Habib University
Operating Systems - CS232

Homework 02 - Report

Process Scheduler

Instructor: Munzir Zafar

Sadiqah Mushtaq - 07152

Contents

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

```
build:
  gcc -o Scheduler Scheduler.c

run:
  ./Scheduler

clean:
  rm -f cheduler
```

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

```c
void sched_FIFO(dlq *const p_fq, int *p_time, int N)
{
  int process_time = 1;
  int tt_cum = 0;
  int rt_cum = 0;
  double throughput = 0.0;
  double tt_avg = 0.0;
  double rt_avg = 0.0;
  *p_time = 1;
  dlq ready_queue;
  ready_queue.head = NULL;
  ready_queue.tail = NULL;

  int running_process_time = 0;
  dlq_node *curr_process = NULL;

  while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process) {
    // Check if any process arrives at the current time and move it to the ready
    // queue
    if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1) {
      dlq_node *temp = remove_from_head(p_fq);
      add_to_tail(&ready_queue, temp);
    }

    if (curr_process) {
      pcb *p = curr_process->data;

      // Simulate execution of the process
      printf("%d:%s:", *p_time, p->pname);

      // Print the contents of the ready queue
      if (is_empty(&ready_queue)) {
        printf("empty");
      } else {
        print_q(&ready_queue);
      }

      printf(":\n");

      p->ptimeleft--;

      if (p->ptimeleft > 0) {
        // Process still needs more time, continue running it
        running_process_time = p->ptimeleft;
      } else {
        // incrementing the cum turnaround time to add this process
        tt_cum += (*p_time) - p->ptimearrival;
        // Free the memory allocated for the completed process
        free(curr_process);
        curr_process = NULL;
        running_process_time = 0;
      }
    } else if (!is_empty(&ready_queue)) {
      curr_process = remove_from_head(&ready_queue);
      pcb *p = curr_process->data;
      // incrementing the cum response time to add this process
      rt_cum += (*p_time) - 1 - p->ptimearrival;
      printf("%d:%s:", *p_time, p->pname);

      // Print the contents of the ready queue
      if (is_empty(&ready_queue)) {
```

```
        printf("empty");
      } else {
        print_q(&ready_queue);
      }

      printf(":\n");

      p->ptimeleft--;
      running_process_time = p->ptimeleft;

    } else {
      // No process is ready to run, so print "idle"
      printf("%d:idle:empty:\n", *p_time);
      process_time++;
    }

    // Increment system time
    (*p_time)++;
    }
    process_time = (*p_time) - process_time; // +1 to accomodate the process coming milliseconf bef it is actually executed
    printf("time: %d\n", process_time);
    printf("No of processes: %d\n", N);
    printf("Cumulative Turnaround: %d\n", tt_cum);

    throughput = (double)N / process_time;
    tt_avg = (double)tt_cum / N;
    rt_avg = (double)rt_cum / N;

    printf("Throughput: %.3f\n", throughput);
    printf("Turnaround Time: %.3f\n", tt_avg);
    printf("Response Time: %.3f\n", rt_avg);
    printf("\n");
}
```

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

```
//implement the SJF scheduling code
void sched_SJF(dlq *const p_fq, int *p_time, int N)
{
    int process_time = 1;
    int tt_cum = 0;
    int rt_cum = 0;
    double throughput = 0.0;
    double tt_avg = 0.0;
    double rt_avg = 0.0;
    *p_time = 1;
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    int running_process_time = 0;
    dlq_node *curr_process = NULL;
```

```c

    while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process) {
        // Check if any process arrives at the current time and move it to the ready queue
        if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1) {
            dlq_node *temp = remove_from_head(p_fq);
            add_to_tail(&ready_queue, temp);
            sort_by_timetocompletion(&ready_queue);

        }

        if (curr_process) {
            pcb *p = curr_process->data;

            // Simulate execution of the process
            printf("%d:%s:", *p_time, p->pname);

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)) {
                printf("empty");
            } else {
                print_q(&ready_queue);
            }

            printf(":\n");

            p->ptimeleft--;

            if (p->ptimeleft > 0) {
                // Process still needs more time, continue running it
                running_process_time = p->ptimeleft;
            } else {
                // incrementing the cum turnaround time to add this process
                tt_cum += (*p_time) - p->ptimearrival;
                // Free the memory allocated for the completed process
                free(curr_process);
                curr_process = NULL;
                running_process_time = 0;
            }
        }
        else if (!is_empty(&ready_queue)) {
            sort_by_timetocompletion(&ready_queue);
            curr_process = remove_from_head(&ready_queue);
            pcb *p = curr_process->data;
            // incrementing the cum response time to add this process
            rt_cum += (*p_time) - 1 - p->ptimearrival;
            printf("%d:%s:", *p_time, p->pname);

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)){
                printf("empty");
            }
            else{
                print_q(&ready_queue);
            }

            printf(":\n");

            p->ptimeleft--;
            running_process_time = p->ptimeleft;

        }
```

```
        else {
            // No process is ready to run, so print "idle"
            printf("%d:idle:empty:\n", *p_time);
            process_time += 1;
        }

        // Increment system time
        *p_time += 1;
    }
    process_time = (*p_time) - process_time;
    printf("time: %d\n", process_time);
    printf("No of processes: %d\n", N);
    // printf("Cum response: %d\n", rt_cum);

    throughput = (double)N / process_time;
    tt_avg = (double)tt_cum / N;
    rt_avg = (double)rt_cum / N;

    printf("Throughput: %.3f\n", throughput);
    printf("Turnaround Time: %.3f\n", tt_avg);
    printf("Response Time: %.3f\n", rt_avg);
    printf("\n");
}
```

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

```
//implement the STCF scheduling code
void sched_STCF(dlq *const p_fq, int *p_time, int N)
{
    int process_time = 1;
    int tt_cum = 0;
    int rt_cum = 0;
    double throughput = 0.0;
    double tt_avg = 0.0;
    double rt_avg = 0.0;
    *p_time = 1;
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    int running_process_time = 0;
    dlq_node *curr_process = NULL;

    while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process)
    {
        // If any process arrives at the current time, move it to the ready queue
        if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1)
        {
            dlq_node *temp = remove_from_head(p_fq);
            // Check if the current process has a shorter time to completion.
            // If yes make it current instead
            if (curr_process && temp && temp->data->ptimeleft < curr_process->data
    ->ptimeleft)
```

```c
            {
                add_to_tail(&ready_queue, curr_process);
                // free(curr_process);
                // curr_process = NULL;
                curr_process = temp;
                running_process_time = temp->data->ptimeleft;
            }

            else {
                add_to_tail(&ready_queue, temp);
            }
            sort_by_timetocompletion(&ready_queue);
        }

        if (curr_process){
            pcb *p = curr_process->data;

            // Simulate execution of the process
            printf("%d:%s:", *p_time, p->pname);

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)){
                printf("empty");
            }
            else{
                print_q(&ready_queue);
            }

            printf(":\n");

            p->ptimeleft--;

            if (p->ptimeleft > 0){
                // Process still needs more time, continue running it
                running_process_time = p->ptimeleft;
            }
            else{
                // Free the memory allocated for the completed process
                // incrementing the cum turnaround time to add this process
                tt_cum += (*p_time) - p->ptimearrival;
                free(curr_process);
                curr_process = NULL;
                running_process_time = 0;
            }
        }
        else if (!is_empty(&ready_queue)){
            // sort_by_timetocompletion(&ready_queue); // Sort by time to completion
            curr_process = remove_from_head(&ready_queue);
            pcb *p = curr_process->data;
            // incrementing the cum response time to add this process
            if(p->check != 1){
                rt_cum += (*p_time) - 1 - p->ptimearrival;
                p->check = 1;

            }
            printf("%d:%s:", *p_time, p->pname);

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)){
                printf("empty");
            }
```

```
            else{
                print_q(&ready_queue);
            }

            printf(":\n");

            p->ptimeleft--;
            running_process_time = p->ptimeleft;
        }
        else{
            // No process is ready to run, so print "idle"
            printf("%d:idle:empty:\n", *p_time);
            process_time++;
        }

        // Increment system time
        (*p_time)++;
    }
    process_time = (*p_time) - process_time;
    printf("time: %d\n", process_time);
    printf("No of processes: %d\n", N);

    throughput = (double)N / process_time;
    tt_avg = (double)tt_cum / N;
    rt_avg = (double)rt_cum / N;

    printf("Throughput: %.3f\n", throughput);
    printf("Turnaround Time: %.3f\n", tt_avg);
    printf("Response Time: %.3f\n", rt_avg);
    printf("\n");
}
```

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

```
// Implement the RR scheduling code
void sched_RR(dlq *const p_fq, int *p_time, int N)
{
    int process_time = 1;
    int tt_cum = 0;
    int rt_cum = 0;
    double throughput = 0.0;
    double tt_avg = 0.0;
    double rt_avg = 0.0;
    *p_time = 1;
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    dlq_node *curr_process = NULL;

    while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process){
        // If any process arrives at the current time, move it to the ready queue
        if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1){
```

```c
                dlq_node *temp = remove_from_head(p_fq);
                add_to_tail(&ready_queue, temp);
            }
            if (!is_empty(&ready_queue)){
                curr_process = remove_from_head(&ready_queue);
                pcb *p = curr_process->data;
                // incrementing the cum response time to add this process
                if(p->check != 1){
                    rt_cum += (*p_time) - 1 - p->ptimearrival;
                    p->check = 1;
                }
                printf("%d:%s:", *p_time, p->pname);

                // Print the contents of the ready queue
                if (is_empty(&ready_queue)){
                    printf("empty");
                }
                else{
                    print_q(&ready_queue);
                }

                printf(":\n");

                p->ptimeleft--;

                if (p->ptimeleft > 0){
                    // Process still needs more time, put it back in the ready queue
                    add_to_tail(&ready_queue, curr_process);
                }
                else if(p->ptimeleft == 0){
                    // incrementing the cum turnaround time to add this process
                    tt_cum += (*p_time) - p->ptimearrival;
                }
                curr_process = NULL;
            }
            else{
                // No process is ready to run, so print "idle" and indicate that the queue is empty
                printf("%d:idle:empty:\n", *p_time);
                process_time++;
            }

            // Increment system time
            (*p_time)++;

        }
        process_time = (*p_time) - process_time;
        printf("time: %d\n", process_time);
        printf("No of processes: %d\n", N);

        throughput = (double)N / process_time;
        tt_avg = (double)tt_cum / N;
        rt_avg = (double)rt_cum / N;

        printf("Throughput: %.3f\n", throughput);
        printf("Turnaround Time: %.3f\n", tt_avg);
        printf("Response Time: %.3f\n", rt_avg);
        printf("\n");
}
```

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

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

//process control block (PCB)
// in case of context switching it stores the information about the process
struct pcb
{
  unsigned int pid;
  char pname[20];
  unsigned int ptimeleft;
  unsigned int ptimearrival;
    int check;
};

typedef struct pcb pcb;

//queue node
struct dlq_node
{
  struct dlq_node *pfwd;
  struct dlq_node *pbck;
  struct pcb *data;
};

typedef struct dlq_node dlq_node;

//queue
struct dlq
{
  struct dlq_node *head;
  struct dlq_node *tail;
};

typedef struct dlq dlq;

//function to add a pcb to a new queue node
dlq_node* get_new_node(pcb *ndata)
{
  if (!ndata)
    return NULL;

  dlq_node *new = malloc(sizeof(dlq_node));
  if(!new)
    {
    fprintf(stderr, "Error: allocating memory\n");exit(1);
  }

  new->pfwd = new->pbck = NULL;
  new->data = ndata;
  return new;
}

//function to add a node to the tail of queue
void add_to_tail (dlq *q, dlq_node *new)
{
```

```
  if (!new)
    return;

  if (q->head==NULL)
    {
    if(q->tail!=NULL)
        {
      fprintf(stderr, "DLList inconsitent.\n"); exit(1);
    }
    q->head = new;
    q->tail = q->head;
  }
  else
    {
    new->pfwd = q->tail;
    new->pbck = NULL;
    new->pfwd->pbck = new;
    q->tail = new;
  }
}

//function to remove a node from the head of queue
dlq_node* remove_from_head(dlq * const q){
  if (q->head==NULL){ //empty
    if(q->tail!=NULL){fprintf(stderr, "DLList inconsitent.\n"); exit(1);}
    return NULL;
  }
  else if (q->head == q->tail) { //one element
    if (q->head->pbck!=NULL || q->tail->pfwd!=NULL) {
      fprintf(stderr, "DLList inconsitent.\n"); exit(1);
    }

    dlq_node *p = q->head;
    q->head = NULL;
    q->tail = NULL;

    p->pfwd = p->pbck = NULL;
    return p;
  }
  else { // normal
    dlq_node *p = q->head;
    q->head = q->head->pbck;
    q->head->pfwd = NULL;

    p->pfwd = p->pbck = NULL;
    return p;
  }
}

//function to print our queue
void print_q (const dlq *q)
{
  dlq_node *n = q->head;
  if (n == NULL)
    return;

  while (n)
    {
    printf("%s(%d),", n->data->pname, n->data->ptimeleft);
    n = n->pbck;
  }
}
```

```c

//function to check if the queue is empty
int is_empty (const dlq *q)
{
  if (q->head == NULL && q->tail==NULL)
    return 1;
  else if (q->head != NULL && q->tail != NULL)
    return 0;
  else
    {
    fprintf(stderr, "Error: DLL queue is inconsistent."); exit(1);
  }
}

//function to sort the queue on completion time
void sort_by_timetocompletion(const dlq *q)
{
  // bubble sort
  dlq_node *start = q->tail;
  dlq_node *end = q->head;

  while (start != end)
    {
    dlq_node *node = start;
    dlq_node *next = node->pfwd;

    while (next != NULL)
        {
      if (node->data->ptimeleft < next->data->ptimeleft)
            {
        // do a swap
        pcb *temp = node->data;
        node->data = next->data;
        next->data = temp;
      }
      node = next;
      next = node->pfwd;
    }
    end = end ->pbck;
  }
}

//function to sort the queue on arrival time
void sort_by_arrival_time (const dlq *q)
{
  // bubble sort
  dlq_node *start = q->tail;
  dlq_node *end = q->head;

  while (start != end)
    {
    dlq_node *node = start;
    dlq_node *next = node->pfwd;

    while (next != NULL)
        {
      if (node->data->ptimearrival < next->data->ptimearrival)
            {
        // do a swap
        pcb *temp = node->data;
        node->data = next->data;
        next->data = temp;
```

```
      }
      node = next;
      next = node->pfwd;
    }
    end = end->pbck;
  }
}

//function to tokenize the one row of data
pcb* tokenize_pdata (char *buf)
{
  pcb* p = (pcb*) malloc(sizeof(pcb));
  if(!p)
    {
        fprintf(stderr, "Error: allocating memory.\n"); exit(1);
    }

  char *token = strtok(buf, ":\n");
  if(!token)
    {
        fprintf(stderr, "Error: Expecting token pname\n"); exit(1);
    }
  strcpy(p->pname, token);

  token = strtok(NULL, ":\n");
  if(!token)
    {
        fprintf(stderr, "Error: Expecting token pid\n"); exit(1);
    }
  p->pid = atoi(token);

  token = strtok(NULL, ":\n");
  if(!token)
    {
        fprintf(stderr, "Error: Expecting token duration\n"); exit(1);
    }

  p->ptimeleft= atoi(token);

  token = strtok(NULL, ":\n");
  if(!token)
    {
        fprintf(stderr, "Error: Expecting token arrival time\n"); exit(1);
    }
  p->ptimearrival = atoi(token);

  token = strtok(NULL, ":\n");
  if(token)
    {
        fprintf(stderr, "Error: Oh, what've you got at the end of the line?\n");
    exit(1);
    }

  return p;
}



void sched_FIFO(dlq *const p_fq, int *p_time, int N)
{
  int process_time = 1;
  int tt_cum = 0;
```

```c
  int rt_cum = 0;
  double throughput = 0.0;
  double tt_avg = 0.0;
  double rt_avg = 0.0;
  *p_time = 1;
  dlq ready_queue;
  ready_queue.head = NULL;
  ready_queue.tail = NULL;

  int running_process_time = 0;
  dlq_node *curr_process = NULL;

  while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process) {
    // Check if any process arrives at the current time and move it to the ready
    // queue
    if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1) {
      dlq_node *temp = remove_from_head(p_fq);
      add_to_tail(&ready_queue, temp);
    }

    if (curr_process) {
      pcb *p = curr_process->data;

      // Simulate execution of the process
      printf("%d:%s:", *p_time, p->pname);

      // Print the contents of the ready queue
      if (is_empty(&ready_queue)) {
        printf("empty");
      } else {
        print_q(&ready_queue);
      }

      printf(":\n");

      p->ptimeleft--;

      if (p->ptimeleft > 0) {
        // Process still needs more time, continue running it
        running_process_time = p->ptimeleft;
      } else {
        // incrementing the cum turnaround time to add this process
        tt_cum += (*p_time) - p->ptimearrival;
        // Free the memory allocated for the completed process
        free(curr_process);
        curr_process = NULL;
        running_process_time = 0;
      }
    } else if (!is_empty(&ready_queue)) {
      curr_process = remove_from_head(&ready_queue);
      pcb *p = curr_process->data;
      // incrementing the cum response time to add this process
      rt_cum += (*p_time) - 1 - p->ptimearrival;
      printf("%d:%s:", *p_time, p->pname);

      // Print the contents of the ready queue
      if (is_empty(&ready_queue)) {
        printf("empty");
      } else {
        print_q(&ready_queue);
      }

```

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

      p->ptimeleft--;
      running_process_time = p->ptimeleft;

    } else {
      // No process is ready to run, so print "idle"
      printf("%d:idle:empty:\n", *p_time);
      process_time++;
    }

    // Increment system time
    (*p_time)++;
    }
    process_time = (*p_time) - process_time; // +1 to accomodate the process coming milliseconf bef it is actually executed
    printf("time: %d\n", process_time);
    printf("No of processes: %d\n", N);
    printf("Cumulative Turnaround: %d\n", tt_cum);

    throughput = (double)N / process_time;
    tt_avg = (double)tt_cum / N;
    rt_avg = (double)rt_cum / N;

    printf("Throughput: %.3f\n", throughput);
    printf("Turnaround Time: %.3f\n", tt_avg);
    printf("Response Time: %.3f\n", rt_avg);
    printf("\n");
}

//implement the SJF scheduling code
void sched_SJF(dlq *const p_fq, int *p_time, int N)
{
    int process_time = 1;
    int tt_cum = 0;
    int rt_cum = 0;
    double throughput = 0.0;
    double tt_avg = 0.0;
    double rt_avg = 0.0;
    *p_time = 1;
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    int running_process_time = 0;
    dlq_node *curr_process = NULL;

    while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process) {
        // Check if any process arrives at the current time and move it to the ready queue
        if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1) {
            dlq_node *temp = remove_from_head(p_fq);
            add_to_tail(&ready_queue, temp);
            sort_by_timetocompletion(&ready_queue);

        }

        if (curr_process) {
            pcb *p = curr_process->data;

            // Simulate execution of the process
            printf("%d:%s:", *p_time, p->pname);
```

```

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)) {
                printf("empty");
            } else {
                print_q(&ready_queue);
            }

            printf(":\n");

            p->ptimeleft--;

            if (p->ptimeleft > 0) {
                // Process still needs more time, continue running it
                running_process_time = p->ptimeleft;
            } else {
                // incrementing the cum turnaround time to add this process
                tt_cum += (*p_time) - p->ptimearrival;
                // Free the memory allocated for the completed process
                free(curr_process);
                curr_process = NULL;
                running_process_time = 0;
            }
        }
        else if (!is_empty(&ready_queue)) {
            sort_by_timetocompletion(&ready_queue);
            curr_process = remove_from_head(&ready_queue);
            pcb *p = curr_process->data;
            // incrementing the cum response time to add this process
            rt_cum += (*p_time) - 1 - p->ptimearrival;
            printf("%d:%s:", *p_time, p->pname);

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)){
                printf("empty");
            }
            else{
                print_q(&ready_queue);
            }

            printf(":\n");

            p->ptimeleft--;
            running_process_time = p->ptimeleft;

        }
        else {
            // No process is ready to run, so print "idle"
            printf("%d:idle:empty:\n", *p_time);
            process_time += 1;
        }

        // Increment system time
        *p_time += 1;
    }
    process_time = (*p_time) - process_time;
    printf("time: %d\n", process_time);
    printf("No of processes: %d\n", N);
    // printf("Cum response: %d\n", rt_cum);

    throughput = (double)N / process_time;
    tt_avg = (double)tt_cum / N;
```

```c
    rt_avg = (double)rt_cum / N;

    printf("Throughput: %.3f\n", throughput);
    printf("Turnaround Time: %.3f\n", tt_avg);
    printf("Response Time: %.3f\n", rt_avg);
    printf("\n");
}

//implement the STCF scheduling code
void sched_STCF(dlq *const p_fq, int *p_time, int N)
{
    int process_time = 1;
    int tt_cum = 0;
    int rt_cum = 0;
    double throughput = 0.0;
    double tt_avg = 0.0;
    double rt_avg = 0.0;
    *p_time = 1;
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    int running_process_time = 0;
    dlq_node *curr_process = NULL;

    while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process)
    {
        // If any process arrives at the current time, move it to the ready queue
        if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1)
        {
            dlq_node *temp = remove_from_head(p_fq);
            // Check if the current process has a shorter time to completion.
            // If yes make it current instead
            if (curr_process && temp && temp->data->ptimeleft < curr_process->data
    ->ptimeleft)
            {
                add_to_tail(&ready_queue, curr_process);
                // free(curr_process);
                // curr_process = NULL;
                curr_process = temp;
                running_process_time = temp->data->ptimeleft;
            }

            else {
                add_to_tail(&ready_queue, temp);
            }
            sort_by_timetocompletion(&ready_queue);
        }

        if (curr_process){
            pcb *p = curr_process->data;

            // Simulate execution of the process
            printf("%d:%s:", *p_time, p->pname);

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)){
                printf("empty");
            }
            else{
                print_q(&ready_queue);
            }
```

```

            printf(":\n");

            p->ptimeleft--;

            if (p->ptimeleft > 0){
                // Process still needs more time, continue running it
                running_process_time = p->ptimeleft;
            }
            else{
                // Free the memory allocated for the completed process
                // incrementing the cum turnaround time to add this process
                tt_cum += (*p_time) - p->ptimearrival;
                free(curr_process);
                curr_process = NULL;
                running_process_time = 0;
            }
        }
        else if (!is_empty(&ready_queue)){
            // sort_by_timetocompletion(&ready_queue); // Sort by time to
    completion
            curr_process = remove_from_head(&ready_queue);
            pcb *p = curr_process->data;
            // incrementing the cum response time to add this process
            if(p->check != 1){
                rt_cum += (*p_time) - 1 - p->ptimearrival;
                p->check = 1;

            }
            printf("%d:%s:", *p_time, p->pname);

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)){
                printf("empty");
            }
            else{
                print_q(&ready_queue);
            }

            printf(":\n");

            p->ptimeleft--;
            running_process_time = p->ptimeleft;
        }
        else{
            // No process is ready to run, so print "idle"
            printf("%d:idle:empty:\n", *p_time);
            process_time++;
        }

        // Increment system time
        (*p_time)++;
    }
    process_time = (*p_time) - process_time;
    printf("time: %d\n", process_time);
    printf("No of processes: %d\n", N);

    throughput = (double)N / process_time;
    tt_avg = (double)tt_cum / N;
    rt_avg = (double)rt_cum / N;

    printf("Throughput: %.3f\n", throughput);
```

```
    printf("Turnaround Time: %.3f\n", tt_avg);
    printf("Response Time: %.3f\n", rt_avg);
    printf("\n");
}


// Implement the RR scheduling code
void sched_RR(dlq *const p_fq, int *p_time, int N)
{
    int process_time = 1;
    int tt_cum = 0;
    int rt_cum = 0;
    double throughput = 0.0;
    double tt_avg = 0.0;
    double rt_avg = 0.0;
    *p_time = 1;
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    dlq_node *curr_process = NULL;

    while (!is_empty(p_fq) || !is_empty(&ready_queue) || curr_process){
        // If any process arrives at the current time, move it to the ready queue
        if (!is_empty(p_fq) && p_fq->head->data->ptimearrival == *p_time - 1){
            dlq_node *temp = remove_from_head(p_fq);
            add_to_tail(&ready_queue, temp);
        }
        if (!is_empty(&ready_queue)){
            curr_process = remove_from_head(&ready_queue);
            pcb *p = curr_process->data;
            // incrementing the cum response time to add this process
            if(p->check != 1){
                rt_cum += (*p_time) - 1 - p->ptimearrival;
                p->check = 1;
            }
            printf("%d:%s:", *p_time, p->pname);

            // Print the contents of the ready queue
            if (is_empty(&ready_queue)){
                printf("empty");
            }
            else{
                print_q(&ready_queue);
            }

            printf(":\n");

            p->ptimeleft--;

            if (p->ptimeleft > 0){
                // Process still needs more time, put it back in the ready queue
                add_to_tail(&ready_queue, curr_process);
            }
            else if(p->ptimeleft == 0){
                // incrementing the cum turnaround time to add this process
                tt_cum += (*p_time) - p->ptimearrival;
            }
            curr_process = NULL;
        }
        else{
```

```c
            // No process is ready to run, so print "idle" and indicate that the queue is empty
            printf("%d:idle:empty:\n", *p_time);
            process_time++;
        }

        // Increment system time
        (*p_time)++;

    }
    process_time = (*p_time) - process_time;
    printf("time: %d\n", process_time);
    printf("No of processes: %d\n", N);

    throughput = (double)N / process_time;
    tt_avg = (double)tt_cum / N;
    rt_avg = (double)rt_cum / N;

    printf("Throughput: %.3f\n", throughput);
    printf("Turnaround Time: %.3f\n", tt_avg);
    printf("Response Time: %.3f\n", rt_avg);
    printf("\n");
}

int main()
{
    /* Enter your code here. Read input from STDIN. Print output to STDOUT */
    int N = 0;
    char tech[20]={'\0'};
    char buffer[100]={'\0'};
    scanf("%d", &N);
    // printf("%d\n", N);
    scanf("%s", tech);
    // printf("%s\n", tech);

    dlq queue;
    queue.head = NULL;
    queue.tail = NULL;
    for(int i=0; i<N; ++i)
    {
        scanf("%s\n", buffer);
        // printf("%s\n", buffer);
        pcb *p = tokenize_pdata(buffer);
        add_to_tail (&queue, get_new_node(p) );
    }
    // print_q(&queue);
    unsigned int system_time = 0;
    sort_by_arrival_time (&queue);
    // print_q (&queue);

    // run scheduler
    if(!strncmp(tech,"FIFO",4))
        sched_FIFO(&queue, &system_time, N);
    else if(!strncmp(tech,"SJF",3))
        sched_SJF(&queue, &system_time, N);
    else if(!strncmp(tech,"STCF",4))
        sched_STCF(&queue, &system_time, N);
    else if(!strncmp(tech,"RR",2))
        sched_RR(&queue, &system_time, N);
    else
        fprintf(stderr, "Error: unknown POLICY\n");
    return 0;
```

```
}
```