

# Proj4 Scheduling Algorithms

---

by 郑航 520021911347

## Proj4 Scheduling Algorithms

- 1 Introduction
  - 1.1 Objectives
  - 1.2 Environment
- 2 Code Frame
- 3 Requirement
  - 3.1 Fundamental Requirement
  - 3.2 Further Challenges
- 4 Detailed Solution
  - 4.1 solution for further challenges
  - 4.2 schedule\_fcfs.c
    - 4.2.1 Global variables
    - 4.2.2 add()
    - 4.2.3 schedule()
    - 4.2.4 Result
  - 4.3 schedule\_sjf.c
    - 4.3.1 Global variables & add()
    - 4.3.2 schedule()
    - 4.3.3 Full Implementation
    - 4.3.4 Result
  - 4.4 schedule\_rr.c
    - 4.4.1 Global variables & add()
    - 4.4.2 schedule()
    - 4.4.3 Full Implementation
    - 4.4.4 Result
  - 4.5 schedule\_priority.c
    - 4.5.1 Global variables & add()
    - 4.5.2 schedule()
    - 4.5.3 Full Implementation
    - 4.5.4 Result
  - 4.6 schedule\_priority\_rr.c
    - 4.6.1 Global variables & add()
    - 4.6.2 schedule()
    - 4.6.3 Full Implementation
    - 4.6.4 Result
- 5 Summary

## 1 Introduction

---

## 1.1 Objectives

- 初步了解UNIX shell的运作，并实现一个简单的shell
- 进一步学习内核编程，实现一个内核模块，可根据pid查询并返回对应进程的信息

## 1.2 Environment

- win10下的VMware Workstation Pro中运行的Ubuntu18.04

## 2 Code Frame

---

本次Project我选择使用**C语言**进行编程，提供的文件数量较多，先在此整理下整个代码的逻辑框架

- driver.c是main函数所在的文件，负责对命令行参数加以处理，读入目标schedule的txt文件，并将其中各个task通过add函数加入到等待队列中，最终调用scheduler函数执行调度
- CPU.c和cpu.h声明并定义了run函数，用于模拟进程调度的过程
- list.c 和list.h是对等待队列的声明和实现，实现了insert、delete和traverse三个函数
- task.h声明了一个结构体task并取别名为Task，是对调度任务的一个抽象，包含name、tid、priority和cpu burst四个数据成员
- schedulers.h声明了add和schedule函数
- 三个txt文件是各调度程序的测试文件，可以在调用具体调度程序时作为参数传入main并读取

需要完成的五个scheduler程序，分别是对五种调度策略的实现

## 3 Requirement

---

### 3.1 Fundamental Requirement

实现五个scheduler程序：

- schedule\_fcfs.c：实现fcfs，即First-come, first-served 的调度算法
- schedule\_sjf.c：实现sjf，即Shortest-job-first 的调度算法
- schedule\_rr.c：实现rr，即Round-robin 的调度算法
- schedule\_priority.c：实现根据Priority 进行调度的调度算法
- schedule\_priority\_rr.c：实现Priority with round-robin，即考虑优先级的RR调度算法

需要在这五个程序中，分别实现add和schedule函数，具体细节后面进行阐述

### 3.2 Further Challenges

- 为了适应SMP环境，防止可能的race condition，将对tid的赋值和递增操作原子化
- 在每个调度算法中，计算其平均周转时间、平均等待时间和平均响应时间

## 4 Detailed Solution

---

## 4.1 solution for further challenges

- 将tid的赋值和递增操作原子化，只需要使用课本介绍的\_\_sync\_fetch\_and\_add函数即可
- 计算平均周转时间、平均等待时间和平均响应时间，一开始的想法是设置一个全局的clock，每个task执行完毕时clock的值即为其周转时间，减去其初始burst时间即为其waiting时间，但这种方式只能完成如所给测试文件一般，所有的task同时到来的情况。针对更一般的task在任意时间到来的情况，这种方法就不可行了。因此最终决定的方法是在每个task中增加一些数据成员，用来记录其运行过程中的各种时间

修改后的task定义如下：

```
1  typedef struct task {
2      char *name;
3      int tid;
4      int priority;
5      int burst;
6
7      int init_burst;           //记录初始的burst（用于waiting time计算）
8      int arrive_time;         //到达时间
9      int finish_time;         //完成时间
10     int responce_time;        //在第一次运行后，记录响应时间
11     int been_executed;        //记录是否为第一次运行
12 } Task;
```

## 4.2 schedule\_fcfs.c

### 4.2.1 Global variables

设立三个全局变量如下，head指向等待队列；tid从0开始，为每个task赋以一个唯一的标识；clock代表时钟，模拟CPU运行的时间

```
1  struct node *head = NULL;
2  int tid = 0;
3  int clock = 0;
```

### 4.2.2 add()

主要进行新task的一系列初始化过程，并将task加入到等待队列中，注意对tid的递增需要使用\_\_sync\_fetch\_and\_add函数

```
1  void add(char *name, int priority, int burst)
2  {
3      Task *tsk = NULL;
4      tsk = (Task *)malloc(sizeof(Task));
5      tsk->name = (char *)malloc(sizeof(char) * (strlen(name) + 1));
6      strcpy(tsk->name, name);
7      tsk->priority = priority;
8      tsk->burst = burst;
9      tsk->tid = __sync_fetch_and_add(&tid, 1);
10     tsk->been_executed=0;
11     tsk->init_burst = burst;
12     tsk->arrive_time = clock;
13 }
```

```

14     insert(&head, tsk);
15 }

```

### 4.2.3 schedule()

主要分为两部分：

- 模拟执行：
  - 根据fcfs的原则，最先加入的task放在等待队列最后的一个，每次取最后一个task进行执行
  - 执行过程中不断进行task的时间相关数据成员的更新
  - 执行结束后从等待队列中删去该task，并将其各时间累加到总时间中
  - 释放该task的资源
- 统计数据的计算和输出

```

1  void schedule()
2  {
3      int total_turnaround = 0;
4      int total_wait = 0;
5      int total_responce = 0;
6      int task_count = 0;
7      while (head)
8      {
9          struct node *nodeptr = head;
10         while (nodeptr->next)           //找到最先进入队列的task
11         {
12             nodeptr = nodeptr->next;
13         }
14         Task *tsk = nodeptr->task;
15         run(tsk, tsk->burst);
16         if (tsk->been_executed == 0)     //检查是否首次被执行，更新response time
17         {
18             tsk->been_executed = 1;
19             task_count++;
20             tsk->responce_time = clock - tsk->arrive_time;
21         }
22         clock += tsk->burst;
23
24         tsk->finish_time = clock;
25         delete (&head, tsk);
26
27         int turnaround = tsk->finish_time - tsk->arrive_time;   //每个task执
行结束后，将时间累加到总时间
28         total_turnaround += turnaround;
29         total_wait += (turnaround - tsk->init_burst);
30         total_responce += tsk->responce_time;
31
32         free(tsk->name);
33         free(tsk);
34     }
35
36     // calculate the statistics
37     double aver_turnaround = ((double)total_turnaround) / task_count;
38     double aver_wait = ((double)total_wait) / task_count;
39     double aver_responce = ((double)total_responce) / task_count;
40     printf("\n");
41     printf("For the total %d tasks:\n", task_count);

```

```

42     printf("The Average Turnaround Time is:      %lf \n", aver_turnaround);
43     printf("The Average Waiting Time is:        %lf \n", aver_wait);
44     printf("The Average Responce Time is:       %lf \n", aver_responce);
45 }

```

## 4.2.4 Result

```

zh@ubuntu:~/project/pro4$ ./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.

For the total 8 tasks:
The Average Turnaround Time is:      94.375000
The Average Waiting Time is:         73.125000
The Average Responce Time is:       73.125000

```

## 4.3 schedule\_sjf.c

### 4.3.1 Global variables & add()

与4.2中完全一致，在此不加赘述

### 4.3.2 schedule()

主要也分为模拟执行和统计数据的计算输出两部分，与4.2.3中类似，唯一不同的是挑选下一个执行的task的方式不同

根据根据sjf的原则，首先执行burst时间最短的task，故每次遍历等待队列，取出其中burst最小的一个task加入执行，具体代码如下：

```

1  struct node *nodeptr = head, *resptr = nodeptr;
2  while (nodeptr)      //找到burst最小的task
3  {
4      if (resptr->task->burst >= nodeptr->task->burst)    //可以保证，若burst相
        同，则按fcfs处理
5          resptr = nodeptr;
6          nodeptr = nodeptr->next;
7  }
8  Task *tsk = resptr->task;

```

### 4.3.3 Full Implementation

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "task.h"
6  #include "list.h"
7  #include "cpu.h"
8  #include "schedulers.h"

```

```

9
10 struct node *head = NULL;
11 int tid = 0;
12 int clock = 0;
13
14 void add(char *name, int priority, int burst)
15 {
16     Task *tsk = NULL;
17     tsk = (Task *)malloc(sizeof(Task));
18     tsk->name = (char *)malloc(sizeof(char) * (strlen(name) + 1));
19     strcpy(tsk->name, name);
20     tsk->priority = priority;
21     tsk->burst = burst;
22     tsk->tid = __sync_fetch_and_add(&tid, 1);
23     tsk->been_executed = 0;
24     tsk->init_burst = burst;
25     tsk->arrive_time = clock;
26
27     insert(&head, tsk);
28 }
29
30 void schedule()
31 {
32     int total_turnaround = 0;
33     int total_wait = 0;
34     int total_responce = 0;
35     int task_count = 0;
36     while (head)
37     {
38         struct node *nodeptr = head, *resptr = nodeptr;
39         while (nodeptr) //找到burst最小的task
40         {
41             if (resptr->task->burst >= nodeptr->task->burst) //可以保证，若
burst相同，则按fcfs处理
42                 resptr = nodeptr;
43             nodeptr = nodeptr->next;
44         }
45         Task *tsk = resptr->task;
46         run(tsk, tsk->burst);
47         if (tsk->been_executed == 0) //检查是否首次被执行，更新response time
48         {
49             tsk->been_executed = 1;
50             task_count++;
51             tsk->responce_time = clock - tsk->arrive_time;
52         }
53         clock += tsk->burst;
54
55         tsk->finish_time = clock;
56         delete (&head, tsk);
57
58         int turnaround = tsk->finish_time - tsk->arrive_time; //每个task执行
结束后，将时间累加到总时间
59         total_turnaround += turnaround;
60         total_wait += (turnaround - tsk->init_burst);
61         total_responce += tsk->responce_time;
62
63         free(tsk->name);
64         free(tsk);

```

```

65     }
66
67     // calculate the statistics
68     double aver_turnaround = ((double)total_turnaround) / task_count;
69     double aver_wait = ((double)total_wait) / task_count;
70     double aver_responce = ((double)total_responce) / task_count;
71     printf("\n");
72     printf("For the total %d tasks:\n", task_count);
73     printf("The Average Turnaround Time is:      %lf \n", aver_turnaround);
74     printf("The Average Waiting Time is:         %lf \n", aver_wait);
75     printf("The Average Responce Time is:         %lf \n", aver_responce);
76 }

```

#### 4.3.4 Result

```

zh@ubuntu:~/project/pro4$ make sjf
gcc -Wall -o sjf driver.o schedule_sjf.o list.o CPU.o
zh@ubuntu:~/project/pro4$ ./sjf schedule.txt
Running task = [T6] [1] [10] for 10 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T8] [10] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.

For the total 8 tasks:
The Average Turnaround Time is:      82.500000
The Average Waiting Time is:         61.250000
The Average Responce Time is:         61.250000

```

### 4.4 schedule\_rr.c

#### 4.4.1 Global variables & add()

与4.2和4.3中完全一致，在此不加赘述

#### 4.4.2 schedule()

主要也分为模拟执行和统计数据的计算输出两部分，与4.3.2中类似

根据rr的原则，每次挑选当前等待队列最前边（即list中最后一个）的task进行执行，执行一段最多为QUANTUM（本例中为10）的时间片。故每次执行前需要对task的剩余burst时间进行条件判断，若task的剩余burst时间大于一个QUANTUM时间，则执行QUANTUM时间后将其重新放入等待队列的末尾；若不足一个QUANTUM则执行剩余时间后将该task释放，并可以开始计算该task相关时间并累加到总时间中，具体代码如下：

```

1  if (tsk->burst > QUANTUM)           //判断一个时间片是否可以完成该task
2  {
3      clock += QUANTUM;
4      tsk->burst -= QUANTUM;
5      run(tsk, QUANTUM);
6      delete(&head, tsk);
7      insert(&head, tsk);
8      continue;
9  }

```

```

10 else
11 {
12     run(tsk, tsk->burst);
13     clock += tsk->burst;
14
15     tsk->finish_time = clock;
16     delete (&head, tsk);
17
18     int turnaround = tsk->finish_time - tsk->arrive_time; //每个task执行结束后，将时间累加到总时间
19     total_turnaround += turnaround;
20     total_wait += (turnaround - tsk->init_burst);
21     total_responce += tsk->responce_time;
22
23     free(tsk->name);
24     free(tsk);
25 }

```

### 4.4.3 Full Implementation

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "task.h"
6  #include "list.h"
7  #include "cpu.h"
8  #include "schedulers.h"
9
10 struct node *head = NULL;
11 int tid = 0;
12 int clock = 0;
13
14 void add(char *name, int priority, int burst)
15 {
16     Task *tsk = NULL;
17     tsk = (Task *)malloc(sizeof(Task));
18     tsk->name = (char *)malloc(sizeof(char) * (strlen(name) + 1));
19     strcpy(tsk->name, name);
20     tsk->priority = priority;
21     tsk->burst = burst;
22     tsk->tid = __sync_fetch_and_add(&tid, 1);
23     tsk->been_executed = 0;
24     tsk->init_burst = burst;
25     tsk->arrive_time = clock;
26
27     insert(&head, tsk);
28 }
29
30 void schedule()
31 {
32     int total_turnaround = 0;
33     int total_wait = 0;
34     int total_responce = 0;
35     int task_count = 0;
36     while (head)
37     {

```



```

38     struct node *nodeptr = head;
39     while (nodeptr->next) //找到最先进入队列的task
40     {
41         nodeptr = nodeptr->next;
42     }
43     Task *tsk = nodeptr->task;
44
45     if (tsk->been_executed == 0) //检查是否首次被执行, 更新response time
46     {
47         tsk->been_executed = 1;
48         task_count++;
49         tsk->responce_time = clock - tsk->arrive_time;
50     }
51
52     if (tsk->burst > QUANTUM) //判断一个时间片是否可以完成该task
53     {
54         clock += QUANTUM;
55         tsk->burst -= QUANTUM;
56         run(tsk, QUANTUM);
57         delete(&head, tsk);
58         insert(&head, tsk);
59         continue;
60     }
61     else
62     {
63         run(tsk, tsk->burst);
64         clock += tsk->burst;
65
66         tsk->finish_time = clock;
67         delete (&head, tsk);
68
69         int turnaround = tsk->finish_time - tsk->arrive_time; //每个task
        执行结束后, 将时间累加到总时间
70         total_turnaround += turnaround;
71         total_wait += (turnaround - tsk->init_burst);
72         total_responce += tsk->responce_time;
73
74         free(tsk->name);
75         free(tsk);
76     }
77 }
78
79 // calculate the statistics
80 double aver_turnaround = ((double)total_turnaround) / task_count;
81 double aver_wait = ((double)total_wait) / task_count;
82 double aver_responce = ((double)total_responce) / task_count;
83 printf("\n");
84 printf("For the total %d tasks:\n", task_count);
85 printf("The Average Turnaround Time is:      %lf \n", aver_turnaround);
86 printf("The Average Waiting Time is:          %lf \n", aver_wait);
87 printf("The Average Responce Time is:         %lf \n", aver_responce);
88 }

```

#### 4.4.4 Result

```
zh@ubuntu:~/project/pro4$ make rr
gcc -Wall -c schedule_rr.c
gcc -Wall -o rr driver.o schedule_rr.o list.o CPU.o
zh@ubuntu:~/project/pro4$ ./rr schedule.txt
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 10 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T8] [10] [15] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T2] [3] [5] for 10 units.
Running task = [T3] [3] [5] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T8] [10] [5] 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.

For the total 8 tasks:
The Average Turnaround Time is:      128.750000
The Average Waiting Time is:        107.500000
The Average Response Time is:       35.000000
```

### 4.5 schedule\_priority.c

#### 4.5.1 Global variables & add()

与4.2和4.3中完全一致，在此不加赘述

#### 4.5.2 schedule()

主要也分为模拟执行和统计数据的计算输出两部分，与4.3.2中类似，唯一不同的是挑选下一个执行的task的方式不同

根据根据priority优先的原则，首先执行priority最大的task，故每次遍历等待队列，取出其中priority最大的一个task加入执行，具体代码如下：

```
1 struct node *nodeptr = head, *resptr = nodeptr;
2 while (nodeptr) //找到priority最大的task
3 {
4     if (resptr->task->priority <= nodeptr->task->priority) //可以保证，若
        priority相同，则按fcfs处理
5         resptr = nodeptr;
6         nodeptr = nodeptr->next;
7 }
8 Task *tsk = resptr->task;
```

### 4.5.3 Full Implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "task.h"
6  #include "list.h"
7  #include "cpu.h"
8  #include "schedulers.h"
9
10 struct node *head = NULL;
11 int tid = 0;
12 int clock = 0;
13
14 void add(char *name, int priority, int burst)
15 {
16     Task *tsk = NULL;
17     tsk = (Task *)malloc(sizeof(Task));
18     tsk->name = (char *)malloc(sizeof(char) * (strlen(name) + 1));
19     strcpy(tsk->name, name);
20     tsk->priority = priority;
21     tsk->burst = burst;
22     tsk->tid = __sync_fetch_and_add(&tid, 1);
23     tsk->been_executed = 0;
24     tsk->init_burst = burst;
25     tsk->arrive_time = clock;
26
27     insert(&head, tsk);
28 }
29
30 void schedule()
31 {
32     int total_turnaround = 0;
33     int total_wait = 0;
34     int total_responce = 0;
35     int task_count = 0;
36     while (head)
37     {
38         struct node *nodeptr = head, *resptr = nodeptr;
39         while (nodeptr) //找到priority最大的task
40         {
41             if (resptr->task->priority <= nodeptr->task->priority) //可以保证, 若priority相同, 则按fcfs处理
42                 resptr = nodeptr;
43             nodeptr = nodeptr->next;
44         }
45         Task *tsk = resptr->task;
46         run(tsk, tsk->burst);
47         if (tsk->been_executed == 0) //检查是否首次被执行, 更新response time
48         {
49             tsk->been_executed = 1;
50             task_count++;
51             tsk->responce_time = clock - tsk->arrive_time;
52         }
53         clock += tsk->burst;
54     }
```

```

55     tsk->finish_time = clock;
56     delete (&head, tsk);
57
58     int turnaround = tsk->finish_time - tsk->arrive_time; //每个task执行
结束后, 将时间累加到总时间
59     total_turnaround += turnaround;
60     total_wait += (turnaround - tsk->init_burst);
61     total_responce += tsk->responce_time;
62
63     free(tsk->name);
64     free(tsk);
65 }
66
67 // calculate the statistics
68 double aver_turnaround = ((double)total_turnaround) / task_count;
69 double aver_wait = ((double)total_wait) / task_count;
70 double aver_responce = ((double)total_responce) / task_count;
71 printf("\n");
72 printf("For the total %d tasks:\n", task_count);
73 printf("The Average Turnaround Time is:      %lf \n", aver_turnaround);
74 printf("The Average Waiting Time is:         %lf \n", aver_wait);
75 printf("The Average Response Time is:        %lf \n", aver_responce);
76 }

```

## 4.5.4 Result

```

zh@ubuntu:~/project/pro4$ make priority
gcc -Wall -c schedule_priority.c
gcc -Wall -o priority driver.o schedule_priority.o list.o CPU.o
zh@ubuntu:~/project/pro4$ ./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 = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T6] [1] [10] for 10 units.

For the total 8 tasks:
The Average Turnaround Time is:      96.250000
The Average Waiting Time is:         75.000000
The Average Response Time is:        75.000000

```

## 4.6 schedule\_priority\_rr.c

### 4.6.1 Global variables & add()

与4.2和4.3中完全一致, 在此不加赘述

### 4.6.2 schedule()

主要也分为模拟执行和统计数据的计算输出两部分, 与4.4中利用rr的调度算法的程序基本类似, 唯一不同的是挑选下一个执行的task时需要优先考虑priority较高的task而不是按照fcfs原则

故只需要在schedule\_rr.c的基础上, 将选择下一个执行task的部分改为与schedule\_priority.c相同即可

### 4.6.3 Full Implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include "task.h"
6  #include "list.h"
7  #include "cpu.h"
8  #include "schedulers.h"
9
10 struct node *head = NULL;
11 int tid = 0;
12 int clock = 0;
13
14 void add(char *name, int priority, int burst)
15 {
16     Task *tsk = NULL;
17     tsk = (Task *)malloc(sizeof(Task));
18     tsk->name = (char *)malloc(sizeof(char) * (strlen(name) + 1));
19     strcpy(tsk->name, name);
20     tsk->priority = priority;
21     tsk->burst = burst;
22     tsk->tid = __sync_fetch_and_add(&tid, 1);
23     tsk->been_executed = 0;
24     tsk->init_burst = burst;
25     tsk->arrive_time = clock;
26
27     insert(&head, tsk);
28 }
29
30 void schedule()
31 {
32     int total_turnaround = 0;
33     int total_wait = 0;
34     int total_responce = 0;
35     int task_count = 0;
36     while (head)
37     {
38         struct node *nodeptr = head, *resptr = nodeptr;
39         while (nodeptr) //找到priority最大的task
40         {
41             if (resptr->task->priority <= nodeptr->task->priority) //可以保证, 若priority相同, 则按fcfs处理
42                 resptr = nodeptr;
43             nodeptr = nodeptr->next;
44         }
45         Task *tsk = resptr->task;
46
47         if (tsk->been_executed == 0) //检查是否首次被执行, 更新response time
48         {
49             tsk->been_executed = 1;
50             task_count++;
51             tsk->responce_time = clock - tsk->arrive_time;
52         }
53
54         if (tsk->burst > QUANTUM) //判断一个时间片是否可以完成该task
```

```

55     {
56         clock += QUANTUM;
57         tsk->burst -= QUANTUM;
58         run(tsk, QUANTUM);
59         delete (&head, tsk);
60         insert(&head, tsk);
61         continue;
62     }
63     else
64     {
65         run(tsk, tsk->burst);
66         clock += tsk->burst;
67
68         tsk->finish_time = clock;
69         delete (&head, tsk);
70
71         int turnaround = tsk->finish_time - tsk->arrive_time; //每个task
执行结束后，将时间累加到总时间
72         total_turnaround += turnaround;
73         total_wait += (turnaround - tsk->init_burst);
74         total_responce += tsk->responce_time;
75
76         free(tsk->name);
77         free(tsk);
78     }
79 }
80
81 // calculate the statistics
82 double aver_turnaround = ((double)total_turnaround) / task_count;
83 double aver_wait = ((double)total_wait) / task_count;
84 double aver_responce = ((double)total_responce) / task_count;
85 printf("\n");
86 printf("For the total %d tasks:\n", task_count);
87 printf("The Average Turnaround Time is:      %lf \n", aver_turnaround);
88 printf("The Average waiting Time is:          %lf \n", aver_wait);
89 printf("The Average Responce Time is:          %lf \n", aver_responce);
90 }

```

#### 4.6.4 Result

```

zh@ubuntu:~/project/pro4$ make priority_rr
gcc -Wall -c -o schedule_priority_rr.o schedule_priority_rr.c
gcc -Wall -o priority_rr driver.o schedule_priority_rr.o list.o CPU.o
zh@ubuntu:~/project/pro4$ ./priority_rr schedule.txt
Running task = [T8] [10] [15] for 10 units.
Running task = [T8] [10] [5] for 10 units.
Running task = [T8] [10] [5] for 5 units.
Running task = [T4] [5] [5] for 10 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T1] [4] [10] 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 = [T7] [3] [20] for 10 units.
Running task = [T2] [3] [5] for 10 units.
Running task = [T3] [3] [5] for 10 units.
Running task = [T7] [3] [10] 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 = [T6] [1] [10] for 10 units.

For the total 8 tasks:
The Average Turnaround Time is:      105.000000
The Average Waiting Time is:         83.750000
The Average Response Time is:        68.750000

```

## 5 Summary

本次project的收获:

- 对课内有关调度算法的理论知识加以实践, 使我对调度算法有了更深刻更具体的认识
- 通过对平均周转时间、平均等待时间以及平均响应时间的具体观察, 可以验证一下课内的结论, 如: SJF的平均等待时间最短, 而一般来说rr的平均响应时间是最短的等等

其他:

- 本次实现等待队列的数据结构是普通的list, 且由于最先加入的task会位于list的最末位, 因此本次project中多了很多遍历list才能找到目标task的行为, 增加了不必要的开销, 是可以通过换用queue或提前处理list中的task顺序等来加以改进的。考虑到本次案例的task数较少以及代码已实现了list, 且实现该部分内容会使得代码结构不那么清晰, 因此没有加以实现