

Project4: Scheduling Algorithms

王凯俊 520021910282

- [1 Introduction](#)
- [2 Implementation](#)
 - [2-1 Check the code already](#)
 - [2-2 Do the algorithm respectively](#)
 - [2-2-1 FCFS algorithm](#)
 - [2-2-2 SJF algorithm](#)
 - [2-2-3 Priority algorithm](#)
 - [2-2-4 RR algorithm](#)
 - [2-2-5 Priority with RR algorithm](#)
- [3 Conclusion and Thoughts](#)

1. 1 Introduction

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

- First-come, first-served (**FCFS**), which schedules tasks in the order in which they request the CPU.
- Shortest-job-first (**SJF**), which schedules tasks in order of the length of the tasks' next CPU burst.
- **Priority** scheduling, which schedules tasks based on priority.
- Round-robin (**RR**) scheduling, where each task is run for a time quantum (or for the remainder of its CPU burst).
- **Priority** with **round-robin**, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.

Priorities range from 1 to 10, where a higher numeric value indicates a higher relative priority. For round-robin scheduling, the length of a time quantum is **10 milliseconds**.

The task's detailed definition is as below:

The schedule of tasks has the form [**task name**] [**priority**] [**CPU burst**], with the following example format:

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

2. 2 Implementation

2.1. 2-1 Check the code already

We first check the code already done to know how can we achieve the code.

In the **task.h**, there are task structure:

```
typedef struct task {
    char *name;
    int tid;
    int priority;
    int burst;
} Task;
```

In **CPU.c**, there is a **run** function to stimulate the execution of CPU:

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

In **driver.c**, there is a **main** function to stimulate the whole process of task running:

```
int main(int argc, char *argv[])
{
    FILE *in;
    char *temp;
    char task[SIZE];
    char *name;
    int priority;
    int burst;
    struct node **head=(struct node**)(malloc(sizeof(struct node**)));
    in = fopen(argv[1], "r");
    while (fgets(task, SIZE, in) != NULL) {
        temp = strdup(task);
        name = strsep(&temp, ",");
        priority = atoi(strsep(&temp, ","));
        burst = atoi(strsep(&temp, ","));
        // add the task to the scheduler's list of tasks
        add(name, priority, burst, head);
        free(temp);
    }
    fclose(in);
    // invoke the scheduler
    schedule(head);
    return 0;
}
```

Then we find it that we should do the **add()** and **schedule** function to make the process go.

2.2. 2-2 Do the algorithm respectively

2.2.1. 2-2-1 FCFS algorithm

We first simply add the new task into the linked list.

```
void add(char *name, int priority, int burst, struct node **head){
    Task *t_new = (Task*)malloc(sizeof(Task*));
    t_new->name = name;
    t_new->priority = priority;
    t_new->burst = burst;
    insert(head, t_new);
}
```

Then we will find the order is just like **LCFS** algorithm, so we need to reverse the whole linked list:

```
while(current_node!=NULL && current_node->next!=NULL)
{
    back_node = current_node->next;
    current_node->next = front_node;
    front_node = current_node;
    current_node = back_node;
}
if(current_node!=NULL && current_node->next==NULL)
    current_node->next = front_node;
*head = current_node;
```

Then we can just run the new task:

```
Task *new_task;
while(current_node!=NULL)
{
    new_task = current_node->task;
    run(new_task, new_task->burst);
    current_node = current_node->next;
}
```

The result is as below:

type:

```
make fcfs
./fcfs schedule.txt
```

The result:

```
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ make fcfs
make: 'fcfs' is up to date.
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./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.
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$
```

2.2.2. 2-2-2 SJF algorithm

Because we want to process the shortest job firstly, so when we add the new task, it's essential to insert the task in the burst order.

So the **add** function is as below:

```
void add(char *name, int priority, int burst, struct node **head){
    Task *t_new = (Task*)malloc(sizeof(Task*));
    t_new->name = name;
    t_new->priority = priority;
    t_new->burst = burst;
    struct node *current_node, *front_node;
    current_node = *head;
    front_node = NULL;
    if(current_node==NULL)
    {
        insert(head, t_new);
        return ;
    }
    while(current_node!=NULL)
    {
        if(burst > current_node->task->burst ){
            front_node = current_node;
            current_node = current_node->next;
        }
        else{
            struct node* node_newtask = malloc(sizeof(struct node));
            node_newtask->task = t_new;
            node_newtask->next = current_node;
            if(front_node==NULL){
                *head = node_newtask;
            }
            else{
                front_node->next = node_newtask;
            }
            return;
        }
    }
}
```

```

    if(current_node==NULL)
    {
        struct node* node_newtask = malloc(sizeof(struct node));
        node_newtask->task = t_new;
        node_newtask->next = NULL;
        front_node->next = node_newtask;
        return;
    }
}

```

And the **schedule** is the same as FCFS algorithm:

```

void schedule(struct node** head)
{
    struct node* current_node;
    current_node = *head;
    while(current_node!=NULL)
    {
        new_task = current_node->task;
        run(new_task, new_task->burst);
        current_node = current_node->next;
    }
}

```

The result is as below:

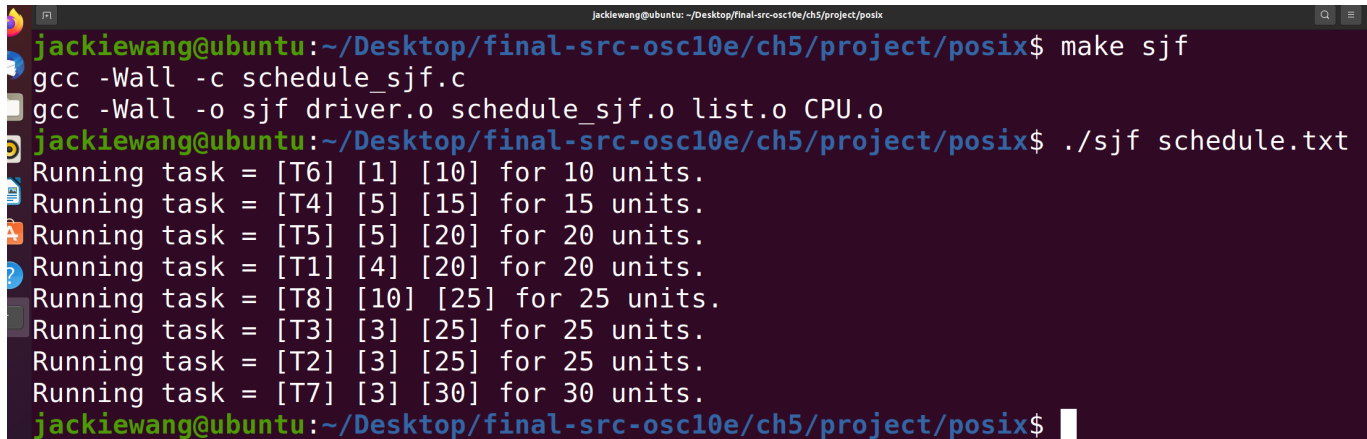
Type:

```

make sjf
./sjf schedule.txt

```

The result:



```

jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ make sjf
gcc -Wall -c schedule_sjf.c
gcc -Wall -o sjf driver.o schedule_sjf.o list.o CPU.o
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./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 = [T3] [3] [25] for 25 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$

```

2.2.3. 2-2-3 Priority algorithm

The **priority** algorithm is almost the same as the **SJF** algorithm. We change the insert criterion from **burst** to **priority**.

So the **add** is as below:

```
void add(char *name, int priority, int burst, struct node **head){
    Task *t_new = (Task*)malloc(sizeof(Task*));
    t_new->name = name;
    t_new->priority = priority;
    t_new->burst = burst;
    struct node *current_node, *front_node;
    current_node = *head;
    front_node = NULL;
    if(current_node==NULL)
    {
        insert(head, t_new);
        return ;
    }
    while(current_node!=NULL)
    {
        if(priority < current_node->task->priority ){
            front_node = current_node;
            current_node = current_node->next;
        }
        else{
            struct node* node_newtask = malloc(sizeof(struct node));
            node_newtask->task = t_new;
            node_newtask->next = current_node;
            if(front_node==NULL){
                *head = node_newtask;
            }
            else{
                front_node->next = node_newtask;
            }
            return;
        }
    }
    if(current_node==NULL)
    {
        struct node* node_newtask = malloc(sizeof(struct node));
        node_newtask->task = t_new;
        node_newtask->next = NULL;
        front_node->next = node_newtask;
        return;
    }
}
```

the **schedule** function:

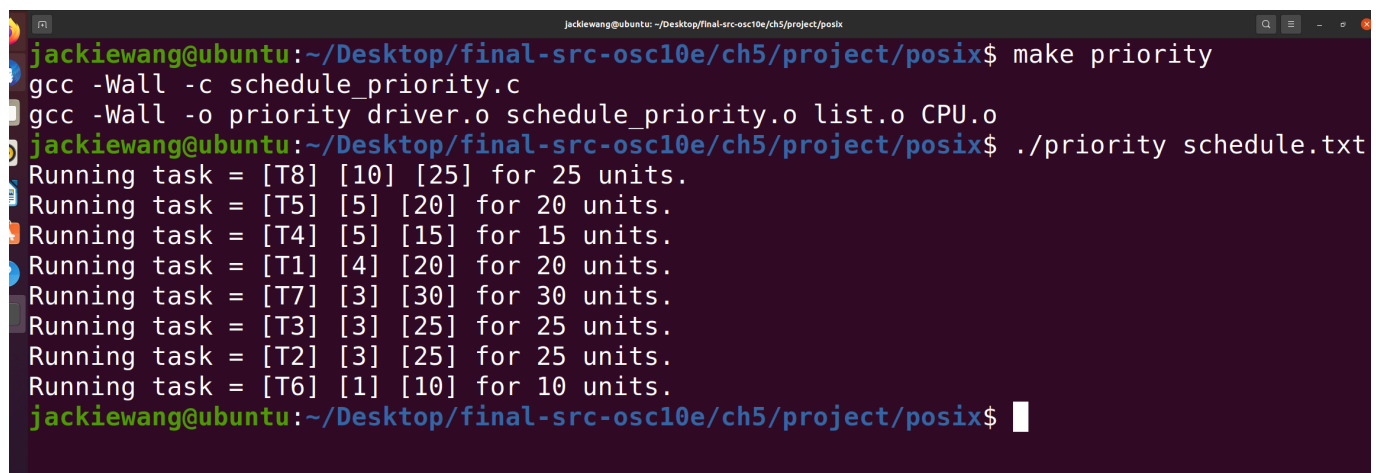
```
void schedule(struct node **head){
    struct node* current_node;
    current_node = *head;
    while(current_node!=NULL)
    {
        new_task = current_node->task;
        run(current_node->task,current_node->task->burst);
        current_node = current_node->next;
    }
}
```

The result is as below:

Type:

```
make priority
./priority schedule.txt
```

The result:



```
jackiewang@ubuntu: ~/Desktop/final-src-osc10e/ch5/project/posix
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ make priority
gcc -Wall -c schedule_priority.c
gcc -Wall -o priority driver.o schedule_priority.o list.o CPU.o
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./priority schedule.txt
Running task = [T8] [10] [25] for 25 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T6] [1] [10] for 10 units.
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$
```

2.2.4. 2-2-4 RR algorithm

Considering the **RR** algorithm, it's similar to the FCFS algorithm, but it needs the switch.

So I think the good way to achieve it is to use a circle linked list.

The **add** function is as below:

```
void add(char *name, int priority, int burst, struct node **head){
    struct node *current_node;
    current_node = *head;

    Task *t_new = (Task*)malloc(sizeof(Task*));
    t_new->name = name;
    t_new->priority = priority;
    t_new->burst = burst;
```

```

if(current_node==NULL)
{
    struct node *new_node = malloc(sizeof(struct node));
    new_node->task = t_new;
    new_node->next = NULL;
    *head = new_node;
    return;
}
while((current_node->next)!=NULL)
{
    current_node=current_node->next;
}
if((current_node->next)==NULL)
{
    struct node *new_node = malloc(sizeof(struct node));
    new_node->task = t_new;
    new_node->next = NULL;
    current_node->next = new_node;
    return;
}
}

```

The **schedule** function is as below:

```

void schedule(struct node **head){
    int quantum = QUANTUM;
    struct node* current_node, *front_node;
    current_node = *head;
    front_node = NULL;
    while((current_node->next)!=NULL)
    {
        current_node = current_node->next;
    }
    current_node->next = *head;
    current_node = *head;
    while(current_node!=NULL)
    {
        if(current_node->task->burst <= quantum)
        {
            run(current_node->task, current_node->task->burst);
            struct node *tmp;
            tmp = front_node->next;

            if(front_node!=NULL)
            {
                front_node->next = current_node->next;
                current_node = current_node->next;
                if(current_node->next == current_node)
                {
                    run(current_node->task, current_node->task->burst);
                    return;
                }
            }
        }
    }
}

```



```

        free(tmp);
        continue;
    }

}

else
{
    run(current_node->task, quantum);
    current_node->task->burst -= quantum;
}

front_node = current_node;
current_node = current_node->next;
}
}

```

The result is as below:

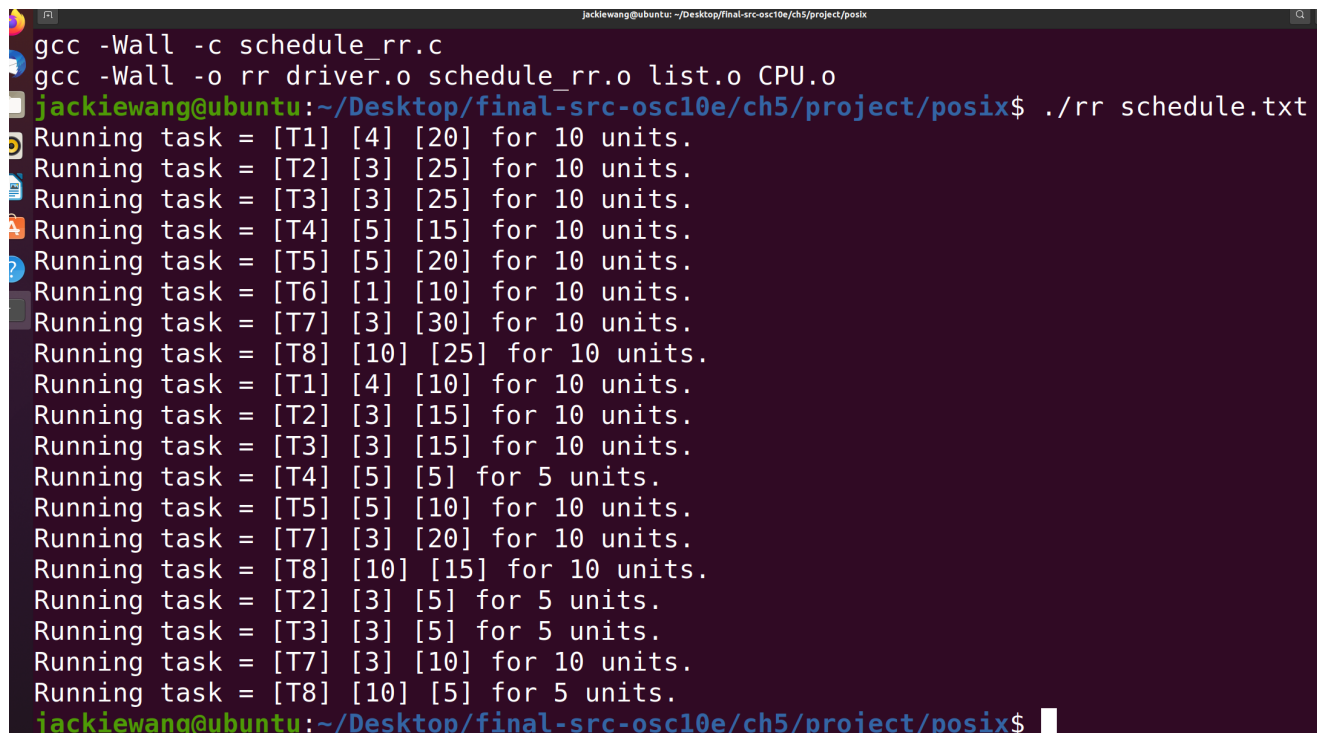
Type:

```

make rr
./rr schedule.txt

```

The result:



```

jackiewang@ubuntu: ~/Desktop/final-src-osc10e/ch5/project/posix
gcc -Wall -c schedule_rr.c
gcc -Wall -o rr driver.o schedule_rr.o list.o CPU.o
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./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.
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$

```

2.2.5. 2-2-5 Priority with RR algorithm

The **Priority with RR** is the combination of the above algorithms.

For the **add** function, We add the new task based on its priority:

```

void add(char *name, int priority, int burst, struct node **head){
    Task *t_new = (Task*)malloc(sizeof(Task*));
    t_new->name = name;
    t_new->priority = priority;
    t_new->burst = burst;
    struct node *current_node, *front_node;
    current_node = *head;
    front_node = NULL;
    if(current_node==NULL)
    {
        insert(head, t_new);
        return ;
    }
    while(current_node!=NULL)
    {
        if(priority < current_node->task->priority ){
            front_node = current_node;
            current_node = current_node->next;
        }
        else{
            struct node* node_newtask = malloc(sizeof(struct node));
            node_newtask->task = t_new;
            node_newtask->next = current_node;
            if(front_node==NULL){
                *head = node_newtask;
            }
            else{
                front_node->next = node_newtask;
            }
            return;
        }
    }
    if(current_node==NULL)
    {
        struct node* node_newtask = malloc(sizeof(struct node));
        node_newtask->task = t_new;
        node_newtask->next = NULL;
        front_node->next = node_newtask;
        return;
    }
}

```

The **schedule** function:

```

void schedule(struct node **head){

    struct node* current_node;
    current_node = *head;
    struct node** circular_list_head=malloc(sizeof(struct node*));
    struct node* current_circular_node=malloc(sizeof(struct node));
    current_circular_node = NULL;

```

```
while(current_node!=NULL)
{
    if(current_circular_node==NULL)
    {
        current_circular_node = current_node;
        *circular_list_head = current_node;
        current_node = current_node->next;

        continue;
    }
    if(current_node->task->priority==current_circular_node->task->priority)
    {
        current_circular_node = current_circular_node->next;
        current_node = current_node->next;

        continue;
    }
    else
    {
        *head = current_node;
        current_circular_node->next = NULL;
        round_robin(circular_list_head);
        *circular_list_head = NULL;
        current_circular_node = *circular_list_head;
    }
}
if(current_circular_node!=NULL)
{
    round_robin(circular_list_head);
}
}
```

The RR function is similar to the above achievement.

The result is as below:

Type:

```
make priority_rr
./priority_rr schedule.txt
```

The result:

```
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ 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
jackiewang@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./priority_rr schedule
txt
Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [15] for 10 units.
Running task = [T8] [10] [5] for 5 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T3] [3] [5] for 5 units.
Running task = [T2] [3] [5] for 5 units.
Running task = [T6] [1] [10] for 10 units.
```

3. 3 Conclusion and Thoughts

In this project, I have learnt various kinds of algorithm to schedule the CPU tasks.

By using the kinked list, I change the item of list to make the schedule algorithm works.

After the project, I have a deeper impression on the CPU schedule algorithm and feel amazed about the genius of the algorithm.