

Habib University
Operating Systems - CS232

Homework 02 - Report
Process Scheduler



Instructor: Munzir Zafar

Sadiqah Mushtaq - 07152

Contents

1	Introduction	3
2	Makefile	3
3	Input	3
4	Output	3
5	Structures and Typedefs	4
6	Scheduling Algorithms	4
6.1	FIFO - First In First Out	4
6.2	SJF - Shortest Job First	6
6.3	STCF - Shortest Time to Completion First	8
6.4	RR - Round Robin	10
7	Performance Metrics of Scheduling Algorithms	12
7.1	Testing Performance Metrics	12
7.1.1	Test Case 0 / Test Case 2	12
7.1.2	Test Case 5	12
7.1.3	Test Case 10	13
7.1.4	Test Case 13	13
7.2	Results - Comparison of Performance Metrics	14
A	Appendix	15
A.1	Code	15

1 Introduction

This assignment is an implementation of a simple process scheduler in C. It defines various data structures and functions for scheduling processes using different algorithms, such as First-Come-First-Served (FIFO), Shortest Job First (SJF), Shortest Time-to-Completion First (STCF), and Round Robin (RR). The code reads process information from the standard input, schedules the processes using the selected algorithm, and calculates performance metrics like throughput, turnaround time, and response time.

2 Makefile

```
1 build:
2     gcc -o Scheduler Scheduler.c
3
4 run:
5     ./Scheduler
6
7 clean:
8     rm -f scheduler
```

The Makefile provides a set of targets for building, running, and cleaning up the project. The "build" target compiles the code and generates an executable named "Scheduler." The "run" target executes the compiled program, while the "clean" target removes the executable to maintain a clean project directory.

3 Input

The input for the scheduler simulation consists of several components. It begins with a line specifying the total number of processes (N) and the chosen scheduling policy (a string representing FIFO, SJF, STCF, or RR). Following this, N lines of data input are provided, each containing the process details separated by colons. The components within a single input line include the process name (**pname**), process ID (**pid**), total runtime (**duration**), and arrival time (**arrivaltime**). The process name is a string with a maximum length of 10 characters, while all other fields are represented as integers.

Sample Input:

```
3
RR
P1:1:2:7:3
P2:1:5:3:5
P3:1:6:2
```

4 Output

The program simulates the scheduler and produces output at each step, following a specific format. Each output line is colon-separated and contains the following elements:

- **time**: Represents the number of clock-ticks that have passed since the system's initiation. Each clock-tick is assumed to last 1 millisecond, and the system starts at time 0.

- **running name:** Specifies the name of the process in the running state during the current clock-tick. If no process is running in a given clock-tick, the output indicates "idle."
- **ready queue names:** Consists of a comma-separated list of process names in the ready state during the current clock-tick, along with their corresponding time-to-completion values enclosed in parentheses. In the case of an empty queue, the output displays "empty."

Sample Output:

```
1:idle:empty:
2:idle:empty:
3:P3:empty:
4:P3:P1(7),:
5:P1:P3(4),:
6:P3:P1(6),P2(3),:
7:P1:P2(3),P3(3),:
8:P2:P3(3),P1(5),:
9:P3:P1(5),P2(2),:
10:P1:P2(2),P3(2),:
11:P2:P3(2),P1(4),:
12:P3:P1(4),P2(1),:
13:P1:P2(1),P3(1),:
14:P2:P3(1),P1(3),:
15:P3:P1(3),:
16:P1:empty:
17:P1:empty:
18:P1:empty:
```

This output format provides a clear and structured representation of the system's behavior during the scheduling process, aiding in the evaluation and analysis of the scheduler's performance.

5 Structures and Typedefs

- `struct pcb` represents the Process Control Block with process information.
- `struct dlq_node` is a node in the doubly-linked queue for processes.
- `struct dlq` represents the doubly-linked queue.
- `pcb` is a typedef for `struct pcb`.
- `dlq_node` is a typedef for `struct dlq_node`.
- `dlq` is a typedef for `struct dlq`.

6 Scheduling Algorithms

6.1 FIFO - First In First Out

```

9 void sched_FIFO(dlq *const p_fq, int *p_time, int N)
10 {
11     int process_time = 1;
12     int tt_cum = 0;
13     int rt_cum = 0;
14     double throughput = 0.0;
15     double tt_avg = 0.0;
16     double rt_avg = 0.0;
17     *p_time = 1;
18     dlq ready_queue;
19     ready_queue.head = NULL;
20     ready_queue.tail = NULL;
21
22     int running_process_time = 0;
23     dlq_node *curr_process = NULL;
24
25     while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process) {
26         // Check if any process arrives at the current time and move it to the ready
27         // queue
28         if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1) {
29             dlq_node *temp = remove_from_head(p_fq);
30             add_to_tail(&ready_queue, temp);
31         }
32
33         if (curr_process) {
34             pcb *p = curr_process->data;
35
36             // Simulate execution of the process
37             printf("%d:%s:", *p_time, p->pname);
38
39             // Print the contents of the ready queue
40             if (is_empty(&ready_queue)) {
41                 printf("empty");
42             } else {
43                 print_q(&ready_queue);
44             }
45
46             printf(":\n");
47
48             p->ptimeleft--;
49
50             if (p->ptimeleft > 0) {
51                 // Process still needs more time, continue running it
52                 running_process_time = p->ptimeleft;
53             } else {
54                 // incrementing the cum turnaround time to add this process
55                 tt_cum += (*p_time) - p->ptimearrival;
56                 // Free the memory allocated for the completed process
57                 free(curr_process);
58                 curr_process = NULL;
59                 running_process_time = 0;
60             }
61         } else if (!is_empty(&ready_queue)) {
62             curr_process = remove_from_head(&ready_queue);
63             pcb *p = curr_process->data;
64             // incrementing the cum response time to add this process
65             rt_cum += (*p_time) - 1 - p->ptimearrival;
66             printf("%d:%s:", *p_time, p->pname);
67
68             // Print the contents of the ready queue
69             if (is_empty(&ready_queue)) {

```

```

70     printf("empty");
71 } else {
72     print_q(&ready_queue);
73 }
74
75     printf(":\n");
76
77     p->ptimeleft--;
78     running_process_time = p->ptimeleft;
79
80 } else {
81     // No process is ready to run, so print "idle"
82     printf("%d:idle:empty:\n", *p_time);
83     process_time++;
84 }
85
86 // Increment system time
87 (*p_time)++;
88 }
89 process_time = (*p_time) - process_time; // +1 to accomodate the process
90 coming millisecond bef it is actually executed
91 printf("time: %d\n", process_time);
92 printf("No of processes: %d\n", N);
93 printf("Cumulative Turnaround: %d\n", tt_cum);
94
95 throughput = (double)N / process_time;
96 tt_avg = (double)tt_cum / N;
97 rt_avg = (double)rt_cum / N;
98
99 printf("Throughput: %.3f\n", throughput);
100 printf("Turnaround Time: %.3f\n", tt_avg);
101 printf("Response Time: %.3f\n", rt_avg);
102 printf("\n");
103 }

```

The sched_FIFO function implements the First-Come-First-Served (FIFO) scheduling policy. It tracks the system time and processes a queue of processes based on their arrival time, allowing each process to complete its execution before moving to the next. The function calculates performance metrics such as throughput, turnaround time, and response time, providing insights into the efficiency of the FIFO scheduling policy.

6.2 SJF - Shortest Job First

```

103 //implement the SJF scheduling code
104 void sched_SJF(dlq *const p_fq, int *p_time, int N)
105 {
106     int process_time = 1;
107     int tt_cum = 0;
108     int rt_cum = 0;
109     double throughput = 0.0;
110     double tt_avg = 0.0;
111     double rt_avg = 0.0;
112     *p_time = 1;
113     dlq ready_queue;
114     ready_queue.head = NULL;
115     ready_queue.tail = NULL;
116
117     int running_process_time = 0;
118     dlq_node *curr_process = NULL;

```

```
119
120 while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process) {
121     // Check if any process arrives at the current time and move it to the
    ready_queue
122     if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1) {
123         dlq_node *temp = remove_from_head(p_fq);
124         add_to_tail(&ready_queue, temp);
125         sort_by_timetocompletion(&ready_queue);
126     }
127
128     if (curr_process) {
129         pcb *p = curr_process->data;
130
131         // Simulate execution of the process
132         printf("%d:%s:", *p_time, p->pname);
133
134         // Print the contents of the ready queue
135         if (is_empty(&ready_queue)) {
136             printf("empty");
137         } else {
138             print_q(&ready_queue);
139         }
140
141         printf(":\n");
142
143         p->ptimeleft--;
144
145         if (p->ptimeleft > 0) {
146             // Process still needs more time, continue running it
147             running_process_time = p->ptimeleft;
148         } else {
149             // incrementing the cum turnaround time to add this process
150             tt_cum += (*p_time) - p->ptimearrival;
151             // Free the memory allocated for the completed process
152             free(curr_process);
153             curr_process = NULL;
154             running_process_time = 0;
155         }
156     }
157
158     else if (!is_empty(&ready_queue)) {
159         sort_by_timetocompletion(&ready_queue);
160         curr_process = remove_from_head(&ready_queue);
161         pcb *p = curr_process->data;
162         // incrementing the cum response time to add this process
163         rt_cum += (*p_time) - 1 - p->ptimearrival;
164         printf("%d:%s:", *p_time, p->pname);
165
166         // Print the contents of the ready queue
167         if (is_empty(&ready_queue)){
168             printf("empty");
169         }
170         else{
171             print_q(&ready_queue);
172         }
173
174         printf(":\n");
175
176         p->ptimeleft--;
177         running_process_time = p->ptimeleft;
178
179     }
```

```

180     else {
181         // No process is ready to run, so print "idle"
182         printf("%d:idle:empty:\n", *p_time);
183         process_time += 1;
184     }
185
186     // Increment system time
187     *p_time += 1;
188 }
189 process_time = (*p_time) - process_time;
190 printf("time: %d\n", process_time);
191 printf("No of processes: %d\n", N);
192 // printf("Cum response: %d\n", rt_cum);
193
194 throughput = (double)N / process_time;
195 tt_avg = (double)tt_cum / N;
196 rt_avg = (double)rt_cum / N;
197
198 printf("Throughput: %.3f\n", throughput);
199 printf("Turnaround Time: %.3f\n", tt_avg);
200 printf("Response Time: %.3f\n", rt_avg);
201 printf("\n");
202 }

```

The sched_SJF function is responsible for Shortest Job First (SJF) scheduling, a policy that prioritizes the execution of processes with the shortest remaining execution time. The function manages a ready queue and sorts it based on the remaining execution times of processes. Performance metrics like throughput, turnaround time, and response time are calculated and reported to evaluate the SJF scheduling policy's effectiveness.

6.3 STCF - Shortest Time to Completion First

```

203 //implement the STCF scheduling code
204 void sched_STCF(dlq *const p_fq, int *p_time, int N)
205 {
206     int process_time = 1;
207     int tt_cum = 0;
208     int rt_cum = 0;
209     double throughput = 0.0;
210     double tt_avg = 0.0;
211     double rt_avg = 0.0;
212     *p_time = 1;
213     dlq ready_queue;
214     ready_queue.head = NULL;
215     ready_queue.tail = NULL;
216
217     int running_process_time = 0;
218     dlq_node *curr_process = NULL;
219
220     while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process)
221     {
222         // If any process arrives at the current time, move it to the ready queue
223         if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1)
224         {
225             dlq_node *temp = remove_from_head(p_fq);
226             // Check if the current process has a shorter time to completion.
227             // If yes make it current instead
228             if (curr_process && temp && temp->data->ptimeleft < curr_process->data
->ptimeleft)

```



```

229         {
230             add_to_tail(&ready_queue, curr_process);
231             // free(curr_process);
232             // curr_process = NULL;
233             curr_process = temp;
234             running_process_time = temp->data->ptimeleft;
235         }
236
237         else {
238             add_to_tail(&ready_queue, temp);
239         }
240         sort_by_timetocompletion(&ready_queue);
241     }
242
243     if (curr_process){
244         pcb *p = curr_process->data;
245
246         // Simulate execution of the process
247         printf("%d:%s:", *p_time, p->pname);
248
249         // Print the contents of the ready queue
250         if (is_empty(&ready_queue)){
251             printf("empty");
252         }
253         else{
254             print_q(&ready_queue);
255         }
256
257         printf("\n");
258
259         p->ptimeleft--;
260
261         if (p->ptimeleft > 0){
262             // Process still needs more time, continue running it
263             running_process_time = p->ptimeleft;
264         }
265         else{
266             // Free the memory allocated for the completed process
267             // incrementing the cum turnaround time to add this process
268             tt_cum += (*p_time) - p->ptimearrival;
269             free(curr_process);
270             curr_process = NULL;
271             running_process_time = 0;
272         }
273     }
274     else if (!is_empty(&ready_queue)){
275         // sort_by_timetocompletion(&ready_queue); // Sort by time to
completion
276         curr_process = remove_from_head(&ready_queue);
277         pcb *p = curr_process->data;
278         // incrementing the cum response time to add this process
279         if(p->check != 1){
280             rt_cum += (*p_time) - 1 - p->ptimearrival;
281             p->check = 1;
282         }
283
284         printf("%d:%s:", *p_time, p->pname);
285
286         // Print the contents of the ready queue
287         if (is_empty(&ready_queue)){
288             printf("empty");
289         }

```

```

290         else{
291             print_q(&ready_queue);
292         }
293
294         printf(":\n");
295
296         p->ptimeleft--;
297         running_process_time = p->ptimeleft;
298     }
299     else{
300         // No process is ready to run, so print "idle"
301         printf("%d:idle:empty:\n", *p_time);
302         process_time++;
303     }
304
305     // Increment system time
306     (*p_time)++;
307 }
308 process_time = (*p_time) - process_time;
309 printf("time: %d\n", process_time);
310 printf("No of processes: %d\n", N);
311
312 throughput = (double)N / process_time;
313 tt_avg = (double)tt_cum / N;
314 rt_avg = (double)rt_cum / N;
315
316 printf("Throughput: %.3f\n", throughput);
317 printf("Turnaround Time: %.3f\n", tt_avg);
318 printf("Response Time: %.3f\n", rt_avg);
319 printf("\n");
320 }

```

The sched.STCF function implements the Shortest Time-to-Completion First (STCF) scheduling policy. It maintains a ready queue and selects the process with the shortest time to completion to execute. The function takes into account both the arrival time and execution time to make efficient scheduling decisions. Performance metrics, including throughput, turnaround time, and response time, are computed and presented for evaluating the STCF policy.

6.4 RR - Round Robin

```

321 // Implement the RR scheduling code
322 void sched_RR(dlq *const p_fq, int *p_time, int N)
323 {
324     int process_time = 1;
325     int tt_cum = 0;
326     int rt_cum = 0;
327     double throughput = 0.0;
328     double tt_avg = 0.0;
329     double rt_avg = 0.0;
330     *p_time = 1;
331     dlq ready_queue;
332     ready_queue.head = NULL;
333     ready_queue.tail = NULL;
334
335     dlq_node *curr_process = NULL;
336
337     while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process){
338         // If any process arrives at the current time, move it to the ready queue
339         if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1){

```

```

340     dlq_node *temp = remove_from_head(p_fq);
341     add_to_tail(&ready_queue, temp);
342 }
343 if (!is_empty(&ready_queue)){
344     curr_process = remove_from_head(&ready_queue);
345     pcb *p = curr_process->data;
346     // incrementing the cum response time to add this process
347     if(p->check != 1){
348         rt_cum += (*p_time) - 1 - p->ptimearrival;
349         p->check = 1;
350     }
351     printf("%d:%s:", *p_time, p->pname);
352
353     // Print the contents of the ready queue
354     if (is_empty(&ready_queue)){
355         printf("empty");
356     }
357     else{
358         print_q(&ready_queue);
359     }
360
361     printf(":\n");
362
363     p->ptimeleft--;
364
365     if (p->ptimeleft > 0){
366         // Process still needs more time, put it back in the ready queue
367         add_to_tail(&ready_queue, curr_process);
368     }
369     else if(p->ptimeleft == 0){
370         // incrementing the cum turnaround time to add this process
371         tt_cum += (*p_time) - p->ptimearrival;
372     }
373     curr_process = NULL;
374 }
375 else{
376     // No process is ready to run, so print "idle" and indicate that the
377     queue is empty
378     printf("%d:idle:empty:\n", *p_time);
379     process_time++;
380 }
381 // Increment system time
382 (*p_time)++;
383
384 }
385 process_time = (*p_time) - process_time;
386 printf("time: %d\n", process_time);
387 printf("No of processes: %d\n", N);
388
389 throughput = (double)N / process_time;
390 tt_avg = (double)tt_cum / N;
391 rt_avg = (double)rt_cum / N;
392
393 printf("Throughput: %.3f\n", throughput);
394 printf("Turnaround Time: %.3f\n", tt_avg);
395 printf("Response Time: %.3f\n", rt_avg);
396 printf("\n");
397 }

```

The sched_RR function simulates Round Robin (RR) scheduling, a policy that allocates a fixed

time quantum to each process, allowing them to execute in a cyclical manner. The function manages a ready queue and schedules processes based on the specified time quantum. Performance metrics, such as throughput, turnaround time, and response time, are computed to assess the effectiveness of the RR scheduling policy.

7 Performance Metrics of Scheduling Algorithms

7.1 Testing Performance Metrics

7.1.1 Test Case 0 / Test Case 2

Sample Input:

P1:1:2:7:3

P2:1:5:3:5

P3:1:6:2

Scheduling Algorithm	FIFO	SJF	STCF	RR
Throughput	0.188	0.188	0.188	0.188
Average Response Time	5.000	3.667	3.667	1.000
Average Turnaround Time	10.333	9.000	9.000	12.333

Table 1: Performance Metrics for Test Case 0/Test Case 2

The throughput remains consistent across all cases. Regarding performance metrics, the Round Robin (RR) scheduling policy exhibits the best average response time, while for the average turnaround time, the Shortest Job First (SJF) and Shortest Time-to-Completion First (STCF) schedulers perform most effectively.

7.1.2 Test Case 5

Sample Input:

P1:1:5:0

P2:2:7:2

P3:3:6:3

P4:4:9:4

P5:5:8:5

P6:6:4:7

Scheduling Algorithm	FIFO	SJF	STCF	RR
Throughput	0.154	0.154	0.154	0.154
Average Response Time	12.667	10.333	10.333	2.500
Average Turnaround Time	19.167	16.833	16.833	26.000

Table 2: Performance Metrics for Test Case 5

The throughput remains consistent across all cases again. Regarding performance metrics, the Round Robin (RR) scheduling policy exhibits the best average response time, while for the average turnaround time, the Shortest Job First (SJF) and Shortest Time-to-Completion First (STCF) schedulers perform most effectively.

7.1.3 Test Case 10

Sample Input:

P1:1:6:0
P2:2:12:2
P3:3:8:4
P4:4:15:5
P5:5:5:7
P6:6:10:9

Scheduling Algorithm	FIFO	SJF	STCF	RR
Throughput	0.107	0.107	0.107	0.107
Average Response Time	18.333	13.667	12.500	2.500
Average Turnaround Time	27.667	23.000	22.667	36.333

Table 3: Performance Metrics for Test Case 10

The throughput remains consistent across all cases. Regarding performance metrics, the Round Robin (RR) scheduling policy exhibits the best average response time, while for the average turnaround time, Shortest Time-to-Completion First (STCF) schedulers perform most effectively.

7.1.4 Test Case 13

Sample Input:

P1:1:10:0
P2:2:15:2
P3:3:8:4
P4:4:12:6
P5:5:6:9
P6:6:4:11
P7:7:7:13
P8:8:9:15

Scheduling Algorithm	FIFO	SJF	STCF	RR
Throughput	0.113	0.113	0.113	0.113
Average Response Time	27.625	18.500	17.875	3.500
Average Turnaround Time	36.500	27.375	27.250	49.625

Table 4: Performance Metrics for Test Case 13

The throughput remains consistent across all cases for a given testcase. Regarding performance metrics, the Round Robin (RR) scheduling policy exhibits the best average response time, while for the average turnaround time, Shortest Time-to-Completion First (STCF) schedulers perform most effectively.

7.2 Results - Comparison of Performance Metrics

The analysis of the performance metrics reveals several interesting insights. Firstly, it's evident that the best throughput remains consistent across all scheduling algorithms, which is expected. This uniformity arises from the fact that the total time taken for the execution of all processes in a test case remains the same, regardless of the scheduling algorithm employed. Therefore, the overall system throughput remains constant as it is determined by the cumulative processing time of the tasks.

Secondly, when considering the average response time, the Round Robin (RR) scheduling algorithm stands out as the best performer. This outcome aligns with the intuitive expectation that RR ensures each process receives a fair share of CPU time as it executes in a round-robin fashion. This equitability in resource allocation leads to relatively shorter response times for processes, making RR the optimal choice in scenarios where responsive task execution is a priority.

Lastly, in terms of average turnaround time, the Shortest Time to Completion First (STCF) scheduling algorithm outperforms the others. This result is coherent with the principle that STCF prioritizes the execution of processes with shorter runtimes, even if a process is currently in progress. By selecting the tasks with the least time left to completion, STCF minimizes the overall turnaround time, making it a suitable choice when efficiency in task completion is the primary concern. These insights underscore the significance of selecting the appropriate scheduling algorithm based on the specific requirements and goals of a computing system.

Note: I have computed the response time to account for the fact that when the first process arrives at for instance at 2ms, it begins execution immediately. Therefore, the response time for this specific process is 0ms, even though it is displayed at 3ms.

A Appendix

A.1 Code

```
398 #include <stdio.h>
399 #include <stdlib.h>
400 #include <string.h>
401
402 //process control block (PCB)
403 // in case of context switching it stores the information about the process
404 struct pcb
405 {
406     unsigned int pid;
407     char pname[20];
408     unsigned int ptimeleft;
409     unsigned int ptimearrival;
410     int check;
411 };
412
413 typedef struct pcb pcb;
414
415 //queue node
416 struct dlq_node
417 {
418     struct dlq_node *pfwd;
419     struct dlq_node *pbck;
420     struct pcb *data;
421 };
422
423 typedef struct dlq_node dlq_node;
424
425 //queue
426 struct dlq
427 {
428     struct dlq_node *head;
429     struct dlq_node *tail;
430 };
431
432 typedef struct dlq dlq;
433
434 //function to add a pcb to a new queue node
435 dlq_node* get_new_node(pcb *ndata)
436 {
437     if (!ndata)
438         return NULL;
439
440     dlq_node *new = malloc(sizeof(dlq_node));
441     if (!new)
442     {
443         fprintf(stderr, "Error: allocating memory\n"); exit(1);
444     }
445
446     new->pfwd = new->pbck = NULL;
447     new->data = ndata;
448     return new;
449 }
450
451 //function to add a node to the tail of queue
452 void add_to_tail (dlq *q, dlq_node *new)
453 {
```

```

454     if (!new)
455         return;
456
457     if (q->head==NULL)
458     {
459         if(q->tail!=NULL)
460         {
461             fprintf(stderr, "DLList inconsitent.\n"); exit(1);
462         }
463         q->head = new;
464         q->tail = q->head;
465     }
466     else
467     {
468         new->pfwd = q->tail;
469         new->pbck = NULL;
470         new->pfwd->pbck = new;
471         q->tail = new;
472     }
473 }
474
475 //function to remove a node from the head of queue
476 dlq_node* remove_from_head(dlq * const q){
477     if (q->head==NULL){ //empty
478         if(q->tail!=NULL){fprintf(stderr, "DLList inconsitent.\n"); exit(1);}
479         return NULL;
480     }
481     else if (q->head == q->tail) { //one element
482         if (q->head->pbck!=NULL || q->tail->pfwd!=NULL) {
483             fprintf(stderr, "DLList inconsitent.\n"); exit(1);
484         }
485
486         dlq_node *p = q->head;
487         q->head = NULL;
488         q->tail = NULL;
489
490         p->pfwd = p->pbck = NULL;
491         return p;
492     }
493     else { // normal
494         dlq_node *p = q->head;
495         q->head = q->head->pbck;
496         q->head->pfwd = NULL;
497
498         p->pfwd = p->pbck = NULL;
499         return p;
500     }
501 }
502
503 //function to print our queue
504 void print_q (const dlq *q)
505 {
506     dlq_node *n = q->head;
507     if (n == NULL)
508         return;
509
510     while (n)
511     {
512         printf("%s(%d)", n->data->pname, n->data->ptimeleft);
513         n = n->pbck;
514     }
515 }

```



```
516
517 //function to check if the queue is empty
518 int is_empty (const dlq *q)
519 {
520     if (q->head == NULL && q->tail==NULL)
521         return 1;
522     else if (q->head != NULL && q->tail != NULL)
523         return 0;
524     else
525     {
526         fprintf(stderr, "Error: DLL queue is inconsistent."); exit(1);
527     }
528 }
529
530 //function to sort the queue on completion time
531 void sort_by_timetocompletion(const dlq *q)
532 {
533     // bubble sort
534     dlq_node *start = q->tail;
535     dlq_node *end = q->head;
536
537     while (start != end)
538     {
539         dlq_node *node = start;
540         dlq_node *next = node->pfwd;
541
542         while (next != NULL)
543         {
544             if (node->data->ptimeleft < next->data->ptimeleft)
545             {
546                 // do a swap
547                 pcb *temp = node->data;
548                 node->data = next->data;
549                 next->data = temp;
550             }
551             node = next;
552             next = node->pfwd;
553         }
554         end = end ->pbck;
555     }
556 }
557
558 //function to sort the queue on arrival time
559 void sort_by_arrival_time (const dlq *q)
560 {
561     // bubble sort
562     dlq_node *start = q->tail;
563     dlq_node *end = q->head;
564
565     while (start != end)
566     {
567         dlq_node *node = start;
568         dlq_node *next = node->pfwd;
569
570         while (next != NULL)
571         {
572             if (node->data->ptimearrival < next->data->ptimearrival)
573             {
574                 // do a swap
575                 pcb *temp = node->data;
576                 node->data = next->data;
577                 next->data = temp;
```

```

578     }
579     node = next;
580     next = node->pfwd;
581 }
582 end = end->pbck;
583 }
584 }
585
586 //function to tokenize the one row of data
587 pcb* tokenize_pdata (char *buf)
588 {
589     pcb* p = (pcb*) malloc(sizeof(pcb));
590     if(!p)
591     {
592         fprintf(stderr, "Error: allocating memory.\n"); exit(1);
593     }
594
595     char *token = strtok(buf, ":\n");
596     if(!token)
597     {
598         fprintf(stderr, "Error: Expecting token pname\n"); exit(1);
599     }
600     strcpy(p->pname, token);
601
602     token = strtok(NULL, ":\n");
603     if(!token)
604     {
605         fprintf(stderr, "Error: Expecting token pid\n"); exit(1);
606     }
607     p->pid = atoi(token);
608
609     token = strtok(NULL, ":\n");
610     if(!token)
611     {
612         fprintf(stderr, "Error: Expecting token duration\n"); exit(1);
613     }
614
615     p->ptimeleft= atoi(token);
616
617     token = strtok(NULL, ":\n");
618     if(!token)
619     {
620         fprintf(stderr, "Error: Expecting token arrival time\n"); exit(1);
621     }
622     p->ptimearrival = atoi(token);
623
624     token = strtok(NULL, ":\n");
625     if(token)
626     {
627         fprintf(stderr, "Error: Oh, what've you got at the end of the line?\n");
        exit(1);
628     }
629
630     return p;
631 }
632
633
634
635 void sched_FIFO(dlq *const p_fq, int *p_time, int N)
636 {
637     int process_time = 1;
638     int tt_cum = 0;

```

```

639 int rt_cum = 0;
640 double throughput = 0.0;
641 double tt_avg = 0.0;
642 double rt_avg = 0.0;
643 *p_time = 1;
644 dlq ready_queue;
645 ready_queue.head = NULL;
646 ready_queue.tail = NULL;
647
648 int running_process_time = 0;
649 dlq_node *curr_process = NULL;
650
651 while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process) {
652     // Check if any process arrives at the current time and move it to the ready
653     // queue
654     if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1) {
655         dlq_node *temp = remove_from_head(p_fq);
656         add_to_tail(&ready_queue, temp);
657     }
658
659     if (curr_process) {
660         pcb *p = curr_process->data;
661
662         // Simulate execution of the process
663         printf("%d:%s:", *p_time, p->pname);
664
665         // Print the contents of the ready queue
666         if (is_empty(&ready_queue)) {
667             printf("empty");
668         } else {
669             print_q(&ready_queue);
670         }
671
672         printf(":\n");
673
674         p->ptimeleft--;
675
676         if (p->ptimeleft > 0) {
677             // Process still needs more time, continue running it
678             running_process_time = p->ptimeleft;
679         } else {
680             // incrementing the cum turnaround time to add this process
681             tt_cum += (*p_time) - p->ptimearrival;
682             // Free the memory allocated for the completed process
683             free(curr_process);
684             curr_process = NULL;
685             running_process_time = 0;
686         }
687     } else if (!is_empty(&ready_queue)) {
688         curr_process = remove_from_head(&ready_queue);
689         pcb *p = curr_process->data;
690         // incrementing the cum response time to add this process
691         rt_cum += (*p_time) - 1 - p->ptimearrival;
692         printf("%d:%s:", *p_time, p->pname);
693
694         // Print the contents of the ready queue
695         if (is_empty(&ready_queue)) {
696             printf("empty");
697         } else {
698             print_q(&ready_queue);
699         }
700     }

```

```

701     printf(":\n");
702
703     p->ptimeleft--;
704     running_process_time = p->ptimeleft;
705
706 } else {
707     // No process is ready to run, so print "idle"
708     printf("%d:idle:empty:\n", *p_time);
709     process_time++;
710 }
711
712 // Increment system time
713 (*p_time)++;
714 }
715 process_time = (*p_time) - process_time; // +1 to accomodate the process
716 coming millisecond bef it is actually executed
717 printf("time: %d\n", process_time);
718 printf("No of processes: %d\n", N);
719 printf("Cumulative Turnaround: %d\n", tt_cum);
720
721 throughput = (double)N / process_time;
722 tt_avg = (double)tt_cum / N;
723 rt_avg = (double)rt_cum / N;
724
725 printf("Throughput: %.3f\n", throughput);
726 printf("Turnaround Time: %.3f\n", tt_avg);
727 printf("Response Time: %.3f\n", rt_avg);
728 printf("\n");
729 }
730 //implement the SJF scheduling code
731 void sched_SJF(dlq *const p_fq, int *p_time, int N)
732 {
733     int process_time = 1;
734     int tt_cum = 0;
735     int rt_cum = 0;
736     double throughput = 0.0;
737     double tt_avg = 0.0;
738     double rt_avg = 0.0;
739     *p_time = 1;
740     dlq ready_queue;
741     ready_queue.head = NULL;
742     ready_queue.tail = NULL;
743
744     int running_process_time = 0;
745     dlq_node *curr_process = NULL;
746
747     while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process) {
748         // Check if any process arrives at the current time and move it to the
749         // ready queue
750         if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1) {
751             dlq_node *temp = remove_from_head(p_fq);
752             add_to_tail(&ready_queue, temp);
753             sort_by_timetocompletion(&ready_queue);
754         }
755
756         if (curr_process) {
757             pcb *p = curr_process->data;
758
759             // Simulate execution of the process
760             printf("%d:%s:", *p_time, p->pname);

```

```

761         // Print the contents of the ready queue
762         if (is_empty(&ready_queue)) {
763             printf("empty");
764         } else {
765             print_q(&ready_queue);
766         }
767
768         printf(":\n");
769
770         p->ptimeleft--;
771
772         if (p->ptimeleft > 0) {
773             // Process still needs more time, continue running it
774             running_process_time = p->ptimeleft;
775         } else {
776             // incrementing the cum turnaround time to add this process
777             tt_cum += (*p_time) - p->ptimearrival;
778             // Free the memory allocated for the completed process
779             free(curr_process);
780             curr_process = NULL;
781             running_process_time = 0;
782         }
783     }
784     else if (!is_empty(&ready_queue)) {
785         sort_by_timetocompletion(&ready_queue);
786         curr_process = remove_from_head(&ready_queue);
787         pcb *p = curr_process->data;
788         // incrementing the cum response time to add this process
789         rt_cum += (*p_time) - 1 - p->ptimearrival;
790         printf("%d:%s:", *p_time, p->pname);
791
792         // Print the contents of the ready queue
793         if (is_empty(&ready_queue)){
794             printf("empty");
795         }
796         else{
797             print_q(&ready_queue);
798         }
799
800         printf(":\n");
801
802         p->ptimeleft--;
803         running_process_time = p->ptimeleft;
804
805     }
806     else {
807         // No process is ready to run, so print "idle"
808         printf("%d:idle:empty:\n", *p_time);
809         process_time += 1;
810     }
811
812     // Increment system time
813     *p_time += 1;
814 }
815 process_time = (*p_time) - process_time;
816 printf("time: %d\n", process_time);
817 printf("No of processes: %d\n", N);
818 // printf("Cum response: %d\n", rt_cum);
819
820 throughput = (double)N / process_time;
821 tt_avg = (double)tt_cum / N;

```

```

823     rt_avg = (double)rt_cum / N;
824
825     printf("Throughput: %.3f\n", throughput);
826     printf("Turnaround Time: %.3f\n", tt_avg);
827     printf("Response Time: %.3f\n", rt_avg);
828     printf("\n");
829 }
830
831 //implement the STCF scheduling code
832 void sched_STCF(dlq *const p_fq, int *p_time, int N)
833 {
834     int process_time = 1;
835     int tt_cum = 0;
836     int rt_cum = 0;
837     double throughput = 0.0;
838     double tt_avg = 0.0;
839     double rt_avg = 0.0;
840     *p_time = 1;
841     dlq ready_queue;
842     ready_queue.head = NULL;
843     ready_queue.tail = NULL;
844
845     int running_process_time = 0;
846     dlq_node *curr_process = NULL;
847
848     while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process)
849     {
850         // If any process arrives at the current time, move it to the ready queue
851         if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1)
852         {
853             dlq_node *temp = remove_from_head(p_fq);
854             // Check if the current process has a shorter time to completion.
855             // If yes make it current instead
856             if (curr_process && temp && temp->data->ptimeleft < curr_process->data-
->ptimeleft)
857             {
858                 add_to_tail(&ready_queue, curr_process);
859                 // free(curr_process);
860                 // curr_process = NULL;
861                 curr_process = temp;
862                 running_process_time = temp->data->ptimeleft;
863             }
864
865             else {
866                 add_to_tail(&ready_queue, temp);
867             }
868             sort_by_timetocompletion(&ready_queue);
869         }
870
871         if (curr_process){
872             pcb *p = curr_process->data;
873
874             // Simulate execution of the process
875             printf("%d:%s:", *p_time, p->pname);
876
877             // Print the contents of the ready queue
878             if (is_empty(&ready_queue)){
879                 printf("empty");
880             }
881             else{
882                 print_q(&ready_queue);
883             }

```

```

884         printf(":\n");
885
886         p->ptimeleft--;
887
888         if (p->ptimeleft > 0){
889             // Process still needs more time, continue running it
890             running_process_time = p->ptimeleft;
891         }
892         else{
893             // Free the memory allocated for the completed process
894             // incrementing the cum turnaround time to add this process
895             tt_cum += (*p_time) - p->ptimearrival;
896             free(curr_process);
897             curr_process = NULL;
898             running_process_time = 0;
899         }
900     }
901 }
902 else if (!is_empty(&ready_queue)){
903     // sort_by_timetocompletion(&ready_queue); // Sort by time to
completion
904     curr_process = remove_from_head(&ready_queue);
905     pcb *p = curr_process->data;
906     // incrementing the cum response time to add this process
907     if(p->check != 1){
908         rt_cum += (*p_time) - 1 - p->ptimearrival;
909         p->check = 1;
910     }
911     printf("%d:%s:", *p_time, p->pname);
912
913     // Print the contents of the ready queue
914     if (is_empty(&ready_queue)){
915         printf("empty");
916     }
917     else{
918         print_q(&ready_queue);
919     }
920 }
921
922     printf(":\n");
923
924     p->ptimeleft--;
925     running_process_time = p->ptimeleft;
926 }
927 else{
928     // No process is ready to run, so print "idle"
929     printf("%d:idle:empty:\n", *p_time);
930     process_time++;
931 }
932
933 // Increment system time
934 (*p_time)++;
935 }
936 process_time = (*p_time) - process_time;
937 printf("time: %d\n", process_time);
938 printf("No of processes: %d\n", N);
939
940 throughput = (double)N / process_time;
941 tt_avg = (double)tt_cum / N;
942 rt_avg = (double)rt_cum / N;
943
944 printf("Throughput: %.3f\n", throughput);

```

```
945     printf("Turnaround Time: %.3f\n", tt_avg);
946     printf("Response Time: %.3f\n", rt_avg);
947     printf("\n");
948 }
949
950
951 // Implement the RR scheduling code
952 void sched_RR(dlq *const p_fq, int *p_time, int N)
953 {
954     int process_time = 1;
955     int tt_cum = 0;
956     int rt_cum = 0;
957     double throughput = 0.0;
958     double tt_avg = 0.0;
959     double rt_avg = 0.0;
960     *p_time = 1;
961     dlq ready_queue;
962     ready_queue.head = NULL;
963     ready_queue.tail = NULL;
964
965     dlq_node *curr_process = NULL;
966
967     while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process){
968         // If any process arrives at the current time, move it to the ready queue
969         if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1){
970             dlq_node *temp = remove_from_head(p_fq);
971             add_to_tail(&ready_queue, temp);
972         }
973         if (!is_empty(&ready_queue)){
974             curr_process = remove_from_head(&ready_queue);
975             pcb *p = curr_process->data;
976             // incrementing the cum response time to add this process
977             if(p->check != 1){
978                 rt_cum += (*p_time) - 1 - p->ptimearrival;
979                 p->check = 1;
980             }
981             printf("%d:%s:", *p_time, p->pname);
982
983             // Print the contents of the ready queue
984             if (is_empty(&ready_queue)){
985                 printf("empty");
986             }
987             else{
988                 print_q(&ready_queue);
989             }
990
991             printf(":\n");
992
993             p->ptimeleft--;
994
995             if (p->ptimeleft > 0){
996                 // Process still needs more time, put it back in the ready queue
997                 add_to_tail(&ready_queue, curr_process);
998             }
999             else if(p->ptimeleft == 0){
1000                 // incrementing the cum turnaround time to add this process
1001                 tt_cum += (*p_time) - p->ptimearrival;
1002             }
1003             curr_process = NULL;
1004         }
1005     }
1006 }
```



```

1006     // No process is ready to run, so print "idle" and indicate that the
1007     queue is empty
1008     printf("%d:idle:empty:\n", *p_time);
1009     process_time++;
1010 }
1011
1012     // Increment system time
1013     (*p_time)++;
1014 }
1015 process_time = (*p_time) - process_time;
1016 printf("time: %d\n", process_time);
1017 printf("No of processes: %d\n", N);
1018
1019 throughput = (double)N / process_time;
1020 tt_avg = (double)tt_cum / N;
1021 rt_avg = (double)rt_cum / N;
1022
1023 printf("Throughput: %.3f\n", throughput);
1024 printf("Turnaround Time: %.3f\n", tt_avg);
1025 printf("Response Time: %.3f\n", rt_avg);
1026 printf("\n");
1027 }
1028
1029 int main()
1030 {
1031     /* Enter your code here. Read input from STDIN. Print output to STDOUT */
1032     int N = 0;
1033     char tech[20]={'\0'};
1034     char buffer[100]={'\0'};
1035     scanf("%d", &N);
1036     // printf("%d\n", N);
1037     scanf("%s", tech);
1038     // printf("%s\n", tech);
1039
1040     dlq queue;
1041     queue.head = NULL;
1042     queue.tail = NULL;
1043     for(int i=0; i<N; ++i)
1044     {
1045         scanf("%s\n", buffer);
1046         // printf("%s\n", buffer);
1047         pcb *p = tokenize_pdata(buffer);
1048         add_to_tail (&queue, get_new_node(p) );
1049     }
1050     // print_q(&queue);
1051     unsigned int system_time = 0;
1052     sort_by_arrival_time (&queue);
1053     // print_q (&queue);
1054
1055     // run scheduler
1056     if(!strcmp(tech,"FIFO",4))
1057         sched_FIFO(&queue, &system_time, N);
1058     else if(!strcmp(tech,"SJF",3))
1059         sched_SJF(&queue, &system_time, N);
1060     else if(!strcmp(tech,"STCF",4))
1061         sched_STCF(&queue, &system_time, N);
1062     else if(!strcmp(tech,"RR",2))
1063         sched_RR(&queue, &system_time, N);
1064     else
1065         fprintf(stderr, "Error: unknown POLICY\n");
1066     return 0;

```

1067 }