

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Ульяновский государственный технический университет»
Кафедра «Вычислительная техника им. П. И. Соснина»

Операционные системы

(название дисциплины)

Лабораторная работа №2

«Алгоритмы планирования»

(название (тема) работы)

Выполнил:
студент группы ИВТАПбд-31
Молчанов А. В.
(Фамилия И. О.)

Проверил:
преподаватель, аспирант кафедры «ВТ»
(должность)
Беляев К. С.
(Фамилия И. О.)

Ульяновск

2023

Постановка задачи

На языке программирования С необходимо реализовать следующие алгоритмы планирования:

- «Первым пришёл – первым обслужен» (FCFS), при котором задачи планируются в том порядке, в котором они запрашивают ЦПУ;
- Shortest-Job-First (SJF), при котором задачи планируются в порядке продолжительности использования ими ЦПУ;
- Приоритетное планирование, при котором задачи планируются на основе приоритета;
- Циклическое планирование (RR), при котором каждая задача выполняется в течение определённого кванта времени или оставшуюся часть времени использования ЦПУ;
- Циклическое планирование с приоритетом: задачи планируются в порядке приоритета и используется циклическое планирование для задач с одинаковым приоритетом.

Предполагается, что все задачи поступают одновременно, поэтому алгоритмы планировщика не обязаны поддерживать процессы с более высоким приоритетом, вытесняя процессы с более низким приоритетом.

В качестве дополнительного задания необходимо подсчитать среднее обратное время, время ожидания и время отклика для каждого алгоритма планирования.

Детали реализации

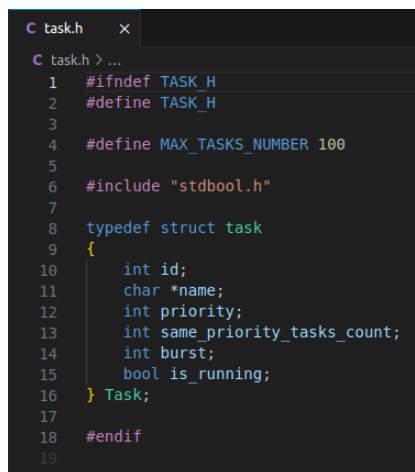
Выполнение процессов будет производиться на виртуальном процессоре, который реализован в виде двух файлов – CPU.h и CPU.c и представляет собой обычный вызов функции run с аргументами самого объекта процесса task и времени его выполнения slice (Рисунок 1).



```
C CPU.c x
C CPU.c > ...
1  #include "CPU.h"
2  #include "stdio.h"
3  #include "task.h"
4
5  void run(Task *task, int slice)
6  {
7      printf("Running task = [%s] [%d] [%d] for %d units.\n", task->name, task->priority, task->burst, slice);
8  }
9
```

Рисунок 1. Фрагмент кода №1

Сущность процесса реализована в заголовочном файле task.h (Рисунок 2) и представляет собой структуру данных с такими полями как идентификатор процесса (id), его наименование (name), приоритет (priority), счётчик количества процессов с таким же приоритетом (same_priority_tasks_count), а также время, которое потребуется на выполнение данного процесса (burst) и булевая переменная is_running, показывающая, был ли он уже запущен на виртуальном процессоре.



```
C task.h x
C task.h > ...
1  #ifndef TASK_H
2  #define TASK_H
3
4  #define MAX_TASKS_NUMBER 100
5
6  #include "stdbool.h"
7
8  typedef struct task
9  {
10     int id;
11     char *name;
12     int priority;
13     int same_priority_tasks_count;
14     int burst;
15     bool is_running;
16 } Task;
17
18 #endif
19
```

Рисунок 2. Фрагмент кода №2

Для хранения множества процессов используется массив объектов task – tasks_array_list или односвязный список объектов node – tasks_linked_list, которые определены в заголовочном файле list.h (Рисунок 3). Также, в данном файле объявлены два массива tasks_waiting_time и tasks_turnaround_time для выполнения дополнительного задания, которое заключается в подсчёте среднего оборотного времени, времени ожидания и времени отклика (в нашем случае совпадает с временем ожидания, так как все процессы поступают в одно и то же время) для каждого из алгоритмов планирования.

```
C list.h x
C list.h > ...
1  #ifndef _LIST_H
2  #define _LIST_H
3
4  #include "task.h"
5
6  // Элемент односвязного списка процессов
7  struct node
8  {
9      Task *task;
10     struct node *next_task;
11 };
12
13 extern struct task *tasks_array_list[MAX_TASKS_NUMBER]; // Массив процессов
14 extern struct node *tasks_linked_list; // Односвязный список процессов
15
16 extern int tasks_waiting_time[MAX_TASKS_NUMBER]; // Время ожидания каждого из процессов
17 extern int tasks_turnaround_time[MAX_TASKS_NUMBER]; // Оборотное время каждого из процессов
18
19 // Вставляет процесс new_task в конец односвязного списка
20 void insert(struct node **start_node, Task *new_task);
21
22 // Удаляет процесс deleting_task из односвязного списка
23 void delete(struct node **start_node, Task *deleting_task);
24
25 // Переворачивает односвязный список (первый его элемент становится последним, а последний первым)
26 struct node* reverse(struct node *start_node);
27
28 #endif
29
```

Рисунок 3. Фрагмент кода №3

Заголовки функций для работы со списком определены в этом же файле list.h (Рисунок 3), а их реализация представлена в файле list.c (Рисунок 4).

```
C list.c x
C list.c > ...
1  #include "list.h"
2  #include "stdlib.h"
3  #include "string.h"
4  #include "task.h"
5
6  void insert(struct node **start_node, Task *new_task)
7  {
8      struct node *new_node = malloc(sizeof(struct node));
9
10     new_node->task = new_task;
11     new_node->next_task = *start_node;
12
13     *start_node = new_node;
14 }
15
16 void delete(struct node **start_node, Task *deleting_task)
17 {
18     struct node *temp_node;
19     struct node *previous_node;
20
21     temp_node = *start_node;
22
23     // Если удаляемый элемент находится в начале списка
24     if (strcmp(deleting_task->name, temp_node->task->name) == 0)
25     {
26         *start_node = (*start_node)->next_task;
27     }
28     else
29     {
30         previous_node = *start_node;
31         temp_node = temp_node->next_task;
32
33         while (strcmp(deleting_task->name, temp_node->task->name) != 0)
34         {
35             previous_node = temp_node;
36             temp_node = temp_node->next_task;
37         }
38
39         previous_node->next_task = temp_node->next_task;
40     }
41 }
42
43 struct node* reverse(struct node *start_node)
44 {
45     if (start_node == NULL || start_node->next_task == NULL)
46         return start_node;
47
48     struct node *reverse_node = reverse(start_node->next_task);
49     start_node->next_task->next_task = start_node;
50
51     start_node->next_task = NULL;
52
53     return reverse_node;
54 }
55
```

Рисунок 4. Фрагмент кода №4

Файл driver.c (Рисунок 5), исходя из названия, приводит всю программу в движение, считывая, сохраняя и планируя необходимые для её корректной работы данные, используя для этого описанные выше структуры и функции.

```
C driver.c x
C driver.c > ...
1  #include "list.h"
2  #include "schedulers.h"
3  #include "stdio.h"
4  #include "stdlib.h"
5  #include "string.h"
6  #include "task.h"
7
8  struct node *tasks_linked_list;
9
10 int main(int argc, char *argv[])
11 {
12     tasks_linked_list = malloc(sizeof(struct node));
13
14     // Открываем для чтения файл schedule.txt со списком процессов
15     FILE *schedule_tasks_list = fopen(argv[1], "r");
16
17     // Строки считанного из файла процесса
18     char task[MAX_TASKS_NUMBER];
19     char *task_temp;
20
21     char *name;
22     int priority;
23     int burst;
24
25     // Пока не дошли до конца файла
26     while (fgets(task, MAX_TASKS_NUMBER, schedule_tasks_list) != NULL)
27     {
28         task_temp = strdup(task);
29
30         name = strtok(task_temp, ",");
31         priority = atoi(strtok(task_temp, ","));
32         burst = atoi(strtok(task_temp, ","));
33
34         // Добавляем данные о процессе в одну из структур данных (массив или односвязный список), выбор которой зависит от алгоритма планирования
35         add_task(name, priority, burst);
36
37         free(task_temp);
38     }
39
40     fclose(schedule_tasks_list);
41
42     // Планируем список считанных и сохранённых выше процессов на выполнение виртуальным процессором
43     schedule();
44
45     return 0;
46 }
47
```

Рисунок 5. Фрагмент кода №5

Заголовочный файл schedulers.h, подключённый к driver.c, содержит в себе две константные переменные для обозначения минимально ($\text{MIN_PRIORITY} = 1$) и максимально ($\text{MAX_PRIORITY} = 10$) возможного приоритета процессов (более высокое числовое значение указывает на более высокий относительный приоритет), а также заголовки таких функций как add_task и schedule вызываемые в драйвере программы (Рисунок 5).

Начнём рассматривать реализацию каждого из алгоритмов планирования с FCFS-планировщика (Рисунок 6), при котором задачи планируются на выполнение виртуальным центральным процессором в том порядке, в котором они его запрашивают.

Стоит отметить, что каждый из планировщиков переопределяет, в соответствии со своим предназначением, функции `add_task` и `schedule` из файла `schedulers.h` описанного выше. И если первая функция, вне зависимости от алгоритма, всегда делает одно и то же – добавляет процессы в очередь, то отличие между ними во второй заключается лишь в том, как и в каком порядке они планируют их выполнение. В случае с алгоритмом планирования FCFS (First Come First Served) всё происходит последовательно.

```
C schedule_fcfs.c x
C schedule_fcfs.c ...
1 #include "CPU.h"
2 #include "list.h"
3 #include "schedulers.h"
4 #include "stdio.h"
5 #include "stdlib.h"
6 #include "task.h"
7
8 int tasks_waiting_time[MAX_TASKS_NUMBER];
9 int tasks_turnaround_time[MAX_TASKS_NUMBER];
10
11 int tasks_number = 0;
12 float average_waiting_time = 0.0;
13 float average_turnaround_time = 0.0;
14
15 void add_task(char *name, int priority, int burst)
16 {
17     ++tasks_number;
18     struct task *new_task = malloc(sizeof(struct task));
19
20     new_task->id = tasks_number;
21     new_task->name = name;
22     new_task->priority = priority;
23     new_task->burst = burst;
24     insert(&tasks_linked_list, new_task);
25 }
26
27 void schedule()
28 {
29     // Переопределяем односвязный список процессов, так как каждый новый процесс добавлялся в его начало
30     tasks_linked_list = reverse(tasks_linked_list);
31     struct node *current_node = tasks_linked_list;
32
33     int task_index = 0, previous_task_burst_time;
34     tasks_waiting_time[task_index] = 0;
35
36     while (current_node != NULL)
37     {
38         if (current_node->task != NULL)
39         {
40             // Для первого процесса время ожидания равно нулю (строка 36)
41             if (task_index == 0)
42             {
43                 // Время ожидания текущего процесса - это время ожидания всех процессов до него + время выполнения предыдущего процесса
44                 tasks_waiting_time[task_index] = tasks_waiting_time[task_index - 1] + previous_task_burst_time;
45                 average_waiting_time += tasks_waiting_time[task_index];
46             }
47
48             run(current_node->task, current_node->task->burst);
49
50             // Обратное время текущего процесса - это его время ожидания + его время выполнения
51             tasks_turnaround_time[task_index] = tasks_waiting_time[task_index] + current_node->task->burst;
52             average_turnaround_time += tasks_turnaround_time[task_index];
53
54             ++task_index;
55             previous_task_burst_time = current_node->task->burst;
56         }
57         // Берём следующий процесс из списка
58         current_node = current_node->next_task;
59     }
60
61     printf("\nAverage waiting time = average response time = %.3f\n", average_waiting_time / (float)tasks_number);
62     printf("Average turnaround time = %.3f\n", average_turnaround_time / (float)tasks_number);
63 }
64
65
66
67
```

Рисунок 6. Фрагмент кода №6

Описание работы SJF-планировщика (Shortest Job First), при котором задачи планируются в порядке продолжительности использования ими ЦПУ приведено в комментариях к программному коду на Рисунке 7 ниже.

```
C schedule_sjf.c X
C schedule_sjf.c > ...
1  #include "CPU.h"
2  #include "list.h"
3  #include "schedulers.h"
4  #include "stdio.h"
5  #include "stdlib.h"
6  #include "task.h"
7
8  struct task *tasks_array_list[MAX_TASKS_NUMBER];
9
10 int tasks_waiting_time[MAX_TASKS_NUMBER];
11 int tasks_turnaround_time[MAX_TASKS_NUMBER];
12
13 int current_task_index = 0;
14 float average_waiting_time = 0.0;
15 float average_turnaround_time = 0.0;
16
17 void add_task(char *name, int priority, int burst)
18 {
19     struct task *new_task = malloc(sizeof(struct task));
20
21     new_task->id = current_task_index + 1;
22     new_task->name = name;
23     new_task->priority = priority;
24     new_task->burst = burst;
25
26     tasks_array_list[current_task_index] = new_task;
27     ++current_task_index;
28 }
29
30 void schedule()
31 {
32     struct task *temp_task;
33
34     for (int i = 0; i < current_task_index; ++i)
35     {
36         for (int j = i + 1; j < current_task_index; ++j)
37         {
38             // Сортируем массив процессов в порядке возрастания времени их выполнения на ЦПУ
39             if (tasks_array_list[i]->burst > tasks_array_list[j]->burst)
40             {
41                 temp_task = tasks_array_list[i];
42                 tasks_array_list[i] = tasks_array_list[j];
43                 tasks_array_list[j] = temp_task;
44             }
45             // Если время одинаково
46             else if (tasks_array_list[i]->burst == tasks_array_list[j]->burst)
47             {
48                 // Сортируем массив процессов в порядке убывания их приоритета
49                 if (tasks_array_list[i]->priority < tasks_array_list[j]->priority)
50                 {
51                     temp_task = tasks_array_list[i];
52                     tasks_array_list[i] = tasks_array_list[j];
53                     tasks_array_list[j] = temp_task;
54                 }
55             }
56         }
57     }
58
59     tasks_waiting_time[0] = 0;
60
61     // Перебираем весь массив процессов
62     for (int task_index = 0; task_index < current_task_index; ++task_index)
63     {
64         // Для первого процесса время ожидания равно нулю (строка 59)
65         if (task_index != 0)
66         {
67             // Время ожидания текущего процесса - это время ожидания всех процессов до него + время выполнения предыдущего процесса
68             tasks_waiting_time[task_index] = tasks_waiting_time[task_index - 1] + tasks_array_list[task_index - 1]->burst;
69             average_waiting_time += tasks_waiting_time[task_index];
70         }
71
72         run(tasks_array_list[task_index], tasks_array_list[task_index]->burst);
73
74         // Обратное время текущего процесса - это его время ожидания + его время выполнения
75         tasks_turnaround_time[task_index] = tasks_waiting_time[task_index] + tasks_array_list[task_index]->burst;
76         average_turnaround_time += tasks_turnaround_time[task_index];
77     }
78
79     printf("\nAverage waiting time = average response time = %.3f.\n", average_waiting_time / (float)current_task_index);
80     printf("Average turnaround time = %.3f.\n", average_turnaround_time / (float)current_task_index);
81 }
82
```

Рисунок 7. Фрагмент кода №7

Приоритетный планировщик, при котором задачи планируются на основе их приоритета отличается от предыдущего алгоритма планирования лишь порядком сортировки массива процессов `tasks_array_list`.

Описание его работы приведено в комментариях к программному коду на Рисунке 8 ниже.

```
C: schedule_priority.c X
C: schedule_priority.c > ...
1 #include "CPU.h"
2 #include "list.h"
3 #include "schedulers.h"
4 #include "stdio.h"
5 #include "stdlib.h"
6 #include "task.h"
7
8 struct task *tasks_array_list[MAX_TASKS_NUMBER];
9
10 int tasks_waiting_time[MAX_TASKS_NUMBER];
11 int tasks_turnaround_time[MAX_TASKS_NUMBER];
12
13 int current_task_index = 0;
14 float average_waiting_time = 0.0;
15 float average_turnaround_time = 0.0;
16
17 void add_task(char *name, int priority, int burst)
18 {
19     struct task *new_task = malloc(sizeof(struct task));
20
21     new_task->id = current_task_index + 1;
22     new_task->name = name;
23     new_task->priority = priority;
24     new_task->burst = burst;
25
26     tasks_array_list[current_task_index] = new_task;
27     ++current_task_index;
28 }
29
30 void schedule()
31 {
32     struct task *temp_task;
33
34     for (int i = 0; i < current_task_index; ++i)
35     {
36         for (int j = i + 1; j < current_task_index; ++j)
37         {
38             // Сортируем массив процессов в порядке убывания их приоритета
39             if (tasks_array_list[i]->priority < tasks_array_list[j]->priority)
40             {
41                 temp_task = tasks_array_list[i];
42                 tasks_array_list[i] = tasks_array_list[j];
43                 tasks_array_list[j] = temp_task;
44             }
45             // Если приоритеты равны
46             else if (tasks_array_list[i]->priority == tasks_array_list[j]->priority)
47             {
48                 // Сортируем массив процессов в порядке возрастания времени их выполнения на ЦПУ
49                 if (tasks_array_list[i]->burst > tasks_array_list[j]->burst)
50                 {
51                     temp_task = tasks_array_list[i];
52                     tasks_array_list[i] = tasks_array_list[j];
53                     tasks_array_list[j] = temp_task;
54                 }
55             }
56         }
57     }
58
59     tasks_waiting_time[0] = 0;
60
61     // Перебираем весь массив процессов
62     for (int task_index = 0; task_index < current_task_index; ++task_index)
63     {
64         // Для первого процесса время ожидания равно нулю (строка 59)
65         if (task_index != 0)
66         {
67             // Время ожидания текущего процесса - это время ожидания всех процессов до него + время выполнения предыдущего процесса
68             tasks_waiting_time[task_index] = tasks_waiting_time[task_index - 1] + tasks_array_list[task_index - 1]->burst;
69             average_waiting_time += tasks_waiting_time[task_index];
70         }
71
72         run(tasks_array_list[task_index], tasks_array_list[task_index]->burst);
73
74         // Обратное время текущего процесса - это его время ожидания + его время выполнения
75         tasks_turnaround_time[task_index] = tasks_waiting_time[task_index] + tasks_array_list[task_index]->burst;
76         average_turnaround_time += tasks_turnaround_time[task_index];
77     }
78
79     printf("\nAverage waiting time = average response time = %.3f.\n", average_waiting_time / (float)current_task_index);
80     printf("Average turnaround time = %.3f.\n", average_turnaround_time / (float)current_task_index);
81 }
82
```

Рисунок 8. Фрагмент кода №8

Реализация и описание основной функции `schedule()` RR-планировщика (Round Robin), при котором каждая задача выполняется в течение определённого кванта времени (константная переменная `QUANTUM = 10` в заголовочном файле `CPU.h`) или оставшуюся часть времени использования виртуального центрального процессора приведены на Рисунке 9 ниже.

```
C schedule_rr.c X
C schedule_rr.c > ...
30 void schedule()
31 {
32     // Переворачиваем односвязный список процессов, так как каждый новый процесс добавлялся в его начало
33     tasks_linked_list = reverse(tasks_linked_list);
34     struct node *current_node = tasks_linked_list;
35
36     int task_index = 0, completed_tasks_count = 0, previous_task_burst_time;
37     tasks_waiting_time[task_index] = 0;
38
39     while (current_node != NULL)
40     {
41         if (completed_tasks_count != tasks_number)
42         {
43             if (current_node->task != NULL)
44             {
45                 // Если процесс уже был запущен на выполнение виртуальным центральным процессором, значит время ожидания для него посчитано
46                 if (task_index != 0 && !current_node->task->is_running)
47                 {
48                     current_node->task->is_running = true;
49
50                     // Время ожидания текущего процесса - это время ожидания всех процессов до него + время выполнения предыдущего процесса
51                     tasks_waiting_time[task_index] = tasks_waiting_time[task_index - 1] + previous_task_burst_time;
52                     average_waiting_time += tasks_waiting_time[task_index];
53                 }
54
55                 // Если время выполнения процесса больше 0 и больше или равно длине кванта времени (10)
56                 if (current_node->task->burst > 0 && current_node->task->burst >= QUANTUM)
57                 {
58                     run(current_node->task, QUANTUM); // Выполняем его в течение кванта времени
59                     previous_task_burst_time = QUANTUM;
60                     current_node->task->burst -= QUANTUM;
61                 }
62
63                 // Иначе, если время выполнения процесса меньше длины кванта времени (10) и не равно 0
64                 else if (current_node->task->burst < QUANTUM && current_node->task->burst != 0)
65                 {
66                     run(current_node->task, current_node->task->burst); // Довыполняем его в течение оставшегося времени
67                     previous_task_burst_time = current_node->task->burst;
68                     current_node->task->burst = 0;
69                 }
70
71                 // Иначе, процесс считается выполненным
72                 else
73                 {
74                     ++completed_tasks_count;
75                     previous_task_burst_time = 0;
76
77                     // Обратное время текущего процесса - это его время ожидания + его время выполнения
78                     tasks_turnaround_time[task_index] = tasks_waiting_time[task_index] + current_node->task->burst;
79                     average_turnaround_time += tasks_turnaround_time[task_index];
80                 }
81
82                 ++task_index;
83
84                 // Берём следующий процесс из списка
85                 current_node = current_node->next_task;
86
87                 if (current_node == NULL)
88                 {
89                     task_index = 0;
90                     current_node = tasks_linked_list;
91                 }
92             }
93             else
94             {
95                 break;
96             }
97         }
98
99         printf("\nAverage waiting time = average response time = %.3f.\n", average_waiting_time / (float)tasks_number);
100         printf("Average turnaround time = %.3f.\n", average_turnaround_time / (float)tasks_number);
101     }
}
```

Рисунок 9. Фрагмент кода №9

Последний из реализованных в данной лабораторной работе алгоритмов планирования – это RR-планировщик с приоритетом, при котором задачи планируются на основе их приоритета, а для задач с одинаковым приоритетом используется циклическое (RR) планирование.

Таким образом, этот тип планировщика совмещает в себе два других алгоритма планирования описанных выше – приоритетное (Рисунок 8) и циклическое обычное (Рисунок 9). Его же более подробное описание можно наблюдать в комментариях к программному коду на Рисунках 10-11 ниже.

```
C schedule_priority_rr.c X
C schedule_priority_rr.c > ...
31
32 void schedule()
33 {
34     struct task *temp_task;
35
36     int tasks_array_list_size = current_task_index, completed_tasks_count = 0, previous_task_burst_time = 0;
37     tasks_waiting_time[0] = 0;
38
39     for (int i = 0; i < tasks_array_list_size; ++i)
40     {
41         for (int j = i + 1; j < tasks_array_list_size; ++j)
42         {
43             // Сортируем массив процессов в порядке убывания их приоритета
44             if (tasks_array_list[i]->priority < tasks_array_list[j]->priority)
45             {
46                 temp_task = tasks_array_list[i];
47                 tasks_array_list[i] = tasks_array_list[j];
48                 tasks_array_list[j] = temp_task;
49             }
50             // Если приоритеты равны
51             else if (tasks_array_list[i]->priority == tasks_array_list[j]->priority)
52             {
53                 // Сортируем массив процессов в порядке возрастания времени их выполнения на ЦПУ
54                 if (tasks_array_list[i]->burst > tasks_array_list[j]->burst)
55                 {
56                     temp_task = tasks_array_list[i];
57                     tasks_array_list[i] = tasks_array_list[j];
58                     tasks_array_list[j] = temp_task;
59                 }
60             }
61         }
62     }
63
64     // Для каждого процесса подсчитываем количество процессов с таким же приоритетом как и у исходного
65     for (int i = 0; i < tasks_array_list_size; ++i)
66     {
67         for (int j = i + 1; j < tasks_array_list_size; ++j)
68         {
69             if (tasks_array_list[i]->priority == tasks_array_list[j]->priority)
70                 ++tasks_array_list[i]->same_priority_tasks_count;
71         }
72     }
73 }
```

Рисунок 10. Фрагмент кода №10

```

if (completed_tasks_count != tasks_array_list_size)
{
    // Если в нём есть несколько процессов с равным приоритетом
    if (tasks_array_list[i]->same_priority_tasks_count != 0)
    {
        int same_priority_tasks_count = tasks_array_list[i]->same_priority_tasks_count;
        int task_index = i;

        while (tasks_array_list[i]->same_priority_tasks_count != 0)
        {
            // Если процесс уже был запущен на выполнение виртуальным центральным процессором, значит время ожидания для него посчитано
            if (task_index != 0 && !tasks_array_list[task_index]->is_running)
            {
                tasks_array_list[task_index]->is_running = true;

                // Время ожидания текущего процесса - это время ожидания всех процессов до него + время выполнения предыдущего процесса
                tasks_waiting_time[task_index] = tasks_waiting_time[task_index - 1] + previous_task_burst_time;
                average_waiting_time += tasks_waiting_time[task_index];

                previous_task_burst_time = 0;
            }

            // Если время выполнения процесса больше 0 и больше или равно длине кванта времени (10)
            if (tasks_array_list[task_index]->burst > 0 && tasks_array_list[task_index]->burst >= QUANTUM)
            {
                run(tasks_array_list[task_index], QUANTUM); // Выполняем его в течение кванта времени
                previous_task_burst_time += QUANTUM;
                tasks_array_list[task_index]->burst -= QUANTUM;
            }

            // Иначе, если время выполнения процесса меньше длины кванта времени (10) и не равно 0
            else if (tasks_array_list[task_index]->burst < QUANTUM && tasks_array_list[task_index]->burst != 0)
            {
                run(tasks_array_list[task_index], tasks_array_list[task_index]->burst); // Довыполняем его в течение оставшегося времени
                previous_task_burst_time += tasks_array_list[task_index]->burst;
                tasks_array_list[task_index]->burst = 0;
            }

            // Иначе, процесс считается выполненным
            else
            {
                ++completed_tasks_count;

                // Обратное время текущего процесса - это его время ожидания + его время выполнения
                tasks_turnaround_time[task_index] = tasks_waiting_time[task_index] + previous_task_burst_time;
                average_turnaround_time += tasks_turnaround_time[task_index];

                --tasks_array_list[i]->same_priority_tasks_count;
            }

            ++task_index;

            if (task_index == i + same_priority_tasks_count + 1)
                task_index = i;
        }

        i += same_priority_tasks_count;
    }
    // Иначе, если только текущий процесс из списка имеет такой приоритет
    else
    {
        if (i != 0 && !tasks_array_list[i]->is_running)
        {
            tasks_array_list[i]->is_running = true;

            tasks_waiting_time[i] = tasks_waiting_time[i - 1] + previous_task_burst_time;
            average_waiting_time += tasks_waiting_time[i];

            previous_task_burst_time = 0;
        }

        run(tasks_array_list[i], tasks_array_list[i]->burst); // Выполняем процесс полностью
        previous_task_burst_time += tasks_array_list[i]->burst;
        tasks_array_list[i]->burst = 0;

        ++completed_tasks_count;

        tasks_turnaround_time[i] = tasks_waiting_time[i] + previous_task_burst_time;
        average_turnaround_time += tasks_turnaround_time[i];
    }
}

```

Рисунок 11. Фрагмент кода №11

Тестирование

```
ayrtom@asus-m15021a:~/Университет/Операционные системы/Лабораторные работы/2/framework$ ./fcfs schedule.txt
Running task = [T1] [4] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T8] [10] [25] for 25 units.

Average waiting time = average response time = 73.125.
Average turnaround time = 94.375.
```

Рисунок 12. FCFS-планировщик

```
ayrtom@asus-m15021a:~/Университет/Операционные системы/Лабораторные работы/2/framework$ ./sjf schedule.txt
Running task = [T6] [1] [10] for 10 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T8] [10] [25] for 25 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.

Average waiting time = average response time = 61.250.
Average turnaround time = 82.500.
```

Рисунок 13. SJF-планировщик

```
ayrtom@asus-m15021a:~/Университет/Операционные системы/Лабораторные работы/2/framework$ ./priority schedule.txt
Running task = [T8] [10] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T6] [1] [10] for 10 units.

Average waiting time = average response time = 75.000.
Average turnaround time = 96.250.
```

Рисунок 14. Приоритетный планировщик

```
ayrtom@asus-m1502la:~/Университет/Операционные системы/Лабораторные работы/2/Framework$ ./rr schedule.txt
Running task = [T1] [4] [20] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T8] [10] [25] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T8] [10] [15] for 10 units.
Running task = [T2] [3] [5] for 5 units.
Running task = [T3] [3] [5] for 5 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T8] [10] [5] for 5 units.

Average waiting time = average response time = 35.000.
Average turnaround time = 123.125.
```

Рисунок 15. RR-планировщик

```
ayrtom@asus-m1502la:~/Университет/Операционные системы/Лабораторные работы/2/Framework$ ./priority_rr schedule.txt
Running task = [T8] [10] [25] for 25 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T3] [3] [5] for 5 units.
Running task = [T2] [3] [5] for 5 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T6] [1] [10] for 10 units.

Average waiting time = average response time = 68.750.
Average turnaround time = 76.875.
```

Рисунок 16. RR-планировщик с приоритетом