Operating Systems Assignment No.2 Report

Owais Waheed (ow07611)

October 12, 2023

1 Code Implementation

This section explains the algorithm used to implement the scheduling policies and the implementation of the performance metrics.

1.1 First In First Out

In this policy, processes are executed in the order they arrive. Its implementation is defined below:

Firstly, I am initializing a ready queue which enqueue task according to their arrival time as shown in the listing below.

```
dlq* myq = (dlq*)malloc(sizeof(p_fq));
myq->head = get_new_node(NULL);
myq->tail = myq->head;

sort_by_arrival_time(p_fq);
dlq_node *current = remove_from_head(p_fq);

while (!is_empty(p_fq) && p_fq->head->data->ptimearrival < *p_time)
{
    dlq_node *temp = remove_from_head(p_fq);
    add_to_tail(myq, temp);
}
```

Listing 1: Initializing and Populating Populating Ready Queue

Now, when the process gets completed, the new process is dequeued from the ready queue and is placed on the processor as shown below.

```
if (process->ptimeleft > 1)
{
    process->ptimeleft--;
}
else
{
    free(process);
    current = remove_from_head(myq);
}
```

Listing 2: Updating the task on processor

When the process is running, following statements are getting printed in order to simulate the processor. If no process have arrived yet, then it prints the idle representing processor state else the name of process.

```
if (current->data->ptimearrival >= *p_time)
{
    printf("%d:idle:empty:\n", *p_time);
    continue;
}
pcb *process = current->data;
// Print the state of the selected process
printf("%d:%s:", *p_time, process->pname);
print_q(myq);
```

```
printf(":\n");
```

Listing 3: Printing the state of processor and ready queue

1.2 Shortest Job First

SJF is a non-preemptive policy where processes are scheduled based on their remaining execution time. The process with the shortest remaining time is given priority. The algorithm used for this policy is mostly the same as FIFO. The only difference was that I was sorting the ready queue after the arrival of every new process as shown in Listing 4.

```
while (!is_empty(p_fq) && p_fq->head->data->ptimearrival < *p_time)
{
    dlq_node *temp = remove_from_head(p_fq);
    add_to_tail(myq, temp);
    // Sort myq by remaining time (SJF)
    sort_by_timetocompletion(myq);

}
```

Listing 4: Sorting Ready queue on completion time

1.3 Shortest Time Completion First

STCF is a preemptive policy where the scheduler selects the process with the shortest estimated time to completion. If a shorter job arrives, the running process can be preempted. To implement this logic, we check if the arrived process has shorter time than the currently running process and if that's the case then we move the currently running process to the ready queue and place the arrived task into processor. And then we the rest implementation remains same as SJF. STCF implementation is shown in Listing 5:

```
while (!is_empty(p_fq) && p_fq->head->data->ptimearrival < *p_time)
{
    dlq_node *temp = remove_from_head(p_fq);
    add_to_tail(myq, temp);
    sort_by_timetocompletion(myq);
    int var1 = (current->data->ptimeleft);
    if(  (var1 > (myq->head->data->ptimeleft)))
    {
        add_to_tail(myq, current);
        current = remove_from_head(myq);
        sort_by_timetocompletion(myq);
    }
}
```

Listing 5: Updating Ready Queue according to STCF

1.4 Round Robin

RR is a policy which gives equal time slices to every process regardless of their completion time. To implement this logic, I enqueued the process running to ready queue at the end of every clock cycle with updated arrival time and dequeued the next process in line to place it on the processor as shown in Listing 6:

```
//update the remaining time
process->ptimeleft--;

if (process->ptimeleft == 0)
{
    current = remove_from_head(myq);
}
else
{
    current->data->ptimearrival = *p_time;
    add_to_tail(myq, current);
```

```
    current = remove_from_head(myq);
}
```

Listing 6: Implementation of Round Robin

1.5 Performance Metrics

To calculate the performance of the policies, I have made two queues one for response time and one for turnaround time. The queues are initialized with process arrival time. Now when the process is scheduled I update its value in response time queue with the formula:

$$TotalResponseTime = firstresponsetime - arrivaltime \tag{1}$$

For turn around time, I will update the process's value in turnaround queue with this formula:

$$TurnaroundTime = Completedtime - arrivaltime \tag{2}$$

Now for calculating average, I pass these queues to a function which sums the all values and then divide it by the number of process. This function is given in the below listing:

```
void perf_metric(int response_time_q[] , int turnaround_time_q[] , int total_proc , int
    throughput)
{
    float avg_turnaround;
    float avg_response;

    for(int i=0;i<total_proc ; i++)
    {
        avg_turnaround += turnaround_time_q[i];

        avg_response+= response_time_q[i];
    }

    avg_response = avg_response/total_proc;
    avg_turnaround = avg_turnaround/total_proc;


    printf("Performance Metrics : \n Throughput: ");
    printf("%d \n Average Response Time: %f ms \n Average Turnaround Time: %f ms \n" ,
    throughput , avg_response, avg_turnaround);
}
```

Listing 7: Performance Metrics for Scheduling Policy

2 Results

Test Case	Scheduling Policy	Throughput	Avg Response Time(ms)	Avg Turnaround Time(ms)
0	RR	3	1.000	12.333
	FIFO	3	5.000	10.333
	SJF	3	3.667	9.000
	STCF	3	4.333	9.000
5	RR	6	2.500	26.000
	FIFO	6	12.667	19.167
	SJF	6	10.333	16.833
	STCF	6	5.333	16.833
10	RR	3	2.500	36.333
	FIFO	5	18.333	27.667
	SJF	6	13.667	23.000
	STCF	5	8.667	22.667
13	RR	2	3.500	49.625
	FIFO	4	27.625	36.500
	SJF	6	18.500	27.375
	STCF	8	14.000	27.250

Table 1: Performance Metrics for Scheduling Policies on Given Test Cases

3 Discussion

3.1 Average Throughput

When I calculated average throughput, it remained same for every process because the throughput is the property of processor and at the end, every task has to run on processor regardless of its time so the throughput should remain same for every policy. In the above table, I have calculated no.of processes completed in first 50 milli-seconds but again throughput is not the correct metric to decide a scheduling policy.

3.2 Average Response Time

In the table, RR consistently shows the lowest average response times in all test cases. This indicates that RR offers efficient process initiation. After RR, STCF has the lowest response time, since, it is a pre-emptive policy which means it will have a good response time but it may lead lengthy jobs to starvation.

3.3 Average Turnaround Time

In the above test cases, SJF and STCF often demonstrate lower average turnaround times, indicating that these policies lead to faster task completion. In some cases, STCF has slightly lower time than SJF, which makes it the best policy when considering turnaround time. RR exhibits the highest turnaround times, indicating that it may not always prioritize quick task completion.

4 Conclusion

In conclusion, the choice of scheduling policy should align with the specific needs and priorities of a given system. RR offers fairness but may lead to longer waiting times, while FIFO is predictable but may not always optimize execution. SJF and STCF excel at reducing response times and maximizing throughput but may not guarantee fairness. Careful consideration of the system's requirements is essential to choose the most suitable scheduling policy to optimize resource utilization and ensure efficient task execution.

A Appendix

A.1 First In First Out

```
//implement the FIFO scheduling code
void sched_FIFO(dlq *const p_fq, int *p_time)
{
    dlq* myq = (dlq*)malloc(sizeof(p_fq));
    myq->head = get_new_node(NULL);
    myq->tail = myq->head;

    sort_by_arrival_time(p_fq);
    dlq_node *current = remove_from_head(p_fq);

    while (current != NULL)
    {
        (*p_time)++;
        //move to myq (ready queue) when the arrival time == system time
        while (!is_empty(p_fq) && p_fq->head->data->ptimearrival < *p_time)
        {
            dlq_node *temp = remove_from_head(p_fq);
            add_to_tail(myq, temp);
        }
        // If no process has not arrived, print idle and continue
        if (current->data->ptimearrival >= *p_time)
        {
            printf("%d:idle:empty:\n", *p_time);
            continue;
        }

        if(check == '1')
        {
            response_time_q[now] = *p_time - response_time_q[now]-1;
            check = '0';
        }

        pcb *process = current->data;
        // Print the state of the selected process
        printf("%d:%s:", *p_time, process->pname);
        print_q(myq);
        printf(":\n");

        // Update the remaining time for the process
        if (process->ptimeleft > 1)
        {
            process->ptimeleft--;
        }
        else
        {
            free(process);
            current = remove_from_head(myq);
        }
    }
    // perf_metric(response_time_q , turnaround_time_q , total_proc , throughput );
}
```

Listing 8: FIFO Scheduler Function

A.2 Shortest Job First

```
//implement the FIFO scheduling code
void sched_SJF(dlq *const p_fq, int *p_time)
{
    dlq* myq = (dlq*)malloc(sizeof(p_fq));
    myq->head = get_new_node(NULL);
    myq->tail = myq->head;
```

```
7
8       sort_by_arrival_time(p_fq);
9       dlq_node *current = remove_from_head(p_fq);
10
11      while (current != NULL)
12      {
13          (*p_time)++;
14          //move to myq (ready queue) when the arrival time == system time
15          while (!is_empty(p_fq) && p_fq->head->data->ptimearrival < *p_time)
16          {
17              dlq_node *temp = remove_from_head(p_fq);
18              add_to_tail(myq, temp);
19              // Sort myq by remaining time (SJF)
20              sort_by_timetocompletion(myq);
21          }
22          // If no process has not arrived, print idle and continue
23          if (current->data->ptimearrival >= *p_time)
24          {
25              printf("%d:idle:empty:\n", *p_time);
26              continue;
27          }
28
29          if(check == '1')
30          {
31              response_time_q[now] = *p_time - response_time_q[now]-1;
32              check = '0';
33          }
34
35          pcb *process = current->data;
36          // Print the state of the selected process
37          printf("%d:%s:", *p_time, process->pname);
38          print_q(myq);
39          printf(":\n");
40
41          // Update the remaining time for the process
42          if (process->ptimeleft > 1)
43          {
44              process->ptimeleft--;
45          }
46          else
47          {
48              free(process);
49              current = remove_from_head(myq);
50          }
51      }
52      // perf_metric(response_time_q , turnaround_time_q , total_proc , throughput );
53  }
```

Listing 9: SJF Scheduler Function

A.3 Shortest Time Completion First

```
1   //implement the FIFO scheduling code
2   void sched_STCF(dlq *const p_fq, int *p_time)
3   {
4       dlq* myq = (dlq*)malloc(sizeof(p_fq));
5       myq->head = get_new_node(NULL);
6       myq->tail = myq->head;
7
8       sort_by_arrival_time(p_fq);
9       dlq_node *current = remove_from_head(p_fq);
10
11      while (current != NULL)
12      {
13          (*p_time)++;
14          //move to myq (ready queue) when the arrival time == system time
```

```
        while (!is_empty(p_fq) && p_fq->head->data->ptimearrival < *p_time)
        {
            dlq_node *temp = remove_from_head(p_fq);
            add_to_tail(myq, temp);
            sort_by_timetocompletion(myq);

            int var1 = (current->data->ptimeleft);
            if(  (var1 > (myq->head->data->ptimeleft)))
            {
                add_to_tail(myq, current);
                current = remove_from_head(myq);

                sort_by_timetocompletion(myq);

            }
        }
        // If no process has not arrived, print idle and continue
        if (current->data->ptimearrival >= *p_time)
        {
            printf("%d:idle:empty:\n", *p_time);
            continue;
        }

        if(check == '1')
        {
            response_time_q[now] = *p_time - response_time_q[now]-1;
            check = '0';
        }

        pcb *process = current->data;
        // Print the state of the selected process
        printf("%d:%s:", *p_time, process->pname);
        print_q(myq);
        printf(":\n");

        // Update the remaining time for the process
        if (process->ptimeleft > 1)
        {
            process->ptimeleft--;
        }
        else
        {
            free(process);
            current = remove_from_head(myq);
        }
    }
    // perf_metric(response_time_q , turnaround_time_q , total_proc , throughput );
}
```

Listing 10: STCF Scheduler Function

A.4 Round Robin

```
//implement the FIFO scheduling code
void sched_RR(dlq *const p_fq, int *p_time)
{
    dlq* myq = (dlq*)malloc(sizeof(p_fq));
    myq->head = get_new_node(NULL);
    myq->tail = myq->head;

    sort_by_arrival_time(p_fq);
    dlq_node *current = remove_from_head(p_fq);

    while (current != NULL)
    {
        (*p_time)++;
```

```
        //move to myq (ready queue) when the arrival time == system time
        while (!is_empty(p_fq) && p_fq->head->data->ptimearrival < *p_time)
        {
            dlq_node *temp = remove_from_head(p_fq);
            add_to_tail(myq, temp);
        }
        // If no process has not arrived, print idle and continue
        if (current->data->ptimearrival >= *p_time)
        {
            printf("%d:idle:empty:\n", *p_time);
            continue;
        }

        if(check == '1')
        {
            response_time_q[now] = *p_time - response_time_q[now]-1;
            check = '0';
        }

        pcb *process = current->data;
        // Print the state of the selected process
        printf("%d:%s:", *p_time, process->pname);
        print_q(myq);
        printf(":\n");

        //update the remaining time
        process->ptimeleft--;

        if (process->ptimeleft == 0)
        {
           current = remove_from_head(myq);
        }
        else
        {
            current->data->ptimearrival = *p_time;
            add_to_tail(myq, current);
            current = remove_from_head(myq);
        }
    }
    // perf_metric(response_time_q , turnaround_time_q , total_proc , throughput );
}
```

Listing 11: RR Scheduler Function