Project 4: Scheduling Algorithms

CS307-Operating System (D), CS356-Operating System Course Design, Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang)   Student ID: 518030910150   Email: galaxies@sjtu.edu.cn

1 Scheduling Algorithms

1.1 Requirements

In this project, we are required to implement several different process scheduling algorithms. The scheduler will be assigned as a predefined set of tasks and will schedule the task based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduing 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 the 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 the 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 implementation of this project may be completed in either C or Java, and the program files supporting both of these languages are provided in the source code download for the textbook. These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form ***[task name] [priority] [CPU burst]***.

There are a few different strategies for organizing the list of tasks. One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling algorithm. You are likely to find the functionality of a general list to be more suitable when completing this project.

Further Challenges: two additional challenges are presented for this project:

- **a.** Each task provided to the scheduler is assigned a unique task (`tid`). If a scheduler is running in a 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.
- **b.** Calculate the average turnaround time, waiting time and response time for each of the scheduling algorithms.

1.2 Solutions

I use C programming language to implement this project, and I also complete the further challenges in my code, which I'll explain later. We will use the data stored in `schedule.txt` to test our algorithms.

1.2.1 FCFS Scheduler

Here are the brief methods about implementing FCFS scheduler.

- The introduction of FCFS scheduling algorithm is in textbook, and we don't discuss the details of the algorithm here. You can refer to textbook to get more information.
- To complete the FCFS scheduler, we need to find out the first-come process in the process list. Since we add every new process in the beginning of our process list in `insert(...)` function in `list.c`, the first-come process is in the end of the list. Therefore we can find the last process of the list and make it the next task to complete.
- To solve the further challenge **a**, we add a task identifiers variable `tid_value` in code `task.h` and `task.c`, and we use the atomic function `__sync_fetch_and_add(&x, y)` provided by Linux to solve the racing conditions. This function is atomic and has the functionality of adding the variable `x` by `y` and returning the value of the original `x`. Therefore, we can use it to update `tid_value` and allocate task identifiers.
- We also make a few modifications to the `Makefile` file because we add `task.c` to implement the task identifer. You can refer to the code for details.
- To solve the further challenge **b**, we add some extra variables in the definition of `struct task`. The extra variables record the arrival time, the waiting time, the time of the last execution, the response time and the turnaround time of the current task. And we update these variables when the task is executed. Therefore, when we finish the task, we can add the counting elements into global counting elements to calculate the average waiting time, average response time and average turnaround time. When all the tasks are finished, we will print these statistics in the screen.

Here is the specific implementation of the FCFS scheduler. (`schedule_fcfs.c`).

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

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

struct node *head = NULL;

// Extra variables for calculating the response time, etc.
// |-- current time.
int time = 0;
// |-- task count.
int tsk_cnt = 0;
// |-- total waiting time.
int tot_wait_time = 0;
// |-- total response time.
int tot_resp_time = 0;
// |-- total turnaround time.
int tot_turn_time = 0;
```

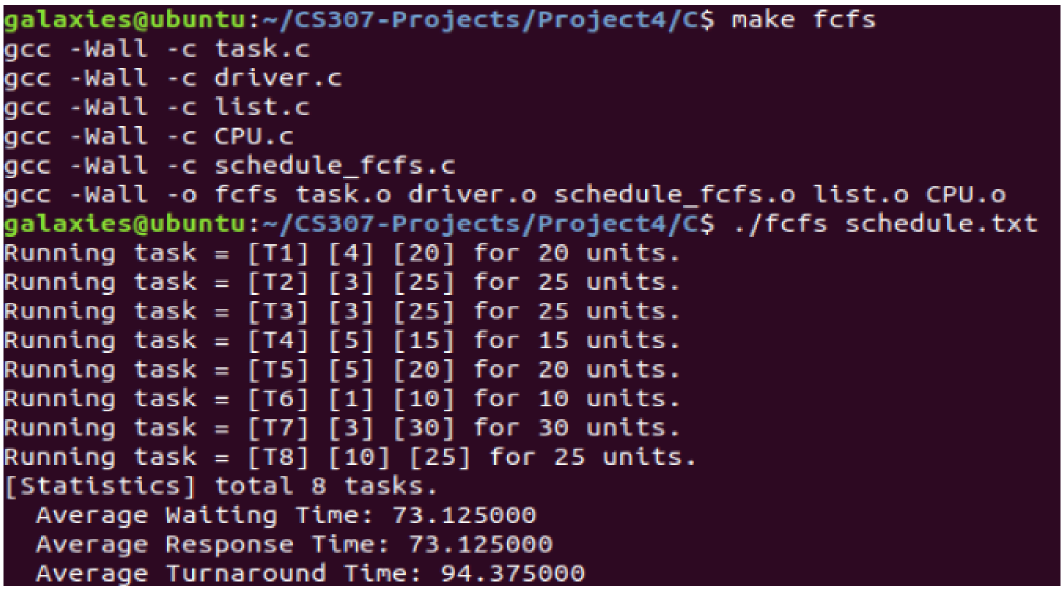
```

void add(char *name, int priority, int burst) {
  Task *tsk;
  tsk = (Task *) malloc (sizeof(Task));
  // avoid racing conditions on tid_value .
  tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
  tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
  strcpy(tsk -> name, name);
  tsk -> priority = priority;
  tsk -> burst = burst;
  // Record some extra parameters here to calculate the response time, etc.
  tsk -> arri_time = time;
  tsk -> last_exec = time;
  tsk -> wait_time = 0;
  tsk -> resp_time = 0;
  tsk -> turn_time = 0;
  insert(&head, tsk);
}

int schedule_next() {
  if (head == NULL) return 1;

  struct node *cur = head;
  while (cur -> next != NULL) cur = cur -> next;

  Task *tsk = cur -> task;
  run(tsk, tsk -> burst);
  delete(&head, tsk);

  time += tsk -> burst;

  // Update some parameters.
  // |-- last waiting time.
  int last_wait_time = time - tsk -> last_exec - tsk -> burst;
  // |-- update waiting time.
  tsk -> wait_time += last_wait_time;
  // |-- update response time.
  if (tsk -> last_exec == tsk -> arri_time)
    tsk -> resp_time = last_wait_time;
  // |-- update last execution time.
  tsk -> last_exec = time;

  // The task is finished, update more parameters.
  // |-- update turnaround time.
  tsk -> turn_time = time - tsk -> arri_time;

  // Update final counting parameters.
  tsk_cnt += 1;
  tot_wait_time += tsk -> wait_time;
  tot_resp_time += tsk -> resp_time;
```

```c
  tot_turn_time += tsk -> turn_time;

  free(tsk -> name);
  free(tsk);
  return 0;
}

void print_statistics() {
  printf("[Statistics] total %d tasks.\n", tsk_cnt);
  double avg_wait_time = 1.0 * tot_wait_time / tsk_cnt;
  double avg_resp_time = 1.0 * tot_resp_time / tsk_cnt;
  double avg_turn_time = 1.0 * tot_turn_time / tsk_cnt;
  printf(" Average Waiting Time: %.6lf\n", avg_wait_time);
  printf(" Average Response Time: %.6lf\n", avg_resp_time);
  printf(" Average Turnaround Time: %.6lf\n", avg_turn_time);
}

void schedule() {
  while (schedule_next() == 0);
  print_statistics();
}
```

You can enter these instructions to test the FCFS scheduler.

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

Here is the execution result of the program (Fig. 1).



```
galaxies@ubuntu:~/CS307-Projects/Project4/C$ make fcfs
gcc -Wall -c task.c
gcc -Wall -c driver.c
gcc -Wall -c list.c
gcc -Wall -c CPU.c
gcc -Wall -c schedule_fcfs.c
gcc -Wall -o fcfs task.o driver.o schedule_fcfs.o list.o CPU.o
galaxies@ubuntu:~/CS307-Projects/Project4/C$ ./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.
[Statistics] total 8 tasks.
  Average Waiting Time: 73.125000
  Average Response Time: 73.125000
  Average Turnaround Time: 94.375000
```

Figure 1: The execution result of FCFS scheduler

1.2.2 SJF Scheduler

Here are the brief methods about implementing SJF scheduler.

- The introduction of SJF scheduling algorithm is in textbook, and we don't discuss the details of the algorithm here. You can refer to textbook to get more information.
- To complete the SJF scheduler, we need to find out the shortest-burst-time process in the process list. Therefore we need to traverse the list to find out the shortest task and make it the next task to execute.
- We use the similar method as the FCFS scheduler to solve the further challenge **a** and **b**. You can refer to the corresponding section for details.

Here is the specific implementation of the SJF scheduler. (`schedule_sjf.c`).

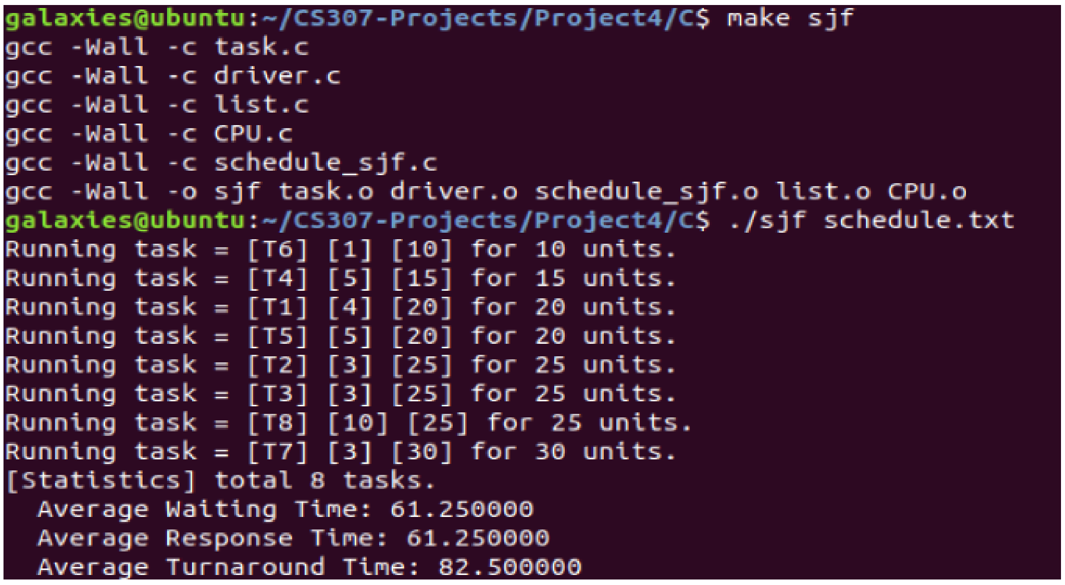
```c
# include <stdio.h>
# include <stdlib.h>
# include <string.h>

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

struct node *head = NULL;

// Extra variables for calculating the response time, etc.
// |-- current time.
int time = 0;
// |-- task count.
int tsk_cnt = 0;
// |-- total waiting time.
int tot_wait_time = 0;
// |-- total response time.
int tot_resp_time = 0;
// |-- total turnaround time.
int tot_turn_time = 0;

void add(char *name, int priority, int burst) {
  Task *tsk;
  tsk = (Task *) malloc (sizeof(Task));
  // avoid racing conditions on tid_value.
  tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
  tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
  strcpy(tsk -> name, name);
  tsk -> priority = priority;
  tsk -> burst = burst;
  // Record some extra parameters here to calculate the response time, etc.
  tsk -> arri_time = time;
  tsk -> last_exec = time;
  tsk -> wait_time = 0;
  tsk -> resp_time = 0;
  tsk -> turn_time = 0;
  insert(&head, tsk);
}
```

```c

int schedule_next() {
  if (head == NULL) return 1;

  struct node *cur = head, *pos = head -> next;
  while (pos != NULL) {
    if (pos -> task -> burst <= cur -> task -> burst) cur = pos;
    pos = pos -> next;
  }

  Task *tsk = cur -> task;
  run(tsk, tsk -> burst);
  delete(&head, tsk);

  time += tsk -> burst;

  // Update some parameters.
  // |-- last waiting time.
  int last_wait_time = time - tsk -> last_exec - tsk -> burst;
  // |-- update waiting time.
  tsk -> wait_time += last_wait_time;
  // |-- update response time.
  if (tsk -> last_exec == tsk -> arri_time)
    tsk -> resp_time = last_wait_time;
  // |-- update last execution time.
  tsk -> last_exec = time;

  // The task is finished, update more parameters.
  // |-- update turnaround time.
  tsk -> turn_time = time - tsk -> arri_time;

  // Update final counting parameters.
  tsk_cnt += 1;
  tot_wait_time += tsk -> wait_time;
  tot_resp_time += tsk -> resp_time;
  tot_turn_time+= tsk -> turn_time;

  free(tsk -> name);
  free(tsk);
  return 0;
}

void print_statistics() {
  printf("[Statistics] total %d tasks.\n", tsk_cnt);
  double avg_wait_time = 1.0 * tot_wait_time / tsk_cnt;
  double avg_resp_time = 1.0 * tot_resp_time / tsk_cnt;
  double avg_turn_time = 1.0 * tot_turn_time / tsk_cnt;
  printf(" Average Waiting Time: %.6lf\n", avg_wait_time);
  printf(" Average Response Time: %.6lf\n", avg_resp_time);
  printf(" Average Turnaround Time: %.6lf\n", avg_turn_time);
}

void schedule() {
  while (schedule_next() == 0);
  print_statistics();
}
```

You can enter these instructions to test the SJF scheduler.

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

Here is the execution result of the program (Fig. 2).



```
galaxies@ubuntu:~/CS307-Projects/Project4/C$ make sjf
gcc -Wall -c task.c
gcc -Wall -c driver.c
gcc -Wall -c list.c
gcc -Wall -c CPU.c
gcc -Wall -c schedule_sjf.c
gcc -Wall -o sjf task.o driver.o schedule_sjf.o list.o CPU.o
galaxies@ubuntu:~/CS307-Projects/Project4/C$ ./sjf schedule.txt
Running task = [T6] [1] [10] for 10 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T8] [10] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
[Statistics] total 8 tasks.
  Average Waiting Time: 61.250000
  Average Response Time: 61.250000
  Average Turnaround Time: 82.500000
```

Figure 2: The execution result of SJF scheduler

1.2.3 Priority Scheduler

Here are the brief methods about implementing priority scheduler.

- The introduction of priority-based scheduling algorithm is in textbook, and we don't discuss the details of the algorithm here. You can refer to textbook to get more information.
- To complete the priority scheduler, we need to find out the highest-priority process in the process list. Therefore we need to traverse the list to find out the task with the highest priority and make it the next task to execute.
- We use the similar method as the FCFS scheduler to solve the further challenge **a** and **b**. You can refer to the corresponding section for details.

Here is the specific implementation of the priority scheduler. (`schedule_priority.c`).

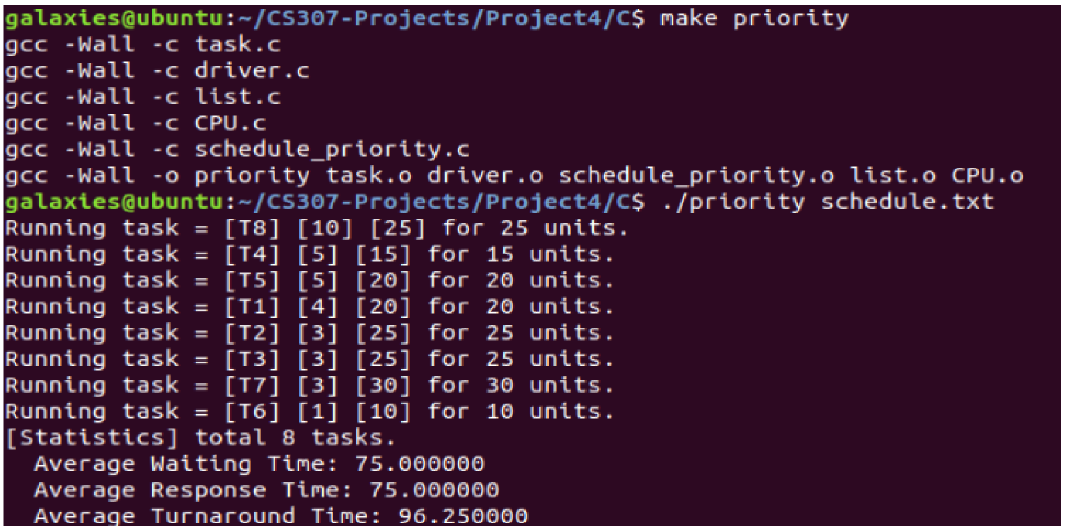
```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>

# include "task.h"
```

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

struct node *head = NULL;

// Extra variables for calculating the response time, etc.
// |-- current time.
int time = 0;
// |-- task count.
int tsk_cnt = 0;
// |-- total waiting time.
int tot_wait_time = 0;
// |-- total response time.
int tot_resp_time = 0;
// |-- total turnaround time.
int tot_turn_time = 0;

void add(char *name, int priority, int burst) {
  Task *tsk;
  tsk = (Task *) malloc (sizeof(Task));
  // avoid racing conditions on tid_value .
  tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
  tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
  strcpy(tsk -> name, name);
  tsk -> priority = priority;
  tsk -> burst = burst;
  // Record some extra parameters here to calculate the response time, etc.
  tsk -> arri_time = time;
  tsk -> last_exec = time;
  tsk -> wait_time = 0;
  tsk -> resp_time = 0;
  tsk -> turn_time = 0;
  insert(&head, tsk);
}

int schedule_next() {
  if (head == NULL) return 1;

  struct node *cur = head, *pos = head -> next;
  while (pos != NULL) {
    if (pos -> task -> priority >= cur -> task -> priority) cur = pos;
    pos = pos -> next;
  }

  Task *tsk = cur -> task;
  run(tsk, tsk -> burst);
  delete(&head, tsk);

  time += tsk -> burst;

  // Update some parameters.
  // |-- last waiting time.
  int last_wait_time = time - tsk -> last_exec - tsk -> burst;
  // |-- update waiting time.
  tsk -> wait_time += last_wait_time;
  // |-- update response time.
  if (tsk -> last_exec == tsk -> arri_time)
    tsk -> resp_time = last_wait_time;
  // |-- update last execution time.
  tsk -> last_exec = time;

  // The task is finished , update more parameters.
  // |-- update turnaround time.
  tsk -> turn_time = time - tsk -> arri_time;

  // Update final counting parameters.
  tsk_cnt += 1;
  tot_wait_time += tsk -> wait_time;
  tot_resp_time += tsk -> resp_time;
  tot_turn_time+= tsk -> turn_time;

  free(tsk -> name);
  free(tsk);
  return 0;
}

void print_statistics() {
  printf("[Statistics] total %d tasks.\n", tsk_cnt);
  double avg_wait_time = 1.0 * tot_wait_time / tsk_cnt;
  double avg_resp_time = 1.0 * tot_resp_time / tsk_cnt;
  double avg_turn_time = 1.0 * tot_turn_time / tsk_cnt;
  printf(" Average Waiting Time: %.6lf\n", avg_wait_time);
  printf(" Average Response Time: %.6lf\n", avg_resp_time);
  printf(" Average Turnaround Time: %.6lf\n", avg_turn_time);
}

void schedule() {
  while (schedule_next() == 0);
  print_statistics();
}
```

You can enter these instructions to test the priority scheduler.

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

Here is the execution result of the program (Fig. 3).



```
galaxies@ubuntu:~/CS307-Projects/Project4/C$ make priority
gcc -Wall -c task.c
gcc -Wall -c driver.c
gcc -Wall -c list.c
gcc -Wall -c CPU.c
gcc -Wall -c schedule_priority.c
gcc -Wall -o priority task.o driver.o schedule_priority.o list.o CPU.o
galaxies@ubuntu:~/CS307-Projects/Project4/C$ ./priority schedule.txt
Running task = [T8] [10] [25] for 25 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 = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T6] [1] [10] for 10 units.
[Statistics] total 8 tasks.
  Average Waiting Time: 75.000000
  Average Response Time: 75.000000
  Average Turnaround Time: 96.250000
```

Figure 3: The execution result of priority scheduler

1.2.4 RR Scheduler

Here are the brief methods about implementing RR scheduler.

- The introduction of RR (Round-Robin) scheduling algorithm is in textbook, and we don't discuss the details of the algorithm here. You can refer to textbook to get more information.
- To complete the RR scheduler, we need to execute the tasks in a certain turn. We execute the current task for a time slice, and if the current task is not finished, we will put it into the beginning of the list, waiting for the next execution.
- We use the similar method as the FCFS scheduler to solve the further challenge **a** and **b**. You can refer to the corresponding section for details.

Here is the specific implementation of the RR scheduler. (`schedule_rr.c`).

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

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

struct node *head = NULL;

// Extra variables for calculating the response time, etc.
// |-- current time.
int time = 0;
// |-- task count.
int tsk_cnt = 0;
// |-- total waiting time.
```

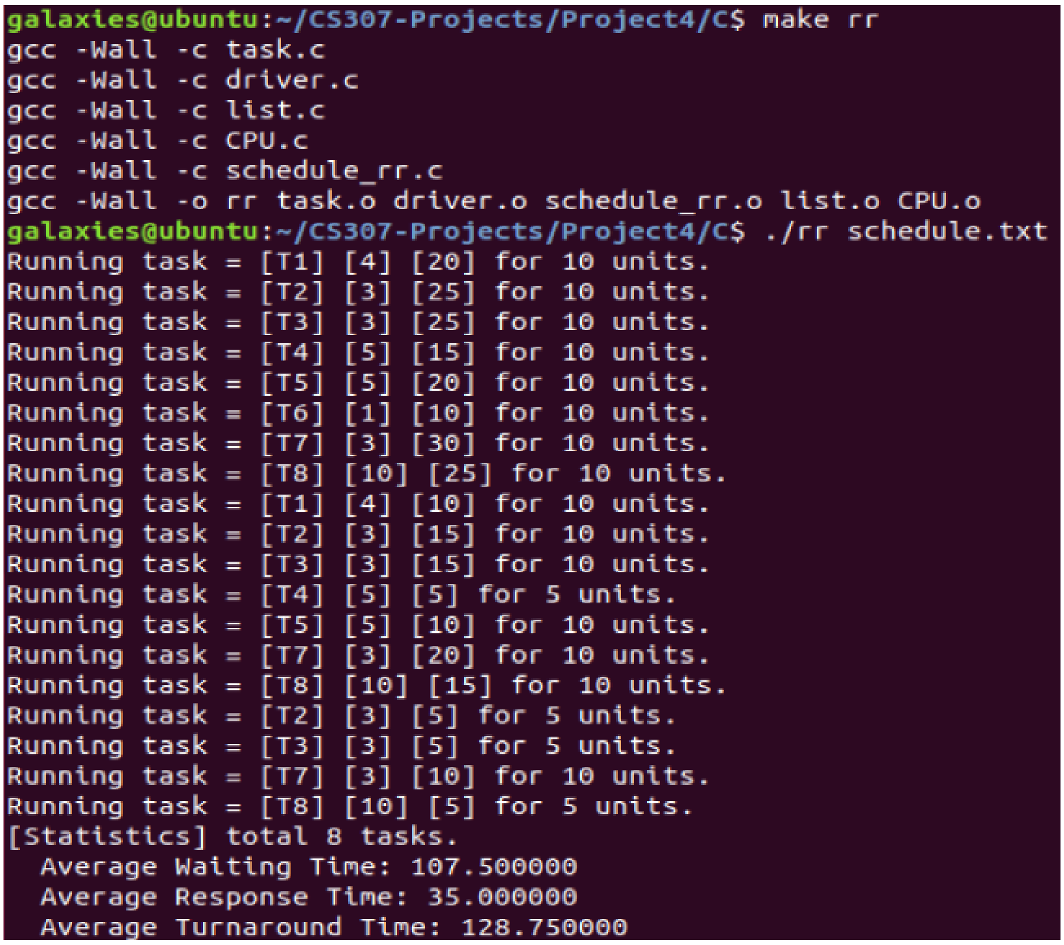
```c
int tot_wait_time = 0;
// |-- total response time.
int tot_resp_time = 0;
// |-- total turnaround time.
int tot_turn_time = 0;

void add(char *name, int priority, int burst) {
  Task *tsk;
  tsk = (Task *) malloc (sizeof(Task));
  // avoid racing conditions on tid_value .
  tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
  tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
  strcpy(tsk -> name, name);
  tsk -> priority = priority;
  tsk -> burst = burst;
  // Record some extra parameters here to calculate the response time, etc .
  tsk -> arri_time = time;
  tsk -> last_exec = time;
  tsk -> wait_time = 0;
  tsk -> resp_time = 0;
  tsk -> turn_time = 0;
  insert(&head, tsk);
}

int schedule_next() {
  if (head == NULL) return 1;

  struct node *cur = head;
  while (cur -> next != NULL) cur = cur -> next;

  Task *tsk = cur -> task;
  if (tsk -> burst <= QUANTUM) {
    run(tsk, tsk -> burst);
    delete(&head, tsk);

    time += tsk -> burst;

    // Update some parameters.
    // |-- last waiting time.
    int last_wait_time = time - tsk -> last_exec - tsk -> burst;
    // |-- update waiting time.
    tsk -> wait_time += last_wait_time;
    // |-- update response time.
    if (tsk -> last_exec == tsk -> arri_time)
      tsk -> resp_time = last_wait_time;
    // |-- update last execution time.
    tsk -> last_exec = time;

    // The task is finished , update more parameters.
    // |-- update turnaround time.
    tsk -> turn_time = time - tsk -> arri_time;

    // Update final counting parameters.
    tsk_cnt += 1;
    tot_wait_time += tsk -> wait_time;
    tot_resp_time += tsk -> resp_time;
    tot_turn_time+= tsk -> turn_time;

    free(tsk -> name);
    free(tsk);
  } else {
    run(tsk, QUANTUM);
    delete(&head, tsk);

    time += QUANTUM;
    tsk -> burst = tsk -> burst - QUANTUM;

    // Update some parameters.
    // |-- last waiting time.
    int last_wait_time = time - tsk -> last_exec - QUANTUM;
    // |-- update waiting time.
    tsk -> wait_time += last_wait_time;
    // |-- update response time.
    if (tsk -> last_exec == tsk -> arri_time)
      tsk -> resp_time = last_wait_time;
    // |-- update last execution time.
    tsk -> last_exec = time;

    insert(&head, tsk);
  }
  return 0;
}

void print_statistics() {
  printf("[Statistics] total %d tasks.\n", tsk_cnt);
  double avg_wait_time = 1.0 * tot_wait_time / tsk_cnt;
  double avg_resp_time = 1.0 * tot_resp_time / tsk_cnt;
  double avg_turn_time = 1.0 * tot_turn_time / tsk_cnt;
  printf(" Average Waiting Time: %.6lf\n", avg_wait_time);
  printf(" Average Response Time: %.6lf\n", avg_resp_time);
  printf(" Average Turnaround Time: %.6lf\n", avg_turn_time);
}

void schedule() {
  while (schedule_next() == 0);
  print_statistics();
}
```

You can enter these instructions to test the RR scheduler.

```
make rr
```

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

Here is the execution result of the program (Fig. 4).



```
galaxies@ubuntu:~/CS307-Projects/Project4/C$ make rr
gcc -Wall -c task.c
gcc -Wall -c driver.c
gcc -Wall -c list.c
gcc -Wall -c CPU.c
gcc -Wall -c schedule_rr.c
gcc -Wall -o rr task.o driver.o schedule_rr.o list.o CPU.o
galaxies@ubuntu:~/CS307-Projects/Project4/C$ ./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.
[Statistics] total 8 tasks.
  Average Waiting Time: 107.500000
  Average Response Time: 35.000000
  Average Turnaround Time: 128.750000
```

Figure 4: The execution result of RR scheduler

1.2.5 Priority-RR Scheduler

Here are the brief methods about implementing Priority-RR scheduler.

- The introduction of Priority-RR scheduling algorithm is in textbook, and we don't discuss the details of the algorithm here. You can refer to textbook to get more information.
- To complete the Priority-RR scheduler, we can implement several list and use list i to store the tasks with priority i. Therefore we can check the list in the priority turn and implement RR scheduler in each list.
- We use the similar method as the FCFS scheduler to solve the further challenge **a** and **b**. You can refer to the corresponding section for details.

Here is the specific implementation of the Priority-RR scheduler. (`schedule_priority_rr.c`).

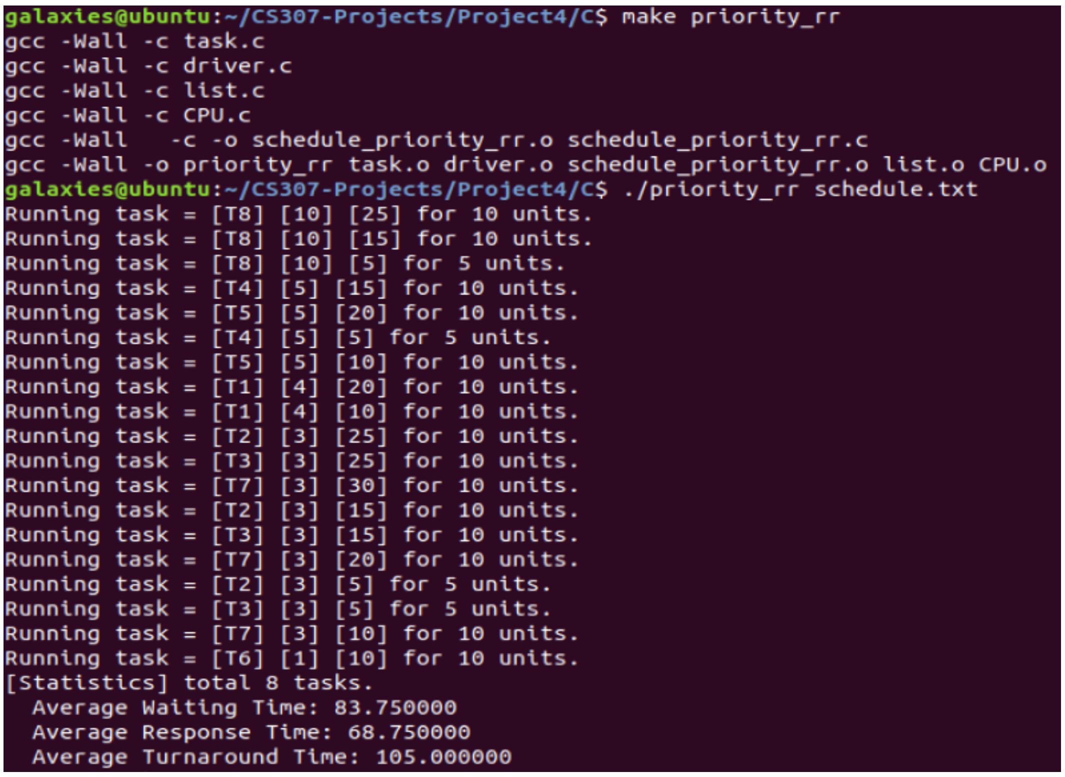
```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>

# include "task.h"
```

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

# define PRIORITY_NUMBER_SIZE (MAX_PRIORITY - MIN_PRIORITY + 1)
# define PRIORITY_BASE MIN_PRIORITY

struct node *head[PRIORITY_NUMBER_SIZE] = {};

// Extra variables for calculating the response time, etc.
// |-- current time.
int time = 0;
// |-- task count.
int tsk_cnt = 0;
// |-- total waiting time.
int tot_wait_time = 0;
// |-- total response time.
int tot_resp_time = 0;
// |-- total turnaround time.
int tot_turn_time = 0;

void add(char *name, int priority, int burst) {
  Task *tsk;
  tsk = (Task *) malloc (sizeof(Task));
  // avoid racing conditions on tid_value .
  tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
  tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
  strcpy(tsk -> name, name);
  tsk -> priority = priority;
  tsk -> burst = burst;
  // Record some extra parameters here to calculate the response time, etc.
  tsk -> arri_time = time;
  tsk -> last_exec = time;
  tsk -> wait_time = 0;
  tsk -> resp_time = 0;
  tsk -> turn_time = 0;
  insert(&head[priority - PRIORITY_BASE], tsk);
}

int schedule_next(int priority) {
  if (head[priority - PRIORITY_BASE] == NULL) return 1;

  struct node *cur = head[priority - PRIORITY_BASE];
  while (cur -> next != NULL) cur = cur -> next;

  Task *tsk = cur -> task;
  if (tsk -> burst <= QUANTUM) {
    run(tsk, tsk -> burst);
    delete(&head[priority - PRIORITY_BASE], tsk);

    time += tsk -> burst;

    // Update some parameters.
    // |-- last waiting time.
    int last_wait_time = time - tsk -> last_exec - tsk -> burst;
    // |-- update waiting time.
    tsk -> wait_time += last_wait_time;
    // |-- update response time.
    if (tsk -> last_exec == tsk -> arri_time)
      tsk -> resp_time = last_wait_time;
    // |-- update last execution time.
    tsk -> last_exec = time;

    // The task is finished, update more parameters.
    // |-- update turnaround time.
    tsk -> turn_time = time - tsk -> arri_time;

    // Update final counting parameters.
    tsk_cnt += 1;
    tot_wait_time += tsk -> wait_time;
    tot_resp_time += tsk -> resp_time;
    tot_turn_time+= tsk -> turn_time;

    free(tsk -> name);
    free(tsk);
  } else {
    run(tsk, QUANTUM);
    delete(&head[priority - PRIORITY_BASE], tsk);

    time += QUANTUM;
    tsk -> burst = tsk -> burst - QUANTUM;

    // Update some parameters.
    // |-- last waiting time.
    int last_wait_time = time - tsk -> last_exec - QUANTUM;
    // |-- update waiting time.
    tsk -> wait_time += last_wait_time;
    // |-- update response time.
    if (tsk -> last_exec == tsk -> arri_time)
      tsk -> resp_time = last_wait_time;
    // |-- update last execution time.
    tsk -> last_exec = time;

    insert(&head[priority - PRIORITY_BASE], tsk);
  }
  return 0;
}

void print_statistics() {
  printf("[Statistics] total %d tasks.\n", tsk_cnt);
  double avg_wait_time = 1.0 * tot_wait_time / tsk_cnt;
  double avg_resp_time = 1.0 * tot_resp_time / tsk_cnt;
  double avg_turn_time = 1.0 * tot_turn_time / tsk_cnt;
  printf(" Average Waiting Time: %.6lf\n", avg_wait_time);
  printf(" Average Response Time: %.6lf\n", avg_resp_time);
  printf(" Average Turnaround Time: %.6lf\n", avg_turn_time);
}

void schedule() {
  for (int i = MAX_PRIORITY; i >= MIN_PRIORITY; -- i)
    while (schedule_next(i) == 0);
  print_statistics();
}
```

You can enter these instructions to test the Priority-RR scheduler.

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

Here is the execution result of the program (Fig. 5).



```
galaxies@ubuntu:~/CS307-Projects/Project4/C$ make priority_rr
gcc -Wall -c task.c
gcc -Wall -c driver.c
gcc -Wall -c list.c
gcc -Wall -c CPU.c
gcc -Wall   -c -o schedule_priority_rr.o schedule_priority_rr.c
gcc -Wall -o priority_rr task.o driver.o schedule_priority_rr.o list.o CPU.o
galaxies@ubuntu:~/CS307-Projects/Project4/C$ ./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 = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T7] [3] [20] 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 = [T6] [1] [10] for 10 units.
[Statistics] total 8 tasks.
  Average Waiting Time: 83.750000
  Average Response Time: 68.750000
  Average Turnaround Time: 105.000000
```

Figure 5: The execution result of Priority-RR scheduler

1.3 Summary

From the execution results (Fig. 1 to Fig. 5), we can check the performance metrics of these schedulers, and it is easy to verify that SJF scheduler has the shortest average waiting time. Since this project wants the scheduler to use the same data structure to organize the unfinished tasks, we can use the unordered list for convenience, or we can use queue and heap (priority queue) to optimize the scheduler for better performance.

2 Personal Thoughts

This project improves my understanding about scheduling algorithms, and I also gain knowledge about atomic variables during solving the further challenges, which benefits me a lot.

By the way, you can find all the source codes in the "src" folder. You can also refer to my github to see my codes of this project, and they are in the `Project4` folder.