Task Based Assignment

26.March.2020

Venkata Prasad.S
Registration no:11701894
Roll no:02
Section:EE031

Question

2. Considering the arrival time and burst time requirement of the process the scheduler schedules the processes by interrupting the processor after every 6 units of time and does consider the completion of the process in this iteration. The scheduler then checks for the number of processes waiting for the processor and allots the processor to the process but interrupts the processor every 10 units of time and considers the completion of the processes in this iteration. The scheduler checks the number of processes waiting in the queue for the processor after the second iteration and gives the processor to the process which needs more time to complete than the other processes to go in the terminated state.

The inputs for the number of requirements, arrival time and burst time should be provided by the user.

Consider the following units for reference. Process Arrival time Burst time

P1	0	20
P2	5	36
P3	13	19
P4	26	42

Link to Program with Execution:

https://onlinegdb.com/Sydomo0UL

Explanation:

Round Robin Scheduling: Each process is assigned a fixed timeTime Quantum/Time Slice in a cyclic way.It is designed especially for the time-sharing system. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. To implement Round Robin scheduling, we keep the ready queue as a First In First Out queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Explaining question in relative to answer:

Formula used in the program:

```
Tworst = {(dti+ sti + eti ), + (dti+ sti +
eti )2 +...+ (dti+ sti + eti )N., + (dti+
sti + eti  + eti) N} + tISR
```

As asked in question we need to implement a Round Robin algorithm which will changes in terms of its time break by quantum and it need to gets interrupting the process by 10 units of time as said in the program had been developed with taking those entities and regular time interruption with 6 units of time in this iteration and scheduler checks the number of processes waiting in the queue for the processor after the second iteration and gives the processor to the process which needs more time to complete than the other processes

Explaining Code:

Declaration of variables data types and data structure name had been declared at first

```
#define no 100
typedef struct Processes {
        int p_no;
        int arr_time;
        int burst_time;
} Process;
Process * queue[no];
int front = 0, rear = -1, processed = 0, curr_time = 0, tq = 6;
int last_front = 0, last_rear = -1;
```

this is used for the constant is been declared over here

here we are taken up data which we constantly use so they have been taken as to represent to all that data types

here we declared different variables

And from here we are going to take the inputs and make it to declare the value in effectively

```
void swap(Process * a, Process * b) {
        Process temp = * a;
        * a = * b;
        * b = temp;
}

                if (swapped == 0)
                        break;
        }
}
void execute() {
        int i;

        if (front-1 == rear) {
                printf("CPU idle for 1 second.\n");
                curr_time += 1;
        }

        err_flag = 0;
```

```
        Process p[n];
        for (i = 0; i < n; ++i) {
                printf("\n");
                printf("Enter arrival time of process %d: ", i+1);
                scanf("%d", &p[i].arr_time);
                printf("Enter burst time of process %d: ", i+1);
                scanf("%d", &p[i].burst_time);
                p[i].p_no = i+1;
        }
        sort(&p[0], n); Sort the processes according to the arrival time of each process.
        while (1) {
                enqueue(p, n);
                printf("\nIn queue: ");
                for (i = 0; i <= rear; ++i) {
                        printf("%d ", queue[i]->p_no);
                }
                printf("\nFront = %d, Rear = %d.\n\n", front, rear);
                execute();  If all the processes have been processed, break from the loop.
                if (processed == n)
                        break;
        }
        return 0;
}
```

Full program:

```
#include <stdio.h>
#define no 100
typedef struct Processes {
```

```
        int p_no;
        int arr_time;
        int burst_time;
} Process;
Process * queue[no];
int front = 0, rear = -1, processed = 0, curr_time = 0, tq = 6;
int last_front = 0, last_rear = -1;
void swap(Process * a, Process * b) {
        Process temp = * a;
        * a = * b;
        * b = temp;
}
void sort(Process p[], int n) {
        int i, j;
        short swapped;
        for (i = 0; i < n; ++i) {
                swapped = 0;
                for (j = 0; j < n-i-1; ++j)
                {
                        if (p[j].arr_time > p[j+1].arr_time)
                        {
                                swap(&p[j], &p[j+1]);
                                swapped = 1;
                        }
                }
                if (swapped == 0)
                        break;
        }
}
void enqueue(Process p[], int n) {
```

```
        int i, j, can_insert;
        for (i = 0; i < n; ++i)
        {
                can_insert = 1;

                if (p[i].arr_time <= curr_time && p[i].burst_time > 0)
                {
                        if (front == 0) {
                                queue[++rear] = &p[i];
                        }
                        else
                        {
                                for (j = last_front; j <= last_rear; ++j) {
                                        if (queue[j]->p_no == p[i].p_no)
                                                can_insert = 0;
                                }
                                if (can_insert == 1)
                                        queue[++rear] = &p[i];
                        }
                }
        }
        for (i = last_front; i <= last_rear; ++i)
        {
                if (queue[i]->burst_time > 0)
                        queue[++rear] = queue[i];
        }
}
void execute() {
        int i;
```

```
	if (front-1 == rear) {
		printf("CPU idle for 1 second.\n");
		curr_time += 1;
	}
	else {
		last_front = front;
		last_rear = rear;
		for (i = front; i <= rear; ++i, ++front)
		{
			if (queue[i]->burst_time > tq)
			{
				queue[i]->burst_time -= tq;
				curr_time += tq;
				printf("Process number %d excuted till %d seconds.\n",
queue[i]->p_no, curr_time);
			}
			else if (queue[i]->burst_time > 0)
			{
				curr_time += queue[i]->burst_time;
				queue[i]->burst_time = 0;
				printf("Process number %d excuted till %d seconds.\n",
queue[i]->p_no, curr_time);
				++processed;
			}
		}
	}
}
int main() {
	int n, i;
	short err_flag = 0;
	do {
```

```
            if (err_flag == 1)
                fprintf(stderr, "\nNumber of processes should be greater than 1.\n");
            printf("Enter the number of processes: ");
            scanf("%d", &n);
            err_flag = 1;
        } while (n < 1);
        err_flag = 0;
        Process p[n];
        for (i = 0; i < n; ++i) {
            printf("\n");
            printf("Enter arrival time of process %d: ", i+1);
            scanf("%d", &p[i].arr_time);
            printf("Enter burst time of process %d: ", i+1);
            scanf("%d", &p[i].burst_time);
            p[i].p_no = i+1;
        }
        sort(&p[0], n);
        while (1) {
            enqueue(p, n);
            printf("\nIn queue: ");
            for (i = 0; i <= rear; ++i) {
                printf("%d ", queue[i]->p_no);
            }
            printf("\nFront = %d, Rear = %d.\n\n", front, rear);
            execute();
            if (processed == n)
                break;
        }
        return 0;
}
```

GitHub link to Repository:

https://github.com/venkatVAD/operating-systems