

操作系统课程设计 Project 4

Scheduling Algorithms

姓名：郭倩昀

班级：F1903303

学号：519021910095

Email: guoqianyun@sjtu.edu.cn

2021 年 5 月 22 日

目录

1 Scheduling Algorithms	2
1.1 实验内容与目标	2
1.2 FCFS	2
1.2.1 实验过程及步骤	2
1.2.2 实验代码	3
1.2.3 实验测试	5
1.3 SJF	5
1.3.1 实验过程及步骤	5
1.3.2 实验代码	5
1.3.3 实验测试	7
1.4 Priority scheduling	7
1.4.1 实验过程及步骤	7
1.4.2 实验代码	8
1.4.3 实验测试	9
1.5 RR	10
1.5.1 实验过程及步骤	10
1.5.2 实验代码	10
1.5.3 实验测试	12
1.6 Priority with round-robin	12
1.6.1 实验过程及步骤	12
1.6.2 实验代码	13
1.6.3 实验测试	15

2 Conclusion	16
2.1 问题与解决方案	16
2.2 实验心得	16

1 Scheduling Algorithms

1.1 实验内容与目标

- 利用 C 语言实现不同的进程调度算法
根据本 project 提供的 Makefile 文件，结合提供 cpu, list, driver, task 文件分析，需要我们设计 schedule_rr.c, schedule_sjf.c, schedule_fcfs.c, schedule_priority.c, schedule_priority_rr.c 一共五个程序分别实现 round-robin(RR), shortest-job-first(SJF), first-come,first-served(FCFS), priority scheduling, priority with round-robin 五种进程调度算法。
- 分配 tid 时应对竞态条件
- 分别为调度算法计算平均周转时间，平均等待时间，平均响应时间。

1.2 FCFS

1.2.1 实验过程及步骤

- 修改 task.h 文件
为支持给调度算法计算平均周转时间，平均等待时间，平均响应时间的功能，修改 task.h 文件，给结构体 task 增加辅助计算的记录进程的 arrival time, waiting time, last execution time, response time, turnaround time 等信息。
- 创建全局变量
创建任务列表头结点 head。另外为支持计算，分别创建全局变量记录当前时间，任务计数，总体等待时间，总体响应时间，总体周转时间和进程 tid 数值。
- 设计 add 函数
根据参数的进程名称，优先级和运行时间创建 Task 类型的对象并相应赋值，其中任务的 tid 利用 __sync_fetch_and_add 函数获得。将 Task 类中 arrival time 和 last execution time 设为当前时间，其他初始化为 0。最后将创建好的 task 插入到任务列表中。
- 设计 next_tsk() 函数运行下一个进程
由于是 FCFS 调度，根据任务列表插入方式，最先插入的进程在任务列表的最后，每次查找任务列表的最后一个并执行，然后将任务从任务列表中移除，并更新当前时间。另外需要为该进程计算相应的周转时间，等待时间，响应时间并更新全局变量记录信息。
- 设计调度函数 schedule()
当任务列表不为空的时候，就调用 next_tsk() 函数运行下一个进程。当任务列表全部完成，打印计算结果。

1.2.2 实验代码

task.h

```
1 #ifndef TASK_H
2 #define TASK_H
3 // representation of a task
4 typedef struct task {
5     char *name;
6     int tid;
7     int priority;
8     int burst;
9     //for calculating
10    int arv_time; //arrival time.
11    int wt_time; //waiting time.
12    int last_exe_time; //last execution time.
13    int rsp_time; //response time.
14    int ta_time; //turnaround time.
15 } Task;
16 #endif
```

shedule_fcfs.c

```
1 # include <stdio.h>
2 # include <stdlib.h>
3 # include <string.h>
4
5 # include "task.h"
6 # include "list.h"
7 # include "cpu.h"
8 # include "schedulers.h"
9
10 struct node *head = NULL;
11
12 //for calculating
13 int time = 0; //current time.
14 int tsk_cnt = 0; //task count.
15 int tt_wt_time = 0; //total waiting time.
16 int tt_rsp_time = 0; //total response time.
17 int tt_ta_time = 0; //total turnaround time.
18 int tid_value = 0; //task identifier
19 void add(char *name, int priority, int burst) {
20     Task *tsk;
21     tsk = (Task *) malloc (sizeof(Task));
22     //avoid racing conditions
23     tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
24     tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
25     strcpy(tsk -> name, name);
26     tsk -> priority = priority;
27     tsk -> burst = burst;
28
29     //for calculating
30     tsk -> arv_time = time; //arrival time.
31     tsk -> wt_time = 0; //waiting time.
32     tsk -> last_exe_time = time; //last execution time.
33     tsk -> rsp_time = 0; //response time.
```

```

34     tsk -> ta_time = 0; //turnaround time.
35     insert(&head, tsk); //add to list
36 }
37
38 void next_tsk() {
39     if (head == NULL) return;
40
41     struct node *cur = head;
42     while (cur -> next != NULL) //fcfs next last one
43         cur = cur -> next;
44
45     Task *tsk = cur -> task;
46     run(tsk, tsk -> burst); //execute
47     delete(&head, tsk); //remove from list
48     time += tsk -> burst; //current time
49
50     //for calculating
51     //last waiting time.
52     int last_wt_time = time - tsk -> last_exe_time - tsk -> burst;
53     //waiting time.
54     tsk -> wt_time += last_wt_time;
55     //response time.
56     if(tsk->last_exe_time==tsk->arv_time)//first exe
57         tsk -> rsp_time = last_wt_time;
58     //last execution time.
59     tsk -> last_exe_time = time;
60     //turnaround time.
61     tsk -> ta_time = time - tsk -> arv_time;
62
63     //total data
64     tsk_cnt += 1;
65     tt_wt_time += tsk -> wt_time;
66     tt_rsp_time += tsk -> rsp_time;
67     tt_ta_time += tsk -> ta_time;
68
69     //free
70     free(tsk -> name);
71     free(tsk);
72     //return 0;
73 }
74
75 void print() {
76     printf("\nTotal %d tasks.\n", tsk_cnt);
77     double avg_wt_time = 1.0 * tt_wt_time / tsk_cnt;
78     double avg_rsp_time = 1.0 * tt_rsp_time / tsk_cnt;
79     double avg_ta_time = 1.0 * tt_ta_time / tsk_cnt;
80     printf("Average Waiting Time: %.6lf\n", avg_wt_time);
81     printf("Average Response Time: %.6lf\n", avg_rsp_time);
82     printf("Average Turnaround Time: %.6lf\n", avg_ta_time);
83 }
84
85 void schedule() {
86     while (head != NULL)
87         next_tsk();
88     print();
89 }

```

1.2.3 实验测试

- fcfs 测试 (图 1)

```
gqy@gqy-VirtualBox:~/os_proj4$ ./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.

Total 8 tasks.
Average Waiting Time: 73.125000
Average Response Time: 73.125000
Average Turnaround Time: 94.375000
```

图 1: fcfs 测试

测试指令如下

```
1 make fcfs
2 ./fcfs schedule.txt
```

首先用 Makefile 文件编译, 生成 fcfs 可执行文件, 输入 ./fcfs schedule.txt 对 schedule.txt 的信息进行调度, 测试结果如图 1。

1.3 SJF

1.3.1 实验过程及步骤

- 创建全局变量, 设计 add 函数, 设计调度函数 schedule() 部分与 FCFS 类似, 不作赘述
- 设计 next_tsk() 函数运行下一个进程

由于是 SJF 调度, 每次查找任务列表中运行时间最短的一个任务执行, 然后将任务从任务列表中移除, 并更新当前时间。寻找最短运行时间任务的时候需要遍历任务列表, 用两个当前任务指针指向当前运行时间最短任务, 另一个指针向后查找是否有更短运行时间的任务, 如果有, 就更新当前任务指针。另外需要为该进程计算相应的周转时间, 等待时间, 响应时间并更新全局变量记录信息。

1.3.2 实验代码

shedule_sjf.c

```
1 # include <stdio.h>
2 # include <stdlib.h>
3 # include <string.h>
4
5 # include "task.h"
6 # include "list.h"
7 # include "cpu.h"
8 # include "schedulers.h"
9
10 struct node *head = NULL;
11
```

```

12 //for calculating
13 int time = 0; //current time.
14 int tsk_cnt = 0; //task count.
15 int tt_wt_time = 0; //total waiting time.
16 int tt_rsp_time = 0; //total response time.
17 int tt_ta_time = 0; //total turnaround time.
18 int tid_value = 0; //task identifier
19 void add(char *name, int priority, int burst) {
20     Task *tsk;
21     tsk = (Task *) malloc (sizeof(Task));
22     //avoid racing conditions
23     tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
24     tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
25     strcpy(tsk -> name, name);
26     tsk -> priority = priority;
27     tsk -> burst = burst;
28
29     //for calculating
30     tsk -> arv_time = time; //arrival time.
31     tsk -> wt_time = 0; //waiting time.
32     tsk -> last_exe_time = time; //last execution time.
33     tsk -> rsp_time = 0; //response time.
34     tsk -> ta_time = 0; //turnaround time.
35     insert(&head, tsk); //add to list
36 }
37
38 void next_tsk() {
39     if (head == NULL) return;
40
41     struct node *cur = head;
42     struct node *dex = head -> next;
43     while (dex != NULL) //sjf next
44     {
45         if(dex->task->burst <= cur->task->burst) cur = dex;
46         dex = dex -> next;
47     }
48     Task *tsk = cur -> task;
49     run(tsk, tsk -> burst); //execute
50     delete(&head, tsk); //remove from list
51     time += tsk -> burst; //current time
52
53     //for calculating
54     //last waiting time.
55     int last_wt_time = time - tsk -> last_exe_time - tsk -> burst;
56     //waiting time.
57     tsk -> wt_time += last_wt_time;
58     //response time.
59     if(tsk->last_exe_time==tsk->arv_time)//first exe
60         tsk -> rsp_time = last_wt_time;
61     //last execution time.
62     tsk -> last_exe_time = time;
63     //turnaround time.
64     tsk -> ta_time = time - tsk -> arv_time;
65
66     //total data
67     tsk_cnt += 1;
68     tt_wt_time += tsk -> wt_time;

```

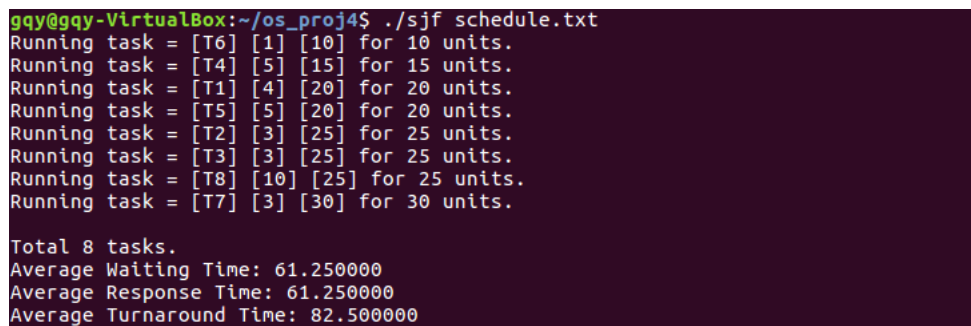
```

69     tt_rsp_time += tsk -> rsp_time;
70     tt_ta_time += tsk -> ta_time;
71
72     //free
73     free(tsk -> name);
74     free(tsk);
75 }
76
77 void print() {
78     printf("\nTotal %d tasks.\n", tsk_cnt);
79     double avg_wt_time = 1.0 * tt_wt_time / tsk_cnt;
80     double avg_rsp_time = 1.0 * tt_rsp_time / tsk_cnt;
81     double avg_ta_time = 1.0 * tt_ta_time / tsk_cnt;
82     printf("Average Waiting Time: %.6lf\n", avg_wt_time);
83     printf("Average Response Time: %.6lf\n", avg_rsp_time);
84     printf("Average Turnaround Time: %.6lf\n", avg_ta_time);
85 }
86
87 void schedule() {
88     while (head != NULL)
89         next_tsk();
90     print();
91 }

```

1.3.3 实验测试

- sjf 测试 (图 2)



```

gqy@gqy-VirtualBox:~/os_proj4$ ./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.

Total 8 tasks.
Average Waiting Time: 61.250000
Average Response Time: 61.250000
Average Turnaround Time: 82.500000

```

图 2: sjf 测试

测试指令如下

```

1 make sjf
2 ./sjf schedule.txt

```

首先用 Makefile 文件编译，生成 sjf 可执行文件，输入 ./sjf schedule.txt 对 schedule.txt 的信息进行调度，测试结果如图 2。

1.4 Priority scheduling

1.4.1 实验过程及步骤

- 创建全局变量，设计 add 函数，设计调度函数 schedule() 部分与 FCFS 类似，不作赘述

- 设计 next_tsk() 函数运行下一个进程

由于是 Priority scheduling 调度，每次查找任务列表中优先级最高的一个任务执行，然后将任务从任务列表中移除，并更新当前时间。寻找最高优先级任务的时候需要遍历任务列表，用两个当前任务指针指向当前优先级最高任务，另一个指针向后查找是否有更高优先级的任务，如果有，就更新当前任务指针。另外需要为该进程计算相应的周转时间，等待时间，响应时间并更新全局变量记录信息。

1.4.2 实验代码

shedule_priority.c

```

1  # include <stdio.h>
2  # include <stdlib.h>
3  # include <string.h>
4
5  # include "task.h"
6  # include "list.h"
7  # include "cpu.h"
8  # include "schedulers.h"
9
10 struct node *head = NULL;
11
12 //for calculating
13 int time = 0;          //current time.
14 int tsk_cnt = 0;       //task count.
15 int tt_wt_time = 0;    //total waiting time.
16 int tt_rsp_time = 0;   //total response time.
17 int tt_ta_time = 0;    //total turnaround time.
18 int tid_value = 0;     //task identifier
19 void add(char *name, int priority, int burst) {
20     Task *tsk;
21     tsk = (Task *) malloc (sizeof(Task));
22     //avoid racing conditions
23     tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
24     tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
25     strcpy(tsk -> name, name);
26     tsk -> priority = priority;
27     tsk -> burst = burst;
28
29     //for calculating
30     tsk -> arv_time = time; //arrival time.
31     tsk -> wt_time = 0;    //waiting time.
32     tsk -> last_exe_time = time; //last execution time.
33     tsk -> rsp_time = 0;   //response time.
34     tsk -> ta_time = 0;    //turnaround time.
35     insert(&head, tsk); //add to list
36 }
37
38 void next_tsk() {
39     if (head == NULL) return;
40
41     struct node *cur = head;
42     struct node *dex = head -> next;
43     while (dex != NULL) //next priority highest
44     {

```



```

45         if(dex->task->priority >= cur->task->priority) cur = dex;
46         dex = dex -> next;
47     }
48     Task *tsk = cur -> task;
49     run(tsk, tsk -> burst); //execute
50     delete(&head, tsk); //remove from list
51     time += tsk -> burst; //current time
52
53     //for calculating
54     //last waiting time.
55     int last_wt_time = time - tsk -> last_exe_time - tsk -> burst;
56     //waiting time.
57     tsk -> wt_time += last_wt_time;
58     //response time.
59     if(tsk->last_exe_time==tsk->arv_time)//first exe
60         tsk -> rsp_time = last_wt_time;
61     //last execution time.
62     tsk -> last_exe_time = time;
63     //turnaround time.
64     tsk -> ta_time = time - tsk -> arv_time;
65
66     //total data
67     tsk_cnt += 1;
68     tt_wt_time += tsk -> wt_time;
69     tt_rsp_time += tsk -> rsp_time;
70     tt_ta_time += tsk -> ta_time;
71
72     //free
73     free(tsk -> name);
74     free(tsk);
75 }
76
77 void print() {
78     printf("\nTotal %d tasks.\n", tsk_cnt);
79     double avg_wt_time = 1.0 * tt_wt_time / tsk_cnt;
80     double avg_rsp_time = 1.0 * tt_rsp_time / tsk_cnt;
81     double avg_ta_time = 1.0 * tt_ta_time / tsk_cnt;
82     printf("Average Waiting Time: %.6lf\n", avg_wt_time);
83     printf("Average Response Time: %.6lf\n", avg_rsp_time);
84     printf("Average Turnaround Time: %.6lf\n", avg_ta_time);
85 }
86
87 void schedule() {
88     while (head != NULL)
89         next_tsk();
90     print();
91 }

```

1.4.3 实验测试

- priority 测试 (图 3)

测试指令如下

```

1 make priority
2 ./priority schedule.txt

```

```

gqy@gqy-VirtualBox:~/os_proj4$ ./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.

Total 8 tasks.
Average Waiting Time: 75.000000
Average Response Time: 75.000000
Average Turnaround Time: 96.250000

```

图 3: priority 测试

首先用 Makefile 文件编译，生成 priority 可执行文件，输入 ./priority schedule.txt 对 schedule.txt 的信息进行调度，测试结果如图 3。

1.5 RR

1.5.1 实验过程及步骤

- 创建全局变量，设计 add 函数，设计调度函数 schedule() 部分与 FCFS 类似，不作赘述
- 设计 next_tsk() 函数运行下一个进程

由于是 RR 调度，每次执行任务列表中最先到达的任务，执行小于（剩余时间小于一个时间片）或等于一个时间片的时间，并更新当前时间。如果任务完成，就从任务列表删除该任务，并为它计算相应的周转时间，等待时间，响应时间并更新全局变量记录信息；否则将任务更新信息后插入回到任务列表。根据任务列表插入方式，最先插入的进程在任务列表的最后，所以每次查找任务列表的最后一个并执行。

1.5.2 实验代码

shedule_rr.c

```

1  # include <stdio.h>
2  # include <stdlib.h>
3  # include <string.h>
4
5  # include "task.h"
6  # include "list.h"
7  # include "cpu.h"
8  # include "schedulers.h"
9
10 struct node *head = NULL;
11
12 //for calculating
13 int time = 0;          //current time.
14 int tsk_cnt = 0;       //task count.
15 int tt_wt_time = 0;    //total waiting time.
16 int tt_rsp_time = 0;   //total response time.
17 int tt_ta_time = 0;    //total turnaround time.
18 int tid_value = 0;     //task identifier
19 void add(char *name, int priority, int burst) {

```

```

20 Task *tsk;
21 tsk = (Task *) malloc (sizeof(Task));
22 //avoid racing conditions
23 tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
24 tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
25 strcpy(tsk -> name, name);
26 tsk -> priority = priority;
27 tsk -> burst = burst;
28
29 //for calculating
30 tsk -> arv_time = time; //arrival time.
31 tsk -> wt_time = 0; //waiting time.
32 tsk -> last_exe_time = time; //last execution time.
33 tsk -> rsp_time = 0; //response time.
34 tsk -> ta_time = 0; //turnaround time.
35 insert(&head, tsk); //add to list
36 }
37
38 void next_tsk() {
39     if (head == NULL) return;
40
41     struct node *cur = head;
42     while (cur -> next != NULL) //next rr last one
43         cur = cur -> next;
44     Task *tsk = cur -> task;
45     if(tsk -> burst <= QUANTUM) //less than QUANTUM finish
46     {
47         run(tsk, tsk -> burst); //execute burst time
48         delete(&head, tsk); //remove from list
49         time += tsk -> burst; //current time
50
51         //for calculating
52         //last waiting time.
53         int last_wt_time = time - tsk -> last_exe_time - tsk -> burst;
54         //waiting time.
55         tsk -> wt_time += last_wt_time;
56         //response time.
57         if(tsk->last_exe_time==tsk->arv_time)//first exe
58             tsk -> rsp_time = last_wt_time;
59         //last execution time.
60         tsk -> last_exe_time = time;
61         //turnaround time.
62         tsk -> ta_time = time - tsk -> arv_time;
63
64         //total data
65         tsk_cnt += 1;
66         tt_wt_time += tsk -> wt_time;
67         tt_rsp_time += tsk -> rsp_time;
68         tt_ta_time += tsk -> ta_time;
69
70         //free
71         free(tsk -> name);
72         free(tsk);
73     }
74     else //more than QUANTUM
75     {
76         run(tsk, QUANTUM); //execute QUANTUM
77         delete(&head, tsk);

```

```

77     time += QUANTUM;    //update current time
78     //update task info for calculating
79     tsk -> burst = tsk -> burst - QUANTUM;
80     //last waiting time.
81     int last_wt_time = time - tsk -> last_exe_time - QUANTUM;
82     //waiting time.
83     tsk -> wt_time += last_wt_time;
84     if(tsk->last_exe_time==tsk->arv_time)//first exe
85         tsk -> rsp_time = last_wt_time;
86     //last execution time.
87     tsk -> last_exe_time = time;
88
89     insert(&head, tsk); //add to list
90 }
91 }
92
93 void print() {
94     printf("\nTotal %d tasks.\n", tsk_cnt);
95     double avg_wt_time = 1.0 * tt_wt_time / tsk_cnt;
96     double avg_rsp_time = 1.0 * tt_rsp_time / tsk_cnt;
97     double avg_ta_time = 1.0 * tt_ta_time / tsk_cnt;
98     printf("Average Waiting Time: %.6lf\n", avg_wt_time);
99     printf("Average Response Time: %.6lf\n", avg_rsp_time);
100    printf("Average Turnaround Time: %.6lf\n", avg_ta_time);
101 }
102
103 void schedule() {
104     while (head != NULL)
105         next_tsk();
106     print();
107 }

```

1.5.3 实验测试

- rr 测试 (图 4)

测试指令如下

```

1 make rr
2 ./rr schedule.txt

```

首先用 Makefile 文件编译，生成 rr 可执行文件，输入./rr schedule.txt 对 schedule.txt 的信息进行调度，测试结果如图 4。

1.6 Priority with round-robin

1.6.1 实验过程及步骤

- 创建全局变量

由于 Priority with round-robin 调度需要根据优先级进行调度，所以全局变量的任务列表头结点设计为给每个优先级创建一个头结点，便于分优先级调度。

- 设计 add 函数部分

该函数其他部分与 FCFS 类似，只是插入任务的时候根据当前任务的优先级插入到相应优先级的头

```

gqy@gqy-VirtualBox:~/os_proj4$ ./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.

Total 8 tasks.
Average Waiting Time: 107.500000
Average Response Time: 35.000000
Average Turnaround Time: 128.750000

```

图 4: rr 测试

结点所在的列表中。

- 设计 next_tsk() 函数运行下一个进程

由于是 Priority with round-robin 调度，这里设计的 next_tsk() 函数有传入参数 priority，表示选择该优先级下的任务列表的下一个任务执行。其他部分的实现跟 RR 调度类似，这里不作赘述。

- 设计调度函数 schedule()

由于是 Priority with round-robin 调度，所以调度函数从最高优先级开始，根据优先级从高到低为每一个优先级队列调度，只有高优先级任务列表完成运行才会开始低优先级任务列表的调度。全部任务运行结束后打印计算结果。

1.6.2 实验代码

shedule_priority_rr.c

```

1 # include <stdio.h>
2 # include <stdlib.h>
3 # include <string.h>
4
5 # include "task.h"
6 # include "list.h"
7 # include "cpu.h"
8 # include "schedulers.h"
9
10 struct node *head[MAX_PRIORITY - MIN_PRIORITY + 1] = {};
11
12 //for calculating
13 int time = 0;          //current time.
14 int tsk_cnt = 0;       //task count.
15 int tt_wt_time = 0;    //total waiting time.
16 int tt_rsp_time = 0;   //total response time.
17 int tt_ta_time = 0;    //total turnaround time.

```

```

18 int tid_value = 0; //task identifier
19 void add(char *name, int priority, int burst) {
20     Task *tsk;
21     tsk = (Task *) malloc (sizeof(Task));
22     //avoid racing conditions
23     tsk -> tid = __sync_fetch_and_add(&tid_value, 1);
24     tsk -> name = (char *) malloc (sizeof(char) * (1 + strlen(name)));
25     strcpy(tsk -> name, name);
26     tsk -> priority = priority;
27     tsk -> burst = burst;
28
29     //for calculating
30     tsk -> arv_time = time; //arrival time.
31     tsk -> wt_time = 0; //waiting time.
32     tsk -> last_exe_time = time; //last execution time.
33     tsk -> rsp_time = 0; //response time.
34     tsk -> ta_time = 0; //turnaround time.
35     insert(&head[priority- MIN_PRIORITY], tsk); //add to list
36 }
37
38 void next_tsk(int priority) {
39     if (head[priority- MIN_PRIORITY] == NULL) return;
40
41     struct node *cur = head[priority- MIN_PRIORITY];
42     while (cur -> next != NULL) //next priority_rr last one
43         cur = cur -> next;
44     Task *tsk = cur -> task;
45     if(tsk -> burst <= QUANTUM) //less than QUANTUM finish
46     {
47         run(tsk, tsk -> burst); //execute burst time
48         delete(&head[priority- MIN_PRIORITY], tsk); //remove from list
49         time += tsk -> burst; //current time
50
51         //for calculating
52         //last waiting time.
53         int last_wt_time = time - tsk -> last_exe_time - tsk -> burst;
54         //waiting time.
55         tsk -> wt_time += last_wt_time;
56         //response time.
57         if(tsk->last_exe_time==tsk->arv_time)//first exe
58             tsk -> rsp_time = last_wt_time;
59         //last execution time.
60         tsk -> last_exe_time = time;
61         //turnaround time.
62         tsk -> ta_time = time - tsk -> arv_time;
63
64         //total data
65         tsk_cnt += 1;
66         tt_wt_time += tsk -> wt_time;
67         tt_rsp_time += tsk -> rsp_time;
68         tt_ta_time += tsk -> ta_time;
69
70         //free
71         free(tsk -> name);
72         free(tsk);
73     }
74     else //more than QUANTUM
75     {

```

```

75     run(tsk, QUANTUM); //execute QUANTUM
76     delete(&head[priority- MIN_PRIORITY], tsk);
77     time += QUANTUM; //update current time
78     //update task info for calculating
79     tsk -> burst = tsk -> burst - QUANTUM;
80     //last waiting time.
81     int last_wt_time = time - tsk -> last_exe_time - QUANTUM;
82     //waiting time.
83     tsk -> wt_time += last_wt_time;
84     if(tsk->last_exe_time==tsk->arv_time)//first exe
85         tsk -> rsp_time = last_wt_time;
86     //last execution time.
87     tsk -> last_exe_time = time;
88
89     insert(&head[priority- MIN_PRIORITY], tsk); //add to list
90 }
91 }
92
93 void print() {
94     printf("\nTotal %d tasks.\n", tsk_cnt);
95     double avg_wt_time = 1.0 * tt_wt_time / tsk_cnt;
96     double avg_rsp_time = 1.0 * tt_rsp_time / tsk_cnt;
97     double avg_ta_time = 1.0 * tt_ta_time / tsk_cnt;
98     printf("Average Waiting Time: %.6lf\n", avg_wt_time);
99     printf("Average Response Time: %.6lf\n", avg_rsp_time);
100    printf("Average Turnaround Time: %.6lf\n", avg_ta_time);
101 }
102
103 void schedule() {
104     for(int i = MAX_PRIORITY; i >= MIN_PRIORITY ; --i)
105     {
106         while (head[i- MIN_PRIORITY] != NULL)
107             next_tsk(i);
108     }
109     print();
110 }

```

1.6.3 实验测试

- Priority with round-robin 测试 (图 5)

测试指令如下

```

1 make priority_rr
2 ./priority_rr schedule.txt

```

首先用 Makefile 文件编译，生成 priority_rr 可执行文件，输入 ./priority_rr schedule.txt 对 schedule.txt 的信息进行调度，测试结果如图 5。

```
gqy@gqy-VirtualBox:~/os_proj4$ ./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.

Total 8 tasks.
Average Waiting Time: 83.750000
Average Response Time: 68.750000
Average Turnaround Time: 105.000000
```

图 5: Priority with round-robin 测试

2 Conclusion

2.1 问题与解决方案

本次 project4 分别实现了 round-robin(RR), shortest-job-first(SJF), first-come,first-served(FCFS), priority scheduling, priority with round-robin 五种进程调度算法,总体上难度不大,只要对所学知识有了解足够透彻就可以完成。其中 priority with round-robin 算法要结合优先级和 RR 算法调度,相较于别的算法实现起来比较困难,经过一番思考后决定采用分优先级的任务列表来实现,最终顺利完成了 project。

2.2 实验心得

本次 project4 将五种进程调度算法实现出来,是对所学知识的一次很好地运用,算法实现难度并不大,但是需要仔细思考每一种算法的核心思想,采用合适的方法实现出来,比如在计算相应的时间信息的时候,需要在每次执行任务的时候及时维护更新时间信息,特别是 RR 算法以及 priority with round-robin 算法。总的来说本次 project 进一步加深了我对各类进程调度算法的理解,在算法实现的过程中也非常有成就感,让我受益匪浅。