# Report for OS Project 4

## Scheduling Algorithms

陈文迪 519021910071

### I. 实验任务

1. 实现多种进程调度算法。这些调度算法是**离线**的，因为调度器可以在一开始获取所有的进程任务信息如优先级和CPU burst，所有的进程都在0时刻到达。具体包括的调度算法如下：
    - First-come, first-served (FCFS)
    - Shortest-job-first (SJF)
    - 优先级调度，数值越高优先级越高
    - Round-robin (RR)，时间片长度为10毫秒
    - 结合RR调度的优先级调度，对相同优先级的进程采取RR调度
2. 使用原子整型变量来解决进程的进程号分配问题。
3. 实现一些基本的数据统计功能，如：平均周转时间、平均等待时间和平均相应时间。

### II. 实验思路

#### 1. 数据结构概览

在本次项目中，我们使用一个链表来管理全部待调度的进程。在本书所附带的源代码中已经提供了相应的具体实现，但经过仔细阅读过后我们发现了些许问题—由于全部的进程在本项目中都被记录在一个`Task`结构体中，而其中的进程名是一个需要动态分配的字符串数组，而在链表的删除函数中却没有实现该内存的释放。因此，我们需要修改原有的`delete()`函数。

```
// delete the selected task from the list
void delete(struct node **head, Task *task) {
    struct node *temp;
    struct node *prev;

    temp = *head;
    // special case - beginning of list
    if (strcmp(task->name,temp->task->name) == 0) {
        free(temp->task->name);
        free(temp->task);
        *head = (*head)->next;
        free(temp);
    }
    else {
        // interior or last element in the list
        prev = *head;
        temp = temp->next;
        while (strcmp(task->name,temp->task->name) != 0) {
            prev = temp;
            temp = temp->next;
        }
        free(temp->task->name);
        free(temp->task);
        prev->next = temp->next;
        free(temp);
    }
```

```
}
```

由于我们需要统计每个进程的平均相应时间，所以我们需要知道目前执行的进程是不是第一次执行，于是我们修改了`Task`结构体，用一个变量`flag`来表示是否是第一次执行该进程。

```
// representation of a task
typedef struct task {
    char *name;
    int tid;
    int priority;
    int burst;
    int flag;
} Task;
```

**2. 基本架构**

在本次项目设计中，依照项目源代码，我们需要完成的`add()`和`schedule()`两个关键函数。其中`add()`函数的功能为将进程任务插入到进程链表中，记录下进程的基本信息，并为其分配一个专有的进程号。需要注意的是，本项目源代码中的链表实现是从头部插入新结点，因此整个链表的顺序**与插入顺序相反**。此外，为了在SMP系统中进程号的分配也可以顺利进行，我们使用了原子操作函数`__sync_fetch_and_add()`来解决同步问题。

```
void add(char *name, int priority, int burst){
    Task *newtask = (Task *)malloc(sizeof(Task ));
    newtask->priority = priority;
    newtask->burst = burst;
    newtask->name = (char*)malloc(sizeof(char)*100);
    newtask->tid = __sync_fetch_and_add(&tid,1);
    strcpy(newtask->name,name);
    insert(&head,newtask);

    newtask->flag = 1;
    count++;
    all_burst+=burst;
}
```

**3. 数据统计**

为了能实现对平均周转时间、平均等待时间和平均相应时间的统计，我们采用了一个统一的统计策略来实现对任意调度算法都可以进行调度，其基本思路如下。我们统计如下数据：总周转时间`turnaround`，总相应时间`response`，总CPU burst时间`all_burst`和总进程数`count`。并用一个全局变量`time`来记录目前的系统时间，而`time`需要在每次CPU执行后更新。其中`count`和`all_burst`在`add()`中即可统计完毕，`response`可以通过`flag`在第一次执行时利用`time`计算出来，而当一个进程全部执行完后，我们可以将当时的`time`作为它的周转时间。最后，等待时间可以通过公式$\text{waiting time} = \text{turnaround time} - \text{all CPU burst}$得出。这就完成了全部数据的统计。

**4. FCFS 调度算法**

FCFS调度算法即先服务先请求的进程。

**实现方法**

根据本项目中的链表结构，链表中的进程顺序与请求顺序恰好是相反的。因此，我们可以这样实现FCFS调度算法：每次遍历到链表的结尾，使用`run()`运行链表结尾的进程，并将其从进程链表中删除。不断执行上述过程，直到进程链表为空。最后，将统计数据打印出来。

核心代码如下：

```
void schedule(){
    while(head){
        struct node*temp = head;
        while(temp->next) temp = temp->next;
        if(temp->task->flag){
            temp->task->flag=0;
            response+=time;
        }
        run(temp->task,temp->task->burst);
        time+=temp->task->burst;
        turnaround+=time;
        delete(&head,temp->task);
    }
    printf("Average Turnaround:%.2f\t Average Waiting:%.2f\t Average
Response:%.2f\n",turnaround/(float)count,(turnaround-
all_burst)/(float)count,response/(float)count);
}
```

**5. SJF 调度算法**

SJF调度算法是优先级调度算法的一个特例，每次选择下一个CPU burst最短的进程执行。

**实现方法**

遍历进程链表，保存下链表中执行时间最短的进程节点指针，使用`run()`执行该进程，并将其从进程链表中删除。不断执行上述过程，直到进程链表为空。最后，将统计数据打印出来。

核心代码如下：

```
void schedule(){
    while(head){
        struct node*temp = head;
        struct node*min_node = head;
        int min = temp->task->burst;

        while(temp){
            if(temp->task->burst<min){
                min_node = temp;
                min = temp->task->burst;
            }
            temp = temp->next;
        }
        if(min_node->task->flag){
            response+=time;
            min_node->task->flag = 0;
        }
        run(min_node->task,min_node->task->burst);
        time+=min_node->task->burst;
        turnaround+=time;
        delete(&head,min_node->task);
    }

    printf("Average Turnaround:%.2f\t Average Waiting:%.2f\t Average
Response:%.2f\n",turnaround/(float)count,(turnaround-
all_burst)/(float)count,response/(float)count);
}
```

### 6. 优先级调度算法

优先级调度算法每次选择优先级最高的进程执行。

#### 实现方法

与SJF调度算法的实现极其类似。遍历进程链表，保存下链表中优先级最高的进程节点指针，使用`run()`执行该进程，并将其从进程链表中删除。不断执行上述过程，直到进程链表为空。最后，将统计数据打印出来。

核心代码如下：

```
void schedule(){
    while(head){
        struct node*temp = head;
        struct node*max_node = head;
        int max = temp->task->priority;

        while(temp){
            if(temp->task->priority>max){
                max_node = temp;
                max = temp->task->priority;
            }
            temp = temp->next;
        }
        if(max_node->task->flag){
            response+=time;
            max_node->task->flag = 0;
        }
        run(max_node->task,max_node->task->burst);
        time+=max_node->task->burst;
        turnaround+=time;
        delete(&head,max_node->task);
    }
    printf("Average Turnaround:%.2f\t Average Waiting:%.2f\t Average
Response:%.2f\n",turnaround/(float)count,(turnaround-
all_burst)/(float)count,response/(float)count);
}
```

### 7. RR调度算法

RR调度算法按照到达顺序让每一个进程执行一个时间片长度，由此来避免饥饿产生。

#### 实现方法

由于我们需要轮转执行每一个进程，为了保证在一次轮转过程中，每一个进程都会被执行正好一次，我们需要把这个倒置链表**反转**。通过反转链表之后，我们可以按顺序遍历进程链表，对于剩余执行时间大于时间片长度的，我们通过`run()`让它执行一个时间片长度的时间，并更新其剩余执行时间；对于剩余执行时间小于等于时间片长度的，我们通过`run()`让它将剩余时间全部执行完，并将该进程从进程链表中删除。不断执行上述过程，直到进程链表为空。最后，将统计数据打印出来。

核心代码如下：

```
void schedule(){
    if(head){ //reverse the linked list
        struct node* slow = head;
        struct node* fast = head->next;
```

```
        head->next = NULL;
        while(fast){
            head = fast;
            struct node* temp = fast->next;
            fast->next = slow;
            slow = fast;
            fast = temp;
        }
    }
    while(head){
        struct node* temp = head;
        while(temp){
            if(temp->task->burst>QUANTUM){
                if(temp->task->flag){
                    temp->task->flag=0;
                    response+=time;
                }
                run(temp->task,QUANTUM);
                time+=QUANTUM;

                temp->task->burst = temp->task->burst-QUANTUM;
                temp = temp->next;
            }
            else{
                if(temp->task->flag){
                    temp->task->flag=0;
                    response+=time;
                }
                run(temp->task,temp->task->burst);
                time+=temp->task->burst;
                turnaround+=time;

                struct node* next = temp->next;
                delete(&head,temp->task);
                temp = next;
            }
        }
    }

    printf("Average Turnaround:%.2f\t Average Waiting:%.2f\t Average
Response:%.2f\n",turnaround/(float)count,(turnaround-
all_burst)/(float)count,response/(float)count);
}
```

**8. 结合RR调度的优先级调度算法**

该算法首先执行优先级最高的进程，对于相同优先级的进程之间采用RR调度。

**实现方法**

从算法的描述上我们可以非常方便地实现该算法。首先遍历整个进程链表，保存下链表中优先级最高的进程节点指针和最大优先级地值，并记录最大优先级地进程共有几个。若只有一个，则用`run()`将其执行完之后从进程链表中删除即可；若有多个，则对这几个拥有最大优先级的进程采用RR调度直到，它们全都执行完。不断执行上述过程，直到进程链表为空。最后，将统计数据打印出来。

核心代码如下：

```
void schedule(){
```

```
    while(head){
        struct node*temp = head->next;
        struct node*max_node = head;
        int max = head->task->priority;
        int max_count = 1;


        while(temp){
            if(temp->task->priority>max){
                max_node = temp;
                max = temp->task->priority;
                max_count = 1;
            }
            else if(temp->task->priority==max){
                max_count ++;
            }
            temp = temp->next;
        }

        //printf("max_count:%d\n",max_count);

        if(max_count==1){
            if(max_node->task->flag){
                max_node->task->flag=0;
                response+=time;
            }
            run(max_node->task,max_node->task->burst);
            time+=max_node->task->burst;
            turnaround+=time;

            delete(&head,max_node->task);
        }
        else{
            while(1){
                int flag = 0;
                temp = head;
                while(temp){
                    if(temp->task->priority==max){
                        //printf("process:%s\n",temp->task->name);
                        flag=1;
                        if(temp->task->burst>QUANTUM){
                            if(temp->task->flag){
                                temp->task->flag=0;
                                response+=time;
                            }
                            run(temp->task,QUANTUM);
                            time+=QUANTUM;

                            temp->task->burst = temp->task->burst-QUANTUM;
                            temp = temp->next;
                        }
                        else{
                            if(temp->task->flag){
                                temp->task->flag=0;
                                response+=time;
                            }
                            run(temp->task,temp->task->burst);
                            time+=temp->task->burst;
```

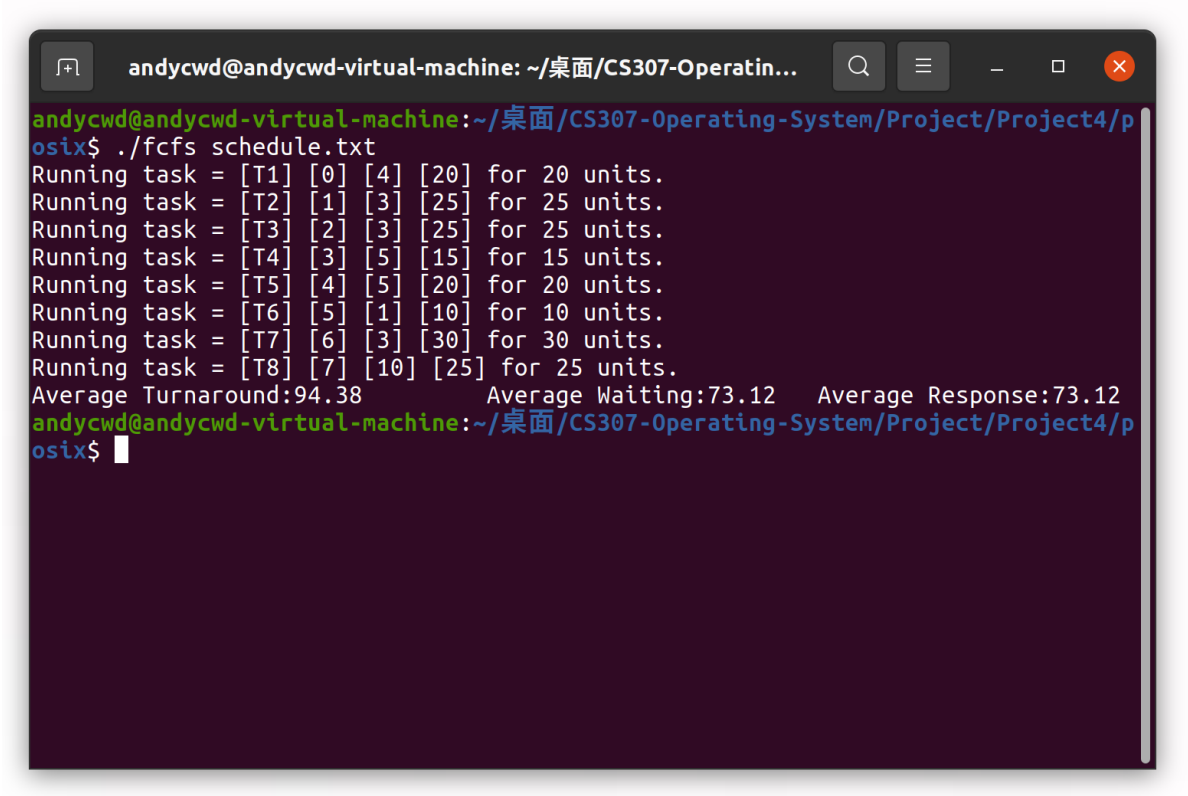
```
                            turnaround+=time;

                            struct node* next = temp->next;
                            delete(&head,temp->task);
                            temp = next;
                        }
                    }
                    else{
                        temp = temp->next;
                    }
                }
                if(flag==0) break;
            }
        }
    }
    printf("Average Turnaround:%.2f\t Average Waiting:%.2f\t Average
Response:%.2f\n",turnaround/(float)count,(turnaround-
all_burst)/(float)count,response/(float)count);
}
```

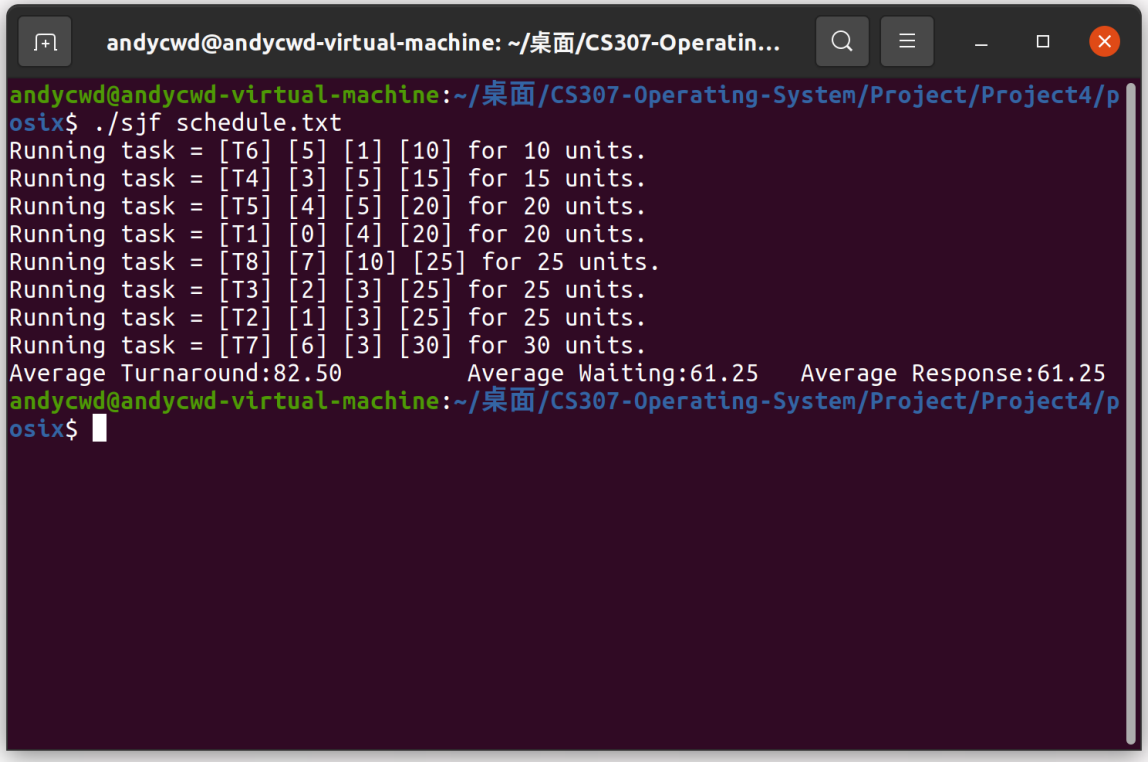
## III. 实验过程

我们对项目进行编译后，利用源代码附带的`schedule.txt`对每个调度算法进行测试。

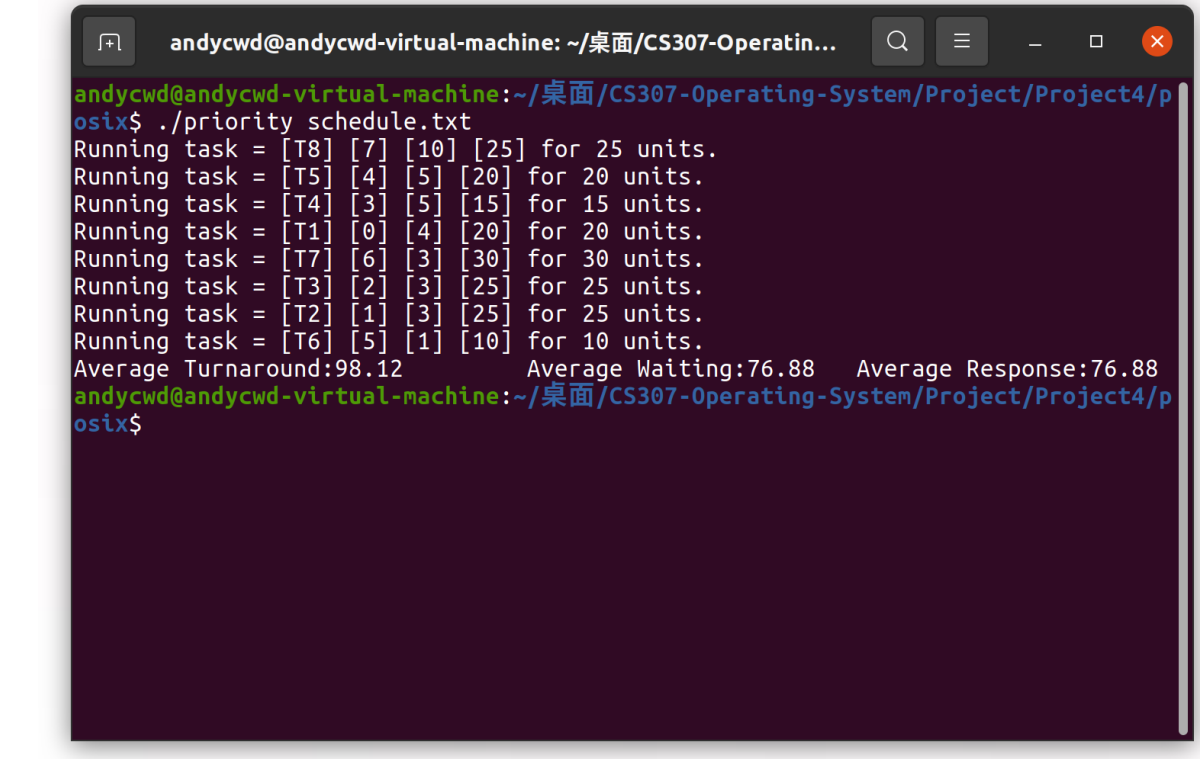
### 1. FCFS 调度算法



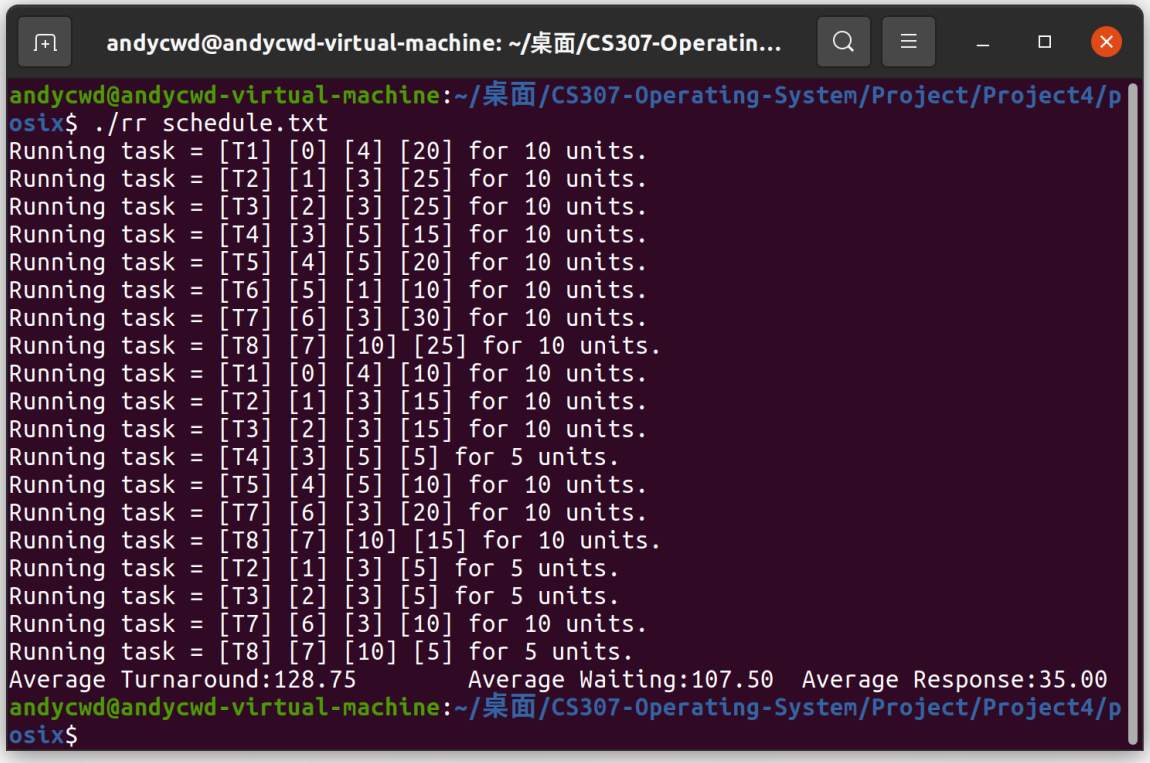

### 2. SJF调度算法



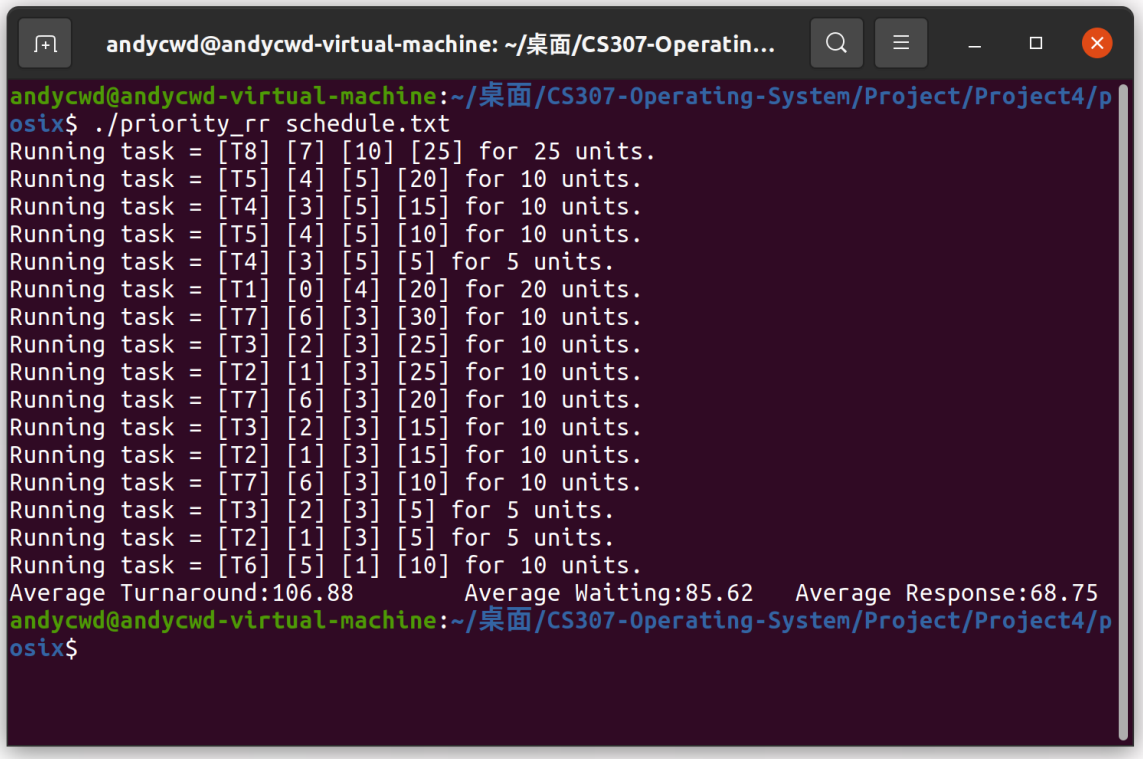

**3. 优先级调度算法**




**4. RR调度算法**




**5. 结合RR调度的优先级调度算法**




**6. 统计数据**

| Algorithm | Average Turnaround Time | Average Waiting Time | Average Response Time |
| --- | --- | --- | --- |
| FCFS | 94.38 | 73.12 | 73.12 |
| SJF | 82.50 | 61.25 | 61.25 |
| Priority | 98.12 | 76.88 | 76.88 |
| RR | 128.75 | 107.50 | 35.00 |
| Priority with RR | 106.88 | 85.62 | 68.75 |

可以看到SJF调度算法具有最短的平均周转时间，而RR调度算法的平均相应时间较短。

## IV. 遇到的问题

**有关c语言编写链表数据结构的细节问题**

由于之前编写数据结构都是采用面向对象的C++语言，因此对于如何使用面向过程的C语言编写链式数据结构不是特别了解。并且C语言中并没有引用传递，需要通过指针的指针实现类似功能。不过，这些问题都是程序设计中的常见问题，通过查阅相关资料都可以轻松解决。

## V. 参考资料

[1] Operating System Concept 10th Edition

[2] C reference - cppreference.com