AIR UNIVERSITY, ISLAMABAD

Lab Task # 06

Submitted to: Ms.Maryam

Submitted by: Muhammad Abdullah | 221546

Date of Submission: March 18, 2024

CPU Scheduling Algorithms

1. First Come First Serve (FCFS):

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

struct fcfs {
    int pid;
    int btime;
    int wtime;
    int ttime;
} p[10];

int main() {
    int i, n;
    int totwtime= 0, totttime = 0;

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

    for (i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &p[i].btime);
    }

    p[0].wtime = 0; p[0].ttime = p[0].btime;
    totttime += p[0].ttime;

    for (i = 1; i < n; i++) {
        p[i].wtime = p[i - 1].wtime + p[i - 1].btime;
        p[i].ttime = p[i].wtime + p[i].btime;
        totttime += p[i].ttime; totwtime += p[i].wtime;
    }

    printf("\nProcess\tWaiting Time\tTurnaround Time\n");

    for(i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", p[i].pid,p[i].wtime, p[i].ttime);
    }

    printf("\nTotal Waiting Time: %d", totwtime);
    printf("\nAverage Waiting Time: %.2f", (float)totwtime / n);
    printf("\nTotal Turnaround Time: %d", totttime);
    printf("\nAverage Turnaround Time: %.2f\n", (float)totttime / n);

    return 0;
}
```

```
┌──(kali㉿kali)-[~]
└─$ nano fcfs.c

┌──(kali㉿kali)-[~]
└─$ gcc fcfs.c -o main

┌──(kali㉿kali)-[~]
└─$ ./main

FCFS scheduling ...
Enter the number of processes: 5
Enter burst time for process 1: 6
Enter burst time for process 2: 2
Enter burst time for process 3: 3
Enter burst time for process 4: 1
Enter burst time for process 5: 5

Process Waiting Time    Turnaround Time
1       0               6
2       6               8
3       8               11
4       11              12
5       12              17

Total Waiting Time: 37
Average Waiting Time: 7.40
Total Turnaround Time: 54
Average Turnaround Time: 10.80
```

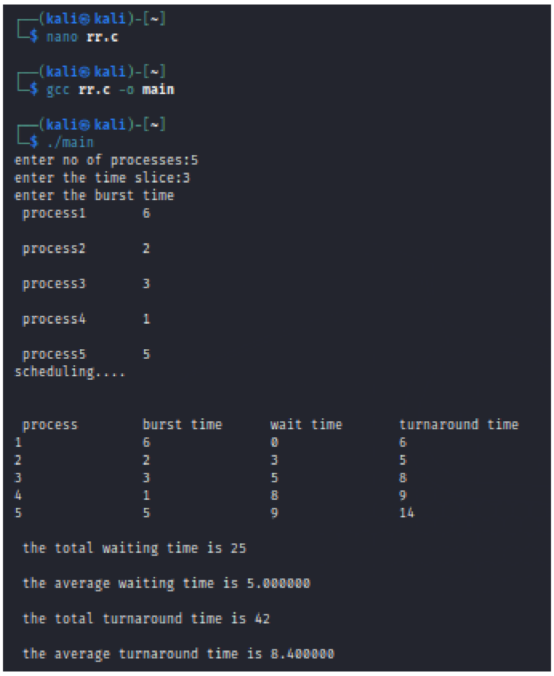
2. Shortest Job First (SJF):

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

typedef struct {
    int pid;
    int btime;
    int wtime;
} sp;

int main() {
    int i, j, n, tbm = 0, towtwtime = 0, totttime = 0;
    sp *p, t;

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

    p = (sp*)malloc(n * sizeof(sp));

    printf("\nEnter the burst time for each process:\n");

    for (i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &p[i].btime);

        p[i].pid = i + 1;
        p[i].wtime = 0;
    }

    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (p[i].btime > p[j].btime) {
                t = p[i]; p[i] = p[j];
                p[j] = t;
            }
        }
    }

    printf("\nProcess scheduling:\n");
    printf("\nProcess\tBurst Time\tWaiting Time\n");

    for (i = 0; i < n; i++) {
        towtwtime += p[i].wtime = tbm; tbm += p[i].btime;
        printf("%d\t%d\t\t%d\n", p[i].pid, p[i].btime, p[i].wtime);
    }

    totttime = tbm + towtwtime;

    printf("\nTotal Waiting Time: %d", towtwtime);
    printf("\nAverage Waiting Time: %.2f", (float)towtwtime / n);
    printf("\nTotal Turnaround Time: %d", totttime);
    printf("\nAverage Turnaround Time: %.2f\n", (float)totttime / n);

    free(p);

    return 0;
}
```

```
┌──(kali㉿kali)-[~]
└─$ nano sjf.c

┌──(kali㉿kali)-[~]
└─$ gcc sjf.c -o main

┌──(kali㉿kali)-[~]
└─$ ./main

SJF scheduling..
Enter the number of processes: 5

Enter the burst time for each process:
Process 1: 6
Process 2: 2
Process 3: 3
Process 4: 1
Process 5: 5

Process scheduling:

Process Burst Time      Waiting Time
4       1               0
2       2               1
3       3               3
5       5               6
1       6               11

Total Waiting Time: 21
Average Waiting Time: 4.20
Total Turnaround Time: 38
Average Turnaround Time: 7.60
```

3. Round Robin (RR):

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

struct rr {
    int pno, btime, sbtime, wtime, lst;
} p[10];

int main() {
    int pp = -1, ts, flag, count, ptm = 0, i, n, twt = 0, totttime = 0;

    printf("enter no of processes:");
    scanf("%d", &n);
    printf("enter the time slice:");
    scanf("%d",&ts);
    printf("enter the burst time");

    for (i = 0; i < n; i++) {
        printf("\n process%d\t", i + 1);
        scanf("%d", &p[i].btime);

        p[i].wtime = p[i].lst = 0;
        p[i].pno = i + 1;
        p[i].sbtime =p[i].btime;
    }

    printf("scheduling....\n");
    do{
        flag = 0;
        for (i = 0; i < n; i++) {
            count = p[i].btime;
            if (count > 0) {
                flag = -1;
                count = (count >= ts) ? ts : count;
                ptm += count;
                p[i].btime -= count;

                if (pp != i) {
                    pp = i; p[i].wtime += ptm - p[i].lst - count;
                    p[i].lst = ptm;
                }
            }
        }
    } while (flag != -1);

    printf("\n\n process\tburst time\twait time\tturnaround time");

    for (i = 0; i < n; i++) {
        printf("\n%d\t\t%d\t\t%d\t\t%d", p[i].pno, p[i].sbtime, p[i].wtime, p[i].sbtime + p[i].wtime);
        twt += p[i].wtime; totttime += p[i].sbtime + p[i].wtime;
    }

    printf("\n\n the total waiting time is %d", twt);
    printf("\n\n the average waiting time is %f", (float)twt / n);
    printf("\n\n the total turnaround time is %d", totttime);
    printf("\n\n the average turnaround time is %f", (float)totttime / n);

    return 0;
}
```



```
┌──(kali㉿kali)-[~]
└─$ nano rr.c

┌──(kali㉿kali)-[~]
└─$ gcc rr.c -o main

┌──(kali㉿kali)-[~]
└─$ ./main
enter no of processes:5
enter the time slice:3
enter the burst time
 process1       6

 process2       2

 process3       3

 process4       1

 process5       5
scheduling....


 process        burst time      wait time       turnaround time
1               6               0               6
2               2               3               5
3               3               5               8
4               1               8               9
5               5               9               14

 the total waiting time is 25

 the average waiting time is 5.000000

 the total turnaround time is 42

 the average turnaround time is 8.400000
```

Task

Source Code:

```c
#include <stdio.h>
#define MAX_PROCESS 10

// Process structure to hold process information
struct Process {
    int id;          // Process ID
    int burst_time;  // Burst time
    int waiting_time;  // Waiting time
    int turnaround_time;  // Turnaround time
};

// Circular Queue structure
struct CircularQueue {
    struct Process *array[MAX_PROCESS]; // Array to hold pointers to processes
    int front, rear;  // Front and rear pointers
    int size;         // Current size of the queue
};

// Function prototypes
void initializeQueue(struct CircularQueue *q);
void enqueue(struct CircularQueue *q, struct Process *p);
struct Process *dequeue(struct CircularQueue *q);
```

```
void roundRobin(struct Process processes[], int n, int time_quantum);

int main() {
    int n, time_quantum;

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

    if (n <= 0 || n > MAX_PROCESS) {
        printf("Invalid number of processes.\n");
        return 1;
    }

    struct Process processes[n];

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

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

    roundRobin(processes, n, time_quantum);

    // Display process details including waiting time and turnaround time
    printf("\nProcess\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].id, processes[i].waiting_time,
processes[i].turnaround_time);
    }

    // Calculate and display average waiting time
    float total_waiting_time = 0;
    for (int i = 0; i < n; i++) {
        total_waiting_time += processes[i].waiting_time;
    }
    float avg_waiting_time = total_waiting_time / n;
    printf("Average Waiting Time: %.2f\n", avg_waiting_time);

    return 0;
}
```

```c
// Initialize circular queue
void initializeQueue(struct CircularQueue *q) {
    q->front = -1;
    q->rear = -1;
    q->size = 0;
}

// Enqueue process into circular queue
void enqueue(struct CircularQueue *q, struct Process *p) {
    if ((q->rear + 1) % MAX_PROCESS == q->front) {
        printf("Queue is full.\n");
        return;
    }
    if (q->front == -1)
        q->front = 0;
    q->rear = (q->rear + 1) % MAX_PROCESS;
    q->array[q->rear] = p;
    q->size++;
}

// Dequeue process from circular queue
struct Process *dequeue(struct CircularQueue *q) {
    if (q->front == -1) {
        printf("Queue is empty.\n");
        return NULL;
    }
    struct Process *p = q->array[q->front];
    if (q->front == q->rear)
        q->front = q->rear = -1;
    else
        q->front = (q->front + 1) % MAX_PROCESS;
    q->size--;
    return p;
}

// Round Robin scheduling algorithm
void roundRobin(struct Process processes[], int n, int time_quantum) {
    struct CircularQueue q;
    initializeQueue(&q);
    int remaining_burst_time[n];

    // Copy burst times to remaining burst times array
    for (int i = 0; i < n; i++) {
        remaining_burst_time[i] = processes[i].burst_time;
    }
```

```
    int current_time = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (remaining_burst_time[i] > 0) {
                done = 0;
                if (remaining_burst_time[i] > time_quantum) {
                    current_time += time_quantum;
                    remaining_burst_time[i] -= time_quantum;
                    enqueue(&q, &processes[i]);
                } else {
                    current_time += remaining_burst_time[i];
                    processes[i].waiting_time = current_time - processes[i].burst_time;
                    remaining_burst_time[i] = 0;
                    processes[i].turnaround_time = current_time;
                }
            }
        }
        if (done == 1)
            break;
        while (q.size > 0) {
            struct Process *p = dequeue(&q);
            if (remaining_burst_time[p->id - 1] > 0) {
                enqueue(&q, p);
                break;
            }
        }
    }
}
```

Console:

```
┌──(kali㉿kali)-[~]
└─$ nano roundRobin.c

┌──(kali㉿kali)-[~]
└─$ gcc roundRobin.c -o main

┌──(kali㉿kali)-[~]
└─$ ./main
Enter the number of processes: 5
Enter burst times for each process:
Burst time for process 1: 6
Burst time for process 2: 2
Burst time for process 3: 3
Burst time for process 4: 1
Burst time for process 5: 5
Enter time quantum: 3

Process Waiting Time    Turnaround Time
1       9               15
2       3               5
3       5               8
4       8               9
5       12              17
Average Waiting Time: 7.40
```

Explanation:

- The main function is the entry point of the program. It prompts the user to enter the number of processes, burst times for each process, and the time quantum for the Round Robin algorithm. It then calls the roundRobin function to simulate the Round Robin scheduling algorithm.
- After the simulation, it displays the waiting time, turnaround time, and average waiting time for each process.
- Defines a structure struct Process to hold information about each process, including its ID, burst time, waiting time, and turnaround time.
- Defines a circular queue structure struct CircularQueue to implement the ready queue for the Round Robin algorithm. It holds an array of pointers to Process structures along with front and rear indices to manage the circular nature of the queue.
- Prototypes for functions initializeQueue, enqueue, dequeue, and roundRobin are provided.These functions are defined later in the code.
- Initializes the circular queue by setting front and rear indices to -1 and size to 0. Adds a process to the circular queue. It checks if the queue is full before enqueueing a process. Utilizes circular indexing for efficient management of the queue.
- Removes a process from the circular queue. It checks if the queue is empty before dequeuing a process. Returns a pointer to the dequeued process.
- Simulates the Round Robin scheduling algorithm. Initializes the circular queue and an array to track remaining burst times for each process. Loops until all processes are completed.
- If the remaining burst time is more than the time quantum, the process is executed for the time quantum, and then enqueued back to the queue.
- If the remaining burst time is less than or equal to the time quantum, the process is executed until completion.
- Waiting time and turnaround time for each process are calculated during execution. Ensures proper handling of processes in the circular queue.
- Displays the waiting time, turnaround time, and average waiting time for each process. Computes and shows the average waiting time for all processes.