

EXP NO: 6

REG.NO :

DATE:

NAME:

CPU SCHEDULING ALGORITHM – FIRST COME FIRST SERVE(FCFS)

AIM:

To implement the FCFS CPU scheduling algorithm using C program.

PROGRAM DESCRIPTION:

In FCFS CPU scheduling algorithm, the first process that arrives in the ready queue is given the CPU time. It is a non-preemptive algorithm, which means other processes should wait till the execution of the current process is complete.

SCHEDULING CRITERIA TO BE CONSIDERED:

Turn-around time : Time difference between completion time and arrival time.

Waiting time : It is the time difference between turnaround time and burst time.

Response time : Amount of time from when a request was submitted until the first response is produced.

Gantt Chart : It is a bar chart that illustrates a particular scheduling algorithm, including the start and finish time of each of the participating processes.

HEADER FILES USED :

- stdio.h

PROGRAM CODE:

```
#include <stdio.h>

int main()
{
    int i, n, pid[7], at[7], bt[7], tot, f, k;
    printf("Enter number of processes: ");
    scanf("%d", &n);
```

```

for (i = 0; i < n; i++) {
    printf("\n enter PID:");
    scanf("%d", &pid[i]);
    printf("\n enter arrival time:");
    scanf("%d", &at[i]);
    printf("\n enter burst time:");
    scanf("%d", &bt[i]);
    tot += bt[i];
}
printf("\n\nPID\tAT\tBT");
printf("\n\n-----");
for (i = 0; i < n; i++) {
    printf("\n\n%d\t%d\t%d", pid[i], at[i], bt[i]);
}
printf("\n\n\n");
for (i = 0; i < n; i++) {
    printf(" ");
}
printf("\n\tGANTT CHART\n");
for (i = 0; i < n; i++) {
    printf("----");
}
printf("\n\n");
for (i = 0; i < n; i++) {
    printf(" ");
}
printf("\n");
}
for (i = 0; i < 16 * n; i++) {
    printf("-");
}
printf("\n");

```

```

for (i = 0; i < n; i++) {
    printf("\t%d\t", pid[i]);
}
printf("\n");
for (i = 0; i < 16 * n; i++) {
    printf("-");
}
printf("\n");

```

```

int sum = at[0];
for (i = 0; i < n; i++) {
    printf("%d\t", sum);
    sum = sum + bt[i];
}

```

```

// WAIT TIME
printf("\n");
int wt[7], tt[7];
int temp = at[0];
float totwt = 0.0, tottt = 0.0;
for (i = 0; i < n; i++) {
    wt[i] = temp - at[i];
    printf("\nWAIT TIME FOR PROCESS %d IS %d", pid[i], wt[i]);
    temp += bt[i];
}

```

```

// TURNAROUND TIME
int b = 0;
for (i = 0; i < n; i++) {
    tt[i] = b + bt[i];
    printf("\nTURN-AROUND TIME FOR PROCESS %d IS %d", pid[i], tt[i]);
    b += bt[i];
}

```

```

}

for (i = 0; i < n; i++) {
    totwt += wt[i];
    tottt += tt[i];
}
} floatavgw=0.0, avgt=0.0;
avgw=totwt/n;
avgt=tottt/n;
printf("\n\n AVERAGE WAIT TIME IS: %f", avgw);
printf("\n\n AVERAGE TURN-AROUND TIME IS: %f", avgt);
return 0;
}

```

SAMPLE INPUT AND OUTPUT:

```

PS D:\Clg\sem 5\OS\Lab> gcc FCFS.c -o fcfs
PS D:\Clg\sem 5\OS\Lab> ./fcfs
Enter number of processes: 3

enter PID: 1

enter arrival time: 0

enter burst time: 24

enter PID: 2

enter arrival time: 0

enter burst time: 3

enter PID: 3

enter arrival time: 0

enter burst time: 3

```

```
PID    AT    BT
-----
```

```
1      0     24
```

```
2      0      3
```

```
3      0      3
```

```
GANTT CHART
```

```
-----
```

```
-----
      1          2          3
-----
0          24          27
```

```
WAIT TIME FOR PROCESS 1 IS 0
```

```
WAIT TIME FOR PROCESS 2 IS 24
```

```
WAIT TIME FOR PROCESS 3 IS 27
```

```
TURN-AROUND TIME FOR PROCESS 1 IS 24
```

```
TURN-AROUND TIME FOR PROCESS 2 IS 27
```

```
TURN-AROUND TIME FOR PROCESS 3 IS 30
```

```
AVERAGE WAIT TIME IS: 17.000000
```

```
AVERAGE TURN-AROUND TIME IS: 27.000000
```

RESULT:

Thus, the FCFS scheduling algorithm has been implemented using C program.

EXP NO : 7

REG NO :

DATE :

NAME :

CPU SCHEDULING ALGORITHM – SHORTEST JOB FIRST (SJF)

AIM :

To implement and compare the **Shortest Job First (SJF)** Scheduling Algorithm in both Pre-emptive (SRTF - Shortest Remaining Time First) and Non-Pre-emptive modes.

PROBLEM DESCRIPTION:

In SJF, processes are selected based on their burst time, with the shortest job being executed first.

- **Non-Pre-emptive SJF:**
 - Once a process is allocated to the CPU, it runs to completion without being interrupted by other processes.
 - In this program, processes are scheduled based on their arrival and burst times. Once a process is selected for execution, it runs till completion without interruption.
- **Pre-emptive SJF (SRTF):**
 - The currently running process can be pre-empted if a new process arrives with a shorter remaining burst time.
 - In this version, the currently executing process can be interrupted if a new process with a shorter remaining burst time arrives.

The algorithms will allow user input for the number of processes, their burst time, and arrival time. In non-pre-emptive scheduling, the process selection is made once, while in pre-emptive scheduling, the process selection can change dynamically as new processes arrive. Both implementations aim to minimize average waiting and turnaround time for all processes.

Key metrics calculated:

1. **Waiting Time (WT):** The total time a process spends waiting in the ready queue before its execution.
2. **Turnaround Time (TAT):** The total time taken by a process from its arrival in the system to its completion.
3. **Completion Time (CT):** The time at which a process completes its execution.

ALGORITHM:

1) NON PREEMPTIVE:

1. **Input:** Number of processes, arrival time, and burst time.
2. **Initialize:** Set currentTime = 0 and mark all processes as incomplete.
3. **Find Process:** At each step, select the **arrived** process with the **shortest burst time** that hasn't been completed yet.
4. **Execute:** Update wait time, completion time, and turnaround time for the selected process. Update currentTime.
5. **Repeat:** Continue until all processes are completed, then display results.

2) PREEMPTIVE:

1. **Input:** Number of processes, arrival time, and burst time.
2. **Initialize:** Set currentTime = 0 and track remaining burst time for all processes.
3. **Find Process:** At each time unit, select the **arrived** process with the **shortest remaining burst time**.
4. **Execute:** Pre-empt if necessary, update remaining burst time, and calculate waiting time when a process completes.
5. **Repeat:** Continue until all processes are completed, then display results.

PROGRAM CODE:

// NON-PREEMPTIVE

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

```
#include <ctype.h>
```

```
#define MAX 50
```

```
typedef struct {
```

```
    int pid;
```

```
    int burst;
```

```
    int arrival;
```

```
    int wait;
```

```
    int turnAround;
```

```
    int completion;
```

```
    int completed;
```

```
} Process;
```

```
void cal_wt_tat(Process processes[], int size) {
```

```

int currentTime = 0, completedProcesses = 0;
while (completedProcesses < size) {
    int idx = -1, minBurst = 9999;
    for (int i = 0; i < size; i++) {
        if (processes[i].arrival <= currentTime && processes[i].completed == 0) {
            if (processes[i].burst < minBurst) {
                minBurst = processes[i].burst;
                idx = i;
            }
        }
    }
    if (idx != -1) {
        processes[idx].wait = currentTime - processes[idx].arrival;
        currentTime += processes[idx].burst;
        processes[idx].completion = currentTime;
        processes[idx].turnAround = processes[idx].wait + processes[idx].burst;
        processes[idx].completed = 1;
        completedProcesses++;
    } else {
        currentTime++;
    }
}

void display(Process processes[], int size) {
    printf("PID\tArrival Time\tBurst Time\tWait Time\tTurn Around Time\tCompletion Time\n");
    for (int i = 0; i < size; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].arrival, processes[i].burst,
        processes[i].wait, processes[i].turnAround, processes[i].completion);
    }
}

int main() {

```



```

Process processes[MAX];

int size;

printf("Enter the number of processes you want to schedule: ");

scanf("%d", &size);

char choice;

printf("Proceed with default arrival time? (Y/n): ");

scanf(" %c", &choice);

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

    processes[i].pid = i + 1;

    if (tolower(choice) == 'n') {

        printf("Enter arrival time of process %d: ", processes[i].pid);

        scanf("%d", &processes[i].arrival);

    } else {

        processes[i].arrival = 0;

    }

    printf("Enter burst time of process %d: ", processes[i].pid);

    scanf("%d", &processes[i].burst);

    processes[i].completed = 0;

}

cal_wt_tat(processes, size);

display(processes, size);

return 0;

}

```

// PREEMPTIVE

```

#include <stdio.h>

#include <limits.h>

```

```

struct Process {

    int pid;

    int bt;

    int art;

};

```

```

void findWaitingTime(struct Process p[], int n, int wt[]) {
    int rt[n];

    for (int i = 0; i < n; i++) rt[i] = p[i].bt;

    int complete = 0, t = 0, minm = INT_MAX, shortt = 0, check = 0;

    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if (p[j].art <= t && rt[j] < minm && rt[j] > 0) {
                minm = rt[j];
                shortt = j;
                check = 1;
            }
        }
        if (check == 0) {
            t++;
            continue;
        }
        rt[shortt]--;
        minm = rt[shortt] ? rt[shortt] : INT_MAX;

        if (rt[shortt] == 0) {
            complete++;
            check = 0;
            int finish_time = t + 1;
            wt[shortt] = finish_time - p[shortt].bt - p[shortt].art;
            if (wt[shortt] < 0) wt[shortt] = 0;
        }
        t++;
    }
}

```

```

void findTurnAroundTime(struct Process p[], int n, int wt[], int tat[]) {
    for (int i = 0; i < n; i++) tat[i] = p[i].bt + wt[i];
}

```

```

void findavgTime(struct Process p[], int n) {
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(p, n, wt);
    findTurnAroundTime(p, n, wt, tat);

    printf("P  BT  WT  TAT\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d  %d  %d  %d\n", p[i].pid, p[i].bt, wt[i], tat[i]);
    }
    printf("Avg WT = %.2f\n", (float)total_wt / n);
    printf("Avg TAT = %.2f\n", (float)total_tat / n);
}

```

```

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

    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter Burst Time and Arrival Time for Process %d: ", p[i].pid);
        scanf("%d %d", &p[i].bt, &p[i].art);
    }
    findavgTime(p, n);
}

```

```
return 0;
}
```

SAMPLE INPUT AND OUTPUT:

(Non Pre-emptive) Case 1: With given arrival time

```
k e z i a@kez MINGW64 /c/[0] college/2024 - 2025 Y3/[1] S5/osLab/ex7
$ ./sjf_scheduling
Enter the number of processes you want to schedule: 5
Proceed with default arrival time? (Y/n): n
Enter arrival time of process 1: 0
Enter burst time of process 1: 5
Enter arrival time of process 2: 1
Enter burst time of process 2: 3
Enter arrival time of process 3: 2
Enter burst time of process 3: 2
Enter arrival time of process 4: 3
Enter burst time of process 4: 1
Enter arrival time of process 5: 4
Enter burst time of process 5: 1
```

PID	Arrival Time	Burst Time	Wait Time	Turn Around Time	Completion Time
1	0	5	0	5	5
2	1	3	8	11	12
3	2	2	5	7	9
4	3	1	2	3	6
5	4	1	2	3	7

(Non Pre-emptive) Case 2: Default arrival time

```
$ ./sjf_scheduling
Enter the number of processes you want to schedule: 5
Proceed with default arrival time? (Y/n): y
Enter burst time of process 1: 5
Enter burst time of process 2: 3
Enter burst time of process 3: 2
Enter burst time of process 4: 6
Enter burst time of process 5: 1
```

PID	Arrival Time	Burst Time	Wait Time	Turn Around Time	Completion Time
1	0	5	6	11	11
2	0	3	3	6	6
3	0	2	1	3	3
4	0	6	11	17	17
5	0	1	0	1	1

(Pre-emptive):

```
Enter the number of processes: 5
Enter Burst Time and Arrival Time for Process 1: 6 2
Enter Burst Time and Arrival Time for Process 2: 2 5
Enter Burst Time and Arrival Time for Process 3: 8 1
Enter Burst Time and Arrival Time for Process 4: 3 0
Enter Burst Time and Arrival Time for Process 5: 4 4
P  BT  WT  TAT
1  6   7  13
2  2   0   2
3  8  14  22
4  3   0   3
5  4   2   6
Avg WT = 4.60
Avg TAT = 9.20

=== Code Execution Successful ===
```

RESULT:

Therefore, SJF has been programmed, implemented and successfully executed using C.

EXP NO : 8

REG NO :

DATE :

NAME :

CPU SCHEDULING ALGORITHM – PRIORITY SCHEDULING

AIM:

To develop a C program to implement the Priority Scheduling Algorithms (Preemptive and Non Preemptive).

CODE:

1) Preemptive:

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

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

```
struct queue {
```

```
int arr[100];
```

```
int size;
```

```
int front, rear;
```

```
};
```

```
struct node {
```

```
int pid;
```

```
int st;
```

```
int ft;
```

```
struct node *next;
```

```
};
```

```
struct table {
```

```
int pid;
```

```
int at;
```

```
int bt;
```

```
int st;
```

```
int ft;
```

```
int wt;
```

```
int rt;
```

```
int tat;
```

```
int pri;
```

```
int response;
```

```
};
```

```
void printTable(struct table* t[], int n) {
```

```
printf(&quot;\nPROCESS ID\tARRIVAL TIME\tBURST TIME\tPRIORITY\tTAT\tWAITING  
TIME\n&quot;);
```

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

```
printf(&quot;\n%d\t %d\t %d\t %d\t %d\t %d\n&quot;, t[i]->pid, t[i]->at, t[i]->bt,  
t[i]->pri,
```

```
t[i]->tat, t[i]->wt);
```

```
}
```

```
}
```

```
void getInput(struct table* t1[], int np) {
```

```
for (int j = 0; j < np; j++) {
```

```
int i = j;
```

```
printf(&quot;ENTER THE %d PROCESS ID : &quot;, i + 1);
```

```
scanf(&quot;%d&quot;, &t1[i]->pid);
```

```
printf(&quot;ENTER THE %d ARRIVAL TIME : &quot;, i + 1);
```

```
scanf(&quot;%d&quot;, &t1[i]->at);
```

```
printf(&quot;ENTER THE %d BURST TIME : &quot;, i + 1);
```

```
scanf(&quot;%d&quot;, &t1[i]->bt);
```

```
printf(&quot;ENTER THE %d PRIORITY : &quot;, i + 1);
```

```
scanf(&quot;%d&quot;, &t1[i]->pri);
```

```
t1[i]->wt = 0;
```

```
t1[i]->st = 0;
```

```
    t1[i]-&gt;response = 0;
}
}
```

```
void create(struct queue* q, int len) {
    q-&gt;size = len;
    q-&gt;front = 0;
    q-&gt;rear = -1;
}
```

```
int isFull(struct queue* q) {
    return (q-&gt;rear == q-&gt;size - 1);
}
```

```
int isEmpty(struct queue* q) {
    return (q-&gt;front &gt; q-&gt;rear);
}
```

```
void enqueue(struct queue* q, int data) {
    if (isFull(q)) {
        printf("&quot;\nQueue is full&quot;);
    } else {
        q-&gt;arr[++q-&gt;rear] = data;
    }
}
```

```
int dequeue(struct queue* q) {
    if (isEmpty(q)) {
        return -1;
    } else {
        return q-&gt;arr[q-&gt;front++];
    }
}
```



```
}
```

```
struct queue* sort(struct queue* q, struct table* t[]) {  
    for (int i = q->front; i <= q->rear; i++) {  
        for (int j = i + 1; j <= q->rear; j++) {  
            if (t[q->arr[i]]->pri > t[q->arr[j]]->pri) {  
                int temp = q->arr[i];  
                q->arr[i] = q->arr[j];  
                q->arr[j] = temp;  
            }  
        }  
    }  
    return q;  
}
```

```
struct node* insert(struct node* header, int pid, int st, int ft) {  
    struct node* newnode = malloc(sizeof(struct node));  
    newnode->pid = pid;  
    newnode->st = st;  
    newnode->ft = ft;  
  
    if (header == NULL) {  
        header = newnode;  
        newnode->next = NULL;  
    } else {  
        struct node* temp = header;  
        while (temp->next != NULL) {  
            temp = temp->next;  
        }  
        temp->next = newnode;  
        newnode->next = NULL;  
    }  
}
```

```
return header;
```

```
}
```

```
void printGanttChart(struct node* header) {
```

```
    struct node* temp = header;
```

```
    while (temp != NULL) {
```

```
        if (temp->pid != -1) {
```

```
            printf("&quot;|tST:%d P_ID:%d FT:%d\\t|&quot;", temp->st, temp->pid + 1, temp->ft);
```

```
        } else {
```

```
            printf("&quot;|tST:%d Idle FT:%d\\t|&quot;", temp->st, temp->ft);
```

```
        }
```

```
        temp = temp->next;
```

```
    }
```

```
    printf("&quot;\\n&quot;);
```

```
}
```

```
void PPS(struct table* t[], int n) {
```

```
    getInput(t, n);
```

```
    struct node* header = malloc(sizeof(struct node));
```

```
    header = NULL;
```

```
    struct queue* q = malloc(sizeof(struct queue));
```

```
    create(q, 100);
```

```
    int proc = 0;
```

```
    int clock_cycle = 0;
```

```
    while (1) {
```

```
        while (proc < n) {
```

```
            if (t[proc]->at == clock_cycle) {
```

```
                enqueue(q, proc);
```

```
            proc++;
```

```
q = sort(q, t);
```

```
} else break;
```

```
}
```

```
if (!isEmpty(q)) {
```

```
int curr_proc = dequeue(q);
```

```
if (t[curr_proc]-&gt;response == 0) {
```

```
t[curr_proc]-&gt;response = 1;
```

```
t[curr_proc]-&gt;st = clock_cycle;
```

```
t[curr_proc]-&gt;rt = t[curr_proc]-&gt;st - t[curr_proc]-&gt;at;
```

```
}
```

```
t[curr_proc]-&gt;bt--;
```

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

```
if (i != curr_proc && t[i]-&gt;at <= clock_cycle && t[i]-&gt;bt > 0) {
```

```
t[i]-&gt;wt++;
```

```
}
```

```
}
```

```
header = insert(header, curr_proc, clock_cycle, clock_cycle + 1);
```

```
if (t[curr_proc]-&gt;bt > 0) {
```

```
enqueue(q, curr_proc);
```

```
q = sort(q, t);
```

```
}
```

```
if (t[curr_proc]-&gt;bt == 0) {
```

```
t[curr_proc]-&gt;ft = clock_cycle + 1;
```

```
}
```

```
} else if (isEmpty(q)) {
```

```
break;
```

```
}
```

```
clock_cycle++;
```

```
}
```

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

```
t[i]>tat = t[i]>ft - t[i]>at;
```

```
}
```

```
printGanttChart(header);
```

```
printTable(t, n);
```

```
float avgwt = 0;
```

```
float avgrt = 0;
```

```
float avgtat = 0;
```

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

```
avgwt = avgwt + t[i]>wt;
```

```
avgrt = avgrt + t[i]>rt;
```

```
avgtat = avgtat + t[i]>tat;
```

```
}
```

```
avgwt = avgwt / n;
```

```
avgrt = avgrt / n;
```

```
avgtat = avgtat / n;
```

```
printf(&quot;Average waiting time: %.2f\n&quot;, avgwt);
```

```
printf(&quot;Average response time: %.2f\n&quot;, avgrt);
```

```
printf(&quot;Average turnaround time: %.2f\n&quot;, avgtat);
```

```
}
```

```
int main() {
```

```
int n;
```

```
printf(&quot;Enter the number of processes: &quot;);
```

```
scanf(&quot;%d&quot;, &n);
```

```

struct table* t[n];

for (int i = 0; i < n; i++) {
t[i] = (struct table*)malloc(sizeof(struct table));
}

PPS(t, n);

return 0;
}

```

2) Non Preemptive:

```

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

```

```

struct node {
int pid;
int st;
int ft;
struct node *next;
};

```

```

struct table {
int pid;
int at;
int bt;
int st;
int ft;
int wt;
int rt;

int tat;

```

```
int pri;
```

```
};
```

```
void printTable(struct table* t[], int n) {
```

```
printf(&quot;\nPROCESS ID\tARRIVAL TIME\tBURST TIME\tPRIORITY\tTAT\tWAITING  
TIME\n&quot;);
```

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

```
printf(&quot;\n%d\t\t %d\t\t %d\t\t %d\t\t %d\t\t %d\n&quot;, t[i]->pid, t[i]->at, t[i]->bt,  
t[i]->pri,
```

```
t[i]->tat, t[i]->wt);
```

```
}
```

```
}
```

```
void getInput(struct table* t1[], int np) {
```

```
for (int j = 0; j < np; j++) {
```

```
int i = j;
```

```
printf(&quot;ENTER THE %d PROCESS ID : &quot;, i + 1);
```

```
scanf(&quot;%d&quot;, &t1[i]->pid);
```

```
printf(&quot;ENTER THE %d ARRIVAL TIME : &quot;, i + 1);
```

```
scanf(&quot;%d&quot;, &t1[i]->at);
```

```
printf(&quot;ENTER THE %d BURST TIME : &quot;, i + 1);
```

```
scanf(&quot;%d&quot;, &t1[i]->bt);
```

```
printf(&quot;ENTER THE %d PRIORITY : &quot;, i + 1);
```

```
scanf(&quot;%d&quot;, &t1[i]->pri);
```

```
t1[i]->wt = 0;
```

```
t1[i]->st = 0;
```

```
}
```

```
}
```

```
struct node* insert(struct node* header, int pid, int st, int ft) {
```

```
struct node* newnode = malloc(sizeof(struct node));
```

```
newnode->pid = pid;
```

```
newnode->st = st;
```

```
newnode->ft = ft;
```

```
if (header == NULL) {
```

```
header = newnode;
```

```
newnode->next = NULL;
```

```
} else {
```

```
struct node* temp = header;
```

```
while (temp->next != NULL) {
```

```
temp = temp->next;
```

```
}
```

```
temp->next = newnode;
```

```
newnode->next = NULL;
```

```
}
```

```
return header;
```

```
}
```

```
void printGanttChart(struct node* header) {
```

```
struct node* temp = header;
```

```
while (temp != NULL) {
```

```
if (temp->pid != -1) {
```

```
printf("&quot;|tST:%d P_ID:%d FT:%d\t|&quot;", temp->st, temp->pid + 1, temp->ft);
```

```
} else {
```

```
printf("&quot;|tST:%d Idle FT:%d\t|&quot;", temp->st, temp->ft);
```

```
}
```

```
temp = temp->next;
```

```
}
```

```
printf("&quot;\n&quot;);
```

```
}
```

```

void NPS(struct table* t[], int n) {
    getInput(t, n);

    struct node* header = malloc(sizeof(struct node));
    header = NULL;

    int clock_cycle = 0, completed = 0;

    while (completed < n) {
        int highest_priority = -1;
        int index = -1;

        for (int i = 0; i < n; i++) {
            if (t[i]->at <= clock_cycle && t[i]->ft == 0) {
                if (index == -1 || t[i]->pri < highest_priority) {
                    highest_priority = t[i]->pri;
                    index = i;
                }
            }
        }

        if (index != -1) {
            t[index]->st = clock_cycle;
            t[index]->wt = t[index]->st - t[index]->at;
            t[index]->ft = t[index]->st + t[index]->bt;
            t[index]->tat = t[index]->ft - t[index]->at;
            clock_cycle = t[index]->ft;
            completed++;

            header = insert(header, index, t[index]->st, t[index]->ft);
        } else {
            clock_cycle++;
        }
    }
}

```



```
printGanttChart(header);
```

```
printTable(t, n);
```

```
float avgwt = 0, avgtat = 0;
```

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

```
    avgwt += t[i]-wt;
```

```
    avgtat += t[i]-tat;
```

```
}
```

```
avgwt /= n;
```

```
avgtat /= n;
```

```
printf("Average waiting time: %.2f\n", avgwt);
```

```
printf("Average turnaround time: %.2f\n", avgtat);
```

```
}
```

```
int main() {
```

```
    int n;
```

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

```
    scanf("%d", &n);
```

```
    struct table* t[n];
```

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

```
        t[i] = (struct table*)malloc(sizeof(struct table));
```

```
    }
```

```
    NPS(t, n);
```

```
    return 0;
```

```
}
```

Test Cases:

Preemptive:

```

Enter the number of processes: 4
ENTER THE 1 PROCESS ID : 1
ENTER THE 1 ARRIVAL TIME : 0
ENTER THE 1 BURST TIME : 8
ENTER THE 1 PRIORITY : 4
ENTER THE 2 PROCESS ID : 2
ENTER THE 2 ARRIVAL TIME : 1
ENTER THE 2 BURST TIME : 4
ENTER THE 2 PRIORITY : 3
ENTER THE 3 PROCESS ID : 3
ENTER THE 3 ARRIVAL TIME : 2
ENTER THE 3 BURST TIME : 9
ENTER THE 3 PRIORITY : 1
ENTER THE 4 PROCESS ID : 4
ENTER THE 4 ARRIVAL TIME : 3
ENTER THE 4 BURST TIME : 5
ENTER THE 4 PRIORITY : 2

```

	ST:0 P_ID:1 FT:1		ST:1 P_ID:2 FT:2		ST:2 P_ID:3 FT:11		ST:11 P_ID:4 FT:16		ST:16 P_ID:2 FT:19	
	ST:19 P_ID:1 FT:26									

PROCESS ID	ARRIVAL TIME	BURST TIME	PRIORITY	TAT	WAITING TIME
1	0	0	4	26	18
2	1	0	3	18	14
3	2	0	1	9	0
4	3	0	2	13	8

Non-preemptive:

```

Enter the number of processes: 4
ENTER THE 1 PROCESS ID : 1
ENTER THE 1 ARRIVAL TIME : 0
ENTER THE 1 BURST TIME : 8
ENTER THE 1 PRIORITY : 4
ENTER THE 2 PROCESS ID : 2
ENTER THE 2 ARRIVAL TIME : 1
ENTER THE 2 BURST TIME : 4
ENTER THE 2 PRIORITY : 3
ENTER THE 3 PROCESS ID : 3
ENTER THE 3 ARRIVAL TIME : 2
ENTER THE 3 BURST TIME : 9
ENTER THE 3 PRIORITY : 1
ENTER THE 4 PROCESS ID : 4
ENTER THE 4 ARRIVAL TIME : 3
ENTER THE 4 BURST TIME : 5
ENTER THE 4 PRIORITY : 2
0
Current process: 0
0 1
Current process: 0
0 2 1
Current process: 0
0 2 3 1
Current process: 0

```

	ST:0 P_ID:1 FT:8		ST:8 P_ID:3 FT:17		ST:17 P_ID:4 FT:22		ST:22 P_ID:2 FT:26	
--	------------------	--	-------------------	--	--------------------	--	--------------------	--

PROCESS ID	ARRIVAL TIME	BURST TIME	PRIORITY	TAT	WAITING TIME
1	0	0	4	8	0
2	1	0	3	25	21
3	2	0	1	15	6
4	3	0	2	19	14

Average waiting time: 10.25
 Average response time: 10.25
 Average turnaround time: 16.75

RESULT:

Developed a C program to implement the Priority Scheduling Algorithms (Preemptive and Non Preemptive).

EXP NO : 9

REG NO :

DATE :

NAME :

CPU SCHEDULING ALGORITHM -Round Robin (RR)

AIM:

To implement the Round Robin (RR) CPU scheduling algorithm using C Program

PROGRAM DESCRIPTION:

The Round Robin (RR) scheduling algorithm is a preemptive scheduling method primarily used in time-sharing systems. It allocates a fixed time slice (quantum) to each process in a cyclic order. When a process's allocated time slice expires, it is moved to the back of the queue, and the CPU is assigned to the next process. This method ensures fair allocation of CPU time among all processes, making it effective in scenarios where response time is critical, such as in interactive systems.

SCHEDULING CRITERIA TO BE CONSIDERED:

Turnaround Time (TAT): Total time from process submission to completion.

Waiting Time (WT): Total time a process spends waiting in the ready queue.

Response Time (RT): Time from submission until the first response is produced.

Gantt Chart: Visual representation of the order and duration of process execution over time.

PROGRAM:

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

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

```
#include <math.h>
```

```
typedef struct{
```

```
    long int pid;
```

```
    int bt;
```

```
    int at;
```

```
} process;
```

```
void roundrobin(process pro[], int n, int quantum) {
```

```
    int i, j;
```

```
    float avg, avg1, sum = 0.0, sum1 = 0.0;
```

```

int flag = 0;
int t = 0;
int lines = 0, checkpoints[100], ckindex = 1;
int done[100];
int wt[100];
int bt[100];
int ut[100];
checkpoints[0] = 0;

// Initialize all process variables
for (i = 0; i < n; i++) {
    done[i] = 0;
    wt[i] = 0;
    bt[i] = pro[i].bt;
    ut[i] = 0;
}

printf("\nPrinting Gantt chart:\n\n");

i = 0;
while (flag < n) {
    if (done[i] == 0) {
        if (bt[i] > quantum) {
            // Process executed for quantum time
            t += quantum;
            bt[i] -= quantum;
            checkpoints[ckindex++] = t;
            printf("P%d | ", pro[i].pid);

            // Add waiting time for other processes
            for (j = 0; j < n; j++) {
                if (done[j] == 0 && j != i) {

```

```

        wt[j] += quantum;
    }
}
} else {
    // Process executed for remaining burst time
    t += bt[i];
    checkpoints[ckindex++] = t;
    printf("P%d | ", pro[i].pid);

    // Mark process as done
    done[i] = 1;
    flag++;

    // Add waiting time for other processes
    for (j = 0; j < n; j++) {
        if (done[j] == 0 && j != i) {
            wt[j] += bt[i];
        }
    }

    bt[i] = 0;
}
}

i = (i + 1) % n; // Move to next process
}

printf("\n\n");
// Print the time checkpoints in Gantt chart
printf("Time checkpoints: ");
for (i = 0; i < ckindex; i++) {
    printf("%d ", checkpoints[i]);
}

// Print process details: ID, Burst time, Waiting time, and Turnaround time

```

```

printf("\n\nProcess ID | Burst time | Waiting time | Turn around time\n\n");
for (i = 0; i < n; i++) {
    ut[i] = wt[i] + pro[i].bt; // Calculate turnaround time
    sum += wt[i];
    sum1 += ut[i];
    printf("P%d \t %d \t %d \t %d\n", pro[i].pid, pro[i].bt, wt[i], ut[i]);
}
// Calculate and print average waiting time and turnaround time
avg = sum / n;
avg1 = sum1 / n;
printf("\nAverage waiting time: %.2f\nAverage turn-around time: %.2f\n", avg, avg1);
}

int main() {
    int pno;
    int quantum;
    process pro[100];

    printf("\nEnter the number of processes: ");
    scanf("%d", &pno);

    for (int i = 0; i < pno; i++) {
        printf("\nEnter the process ID: ");
        scanf("%d", &pro[i].pid);
        printf("Enter the burst time: ");
        scanf("%d", &pro[i].bt);
        printf("Enter the arrival time: ");
        scanf("%d", &pro[i].at);
    }
    printf("\nEnter the quantum time slice: ");
    scanf("%d", &quantum);
    roundrobin(pro, pno, quantum);
}

```

```
    return 0;
}
```

SAMPLE INPUT AND OUTPUT:

```
Enter the number of processes: 3

Enter the process ID: 1
Enter the burst time: 24
Enter the arrival time: 0

Enter the process ID: 2
Enter the burst time: 3
Enter the arrival time: 0

Enter the process ID: 3
Enter the burst time: 5
Enter the arrival time: 0

Enter the quantum time slice: 2

Printing Gantt chart:

P1 | P2 | P3 | P1 | P2 | P3 | P1 | P3 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 |
Time checkpoints: 0  2  4  6  8  9 11 13 14 16 18 20 22 24 26 28 30 32

Process ID | Burst time | Waiting time | Turn around time
P1          24           8           32
P2           3           6           9
P3           5           9          14

Average waiting time: 7.67
Average turn-around time: 18.33
```

RESULT:

Therefore, Round Robin algorithm has been programmed, implemented and successfully executed using C.

EXP NO: 10

REG NO:

DATE:

NAME:

INTERPROCESS COMMUNICATION USING SEMAPHORES

AIM:

To implement the producer-consumer problem using semaphores using a C program.

PROBLEM DESCRIPTION:

- Producer-consumer problem is the classic problem of a multiprocess synchronisation problem.
- Two processes: the producer and consumer share a common fixed-size buffer; producer task is to generate "item", put it into buffer and start again.
- Simultaneously, consumer consumes one item at a time from the buffer.
- The program ensures that producer doesn't try to add 'items' into a fully fixed buffer; consumer doesn't try to remove any data from an empty buffer.

STRUCTURE OF A PRODUCER-CONSUMER PROBLEM:

```
// Producer process
do
{
//produce an item in the next produced
wait (empty);
wait(mutex);
//add next-produced to the buffer
signal (mutex);
signal (full);
}
while(true);

// consumer process
do
{
```



```

wait (full);
wait (mutex);
Next.consumed
//remove an item and move it from buffer to next_consumed
signal (mutex);
signal (empty);
    //consume the item in next consumed
}
while(true),

```

About Semaphores used:

Mutex: Variable mutex is used to incorporate consistency between the producer and consumer process. A process while executing, performing wait to prevent parallel, execution and performance signal after execution.

Structure of wait:

```

wait (int x)
{
    x - -;
}

```

Structure of signal:

```

signal (int x)
{
    x++;
}

```

PROGRAM CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define BUFFER_SIZE 5 // Size of the buffer
#define PRODUCE_COUNT 10 // Number of items to produce
int buffer[BUFFER_SIZE]; // The buffer
int in = 0; // Index for the next item to produce
int out = 0; // Index for the next item to consume


sem_t empty; // Semaphore to count empty slots
sem_t full; // Semaphore to count full slots
pthread_mutex_t mutex; // Mutex for critical section
void* producer(void* arg) {
    for (int i = 0; i < PRODUCE_COUNT; i++) {
        int item = rand() % 100; // Produce a random item
        sem_wait(&empty); // Wait for an empty slot
        pthread_mutex_lock(&mutex); // Enter critical section

        // Add the item to the buffer
        buffer[in] = item;
        printf("Produced: %d at %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE; // Move to the next position

        pthread_mutex_unlock(&mutex); // Exit critical section
        sem_post(&full); // Signal that a new item is available
        sleep(1); // Simulate time taken to produce an item
    }
}
```

```

    return NULL;
}

void* consumer(void* arg) {
    for (int i = 0; i < PRODUCE_COUNT; i++) {
        sem_wait(&full); // Wait for a full slot
        pthread_mutex_lock(&mutex); // Enter critical section

        // Remove an item from the buffer
        int item = buffer[out];
        printf("Consumed: %d from %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE; // Move to the next position

        pthread_mutex_unlock(&mutex); // Exit critical section
        sem_post(&empty); // Signal that an empty slot is available
        sleep(2); // Simulate time taken to consume an item
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    // Initialize semaphores and mutex
    sem_init(&empty, 0, BUFFER_SIZE); // Initially all slots are empty
    sem_init(&full, 0, 0); // No slots are full
    pthread_mutex_init(&mutex, NULL); // Initialize mutex

    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for both threads to finish
    pthread_join(prod, NULL);

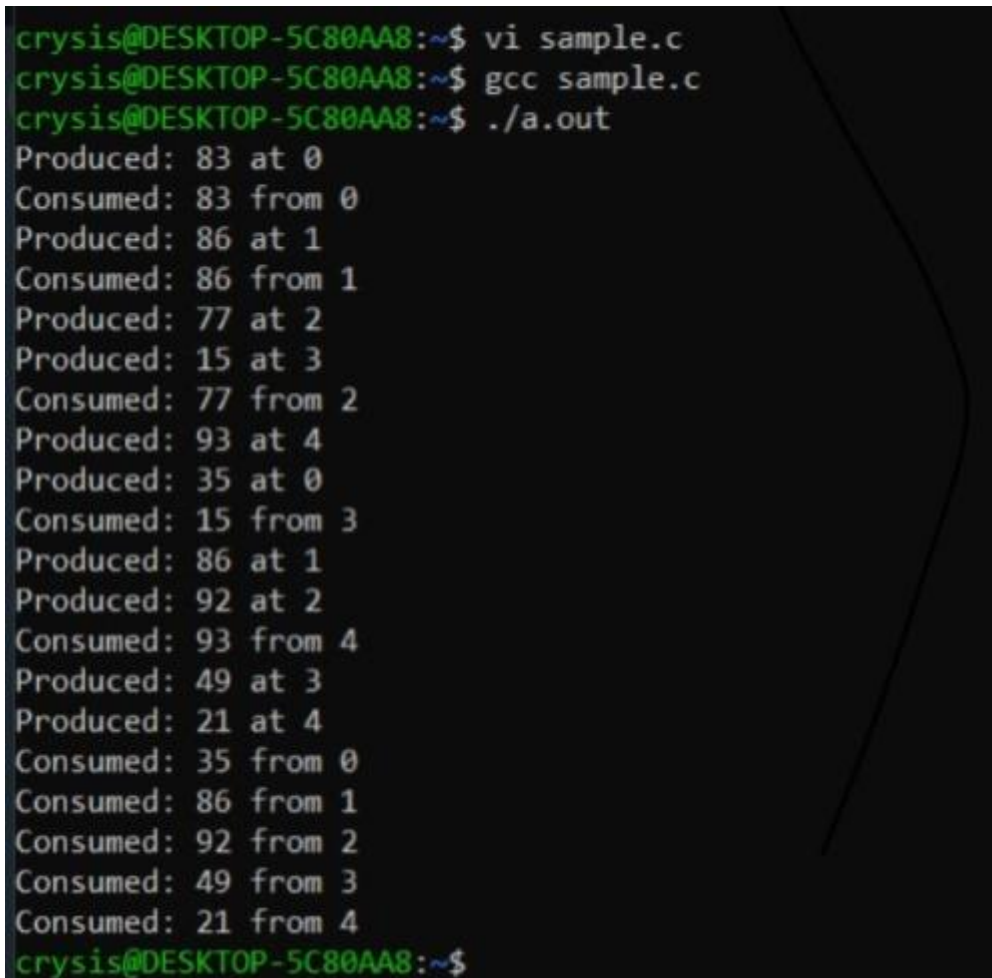
```

```
pthread_join(cons, NULL);

// Clean up
sem_destroy(&empty);
sem_destroy(&full);
pthread_mutex_destroy(&mutex);

return 0;
}
```

SAMPLE INPUT AND OUTPUT:



```
crysis@DESKTOP-5C80AA8:~$ vi sample.c
crysis@DESKTOP-5C80AA8:~$ gcc sample.c
crysis@DESKTOP-5C80AA8:~$ ./a.out
Produced: 83 at 0
Consumed: 83 from 0
Produced: 86 at 1
Consumed: 86 from 1
Produced: 77 at 2
Produced: 15 at 3
Consumed: 77 from 2
Produced: 93 at 4
Produced: 35 at 0
Consumed: 15 from 3
Produced: 86 at 1
Produced: 92 at 2
Consumed: 93 from 4
Produced: 49 at 3
Produced: 21 at 4
Consumed: 35 from 0
Consumed: 86 from 1
Consumed: 92 from 2
Consumed: 49 from 3
Consumed: 21 from 4
crysis@DESKTOP-5C80AA8:~$
```

RESULT:

Thus, the producer - consumer problem has been implemented using semaphores.