OS Project-4 Report

Scheduling Algorithms

Zhou Yiyuan

516021910270
1025152261@qq.com

1 Introduce

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)
- Shortest-job-first (SJF)
- Priority scheduling
- Round-robin (RR) scheduling
- Priority with round-robin

What's more, there are two further challenges:

- Each task provided to the scheduler is assigned a unique task (tid). If a scheduler is running in an SMP environment where each CPU is separately running its own scheduler, there is a possible race condition on the variable that is used to assign task identifiers. Fix this race condition using an atomic integer.
- Calculate the average turnaround time, waiting time, and response time for each of the scheduling algorithms.

2 Framework

The **main()** function is in the driver.c. In the **main()** function, the tasks to be scheduled are read from a file. For every task, **add()** function is invoked to add it to a tasks group. At last, **schedule()** function is invoked to schedule all tasks in the group. In fact, the essential of a scheduling algorithm is **add()** function and **schedule()** function. The only thing we need to do is to use a type of data structure to store tasks and implement **add()** function and **schedule()** function.

For the first challenge, we can assign tid for a task in the **run()** function of **CPU.c**, in use of **__sync_fetch_and_add()** to avoid race conditions.

For the second challenge, a global variable **clock** is introduced to simulate the CPU time and **startWaitingTime**, the time that the task enters the waiting queue, is maintained for every task. We can calculate the turnaround time, waiting time, and response time in **run(task, slice)** function (slice represent how long the task will run in CPU). For every task,

$$waitingTime = clock - startWaitingTime \tag{1}$$

$$turnaroundTime = slide \tag{2}$$

And the respond time is the waiting time of the task if it enters CPU for first time. The **startWaitingTime** is initialized as 0 because all tasks arrive at the same time as the assumption. Thus, if **startWaitingTime**=0, then

$$respondTime = clock - startWaitingTime \tag{3}$$

For two challenges, we only need to modify the **run()** function in **CPU.c** without changing other codes. And the modified CPU.c is shown as below:

```c
/* CPU.c */
/**
 * "Virtual" CPU that also maintains track of system time.
 */

#include <stdio.h>
#include "task.h"

int tid = 0;

// run this task for the specified time slice
void run(Task *task, int slice) {
    extern int turnaround_time;
    extern int waiting_time;
    extern int respond_time;
    extern int clock;

    if(task->startWaitingTime == 0){
        respond_time += clock - task->startWaitingTime;
    }
    waiting_time += clock - task->startWaitingTime;

    task->tid = __sync_fetch_and_add(&tid, 1);
    // printf("tid = %d \n", task->tid);

    printf("Running task = [%s] [%d] [%d] for %d units.\n",task->name,
     ↪  task->priority, task->burst, slice);

    turnaround_time += slice;
    clock += slice;
    task->startWaitingTime = clock;
}
```

3 Details

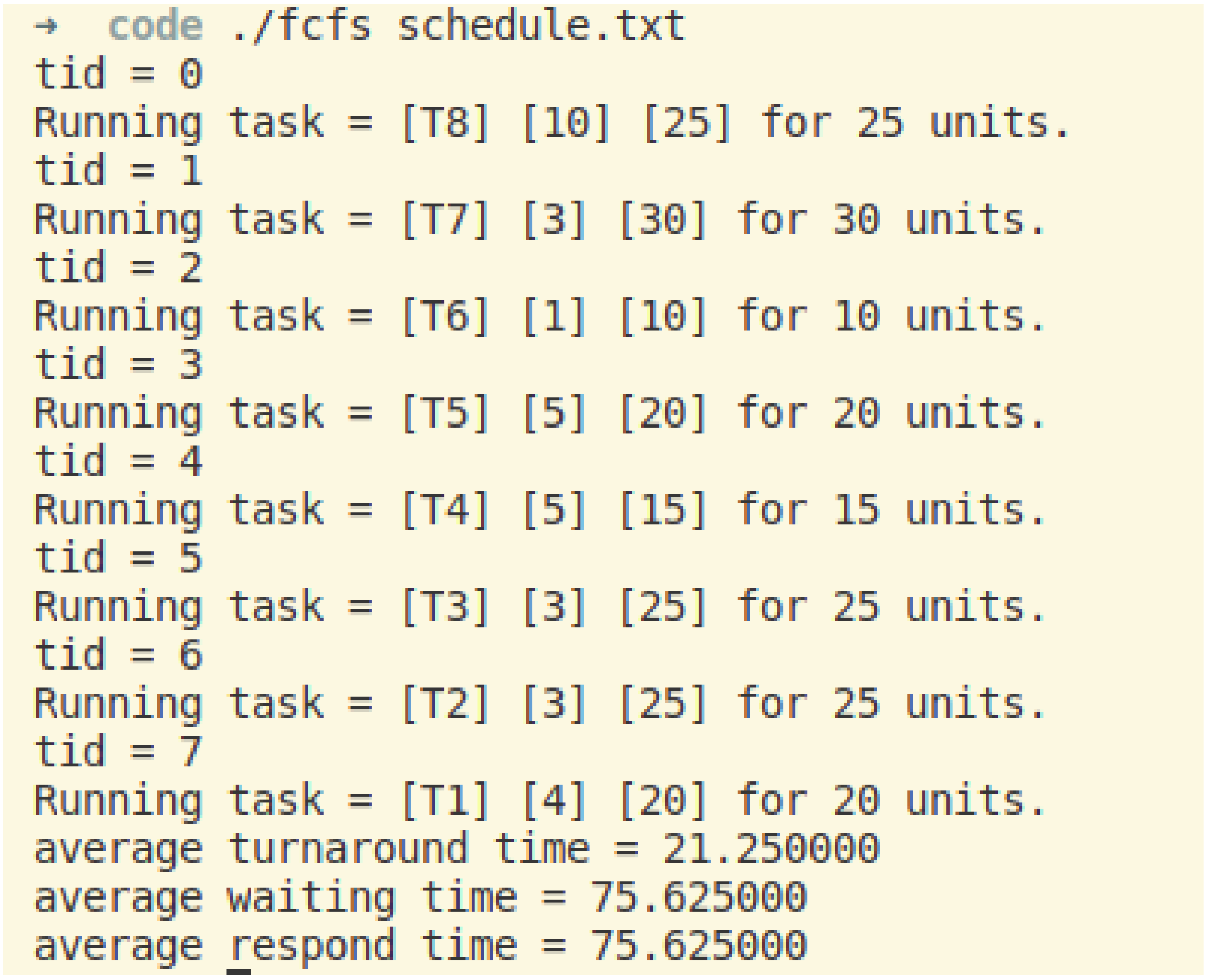
3.1 FCFS

First-come, first-served (FCFS), which schedules tasks in the order in which they request the CPU. As assumption, all tasks arrive at the same time, so the order to run tasks is arbitrary. A linked list is used to store tasks. For simplicity, the new task is inserted to the head of the linked list in **add()** function, and tasks run in the order of the linked list in **schedule()**.

3.1.1 Code

```c
#include <stdlib.h>
#include <stdio.h>

#include "schedulers.h"
#include "list.h"
#include "task.h"
#include "cpu.h"

struct node **taskListHead = NULL;

extern int turnaround_time;
extern int waiting_time;
extern int respond_time;
extern int task_num;

// add a task to the list
void add(char *name, int priority, int burst){
    Task *newTask = malloc(sizeof(Task));

    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;
    newTask->startWaitingTime = 0;

    if(taskListHead == NULL){
        taskListHead = malloc(sizeof(struct node *));
    }

    insert(taskListHead, newTask);

    task_num ++;
}

// invoke the scheduler
void schedule(){
    struct node *temp;

    if(taskListHead == NULL){
        temp = NULL;
    }else{
        temp = *taskListHead;
    }

    while (temp != NULL) {
        run(temp->task, temp->task->burst);
        // printf("[%s] [%d] [%d]\n",temp->task->name, temp->task->priority,
        ↪  temp->task->burst);
        temp = temp->next;
    }

    printf("average turnaround time = %f\n", turnaround_time / (float)task_num);
    printf("average waiting time = %f\n", waiting_time / (float)task_num);
    printf("average respond time = %f\n", respond_time / (float)task_num);
}
```

3.1.2 Result



```
➜  code ./fcfs schedule.txt
tid = 0
Running task = [T8] [10] [25] for 25 units.
tid = 1
Running task = [T7] [3] [30] for 30 units.
tid = 2
Running task = [T6] [1] [10] for 10 units.
tid = 3
Running task = [T5] [5] [20] for 20 units.
tid = 4
Running task = [T4] [5] [15] for 15 units.
tid = 5
Running task = [T3] [3] [25] for 25 units.
tid = 6
Running task = [T2] [3] [25] for 25 units.
tid = 7
Running task = [T1] [4] [20] for 20 units.
average turnaround time = 21.250000
average waiting time = 75.625000
average respond time = 75.625000
```

Figure 1: The result of FCFS

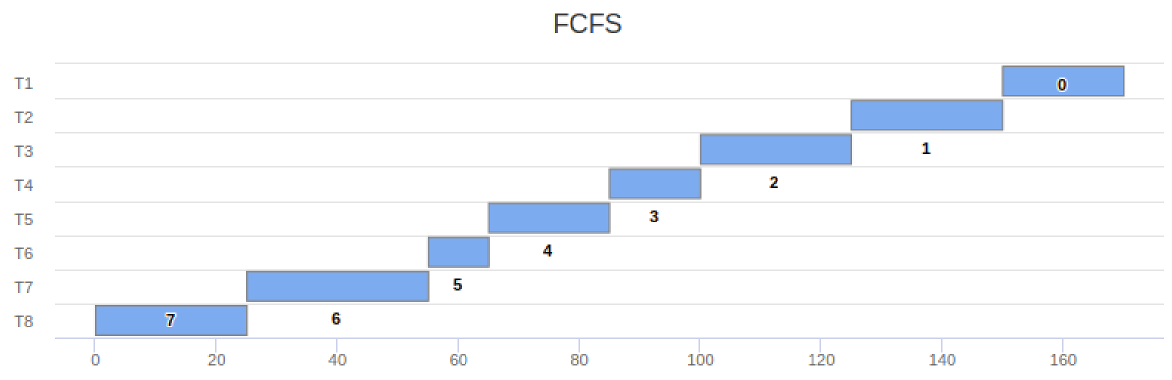


Figure 2: The Gantt Chart of FCFS

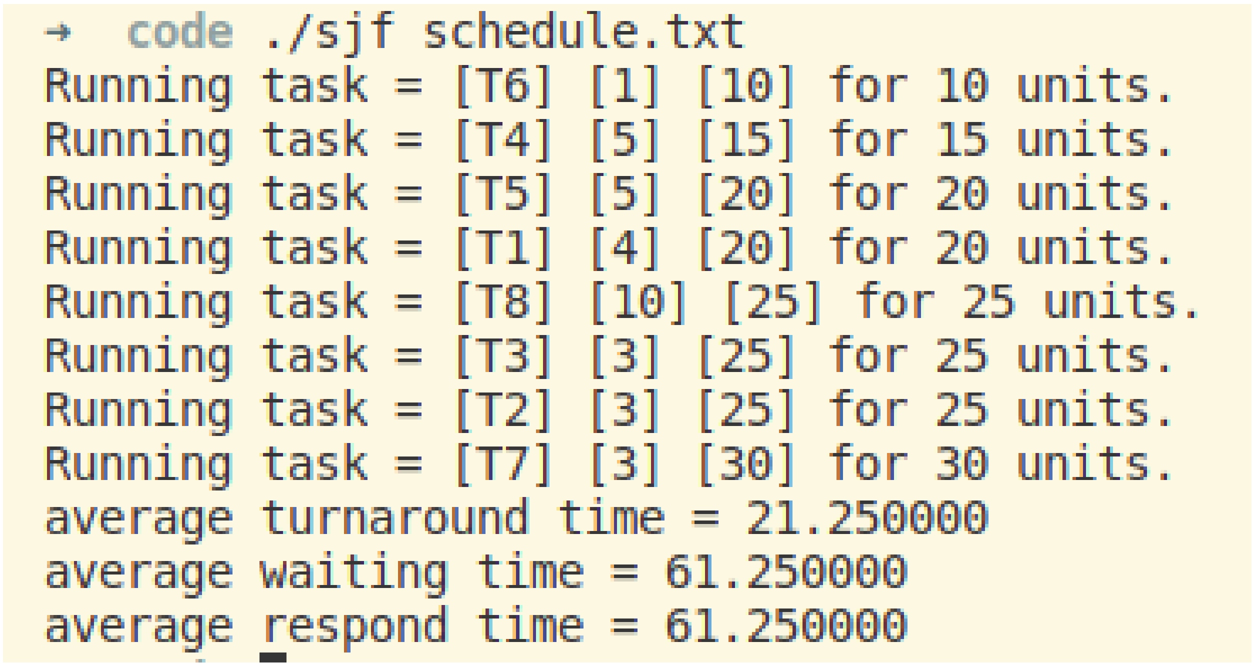
3.2 SJF

Shortest-job-first (SJF), which schedules tasks in order of the length of the tasks' next CPU burst. An ordered linked is used to store tasks, which means the new task is inserted to the linked list in order of length of the burst in **add()**. Then tasks run in the order of the linked list.

3.2.1 Code

```c
#include <stdlib.h>
#include <stdio.h>

#include "schedulers.h"
#include "list.h"
#include "task.h"
#include "cpu.h"

struct node **taskListHead = NULL;

extern int turnaround_time;
extern int waiting_time;
extern int respond_time;
extern int task_num;

// add a task to the list
void add(char *name, int priority, int burst){
    Task *newTask = malloc(sizeof(Task));

    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;
    if(taskListHead == NULL){
        taskListHead = malloc(sizeof(struct node *));
        *taskListHead = NULL;
    }

    struct node *temp = *taskListHead;
    struct node *prev = *taskListHead;

    while(1){
        if(temp != NULL && temp->task->burst < burst){
            prev = temp;
            temp = temp->next;
        }else{
            struct node *newNode = malloc(sizeof(struct node));
            newNode->task = newTask;
            newNode->next = temp;

            if(prev == temp){ //insert into the head
                *taskListHead = newNode;
            }else
            {
                prev->next = newNode;
            }
            break;
        }
    }

    task_num ++;
}

// invoke the scheduler
void schedule(){
    struct node *temp;

    if(taskListHead == NULL){
        temp = NULL;
    }else{
        temp = *taskListHead;
    }

    while (temp != NULL) {
        run(temp->task, temp->task->burst);
        // printf("[%s] [%d] [%d]\n",temp->task->name, temp->task->priority,
        ↪  temp->task->burst);
        temp = temp->next;
    }
    printf("average turnaround time = %f\n", turnaround_time / (float)task_num);
    printf("average waiting time = %f\n", waiting_time / (float)task_num);
    printf("average respond time = %f\n", respond_time / (float)task_num);
}
```

3.2.2 Result



```
→  code ./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.
average turnaround time = 21.250000
average waiting time = 61.250000
average respond time = 61.250000
```

Figure 3: The result of SJF

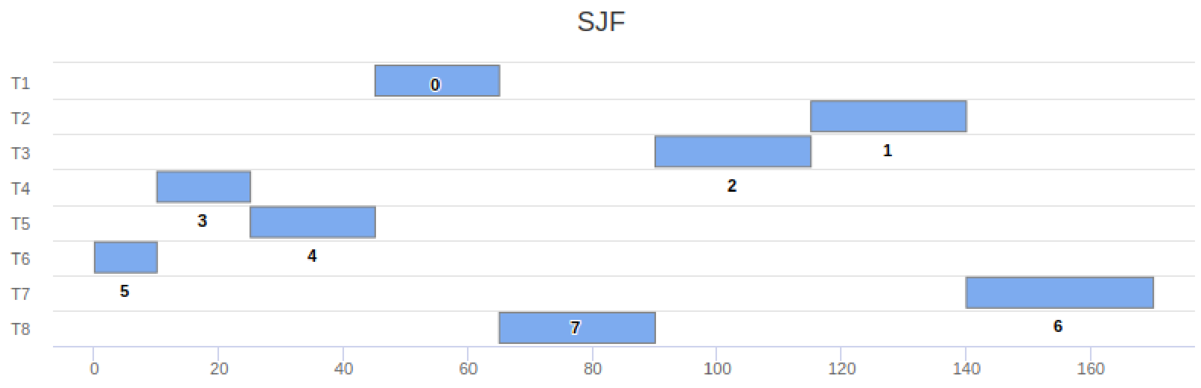


Figure 4: The Gantt Chart of SJF

3.3 Priority Scheduling

Priority scheduling, which schedules tasks based on priority. Similarly, an ordered linked based on priority is used to store tasks, which means the new task is inserted to the linked list in order of its priority in **add()**. Then tasks run in the order of the linked list.

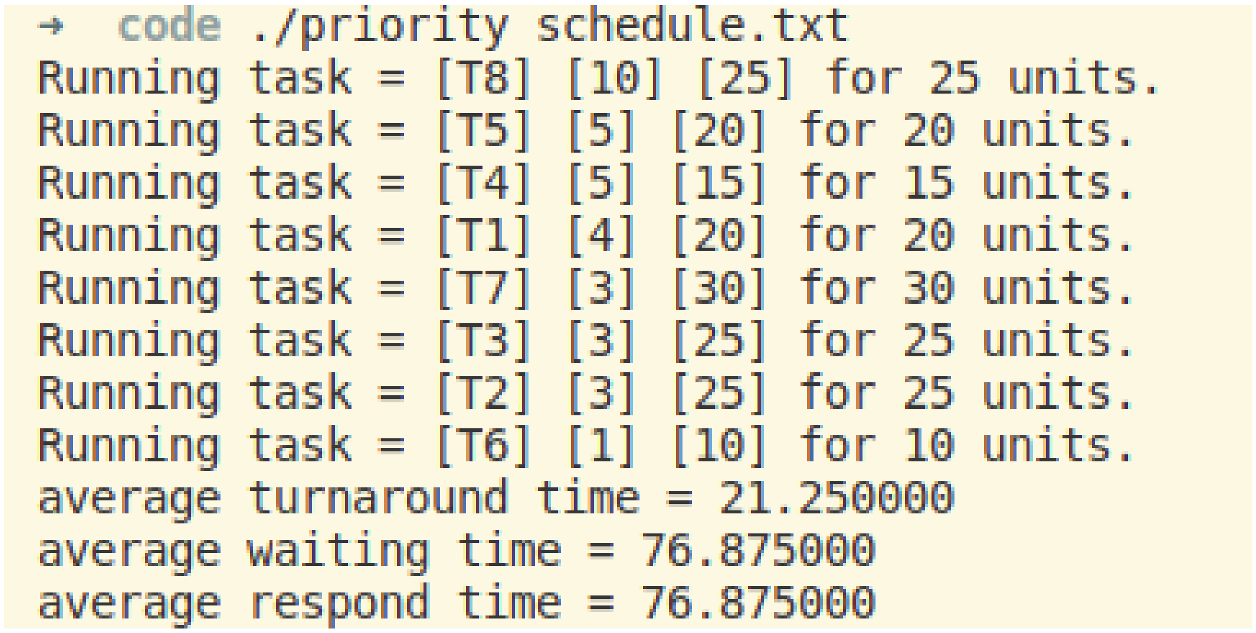
3.3.1 Code

```
/*
a higher numeric value indicates a higher relative priority
*/

#include <stdlib.h>
#include <stdio.h>

#include "schedulers.h"
#include "list.h"
#include "task.h"
#include "cpu.h"

struct node **taskListHead = NULL;

extern int turnaround_time;
extern int waiting_time;
extern int respond_time;
extern int task_num;

// add a task to the list
void add(char *name, int priority, int burst){
    Task *newTask = malloc(sizeof(Task));

    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;

    if(taskListHead == NULL){
        taskListHead = malloc(sizeof(struct node *));
        *taskListHead = NULL;
    }

    struct node *temp = *taskListHead;
    struct node *prev = *taskListHead;

    while(1){
        if(temp != NULL && temp->task->priority > priority){
            prev = temp;
            temp = temp->next;
        }else{
            struct node *newNode = malloc(sizeof(struct node));
            newNode->task = newTask;
            newNode->next = temp;

            if(prev == temp){ //insert into the head
                *taskListHead = newNode;
            }else{
                prev->next = newNode;
            }
            break;
        }
    }

    task_num ++;
}

// invoke the scheduler
void schedule(){
    struct node *temp;

    if(taskListHead == NULL){
        temp = NULL;
    }else{
        temp = *taskListHead;
    }

    while (temp != NULL) {
        run(temp->task, temp->task->burst);
        // printf("[%s] [%d] [%d]\n",temp->task->name, temp->task->priority,
         ↪  temp->task->burst);
        temp = temp->next;
    }

    printf("average turnaround time = %f\n", turnaround_time / (float)task_num);
    printf("average waiting time = %f\n", waiting_time / (float)task_num);
    printf("average respond time = %f\n", respond_time / (float)task_num);
}
```

3.3.2 Result

3.4 RR

Round-robin (RR) scheduling, where each task is run for a time quantum (or for the remainder of its CPU burst). For simplicity, a linked list is used to store tasks and the new task is inserted to the head in **add()**. In **schedule()**, tasks run in the order of the linked list with the following



```
➜  code ./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.
average turnaround time = 21.250000
average waiting time = 76.875000
average respond time = 76.875000
```

Figure 5: The result of priority scheduling

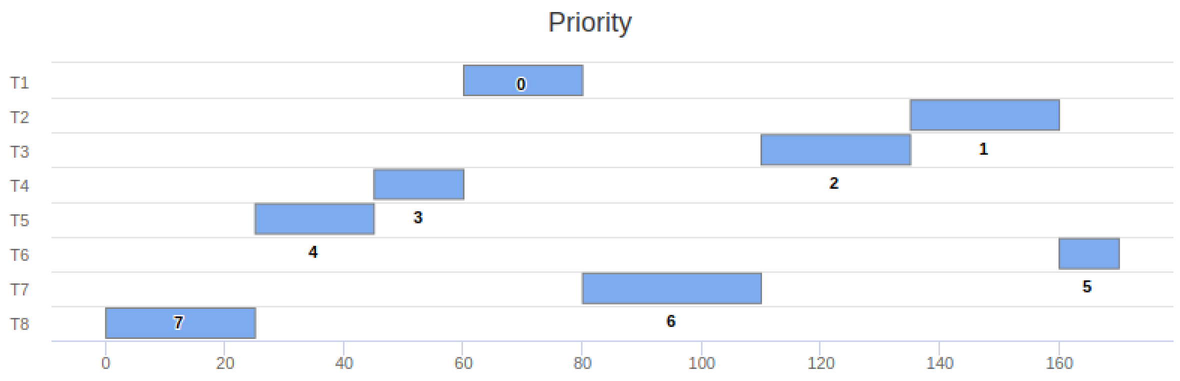


Figure 6: The Gantt Chart of priority scheduling

rules:

- If the task's burst is larger than the quantum, the task runs for a quantum and its burst minus a quantum.
- If the task's burst is smaller than the quantum, the task runs for the remainder of its CPU burst and it will be removed.
- After the rail task runs, the head task will run.
- The running circle will terminate when the linked list is empty.

3.4.1 Code

```c
#include <stdlib.h>
#include <stdio.h>

#include "schedulers.h"
#include "list.h"
#include "task.h"
#include "cpu.h"

struct node **taskListHead = NULL;

extern int turnaround_time;
extern int waiting_time;
extern int respond_time;
extern int task_num;

// add a task to the list
void add(char *name, int priority, int burst){
    Task *newTask = malloc(sizeof(Task));

    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;

    if(taskListHead == NULL){
        taskListHead = malloc(sizeof(struct node *));
    }

    insert(taskListHead, newTask);

    task_num ++;
}

// invoke the scheduler
void schedule(){
    struct node *temp;

    if(taskListHead == NULL){
        temp = NULL;
    }else{
        temp = *taskListHead;
    }

    while (temp != NULL) {
        //run(temp->task, temp->task->burst);
        if(temp->task->burst > QUANTUM){
            run(temp->task, QUANTUM);
            temp->task->burst -= QUANTUM;
        }else
        {
            run(temp->task, temp->task->burst);
            delete(taskListHead, temp->task);
        }

        temp = temp->next;

        if(temp == NULL){ //When reach the rail, return to the head
            temp = *taskListHead;
        }
    }

    printf("average turnaround time = %f\n", turnaround_time / (float)task_num);
    printf("average waiting time = %f\n", waiting_time / (float)task_num);
    printf("average respond time = %f\n", respond_time / (float)task_num);
}
```

3.4.2 Result

3.5 Priority RR Scheduling

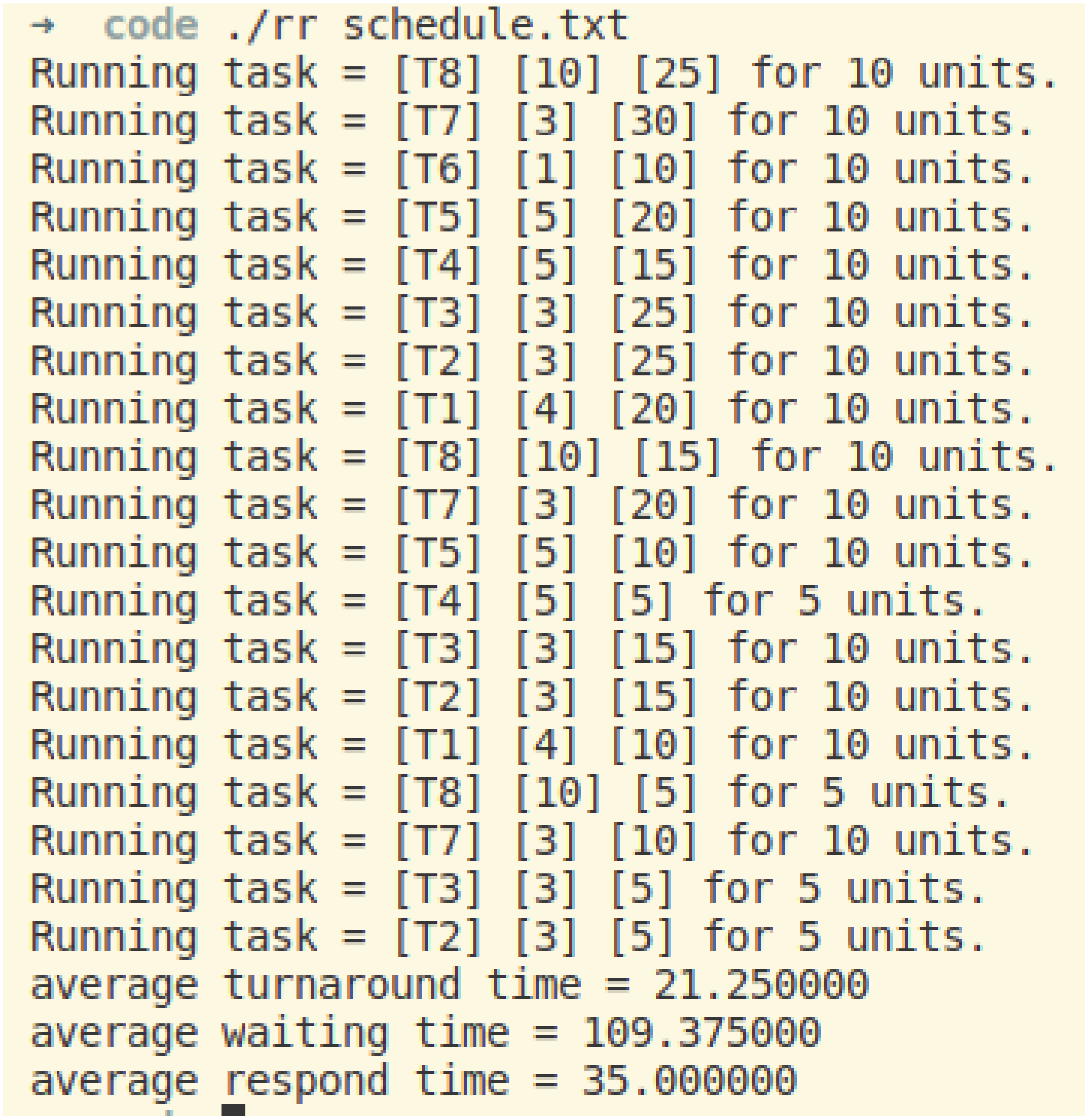
Priority with round-robin, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority. A double nested linked list is used to store tasks. Specifically, a linked lists is used to store tasks with the same priority and another link list is used to store these linked lists in the order of priority.

In **add()**, the new task will be inserted into the head of correspondent linked list if the linked list with the same priority exists, otherwise a new linked list with the priority will be created and the task will be inserted into its head.

In **schedule()**, linked lists will be traversed in order of priority and tasks in every linked lists run as RR scheduling.

3.5.1 Code

```
/*
a higher numeric value indicates a higher relative priority
*/

#include <stdlib.h>
#include <stdio.h>

#include "schedulers.h"
#include "list.h"
```



```
➜  code ./rr schedule.txt
Running task = [T8] [10] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T8] [10] [15] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T8] [10] [5] for 5 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.
average turnaround time = 21.250000
average waiting time = 109.375000
average respond time = 35.000000
```

Figure 7: The result of RR scheduling

```c
#include "task.h"
#include "cpu.h"

//represent a list of tasks with the same priority
typedef struct priority_node {
    struct node *head;
    int priority;
```

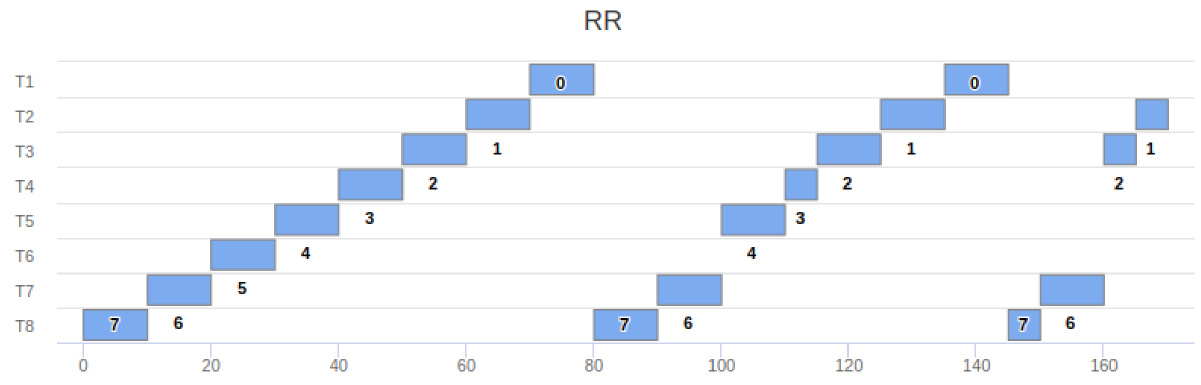
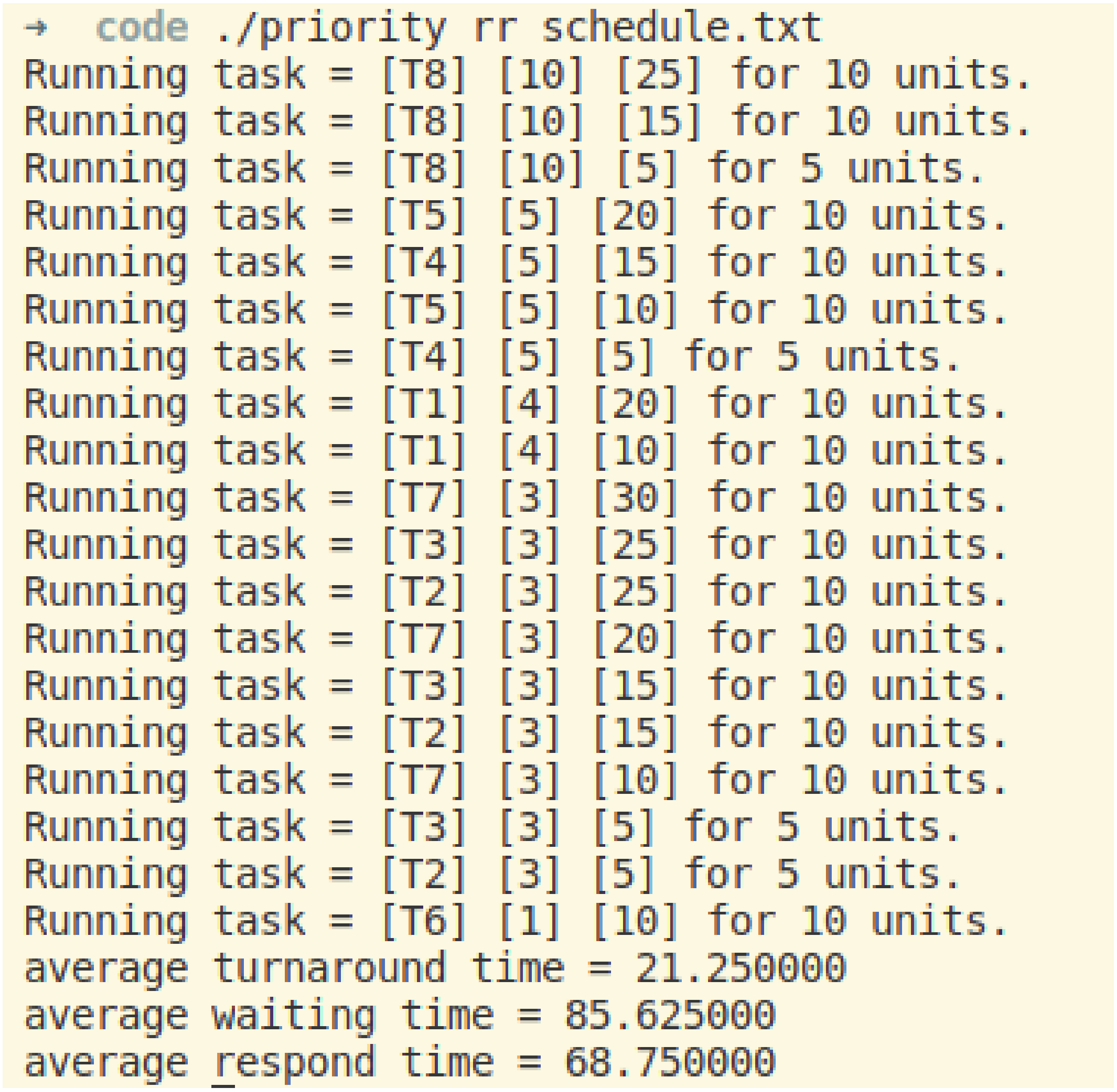


Figure 8: The Gantt Chart of RR scheduling

```
    struct priority_node *next;
} PriNode;

PriNode *taskListHead = NULL;

// add a task to the list
void add(char *name, int priority, int burst){
    Task *newTask = malloc(sizeof(Task));

    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;

    PriNode *pri_temp = taskListHead;
    PriNode *pri_prev = taskListHead;

    while(1){
        if(pri_temp != NULL && pri_temp->priority > priority){
            pri_prev = pri_temp;
            pri_temp = pri_temp->next;
        }else{
            struct node *newNode = malloc(sizeof(struct node));
            newNode->task = newTask;
            newNode->next = NULL;

            if(pri_temp != NULL && pri_temp->priority == priority){
                newNode->next = pri_temp->head;
                pri_temp->head = newNode;
            }else{
                PriNode *newPriNode = malloc(sizeof(PriNode));
                newPriNode->head = newNode;
                newPriNode->priority = priority;
                newPriNode->next = pri_temp;

                if(pri_prev == pri_temp){
                    taskListHead = newPriNode;
                }else{
                    pri_prev->next = newPriNode;
                }
            }
            break;
        }
    }

    task_num ++;
}

// invoke the scheduler
void schedule(){
    PriNode *pri_temp = taskListHead;
    struct node *temp;

    while(pri_temp != NULL){
        temp = pri_temp->head;

        while(temp != NULL){
            if(temp->task->burst > QUANTUM){
                run(temp->task, QUANTUM);
                temp->task->burst -= QUANTUM;
            }else
            {
                run(temp->task, temp->task->burst);
                delete(&(pri_temp->head), temp->task);
            }

            temp = temp->next;

            if(temp == NULL){ //When reach the rail, return to the head
                temp = pri_temp->head;
            }
        }

        pri_temp = pri_temp->next;
    }

    printf("average turnaround time = %f\n", turnaround_time / (float)task_num);
    printf("average waiting time = %f\n", waiting_time / (float)task_num);
    printf("average respond time = %f\n", respond_time / (float)task_num);
```

```
}
```

3.5.2 Result



```
➜  code ./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.
average turnaround time = 21.250000
average waiting time = 85.625000
average respond time = 68.750000
```

Figure 9: The result of priority RR scheduling

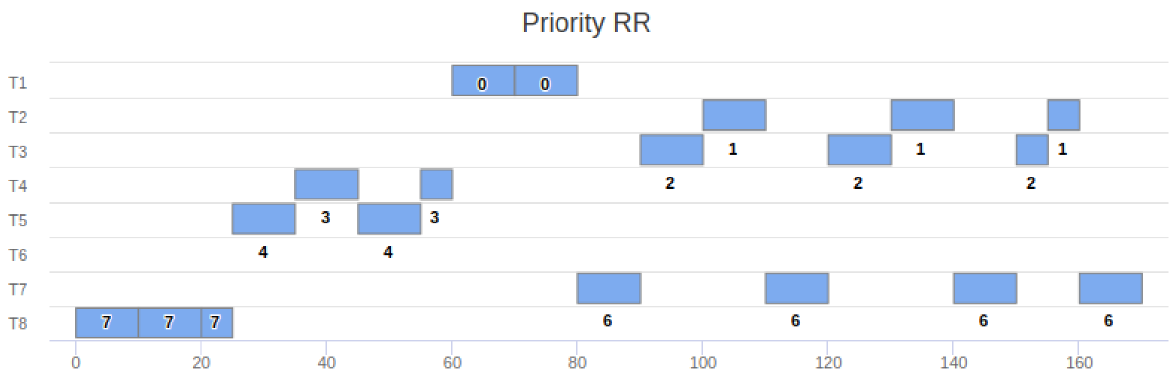


Figure 10: The Gantt Chart of priority RR scheduling