

## 0.1 Project 4 调度算法（完整版含附加题）

- 姓名：Musicminon
- 版本：2022-4-26（最终完整版，含代码和附加题）

### 0.1 Project 4 调度算法（完整版含附加题）

#### 0.1.1 一、介绍

#### 0.1.2 二、输入格式

#### 0.1.3 三、三个基础类与运行逻辑

##### 0.1.3.1 1、Task的作用

##### 0.1.3.2 2、CPU的作用

##### 0.1.3.3 3、Driver的作用

##### 0.1.3.4 4、程序的逻辑

#### 0.1.4 四、FCFS 先到先服务

##### 0.1.4.1 1、链表增元

##### 0.1.4.2 2、翻转链表

##### 0.1.4.3 3、遍历链表

#### 0.1.5 五、SJF 短任务优先

##### 0.1.5.1 一、增加元素

##### 0.1.5.2 二、调度函数

##### 0.1.5.3 三、特别注意

#### 0.1.6 六、优先级调度

##### 0.1.6.1 一、增添元素

##### 0.1.6.2 二、调度执行

#### 0.1.7 七、轮转调度

##### 0.1.7.1 一、实现说明

##### 0.1.7.2 二、相关代码

#### 0.1.8 八、优先级-轮转调度

##### 0.1.8.1 一、原理介绍

##### 0.1.8.2 二、代码演示

#### 0.1.9 九、附加挑战题

##### 0.1.9.1 一、附加题（1）

##### 0.1.9.2 二、附加题（2）

###### 0.1.9.2.1 a) 概念复习

###### 0.1.9.2.2 b) FCFS调度算法 时间计算

###### 0.1.9.2.3 c) SJF调度算法 时间计算

###### 0.1.9.2.4 d) 优先级调度 时间计算

###### 0.1.9.2.5 e) RR调度 时间计算（难）

###### 0.1.9.2.6 f) 优先级-RR调度 时间计算

#### 0.1.10 十、归纳总结

#### 0.1.11 十一、效果演示图

### 0.1.1 一、介绍

该项目涉及实现几种不同的进程调度算法。调度程序将被分配一组预定义的任务，并将根据选定的调度算法调度任务。每个任务都被分配了一个优先级和 CPU 突发。将实现以下调度算法：

- 先到先服务 (FCFS)，它按照任务请求 CPU 的顺序调度任务。
- 短任务优先 (SJF)，按照任务的长度来安排任务任务的下一个 CPU 爆发。
- 优先级调度，根据优先级调度任务。
- 循环 (RR) 调度，其中每个任务运行一个时间量（或其 CPU 突发的剩余部分）。
- Priority 和 round-robin，按优先级顺序调度任务，对同等优先级的任务使用轮询调度

### 0.1.2 二、输入格式

给出的任务调度的格式为 `[taskname][priority][CPUburst]`，示例格式如下（教材）：

```
1 | T1, 4, 20
2 | T2, 2, 25
3 | T3, 3, 25
4 | T4, 3, 15
5 | T5, 10, 1
```

- 因此，任务 T1 具有优先级 4 和 20 毫秒的 CPU 突发，依此类推。假设所有任务同时到达，因此我们的调度程序算法不必支持高优先级进程抢占低优先级进程。此外，任务不必以任何特定顺序放入队列或列表中。
- 如第 5.1.2 节中首次介绍的那样，有几种不同的策略来组织任务列表。一种方法是将所有任务放在一个无序单链表中，其中任务选择的策略取决于调度算法。
- 例如，SJF 调度将搜索列表以找到具有最短下一个 CPU 突发的任务。或者，可以根据调度标准（即，按优先级）对列表进行排序。
- 另一种策略是为每个唯一优先级设置一个单独的队列，如图 5.7 所示。这些方法将在 5.3.6 节中简要讨论。还值得强调的是，我们使用的术语列表和队列有些互换。但是，队列具有非常特定的 FIFO 功能，而列表没有如此严格的插入和删除要求。完成此项目时，您可能会发现通用链表的功能更合适。

### 0.1.3 三、三个基础类与运行逻辑

可以看到在书籍提供给我们的代码中，有下面的一些文件

```
1 | ---src
2 | |---Task.c
3 | |---Task.h
4 | |---cpu.c
5 | |---cpu.h
6 | |---driver.h
```

### 0.1.3.1 1、Task的作用

为了方便完成任务，定义这样的一个Task结构体，包含的成员是名字、ID、优先级、占用的时间。

```
1  #ifndef TASK_H
2  #define TASK_H
3  // representation of a task
4  typedef struct task {
5      char *name;
6      int tid;
7      int priority;
8      int burst;
9  } Task;
10 #endif
```

### 0.1.3.2 2、CPU的作用

在CPU的头文件中，我们定义了一个 `QUANTNUM`，这个变量用来表示在轮转调度过程中，每一个任务最大的占用的时间。

```
1  #define QUANTUM 10
```

同时，我们还定义了一个 `run` 函数，用来表示我们执行了这个任务

```
1  // run this task for the specified time slice
2  void run(Task *task, int slice) {
3      printf("Running task = [%s] [%d] [%d] for %d units.\n", task->name, task-
4      >priority, task->burst, slice);
5  }
```

### 0.1.3.3 3、Driver的作用

- `driver.c` 是模拟任务运行的部分。我们需要读完这部分的所有任务，并将它们传递给调度算法。
- 而我们实际是使用数据结构链表来存储这些任务的。更重要的是，我们使用 `main()` 的参数来传递文件名。
- 我们使用C的几个函数来实现文件的翻译：

```
1  strdup()    //复制字符串
2  strsep()    //从字符串中提取token
3  atoi()      //将字符串转换为整数
```

- 下面是这个程序的主函数，它将包含文件的读取的一整个系列的工作。

```
1  int main(int argc, char *argv[])
2  {
3      FILE *in;
```

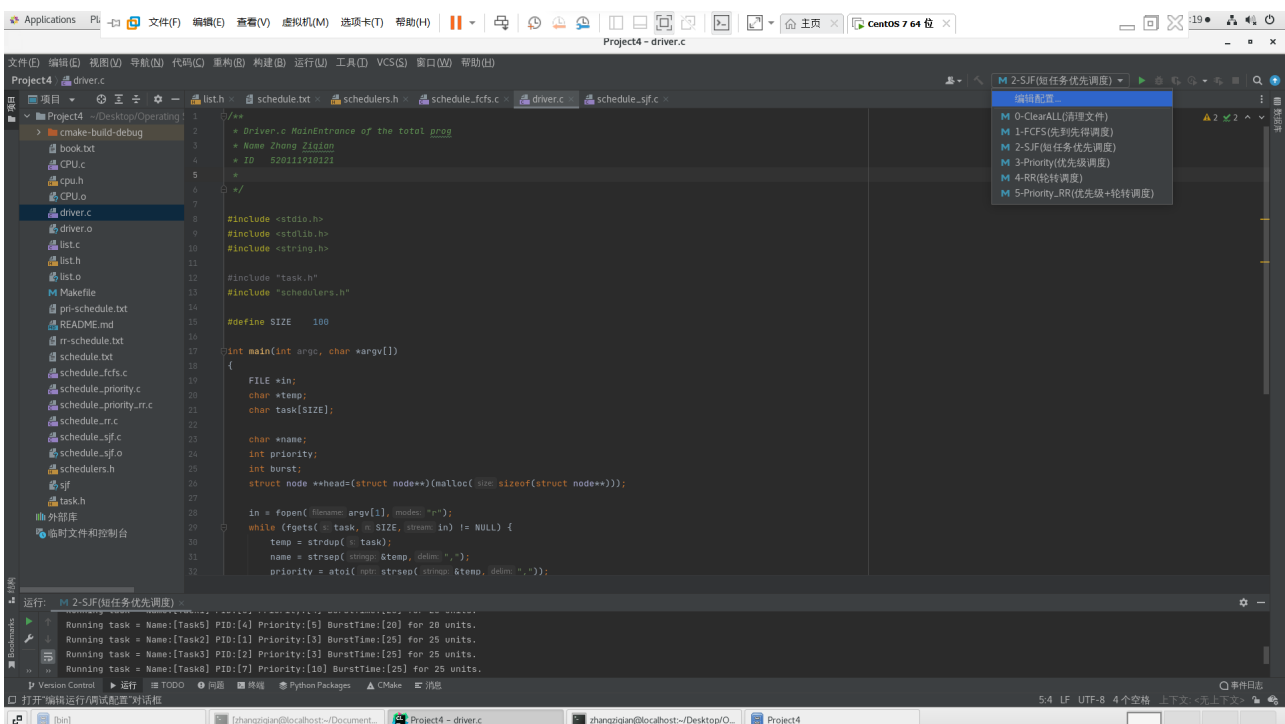
```

4     char *temp;
5     char task[SIZE];
6     char *name;
7     int priority;
8     int burst;
9     struct node **head=(struct node**) (malloc(sizeof(struct node**)));
10    in = fopen(argv[1],"r");
11    while (fgets(task,SIZE,in) != NULL) {
12        temp = strdup(task);
13        name = strtok(&temp, ",");
14        priority = atoi(strtok(&temp, ","));
15        burst = atoi(strtok(&temp, ","));
16        // add the task to the scheduler's list of tasks
17        add(name,priority,burst, head);
18        free(temp);
19    }
20    fclose(in);
21    // invoke the scheduler
22    schedule(head);
23    return 0;
24 }

```

#### 0.1.3.4 4、程序的逻辑

- 这个程序的逻辑就是，选择不同的调度的库函数，分别来采用不同的算法函数，然后按照这对应的算法函数进行任务链表的产生和计算相关的时间长度。主函数完成的仅仅是文件的读取工作。
- 例如下面是我使用CLion在CentOS上面的截图，五种配置可以随心自由的切换，直接选择对应的调度算法即可使用！当然如果是在服务器上运行，也可以采用传统的命令行，两种方法测试都可以正常输出！



#### 0.1.4 四、FCFS 先到先服务

- 这种调度方法是最简单的方法，它根据任务到达CPU的时间来进行，先请求的前响应，直到解决后，就解决下一个任务。
- 实现方法是一个队列，只需要每次有新的任务要到来的时候，就把这个任务放在这个队伍的末尾，当CPU空闲的时候，就直接拿一个队伍首的任务开始执行，直到队伍为空。
- 但是综合考虑，链表也可以很好的实现这个功能，所以选择链表。

##### 0.1.4.1 1、链表增元

- 这个函数的功能相当于给链表增加一个节点，也是和之前说的队列末尾增加元素一个道理。

```
1 void add(char *name, int priority, int burst, struct node **head) {
2     Task *t_new = (Task*)malloc(sizeof(Task*));
3     t_new->name = name;
4     t_new->priority = priority;
5     t_new->burst = burst;
6     insert(head, t_new);
7 }
```

##### 0.1.4.2 2、翻转链表

由于我们每次插入到链表中，都是插入到首部，所以我们对于这个链表需要做一次翻转，完成翻转后，顺序才能正确。

```
1 while(current_node!=NULL && current_node->next!=NULL)
2 {
3     back_node = current_node->next;
4     current_node->next = front_node;
5     front_node = current_node;
6     current_node = back_node;
7 }
```

对于最后一个的节点，需要特殊考虑，删除代码如下：

```
1 if(current_node!=NULL && current_node->next==NULL)
2     current_node->next = front_node;
3 *head = current_node;
```

### 0.1.4.3 3、遍历链表

由于已经翻转了这个链表，直接遍历一下，运行每一个任务就好。

```
1 Task *tmp_task;
2 while(current_node!=NULL)
3 {
4     tmp_task = current_node->task;
5     run(tmp_task, tmp_task->burst); // run the task
6     current_node = current_node->next;
7 }
```

### 0.1.5 五、SJF 短任务优先

- SJF调度算法是考虑进程的长度，当CPU处在空闲的状态，它会选择当然消耗时间最短执行，当然，如果两个任务消耗的时间相同，那么就会按照之前的FCFS原则
- 考虑到这次的任务是要求短任务优先，所以，我们选择在插入的时候找到适合自己的位置。
- 这样完成之后，链表有序，就可以开始执行了。

#### 0.1.5.1 一、增加元素

- 选择在插入的时候找到适合自己的位置。

```
1 void add(char *name, int priority, int burst, struct node **head){
2     Task *t_new = (Task*)malloc(sizeof(Task*));
3     t_new->name = name;
4     t_new->priority = priority;
5     t_new->burst = burst;
6
7     struct node *current_node, *front_node;
8     current_node = *head;
9     front_node = NULL;
10    if(current_node==NULL)
11    {
12        insert(head, t_new);
13        return ;
14    }
15    while(current_node!=NULL)
16    {
17        if(burst >= current_node->task->burst ){
18            front_node = current_node;
19            current_node = current_node->next;
20
21        }
22        else{
23            struct node* node_newtask = malloc(sizeof(struct node));
24            node_newtask->task = t_new;
```

```

25         node_newtask->next = current_node;
26         if(front_node==NULL){
27             *head = node_newtask;
28         }
29         else{
30             front_node->next = node_newtask;
31         }
32         return;
33     }
34 }
35 if(current_node==NULL)
36 {
37     struct node* node_newtask = malloc(sizeof(struct node));
38     node_newtask->task = t_new;
39     node_newtask->next = NULL;
40     front_node->next = node_newtask;
41     return;
42 }
43 }

```

### 0.1.5.2 二、调度函数

按照之前排好的顺序执行。

```

1 void schedule(struct node **head){
2     struct node* current_node;
3     current_node = *head;
4     while(current_node!=NULL)
5     {
6         run(current_node->task,current_node->task->burst);
7         current_node = current_node->next;
8     }
9 }

```

### 0.1.5.3 三、特别注意

- 下面的这一段代码很容易犯错，程序中通常一个等于号的遗漏或者书写，对整个程序的运行逻辑都是巨大的影响
- 因为在时间相同的时候，会按照先来的先处理，后来的后处理，代码是在添加任务的时候，直接把任务放在大小合适的位置，因为在插入的时候，后插入的肯定是后来的，后来的就要放在后面，所以在等于的时候还是要求继续一个 `while` 循环

```

1 if(burst >= current_node->task->burst )
2     // do somethings

```

### 0.1.6 六、优先级调度

- 不难发现，优先级调度和短任务优先调度有异曲同工之妙，因为他们都是比较任务的一个属性，然后来实现一个排序，最后按照这个排序来输出结果。
- 所以，我们需要做的就是改变一个符号、并且改变判断的内容即可

#### 0.1.6.1 一、增添元素

```
1 void add(char *name, int priority, int burst, struct node **head){
2     Task *t_new = (Task*)malloc(sizeof(Task*));
3     t_new->name = name;
4     t_new->priority = priority;
5     t_new->burst = burst;
6
7     struct node *current_node, *front_node;
8     current_node = *head;
9     front_node = NULL;
10    if(current_node==NULL)
11    {
12        insert(head, t_new);
13        return ;
14    }
15    while(current_node!=NULL)
16    {
17        if(priority < current_node->task->priority ){
18            front_node = current_node;
19            current_node = current_node->next;
20
21        }
22        else{
23            struct node* node_newtask = malloc(sizeof(struct node));
24            node_newtask->task = t_new;
25            node_newtask->next = current_node;
26            if(front_node==NULL){
27                *head = node_newtask;
28            }
29            else{
30                front_node->next = node_newtask;
31            }
32            return;
33        }
34    }
35    if(current_node==NULL)
36    {
37        struct node* node_newtask = malloc(sizeof(struct node));
38        node_newtask->task = t_new;
39        node_newtask->next = NULL;
40        front_node->next = node_newtask;
```



```

41         return;
42     }
43 }

```

## 0.1.6.2 二、调度执行

```

1 void schedule(struct node **head){
2     struct node* current_node;
3     current_node = *head;
4     while(current_node!=NULL)
5     {
6         run(current_node->task,current_node->task->burst);
7         current_node = current_node->next;
8     }
9 }

```

## 0.1.7 七、轮转调度

### 0.1.7.1 一、实现说明

- 轮转调度的算法就像FCFS一样，但是每次执行的时候，有一个时间片的限制，一旦任务超出了时间片的限制，就被停止，然后去执行下一个任务。
- 值得注意的是，我们在执行任务的时候，每次执行完，对应的 `bursttime` 要做一个减法，当然，如果这个时间小于那个时间片的限制，就相当于我们把这个任务全部执行完成了，所以时间设置为 0。
- 我们使用了循环链表，这是一个很好的解决这个问题的算法，完成任务就删除掉这个节点，没有完成就放在那里，下次循环的时候自然会到那里，所以直到链表上面没有元素了那么就可以结束了。

### 0.1.7.2 二、相关代码

```

1
2 void add(char *name, int priority, int burst, struct node **head){
3     struct node *current_node;
4     current_node = *head;
5
6     Task *t_new = (Task*)malloc(sizeof(Task*));
7     t_new->name = name;
8     t_new->priority = priority;
9     t_new->burst = burst;
10
11     if(current_node==NULL)
12     {
13         struct node *new_node = malloc(sizeof(struct node));
14         new_node->task = t_new;
15         new_node->next = NULL;
16         *head = new_node;
17         return;

```

```

18     }
19     while((current_node->next) != NULL)
20     {
21         current_node=current_node->next;
22     }
23     if((current_node->next) == NULL)
24     {
25         struct node *new_node = malloc(sizeof(struct node));
26         new_node->task = t_new;
27         new_node->next = NULL;
28         current_node->next = new_node;
29         return;
30     }
31 }
32
33
34
35 void schedule(struct node **head) {
36     int quantum = QUANTUM;
37     struct node* current_node, *front_node;
38     current_node = *head;
39     front_node = NULL;
40     // get tail node
41     while((current_node->next) != NULL)
42     {
43         current_node = current_node->next;
44     }
45     current_node->next = *head;
46     // change into circular list
47
48     current_node = *head;
49     while(current_node != NULL)
50     {
51         if(current_node->task->burst <= quantum)
52         {
53             run(current_node->task, current_node->task->burst);
54             struct node *tmp;
55             tmp = front_node->next;
56
57             if(front_node != NULL)
58             {
59
60                 front_node->next = current_node->next;
61                 current_node = current_node->next;
62                 if(current_node->next == current_node)
63                 {
64                     run(current_node->task, current_node->task->burst);

```

```

65         return;
66     }
67     free(tmp);
68     continue;
69 }
70 }
71 else
72 {
73     run(current_node->task, quantum);
74     current_node->task->burst -= quantum;
75 }
76 front_node = current_node;
77 current_node = current_node->next;
78
79 }
80 }
81
82

```

## 0.1.8 八、优先级-轮转调度

### 0.1.8.1 一、原理介绍

- 这种调度方法结合了轮转和优先级的调度。
  - 将链表中相同优先级的部分转化为循环链表
  - 在这个循环列表上运行 round\_robin()
  - 转向运行下一级优先级列表

### 0.1.8.2 二、代码演示

```

1 void schedule(struct node **head) {
2
3     //int quantum = QUANTUM;
4
5     struct node* current_node;
6     current_node = *head;
7
8     struct node** circular_list_head=malloc(sizeof(struct node*));
9     // struct node** circular_list_tail=malloc(sizeof(struct node*));
10    struct node* current_circular_node=malloc(sizeof(struct node));
11    current_circular_node = NULL;
12
13    while(current_node!=NULL)
14    {
15        if(current_circular_node==NULL)
16        {
17            current_circular_node = current_node;

```

```

18         *circular_list_head = current_node;
19         //printf("!!!%d\n", current_circular_node->task->priority);
20         current_node = current_node->next;
21
22         continue;
23     }
24
25     if(current_node->task->priority==current_circular_node->task-
26 >priority)
27     {
28         current_circular_node = current_circular_node->next;
29         current_node = current_node->next;
30
31         continue;
32     }
33     else
34     {
35         // a level of priority has been found out
36         *head = current_node;
37         current_circular_node->next = NULL;
38         round_robin(circular_list_head);
39         // clear the list
40         *circular_list_head = NULL;
41         current_circular_node = *circular_list_head;
42     }
43     if(current_circular_node!=NULL)
44     {
45         round_robin(circular_list_head);
46     }
47
48 }

```

### 0.1.9 九、附加挑战题

在课本的后面给出了两个挑战题目，第一道题是给每一个任务分配一个编号ID，但是考虑到可能出现竞争状态，所以要使用原子整数。第二题是一个计算题，要求计算每一个算法的一些时间周期。考虑到整个项目的量非常的大，所以这里简答说明第一题，详细说明第二题。

#### 0.1.9.1 一、附加题 (1)

- 在Linux和MacOS系统上，`__sync_fetch_and_add()` 函数可用于原子上增加整数值。例如，以下代码示例原子上将值递增1：

```

1 int value = 0;
2 __sync_fetch_and_add(&value,1);

```

解决方法：

- 定义一个全局变量 `pidNum = 0`，初始化为 0。
- 每次有新来的任务的时候，执行下面的语句，这样可以给这个数据加一个保护锁，在执行加减的时候避免其他新来的进程干扰，从而避免两个任务同时到达的时候，pid可能出现相同的情况。

```

1 void add(char *name, int priority, int burst, struct node **head) {
2     pthread_mutex_t lock;
3     pthread_mutex_init(&lock, NULL);
4     pthread_mutex_lock(&lock);
5
6     Task *t_new = (Task*)malloc(sizeof(Task));
7     t_new->name = name;
8     t_new->priority = priority;
9     t_new->tid = taskSize;
10    taskSize++;
11    t_new->burst = burst;
12    insert(head, t_new);
13    pthread_mutex_unlock(&lock);
14 }

```

- 代码中经过修改也已经体现这个过程。

## 0.1.9.2 二、附加题 (2)

### 0.1.9.2.1 a) 概念复习

- 响应时间：从提交第一个请求到产生第一个响应所用时间。
- 周转时间：从作业提交到作业完成的时间间隔。
- 等待时间：在等待队列里面的时间。

### 0.1.9.2.2 b) FCFS调度算法 时间计算

- 要计算平均周转时间、平均等待时间、平均响应时间：绘制甘特图，结果如下，第一行表示任务，第二行表示时间

$P_0$	$P_1$	$P_2$	$P_3$	$P_{size-1}$
$a_0$	$a_1$	$a_2$	$a_3$	$a_{size-1}$

- 那么，经过iPad上面的一通计算猛如虎（别问，问就是缺纸），有下面的公式可以使用：

$$\text{平均等待时间} = \frac{(size-1) \times a_0 + (size-2) \times a_1 + \dots + 2 \times a_{size-3} + 1 \times a_{size-2}}{size}$$

$$\text{平均响应时间} = \frac{(size-1) \times a_0 + (size-2) \times a_1 + \dots + 2 \times a_{size-3} + 1 \times a_{size-2}}{size}$$

$$\text{平均周转时间} = \frac{(size) \times a_0 + (size-1) \times a_1 + \dots + 2 \times a_{size-2} + 1 \times a_{size-1}}{size}$$

- 为了针对性的解决问题，我们定义一个数组，把这个链表的**burstTime**时间从头到尾遍历一下，然后代入这个计算公式。
- 显然这个函数具有很强的通用性，因为SJF还有优先级调度都可以用这个算法，为什么呢，因为他们在操作的时候都是执行的排序+输出执行的操作，我们进行的只不过是执行之前多了一个预计算的过程，所以不会

改变链表，也非常具有通用性。

```
1 // 接收一个链表，然后计算
2 void timeCalculator(struct node **head) {
3     if(taskSize > 0) {
4         double turnAroundTime = 0;
5         double waitTime = 0;
6         double responseTime = 0;
7
8         int timeArray[taskSize];
9         int i;
10
11        struct node *current_node;
12        current_node = *head;
13
14        for(i=0; i<taskSize; i++){
15            timeArray[i] = current_node->task->burst;
16            current_node = current_node->next;
17
18            turnAroundTime += (taskSize - i) * timeArray[i];
19            responseTime += (taskSize - i - 1) * timeArray[i];
20            waitTime += (taskSize - i - 1) * timeArray[i];
21
22            //printf("%d\t", timeArray[i]);
23        }
24
25        printf("turnAround Time Average is : %lf (ms)\n",turnAroundTime
26        /taskSize);
27        printf("response Time Average is : %lf (ms)\n ",responseTime
28        /taskSize);
29        printf("waitTime Time Average is : %lf (ms)\n",waitTime /taskSize);
30    }
```

#### 0.1.9.2.3 c) SJF调度算法 时间计算

- 国际惯例，遇事不决先抄公式，显然这两个算法是一样的，直接套公式即可。

$$\text{平均等待时间} = \frac{(size - 1) \times a_0 + (size - 2) \times a_1 + \dots + 2 \times a_{size-3} + 1 \times a_{size-2}}{size}$$

$$\text{平均响应时间} = \frac{(size - 1) \times a_0 + (size - 2) \times a_1 + \dots + 2 \times a_{size-3} + 1 \times a_{size-2}}{size}$$

$$\text{平均周转时间} = \frac{(size) \times a_0 + (size - 1) \times a_1 + \dots + 2 \times a_{size-2} + 1 \times a_{size-1}}{size}$$

#### 0.1.9.2.4 d) 优先级调度 时间计算

- 方法和上面的是一模一样的，函数具有通用性，至少在这三个里面是通用的。

#### 0.1.9.2.5 e) RR调度 时间计算（难）

- 要计算这种类型的平均时间，是一件非常麻烦的事情。因为在轮转中存在这样的现象：

- ```
1  # 对于每一个任务来说，可能经过下面的过程
2  |任务到达-----等待中-----|<执行任务>|-----等待中-----|执行任务并完成|
3  |<-----响应时间----->|
4  |<-----周转时间----->|
5  |<-----等待时间1----->|<任务时间>|<-----等待时间2----->|<--任务时间-->
```

- 所以，经过我细心的观察，发现，对于每一个任务来说，周转时间 - *burstTime* = 等待时间总和。
- 这时候就要优化代码了，首先我们要设置全局变量、时间戳等信息。
  - 在执行一个任务剩余时间大于时间片区单位的时候，时间戳要加上轮转的时间片长度
  - 在执行一个任务剩余时间小于或等于时间片区单位的时候，说明这个任务马上完结撒花，就可以时间戳加上任务剩余时间。
  - 当然，对于时间戳操作也可以用锁，但是为了节约代码量，这里不做代码修改。
- 在某个任务要终结的时候，执行完后，要把当前的时间戳写入任务对应ID的数组（周转时间数组）中。
- 在初始化等待时间数组的时候，可以把每个任务的bursttime时间先写入，这样可以节约空间，而不是另外的申请一个数组来保存时间信息。

```
1  #include "schedulers.h"
2  #include "cpu.h"
3  #include <stdio.h>
4  #include <stdlib.h>
5
6
7  int* turnAroundTime;
8  int* waitTime;
9  int* responseTime;
10
11
12  int timeStamp = 0;
13  int taskSize = 0;
14
15  double turnAroundTimeAvg = 0;
16  double waitTimeAvg = 0;
17  double responseTimeAvg = 0;
18
19  void initArray(struct node **head) {
20      turnAroundTime = (int*) malloc(sizeof(int) * taskSize);
21      waitTime = (int*) malloc(sizeof(int) * taskSize);
22      responseTime = (int*) malloc(sizeof(int) * taskSize);
```

```

23
24     int i;
25     struct node *current_node;
26     current_node = *head;
27
28     for (i = 0; i < taskSize; i++) {
29         turnAroundTime[i] = 0;
30         responseTime[i] = 0;
31         waitTime[i] = current_node->task->burst;
32     }
33 }
34
35 void freeArray(){
36     free(turnAroundTime);
37     free(waitTime);
38     free(responseTime);
39 }
40
41
42 void responseTimeCalculator(struct node **head){
43     if(taskSize > 0){
44         struct node *current_node;
45         current_node = *head;
46         int sumTmp = 0;
47         int i;
48
49         for(i=0; i<taskSize; i++){
50             responseTime[i] = sumTmp;
51             if(current_node->task->burst < QUANTUM)
52                 sumTmp += current_node->task->burst;
53             else
54                 sumTmp += QUANTUM;
55             responseTimeAvg += responseTime[i];
56
57             current_node = current_node->next;
58         }
59     }
60 }
61
62 void timeCalculator(struct node **head) {
63     int i;
64     for (i = 0; i < taskSize; i++) {
65         waitTime[i] = turnAroundTime[i] - waitTime[i];
66         waitTimeAvg += waitTime[i];
67         turnAroundTimeAvg += turnAroundTime[i];
68

```



```

69         printf("Pid:[%d] waitTime[%d] turnAroundTime[%d] responseTime[%d]
\n",i,waitTime[i],turnAroundTime[i],responseTime[i]);
70     }
71
72     printf("turnAround Time Average is : %lf (ms)\n",turnAroundTimeAvg
/taskSize);
73     printf("response Time Average is : %lf (ms)\n",responseTimeAvg
/taskSize);
74     printf("waitTime Time Average is : %lf (ms)\n",waitTimeAvg /taskSize);
75 }
76
77
78 void add(char *name, int priority, int burst, struct node **head){
79     struct node *current_node;
80     current_node = *head;
81
82     pthread_mutex_t lock;
83     pthread_mutex_init(&lock,NULL);
84     pthread_mutex_lock(&lock);
85
86
87     Task *t_new = (Task*)malloc(sizeof(Task*));
88     t_new->name = name;
89     t_new->priority = priority;
90     t_new->burst = burst;
91     t_new->tid = taskSize;
92     taskSize++;
93
94     pthread_mutex_unlock(&lock);
95
96     if(current_node==NULL)
97     {
98         struct node *new_node = malloc(sizeof(struct node));
99         new_node->task = t_new;
100         new_node->next = NULL;
101         *head = new_node;
102         return;
103     }
104
105     while((current_node->next)!=NULL)
106     {
107         current_node=current_node->next;
108     }
109     if((current_node->next)==NULL)
110     {
111         struct node *new_node = malloc(sizeof(struct node));
112         new_node->task = t_new;

```

```

113         new_node->next = NULL;
114         current_node->next = new_node;
115         return;
116     }
117
118 }
119
120
121 void schedule(struct node **head) {
122     initArray(head);
123     responseTimeCalculator(head);
124
125     int quantum = QUANTUM;
126     struct node* current_node, *front_node;
127     current_node = *head;
128     front_node = NULL;
129     // get tail node
130     while((current_node->next) != NULL)
131     {
132         current_node = current_node->next;
133     }
134     current_node->next = *head;
135     // change into circular list
136     current_node = *head;
137     while(current_node != NULL)
138     {
139         if(current_node->task->burst <= quantum)
140         {
141             timeStamp += current_node->task->burst;
142             run(current_node->task, current_node->task->burst);
143             turnAroundTime[current_node->task->tid] = timeStamp;
144             current_node->task->burst = 0;
145
146             struct node *tmp;
147
148             if(front_node != NULL)
149             {
150                 tmp = front_node->next;
151                 front_node->next = current_node->next;
152                 current_node = current_node->next;
153                 if(current_node->next == current_node)
154                 {
155                     if(current_node->task->burst <= quantum) {
156                         run(current_node->task, current_node->task->burst);
157                         timeStamp += current_node->task->burst;
158                         current_node->task->burst -= quantum;

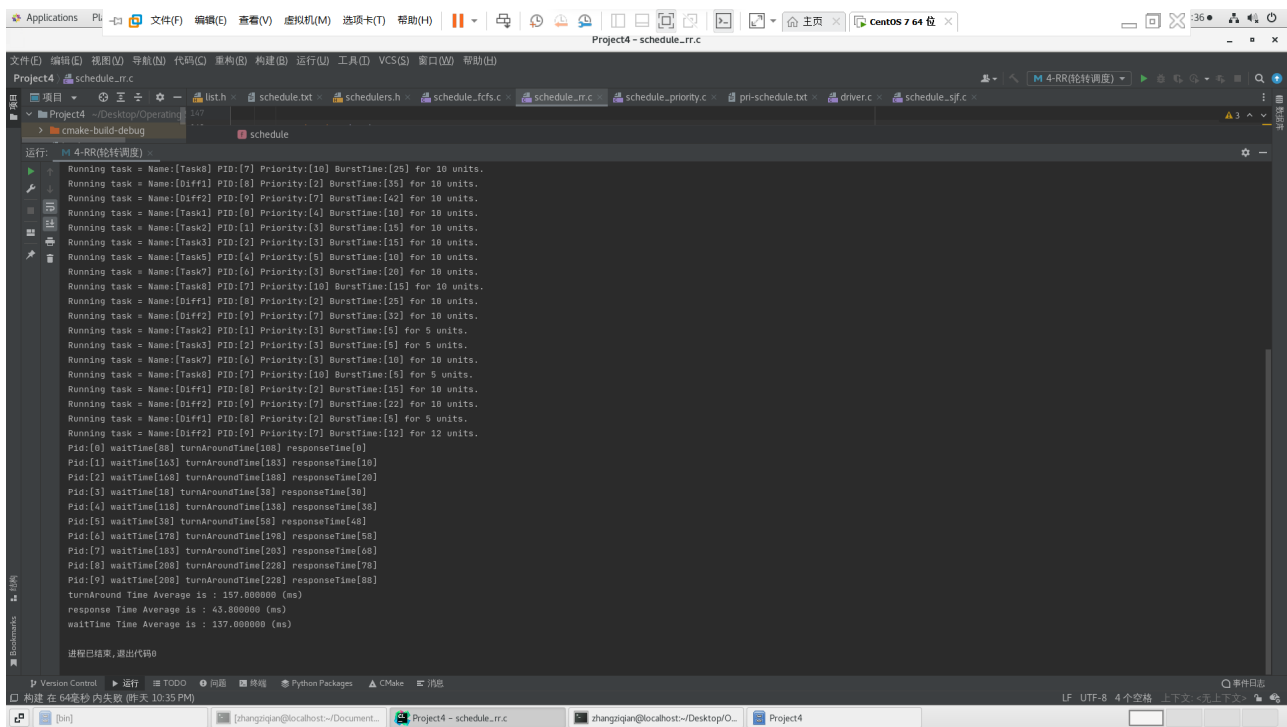
```

```

159         turnAroundTime[current_node->task->tid] =
timeStamp;
160         return;
161     }
162     else{
163         while(current_node->task->burst > quantum){
164             run(current_node->task, quantum);
165             timeStamp+=quantum;
166             current_node->task->burst -= quantum;
167         }
168         run(current_node->task, current_node->task->burst);
169         timeStamp+=current_node->task->burst;
170         current_node->task->burst -= quantum;
171
172         turnAroundTime[current_node->task->tid] =
timeStamp;
173         free(current_node);
174         return;
175     }
176 }
177
178     timeCalculator(head);
179     return;
180 }
181 free(tmp);
182 continue;
183 }
184 }
185 else
186 {
187     timeStamp += quantum;
188     run(current_node->task, quantum);
189     current_node->task->burst -= quantum;
190 }
191 front_node = current_node;
192 current_node = current_node->next;
193 }
194 }

```

- 效果展示



```
Running task = Name:[Task8] PID:[7] Priority:[10] BurstTime:[25] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[35] for 10 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[42] for 10 units.
Running task = Name:[Task1] PID:[0] Priority:[4] BurstTime:[10] for 10 units.
Running task = Name:[Task2] PID:[1] Priority:[3] BurstTime:[15] for 10 units.
Running task = Name:[Task3] PID:[2] Priority:[3] BurstTime:[5] for 10 units.
Running task = Name:[Task6] PID:[4] Priority:[5] BurstTime:[10] for 10 units.
Running task = Name:[Task7] PID:[6] Priority:[3] BurstTime:[20] for 10 units.
Running task = Name:[Task8] PID:[7] Priority:[10] BurstTime:[15] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[25] for 10 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[32] for 10 units.
Running task = Name:[Task2] PID:[1] Priority:[3] BurstTime:[5] for 5 units.
Running task = Name:[Task3] PID:[2] Priority:[3] BurstTime:[5] for 5 units.
Running task = Name:[Task7] PID:[6] Priority:[3] BurstTime:[10] for 10 units.
Running task = Name:[Task8] PID:[7] Priority:[10] BurstTime:[5] for 5 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[15] for 10 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[22] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[5] for 5 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[12] for 12 units.
Pid:[0] waitTime[88] turnAroundTime[108] responseTime[0]
Pid:[1] waitTime[163] turnAroundTime[183] responseTime[10]
Pid:[2] waitTime[168] turnAroundTime[188] responseTime[20]
Pid:[3] waitTime[18] turnAroundTime[38] responseTime[30]
Pid:[4] waitTime[118] turnAroundTime[138] responseTime[38]
Pid:[5] waitTime[38] turnAroundTime[58] responseTime[48]
Pid:[6] waitTime[178] turnAroundTime[198] responseTime[58]
Pid:[7] waitTime[183] turnAroundTime[203] responseTime[68]
Pid:[8] waitTime[208] turnAroundTime[228] responseTime[78]
Pid:[9] waitTime[208] turnAroundTime[228] responseTime[88]
turnAround Time Average is : 157.888889 (ms)
response Time Average is : 43.888889 (ms)
waitTime Time Average is : 137.888889 (ms)
进程已结束,退出代码0
```

#### 0.1.9.2.6 f) 优先级-RR调度 时间计算

特别顺序警告:

- 代码的顺序必须是:
  - run函数!
  - 操作当然时间戳!
  - 修改任务的剩余时间!

仅有下面两种是正确的!!!

```
1 | run(current_node->task, current_node->task->burst);
2 | timeStamp+=current_node->task->burst;
3 | current_node->task->burst -= quantum;
```

```
1 | timeStamp+=current_node->task->burst;
2 | run(current_node->task, current_node->task->burst);
3 | current_node->task->burst -= quantum;
```

和之前的算法类似,所以贴一下代码!

```
1 | #include "schedulers.h"
2 | #include "cpu.h"
3 | #include <stdio.h>
4 | #include <stdlib.h>
5 |
6 | int* turnAroundTime;
7 | int* waitTime;
```

```

8  int* responseTime;
9
10 int timeStamp = 0;
11 int taskSize = 0;
12
13 double turnAroundTimeAvg = 0;
14 double waitTimeAvg = 0;
15 double responseTimeAvg = 0;
16
17 void initArray(struct node **head){
18     turnAroundTime = (int*) malloc(sizeof(int) * taskSize);
19     waitTime = (int*) malloc(sizeof(int) * taskSize);
20     responseTime = (int*) malloc(sizeof(int) * taskSize);
21
22     int i;
23     struct node *current_node;
24     current_node = *head;
25
26     for (i = 0; i < taskSize; i++) {
27         turnAroundTime[i] = 0;
28         responseTime[i] = 0;
29         waitTime[i] = current_node->task->burst;
30     }
31 }
32
33 void freeArray(){
34     free(turnAroundTime);
35     free(waitTime);
36     free(responseTime);
37 }
38
39 void responseTimeCalculator(struct node **head){
40     if(taskSize > 0){
41         struct node *current_node;
42         current_node = *head;
43         int sumTmp = 0;
44         int i;
45
46         for(i=0; i<taskSize; i++){
47             responseTime[i] = sumTmp;
48             if(current_node->task->burst < QUANTUM)
49                 sumTmp += current_node->task->burst;
50             else
51                 sumTmp += QUANTUM;
52             responseTimeAvg += responseTime[i];
53
54             current_node = current_node->next;

```

```

55         }
56     }
57 }
58
59 void timeCalculator(struct node **head) {
60     int i;
61     for (i = 0; i < taskSize; i++) {
62         waitTime[i] = turnAroundTime[i] - waitTime[i];
63         waitTimeAvg += waitTime[i];
64         turnAroundTimeAvg += turnAroundTime[i];
65
66         printf("Pid:[%d] waitTime[%d] turnAroundTime[%d] responseTime[%d]
67 \n",i,waitTime[i],turnAroundTime[i],responseTime[i]);
68     }
69
70     printf("turnAround Time Average is : %lf (ms)\n",turnAroundTimeAvg
71 /taskSize);
72     printf("response Time Average is : %lf (ms)\n",responseTimeAvg
73 /taskSize);
74     printf("waitTime Time Average is : %lf (ms)\n",waitTimeAvg /taskSize);
75 }
76
77 void add(char *name, int priority, int burst, struct node **head){
78     pthread_mutex_t lock;
79     pthread_mutex_init(&lock,NULL);
80     pthread_mutex_lock(&lock);
81
82
83     Task *t_new = (Task*)malloc(sizeof(Task*));
84     t_new->name = name;
85     t_new->priority = priority;
86     t_new->burst = burst;
87     t_new->tid = taskSize;
88     taskSize++;
89
90     pthread_mutex_unlock(&lock);
91
92     struct node *current_node, *front_node;
93     current_node = *head;
94     front_node = NULL;
95     if(current_node==NULL)
96     {
97         insert(head, t_new);
98         return ;

```

```

99     }
100     while (current_node != NULL)
101     {
102         if (priority <= current_node->task->priority ) {
103             front_node = current_node;
104             current_node = current_node->next;
105         }
106         else {
107             struct node* node_newtask = malloc(sizeof(struct node));
108             node_newtask->task = t_new;
109             node_newtask->next = current_node;
110             if (front_node == NULL) {
111                 *head = node_newtask;
112             }
113             else {
114                 front_node->next = node_newtask;
115             }
116             return;
117         }
118     }
119     if (current_node == NULL)
120     {
121         struct node* node_newtask = malloc(sizeof(struct node));
122         node_newtask->task = t_new;
123         node_newtask->next = NULL;
124         front_node->next = node_newtask;
125         return;
126     }
127 }
128
129 void round_robin(struct node **head) {
130     int quantum = QUANTUM;
131     struct node* current_node, *front_node;
132     current_node = *head;
133     front_node = NULL;
134     // get tail node
135     while ((current_node->next) != NULL)
136     {
137         current_node = current_node->next;
138     }
139     current_node->next = *head; // change into circular list
140
141     // front_node init val is tail_node
142     front_node = current_node;
143     // current_node now is head_node
144     current_node = *head;
145

```

```

146 //special situation
147 if(current_node->next==current_node)
148 {
149     int time = current_node->task->burst;
150     while(time > quantum)
151     {
152         run(current_node->task, quantum);
153         timeStamp +=quantum;
154         current_node->task->burst -= quantum;
155         time = current_node->task->burst;
156     }
157     run(current_node->task, current_node->task->burst);
158     timeStamp += current_node->task->burst;
159     turnAroundTime[current_node->task->tid] = timeStamp;
160
161     free(current_node);
162     return;
163 }
164
165 while(current_node!=NULL)
166 {
167     if(current_node->task->burst <= quantum)
168     {
169         run(current_node->task, current_node->task->burst);
170         timeStamp += current_node->task->burst;
171         current_node->task->burst = 0;
172         turnAroundTime[current_node->task->tid] = timeStamp;
173
174         struct node *tmp;
175         // Now, it's important to remove this node
176         if(front_node!=NULL)
177         {
178             tmp = front_node->next;
179             front_node->next = current_node->next;
180             current_node = current_node->next;
181             if(current_node->next == current_node)
182             {
183                 if(current_node->task->burst <= quantum){
184                     run(current_node->task, current_node->task->burst);
185                     timeStamp += current_node->task->burst;
186                     current_node->task->burst = 0;
187                     turnAroundTime[current_node->task->tid] =
timeStamp;
188                     free(current_node);
189                     return;
190                 }
191                 else{

```



```

192         while(current_node->task->burst > quantum){
193             run(current_node->task, quantum);
194             timeStamp +=quantum;
195             current_node->task->burst -= quantum;
196         }
197
198         run(current_node->task, current_node->task->burst);
199         timeStamp += current_node->task->burst;
200         current_node->task->burst = 0;
201
202         turnAroundTime[current_node->task->tid] =
timeStamp;
203         free(current_node);
204         return;
205     }
206 }
207 free(tmp);
208 continue;
209 }
210 }
211 else
212 {
213     run(current_node->task, quantum);
214     timeStamp += quantum;
215     current_node->task->burst -= quantum;
216 }
217 front_node = current_node;
218 current_node = current_node->next;
219 }
220 }
221
222
223 void schedule(struct node **head){
224     initArray(head);
225     responseTimeCalculator(head);
226
227     //int quantum = QUANTUM;
228
229     struct node* current_node;
230     current_node = *head;
231
232     struct node** circular_list_head=malloc(sizeof(struct node*));
233     // struct node** circular_list_tail=malloc(sizeof(struct node*));
234     struct node* current_circular_node;
235     current_circular_node = NULL;
236
237     while(current_node!=NULL)

```

```

238     {
239         if(current_circular_node==NULL)
240         {
241             current_circular_node = current_node;
242             *circular_list_head = current_node;
243             //printf("!!!%d\n", current_circular_node->task->priority);
244             current_node = current_node->next;
245             continue;
246         }
247
248         if(current_node->task->priority==current_circular_node->task-
>priority)
249         {
250             current_circular_node = current_circular_node->next;
251             current_node = current_node->next;
252             continue;
253         }
254         else
255         {
256             // a level of priority has been found out
257             *head = current_node;
258             current_circular_node->next = NULL;
259             round_robin(circular_list_head);
260             // clear the list
261             *circular_list_head = NULL;
262             current_circular_node = *circular_list_head;
263
264         }
265     }
266     if(current_circular_node!=NULL)
267     {
268         round_robin(circular_list_head);
269     }
270     timeCalculator(head);
271 }
272
273

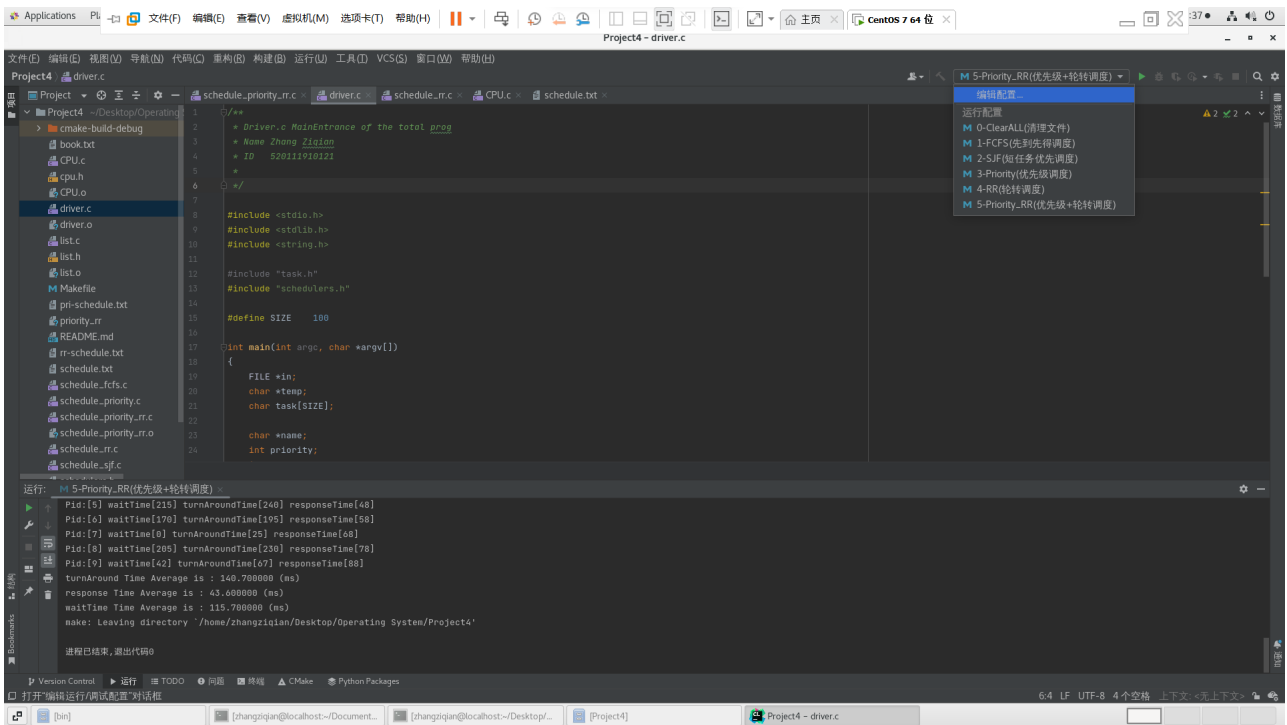
```

#### 0.1.10 十、归纳总结

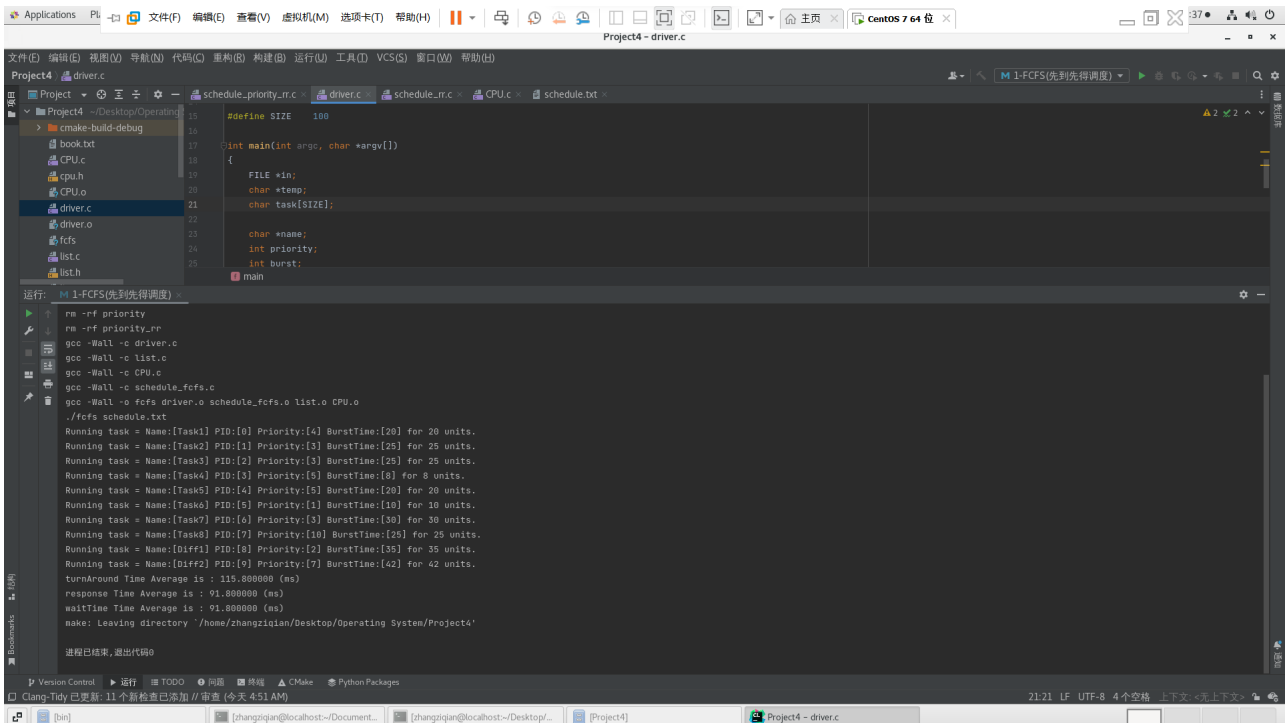
- 这几种算法都是非常重要的调度算法，不仅是操作系统中的应用广泛，在一些服务器系统也很常见。
- 通过这次的练习，复习了我们之前学习过的一些基础的数据结构（例如链表等等），此外对这些基本调度的算法，也有了更加深刻的认识。
- 这个Project的代码量稍微有一些多，但是好在一些基础的代码已经给我们提供了，例如解析输入的内容，所以只需要完成核心的内容部分，所以设计也比较合理。
- 最后一个挑战题目特别有意思，很值得挑战！！受益匪浅！！

## 0.1.11 十一、效果演示图

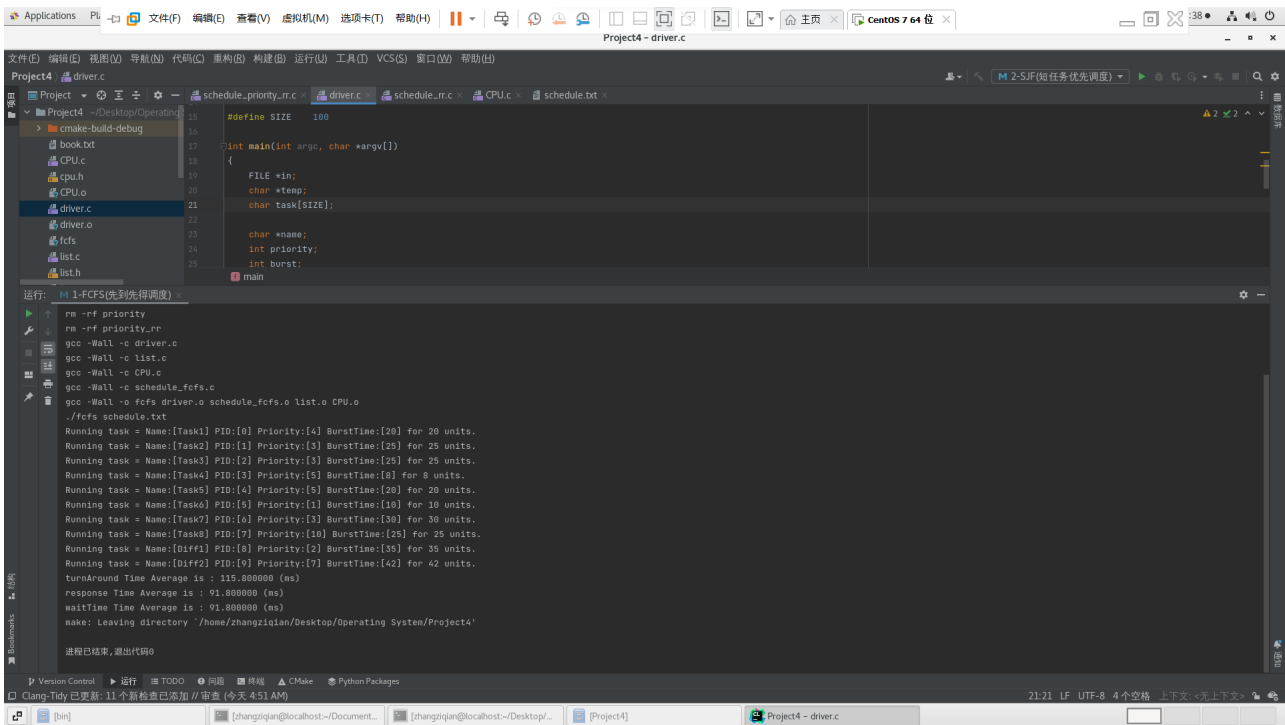
- 配置随心选



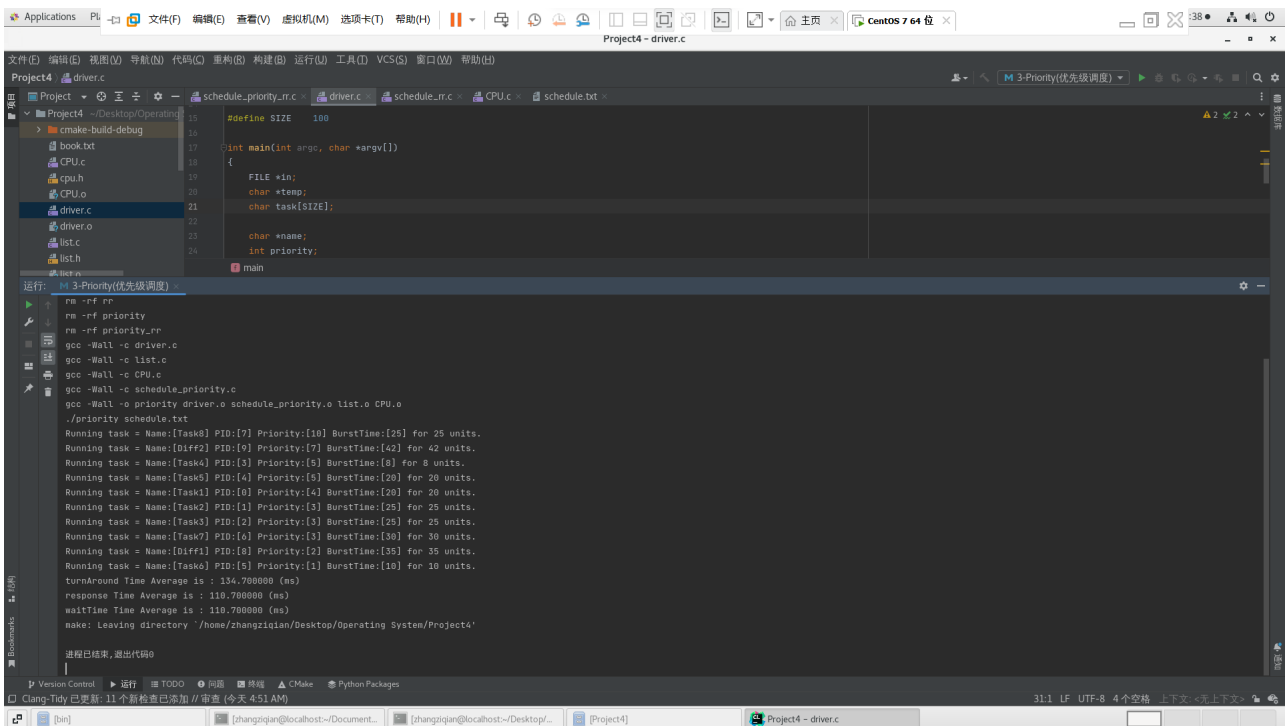
- FCFS演示图



- SJF



## • 优先级



## • RR

```
Running task = Name:[Task7] PID:[6] Priority:[3] BurstTime:[30] for 10 units.
Running task = Name:[Task8] PID:[7] Priority:[10] BurstTime:[25] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[36] for 10 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[42] for 10 units.
Running task = Name:[Task3] PID:[0] Priority:[4] BurstTime:[10] for 10 units.
Running task = Name:[Task2] PID:[1] Priority:[3] BurstTime:[16] for 10 units.
Running task = Name:[Task3] PID:[2] Priority:[3] BurstTime:[16] for 10 units.
Running task = Name:[Task5] PID:[4] Priority:[5] BurstTime:[10] for 10 units.
Running task = Name:[Task7] PID:[6] Priority:[3] BurstTime:[20] for 10 units.
Running task = Name:[Task8] PID:[7] Priority:[10] BurstTime:[15] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[26] for 10 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[32] for 10 units.
Running task = Name:[Task2] PID:[1] Priority:[3] BurstTime:[5] for 5 units.
Running task = Name:[Task3] PID:[2] Priority:[3] BurstTime:[5] for 5 units.
Running task = Name:[Task7] PID:[6] Priority:[3] BurstTime:[10] for 10 units.
Running task = Name:[Task8] PID:[7] Priority:[10] BurstTime:[5] for 9 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[16] for 10 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[22] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[5] for 5 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[12] for 10 units.
Running task = Name:[Diff2] PID:[9] Priority:[7] BurstTime:[2] for 2 units.
Pid:[0] waitTime[88] turnAroundTime[108] responseTime[0]
Pid:[1] waitTime[163] turnAroundTime[183] responseTime[10]
Pid:[2] waitTime[168] turnAroundTime[188] responseTime[20]
Pid:[3] waitTime[18] turnAroundTime[38] responseTime[30]
Pid:[4] waitTime[118] turnAroundTime[138] responseTime[38]
Pid:[5] waitTime[38] turnAroundTime[58] responseTime[48]
Pid:[6] waitTime[178] turnAroundTime[198] responseTime[58]
Pid:[7] waitTime[133] turnAroundTime[153] responseTime[68]
Pid:[8] waitTime[208] turnAroundTime[228] responseTime[78]
Pid:[9] waitTime[228] turnAroundTime[248] responseTime[88]
turnAround Time Average is : 158.200000 (ns)
response Time Average is : 43.800000 (ns)
waitTime Time Average is : 138.200000 (ns)
```

## • 优先级-RR

```
Running task = Name:[Task5] PID:[4] Priority:[5] BurstTime:[10] for 10 units.
Running task = Name:[Task1] PID:[0] Priority:[4] BurstTime:[20] for 10 units.
Running task = Name:[Task1] PID:[0] Priority:[4] BurstTime:[10] for 10 units.
Running task = Name:[Task2] PID:[1] Priority:[3] BurstTime:[26] for 10 units.
Running task = Name:[Task3] PID:[2] Priority:[3] BurstTime:[16] for 10 units.
Running task = Name:[Task7] PID:[6] Priority:[3] BurstTime:[30] for 10 units.
Running task = Name:[Task2] PID:[1] Priority:[3] BurstTime:[16] for 10 units.
Running task = Name:[Task7] PID:[6] Priority:[3] BurstTime:[20] for 10 units.
Running task = Name:[Task2] PID:[1] Priority:[3] BurstTime:[5] for 5 units.
Running task = Name:[Task3] PID:[2] Priority:[3] BurstTime:[5] for 5 units.
Running task = Name:[Task7] PID:[6] Priority:[3] BurstTime:[10] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[36] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[26] for 10 units.
Running task = Name:[Diff1] PID:[8] Priority:[2] BurstTime:[5] for 5 units.
Running task = Name:[Task6] PID:[5] Priority:[1] BurstTime:[10] for 10 units.
Pid:[0] waitTime[90] turnAroundTime[115] responseTime[0]
Pid:[1] waitTime[155] turnAroundTime[180] responseTime[10]
Pid:[2] waitTime[160] turnAroundTime[185] responseTime[20]
Pid:[3] waitTime[50] turnAroundTime[75] responseTime[28]
Pid:[4] waitTime[70] turnAroundTime[95] responseTime[38]
Pid:[5] waitTime[215] turnAroundTime[240] responseTime[48]
Pid:[6] waitTime[170] turnAroundTime[195] responseTime[58]
Pid:[7] waitTime[0] turnAroundTime[25] responseTime[68]
Pid:[8] waitTime[205] turnAroundTime[230] responseTime[78]
Pid:[9] waitTime[42] turnAroundTime[67] responseTime[88]
turnAround Time Average is : 140.700000 (ns)
response Time Average is : 43.800000 (ns)
waitTime Time Average is : 115.700000 (ns)
make: Leaving directory '/home/zhangqian/Desktop/Operating System/Project4'
进程已结束,退出代码0
```