Project4: **Scheduling Algorithms**

Yanjie Ze 519021910706

1 Introduction

1.1 Requirements

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**.

In this project, I implement the algorithms in **C** language.

1.2 Task Definition

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
```

Thus, task T1 has priority 4 and a CPU burst of 20 milliseconds, and so forth. It is assumed that a**ll tasks arrive at the same time,** so our scheduler algorithms do not have to support higher-priority processes preempting processes with

lower priorities. In addition, tasks do not have to be placed into a queue or list in any particular order.

2 Code Frame

2.1 Task

In **task.h**, we define the basic struct **task** which we will use in all of the following programs and algorithms.

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

task consists of:

- name
- tid
- priority
- burst

2.2 CPU

In **cpu.h**, we define the macro **QUANTUM**:

```
#define QUANTUM 10
```

This is the time quantum which **Round Robin** and **Priority Round Robin** need.

In **CPU.c**, we define the function **run**, to represent the task's execution.

```
// 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);
}
```

2.3 Driver

driver.c is the part that simulates the running of tasks.

We need to finish reading all the tasks in this part, and pass them to the scheduling algorithm. And We actually use the data structure **link list** to store these **tasks**.

What's more, we use the arguments of **main()** to pass the file name.

We use several functions of C to implement the translation of the file:

- **strdup()**, duplicate the string
- **strsep()**, extract the token from the string
- **atoi()**, convert the string to the integer

There are two functions we need to implement for the different scheduling algorithms we use: **add()** and **schedule()**.

```
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;
}
```

3 FCFS

3.1 Algorithm

FCFS(first come first serve) is the simplest scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first.

The implementation of the FCFS policy is easily managed with a **FIFO queue**. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

Therefore, using a queue to implement FCFS is natural.

3.2 add()

The function **add()** adds a new task into the head of the link list.

We make use of the function that the link list itself has: **insert()**.

We first create the new **task**, then insert it into the link 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);
}
```

3.3 schedule()

Since when we insert the new task we simply insert it into the front of the head node, we need to **first invert the link 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;
    }
```

The special case:

```
  // process the end
    if(current_node!=NULL && current_node->next==NULL)
        current_node->next = front_node;
    *head = current_node;
```

Then the link list's order is correct. We simply go throght the whole link list.

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

3.4 Program Results

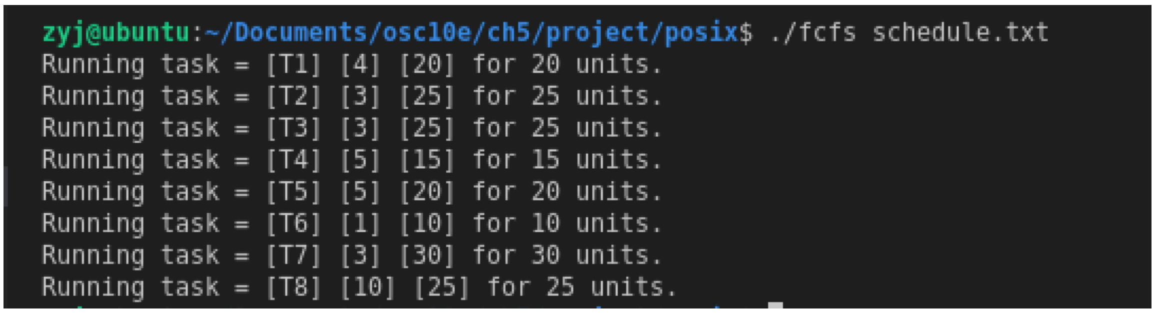
The input file is **schedule.txt**:

```
T1, 4, 20
T2, 3, 25
T3, 3, 25
T4, 5, 15
T5, 5, 20
T6, 1, 10
T7, 3, 30
T8, 10, 25
```

Entering:

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

We get:



```
zyj@ubuntu:~/Documents/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.
```

4 SJF

4.1 Algorithm

SJF (shortest job first) associates with each process the length of the process's next CPU burst. **When the CPU is available, it is assigned to the process that has the smallest next CPU burst**. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

4.2 add()

The function **add()** in SJF is different from that in FCFS.

Since we will run the task with **the smallest CPU burst firs**t, we traverse the link list to find the position where the task should be .

The function first compares the burst of tasks, finds a position, then inserts the task.

```
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;
    }
}
```

4.3 schedule()

The work we have done in **add()** makes us easy to finish **schedule()**.

A single loop is enough.

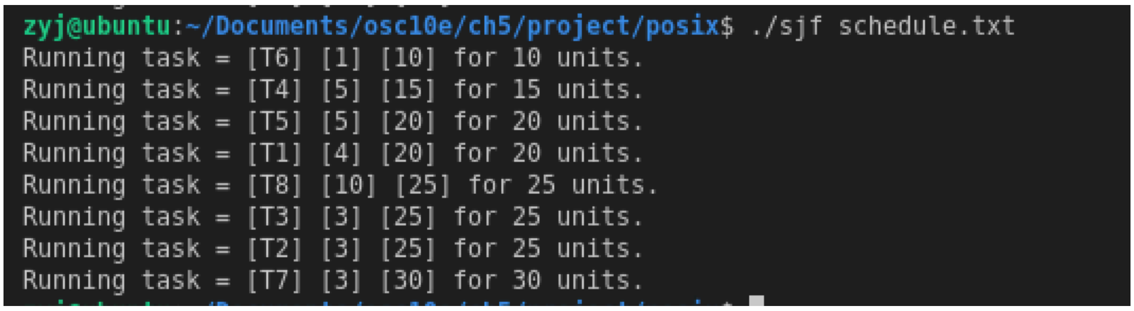
```
void schedule(struct node** head)
{
    struct node* current_node;
    current_node = *head;
    while(current_node!=NULL)
    {
        run(current_node->task,current_node->task->burst);
        current_node = current_node->next;
    }
}
```

4.4 Program Results

Entering:

```
./sjf schedule.txt
```

Then we get:



```
zyj@ubuntu:~/Documents/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.
```

5 Priority

5.1 Algorithm

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (*p*) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

5.2 add()

The basic idea of implementing **add()** in **Priority** is the same as that in **SJF**.

The difference lies in the priority in **SJF** is the burst. The smaller the burst is, the higher priority the task is.

In **Priority** scheduling algorithm, we directly use **priority**.

```c
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;
    }
}
```

5.3 schedule()

The order of the tasks has been modified in **add()**. So we only need to do a traverse.

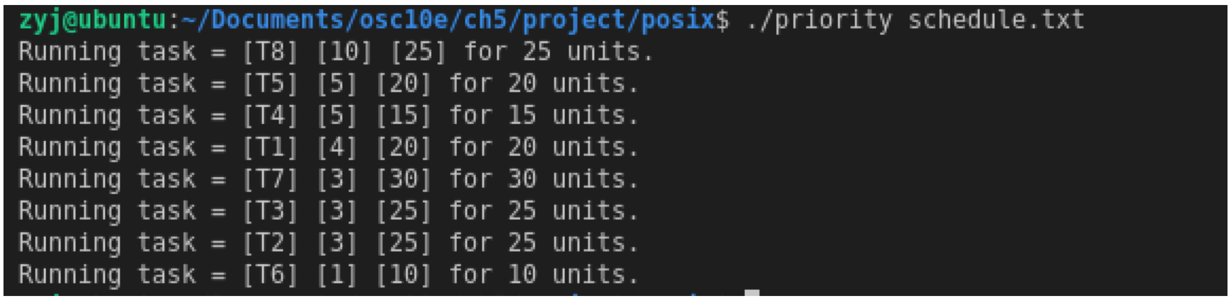
```
void schedule(struct node **head){
    struct node* current_node;
    current_node = *head;
    while(current_node!=NULL)
    {
        run(current_node->task,current_node->task->burst);
        current_node = current_node->next;
    }
}
```

5.4 Program Results

Entering:

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

Then we get:



```
zyj@ubuntu:~/Documents/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.
```

6 RR

6.1 Algorithm

The **round-robin** (**RR**) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

6.2 add()

We directly **add the new task into the tail of the link list.**

If the link list is empty, we need to consider this situation precisely.

```c
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;
    }

}
```

6.3 schedule()

RR is actually combined with **FCFS**.

First, we turn the link list into a **circular link list**:

```
  // get tail node
    while((current_node->next)!=NULL)
    {
        current_node = current_node->next;
    }
    current_node->next = *head;// change into circular list
```

Then,when the burst is not zero, we run each task using **FCFS**, with **a time quantum.**

If one task's burst is zero or less than the time quantum, it is skipped or executed with the remaining burst.

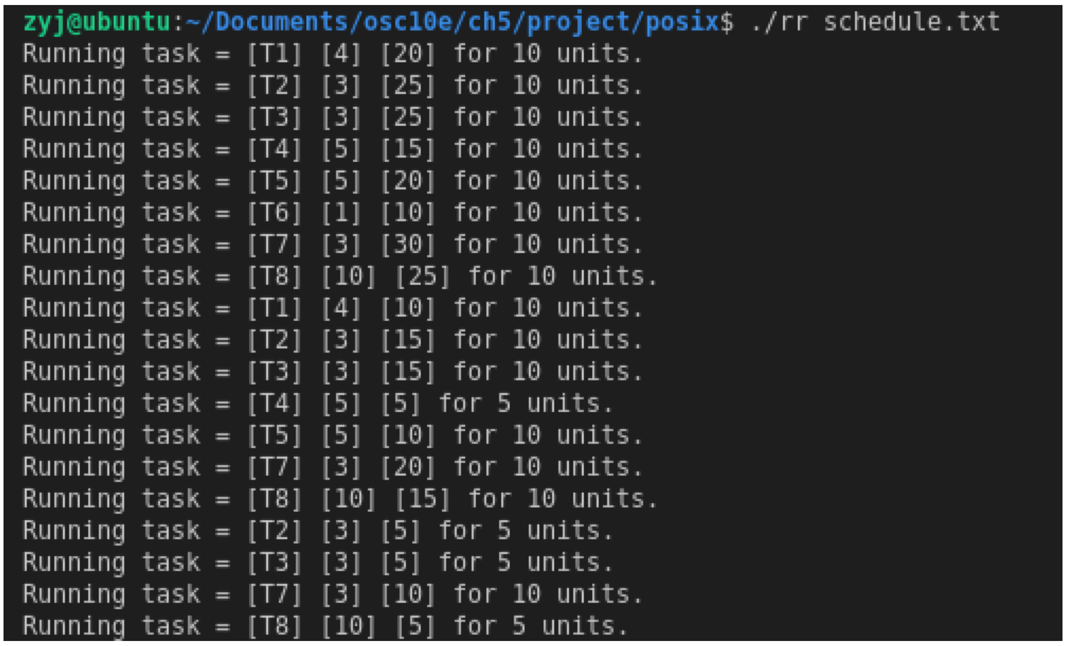
```
 while(current_node!=NULL)
    {
        if(current_node->task->burst <= quantum)
        {
            run(current_node->task, current_node->task->burst);
            struct node *tmp;
            tmp = front_node->next;

            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;

    }
```

6.4 Program Results

Entering:

```
./rr schedule.txt
```

Then we get:



```
zyj@ubuntu:~/Documents/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.
```

7 Priority RR

7.1 Algorithm

Priority RR (**Priority** with **round-robin**) schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority. It is the combination of **Priority** and **Round Robin**.

7.2 add()

This is similar to **add()** in **Priority**.

We insert 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;
    }
}
```

7.3 round_robin()

This function will be used in **schedule()**, to run **Round Robin** .

round_robin() is the same as what we implement in **RR**. We use it again here, since both RR and Priority-RR use **RR**.

Note that this function can **work on a circular link list.** So in **schedule()**, we may transform the link list with the same pirority into a circular link list.

```
void round_robin(struct node **head){
    int quantum = QUANTUM;
    struct node* current_node, *front_node;
    current_node = *head;
    front_node = NULL;
    // get tail node
    while((current_node->next)!=NULL)
    {
        current_node = current_node->next;
    }
    current_node->next = *head;// change into circular list

    current_node = *head;

    //special situation
    if(current_node->next==current_node)
    {
        int time = current_node->task->burst;
        while(time > quantum)
        {
            run(current_node->task, quantum);
            current_node->task->burst -= quantum;
            time = current_node->task->burst;
        }
        run(current_node->task, current_node->task->burst);
        free(current_node);
        return;
    }


    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;

    }
}
```

7.4 schedule()

We seperate the execution of **schedule** into three parts:

1. transform the part of the link list with the same **priority** into a circular list
2. run **round_robin()** on this circular list
3. turn to run the list of next level priority

Then we implement the logic as follows:

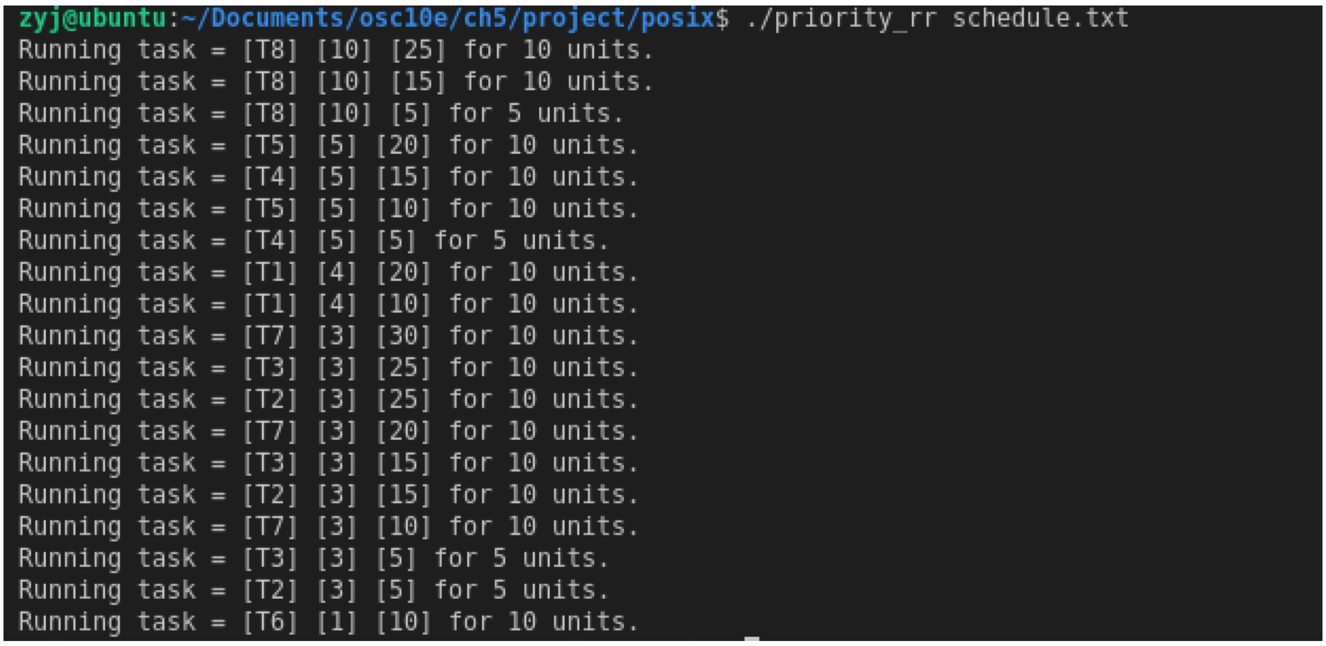
```
void schedule(struct node **head){
    struct node* current_node;
    current_node = *head;

    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
        {
            // a level of priority has been found out
            *head = current_node;
            current_circular_node->next = NULL;
            round_robin(circular_list_head);
            // clear the list
            *circular_list_head = NULL;
            current_circular_node = *circular_list_head;
        }
    }
    if(current_circular_node!=NULL)
    {
        round_robin(circular_list_head);
    }

}
```

7.5 Program Results

Entering:

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

Then we get:



```
zyj@ubuntu:~/Documents/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.
```

8 Makefile

By entering these commands:

```
make fcfs
```

```
make sjf
```

```
make rr
```

```
make priority
```

```
make priority_rr
```

we can complie the corresponding algorithm.

The makefile for this project is shown below.

```
# makefile for scheduling program
#
# make rr - for round-robin scheduling
# make fcfs - for FCFS scheduling
# make sjf - for SJF scheduling
# make priority - for priority scheduling
# make priority_rr - for priority with round robin scheduling

CC=gcc
CFLAGS=-Wall

clean:
  rm -rf *.o
  rm -rf fcfs
  rm -rf sjf
  rm -rf rr
  rm -rf priority
  rm -rf priority_rr

rr: driver.o list.o CPU.o schedule_rr.o
  $(CC) $(CFLAGS) -o rr driver.o schedule_rr.o list.o CPU.o
```

```
sjf: driver.o list.o CPU.o schedule_sjf.o
  $(CC) $(CFLAGS) -o sjf driver.o schedule_sjf.o list.o CPU.o

fcfs: driver.o list.o CPU.o schedule_fcfs.o
  $(CC) $(CFLAGS) -o fcfs driver.o schedule_fcfs.o list.o CPU.o

priority: driver.o list.o CPU.o schedule_priority.o
  $(CC) $(CFLAGS) -o priority driver.o schedule_priority.o list.o CPU.o

schedule_fcfs.o: schedule_fcfs.c
  $(CC) $(CFLAGS) -c schedule_fcfs.c

priority_rr: driver.o list.o CPU.o schedule_priority_rr.o
  $(CC) $(CFLAGS) -o priority_rr driver.o schedule_priority_rr.o list.o
CPU.o

driver.o: driver.c
  $(CC) $(CFLAGS) -c driver.c

schedule_sjf.o: schedule_sjf.c
  $(CC) $(CFLAGS) -c schedule_sjf.c

schedule_priority.o: schedule_priority.c
  $(CC) $(CFLAGS) -c schedule_priority.c

schedule_rr.o: schedule_rr.c
  $(CC) $(CFLAGS) -c schedule_rr.c

list.o: list.c list.h
  $(CC) $(CFLAGS) -c list.c

CPU.o: CPU.c cpu.h
  $(CC) $(CFLAGS) -c CPU.c
```

9 Conclusion and Thoughts

In this project, we go deeper into **Scheduling Algorithms: FCFS, SJF, RR, Priority, Priority-RR** and implement them one by one, which makes us know the features of different algorithms better and deeper.

To implement, we mainly make use of **link list** ,a kind of the classic data structures.

The algorithm we implement can be improved further, since the codes I write are not concise enough. This may be the point where I can progress on further.

In conclusion, I strengthen my comprehension in scheduling algorithms and improve my programming skill in **Project4: Scheduling Algorithms**.