Enter the priority of process 1: 3
Enter the arrival time of process 2: 1
Enter the burst time of process 2: 6
Enter the priority of process 2: 4
Enter the arrival time of process 3: 3
Enter the burst time of process 3: 1
Enter the priority of process 3: 9
Enter the arrival time of process 4: 2
Enter the burst time of process 4: 2
Enter the priority of process 4: 7
Enter the arrival time of process 5: 4
Enter the burst time of process 5: 4
Enter the priority of process 5: 8

--------------------
PID AT  BT  CT  TAT WT
1   0   3   3   3   0
2   1   6   9   8   2
4   2   2   11  9   7
3   3   1   16  13  12
5   4   4   15  11  7
```

The first-fit algorithm searches for the first free partition that is large enough to accommodate the process. The operating system starts searching from the beginning of the memory and allocates the first free partition that is large enough to fit the process.

The best-fit algorithm searches for the smallest free partition that is large enough to accommodate the process. The operating system searches the entire memory and selects the free partition that is closest in size to the process.

Worst Fit Algorithm

The worst-fit algorithm searches for the largest free partition and allocates the process to it. This algorithm is designed to leave the largest possible free partition for future use.

Memory Block					
Processes	212 KB **(P1)**	417KB **(P2)**	112 KB **(P3)**	426 KB **(P4)**	

```
Enter the number of blocks: 5
Enter the sizes of the blocks:
100 500 200 300 600
Enter the number of processes: 4
Enter the sizes of the processes:
212 417 112 400

First Fit Allocation:
Process No.     Process Size    Block No.
1               212             2
2               417             5
3               112             2
4               400             Not Allocated

Best Fit Allocation:
Process No.     Process Size    Block No.
1               212             4
2               417             2
3               112             3
4               400             5

Worst Fit Allocation:
Process No.     Process Size    Block No.
1               212             5
2               417             2
3               112             5
4               400             Not Allocated
```

```
Enter the number of readers and writers: 3

Reader 1 is reading data
Reader 3 is reading data
Writer 3 is trying to enter
Reader 2 is reading data
Writer 1 is trying to enter
Writer 2 is trying to enter
Reader 3 is done reading
Reader 3 is leaving. Remaining readers: 2
Reader 2 is done reading
Reader 2 is leaving. Remaining readers: 1
Reader 1 is done reading
Reader 1 is leaving. Remaining readers: 0
Writer 3 is writing data
Writer 3 is done writing
Writer 1 is writing data
Writer 1 is done writing
Writer 2 is writing data
Writer 2 is done writing
```

```
Enter buffer size: 3

1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
4. Press 4 to Show Buffer Status
Enter your choice: 4

Buffer Status:
Full slots: 0
Empty slots: 3
Total items in buffer: 0

Enter your choice: 1
Producer produces item 1

Enter your choice: 1
Producer produces item 2

Enter your choice: 1
Producer produces item 3

Enter your choice: 1
Buffer is full!

Enter your choice: 2
Consumer consumes item 3
```

```
Enter your choice: 4

Buffer Status:
Full slots: 2
Empty slots: 1
Total items in buffer: 2

Enter your choice: 2
Consumer consumes item 2

Enter your choice: 2
Consumer consumes item 1

Enter your choice: 2
Buffer is empty!

Enter your choice: 4

Buffer Status:
Full slots: 0
Empty slots: 3
Total items in buffer: 0

Enter your choice: 3
```

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter available resources: Resource 1: 3
Resource 2: 3
Resource 3: 2
Enter maximum resources for each process:
Enter maximum resources for process 1: 7 5 3
Enter maximum resources for process 2: 3 2 2
Enter maximum resources for process 3: 9 0 2
Enter maximum resources for process 4: 2 2 2
Enter maximum resources for process 5: 4 3 3
Enter allocated resources for each process:
Enter allocated resources for process 1: 0 1 0
Enter allocated resources for process 2: 2 0 0
Enter allocated resources for process 3: 3 0 2
Enter allocated resources for process 4: 2 1 1
Enter allocated resources for process 5: 0 0 2
System is in a safe state.
Safe sequence is: 1 3 4 0 2
```

```
1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 2
Error opening file!

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 1
Enter ID: 1
Enter Name: Record7
Do you want to add another record? (y/n): y
Enter ID: 2
Enter Name: Record8
Do you want to add another record? (y/n): n
Records written to indexed.txt and index.txt.

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 2

Indexed Records:
ID: 1, Name: Record7
ID: 2, Name: Record8
```

```
1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 3
Enter ID: 4
Enter Name: Record9
Do you want to add another record? (y/n): n
Records written to hashed.txt.

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 4

Hashed Records:
ID: 1, Name: Record1
ID: 2, Name: Record2
ID: 3, Name: Record3
ID: 4, Name: Record9
```

```
1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 5
Enter ID: 4
Enter Name: Record10
Do you want to add another record? (y/n): n
Records written to sequential.txt.

1. Write Indexed File
2. Read Indexed File
3. Write Hashed File
4. Read Hashed File
5. Write Sequential File
6. Read Sequential File
7. Exit
Enter your choice: 6

Sequential Records:
ID: 1, Name: Record4
ID: 2, Name: Record5
ID: 3, Name: Record6
ID: 4, Name: Record10
```