Project 4

秦啸涵　**521021910604**

实现五种调度算法

FCFS

`FCFS` 实现较为简单，在执行完 `add()` 将所有进程入队后，只需按入队的顺序依次执行并更改当前时间即可

```
int schedule_onetask()
{
    if (head == NULL) {
        return 0;
    }
    struct node *cur = head;
    while(cur->next != NULL) {
        cur = cur->next;
    }
    //这里是因为执行add()操作时最先入队的被放在链表尾部，因此从尾部开始执行
    run(cur->task, cur->task->burst);
    delete(&head, cur->task);

    cur->task->wait_time = time_wait;                //当前任务等待时间
    cur->task->response_time = time_wait;            //当先任务响应时间，非抢占，所以等于等待时间

    time += cur->task->burst;
    time_wait += cur->task->burst;
    time_turnaround += cur->task->burst;             //更新全局变量

    cur->task->turnaround_time = time_turnaround;    //当前任务周转时间
```

```
22
23          total_wait += cur->task->wait_time;
24          total_turnaround += cur->task->turnaround_time; //总的等待时
    间，周转时间和响应时间
25          return 1;
26      }
```

实现结果：

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./fcfs schedule.txt
Running task = [T1] [4] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T8] [10] [25] for 25 units.
Average wait time = 73.125000
Average turnaround time = 94.375000
Average response time = 73.125000
```

SJF

`SJF`是最短任务优先算法，在本题中我们只考虑所有任务在时刻0同时到达的情况，因此只需要按照任务的`burst time`时间长短依次执行即可

```
1   int schedule_onetask()
2   {
3       if (head == NULL) {
4           return 0;
5       }
6       struct node *cur = head;
7       struct node *temp = head;
8       while(temp->next != NULL) {
9           temp = temp->next;
10          if (temp->task->burst <= cur->task->burst) {
11              cur = temp;
12          }
13      }
14      //挑选时间最短的去执行
15      run(cur->task, cur->task->burst);
16      delete(&head, cur->task);
```

```
17
18      cur->task->wait_time = time_wait;
19      cur->task->response_time = time_wait;
20
21      time += cur->task->burst;
22      time_wait += cur->task->burst;
23      time_turnaround += cur->task->burst;
24
25      cur->task->turnaround_time = time_turnaround;
26
27      total_wait += cur->task->wait_time;
28      total_turnaround += cur->task->turnaround_time;
29      return 1;
30  }
```

代码结构与 `FCFS` 基本一样(因为也是非抢占)，不同之处是更改了选择执行任务的逻辑

实现结果:

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./sjf schedule.txt
Running task = [T6] [1] [10] for 10 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [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.
Average wait time = 61.250000
Average turnaround time = 82.500000
Average response time = 61.250000
```

Priority

与 `SJF` 基本完全一致，只不过把要排序的从 `burst time` 更换为 `priority`

```
1   int schedule_onetask()
2   {
3       if (head == NULL) {
4           return 0;
5       }
6       struct node *cur = head;
7       struct node *temp = head;
8       while(temp->next != NULL) {
```

```
            temp = temp->next;
            if (temp->task->priority >= cur->task->priority) {
                cur = temp;
            }
        }
        //选择优先级最高的来执行
        run(cur->task, cur->task->burst);
        delete(&head, cur->task);

        cur->task->wait_time = time_wait;
        cur->task->response_time = time_wait;

        time += cur->task->burst;
        time_wait += cur->task->burst;
        time_turnaround += cur->task->burst;

        cur->task->turnaround_time = time_turnaround;

        total_wait += cur->task->wait_time;
        total_turnaround += cur->task->turnaround_time;
        return 1;
    }
```

实现结果：

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./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.
Average wait time = 75.000000
Average turnaround time = 96.250000
Average response time = 75.000000
```

RR

`RR` 算法的实现与以上三种有所不同，因为响应时间不再等于等待时间，在这里我们选择从头到尾扫描任务队列并根据任务剩余时间执行任务：

- 若任务剩余时间大于一个时间片，则分配一个时间片执行

- 若任务剩余时间小于等于一个时间片，则分配剩余时间直接执行完，并将任务从队列中删除

需要注意的点是，为每个任务设置一个`flag`标记，初始为0，当任务第一次执行时，更改为1，以便记录下响应时间。另外在执行前需要对原链表进行`reverse()`操作(原因之前已提到，插入时是从尾部插入)

```
int schedule_onetask()
{
    if (head == NULL) {
        return 0;
    }
    struct node *cur = head;

    while(cur != NULL)
    {
        int onetime = QUANTUM;
        if (cur->task->flag == 0)
        {
            cur->task->response_time = time;
            cur->task->flag = 1;
            total_response += cur->task->response_time;
        }
        if (cur->task->burst_left <= QUANTUM)
        {
            onetime = cur->task->burst_left;
            run(cur->task, onetime);
            cur->task->turnaround_time = time + onetime;  //终止时间
            cur->task->wait_time = cur->task->turnaround_time - cur->task->burst;
            total_wait += cur->task->wait_time;
            total_turnaround += cur->task->turnaround_time;
            delete(&head, cur->task);
        }
        else
        {
            run(cur->task, onetime);
            cur->task->burst_left -= onetime;
        }
        time += onetime;
```

```
            cur = cur->next;
        }

        return 1;
    }
```

实现结果：

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./rr schedule.txt
Running task = [T1] [4] [20] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T8] [10] [25] for 10 units.
Running task = [T1] [4] [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 5 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T8] [10] [25] for 10 units.
Running task = [T2] [3] [25] for 5 units.
Running task = [T3] [3] [25] for 5 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T8] [10] [25] for 5 units.
Average wait time = 107.500000
Average turnaround time = 128.750000
Average response time = 35.000000
```

Priority_RR

不同优先级之间严格按优先级执行，相同优先级之间与 `RR` 相同，因此可以复用 `RR` 代码，只需要更改 `add()` 函数：本来是将所有任务插入到同一个队列，现在为每个优先级单独设置一个队列，任务到来时只插入对应优先级的队列，之后在执行时按优先级从高到低遍历所有队列即可

```
//增加参数priority,表示执行对应priority的队列
int schedule_onetask(int priority)
{
    if (head[priority] == NULL) {
        return 0;
    }
    struct node *cur = head[priority];

```

```
    while(cur != NULL)
    {
        int onetime = QUANTUM;
        if (cur->task->flag == 0)
        {
            cur->task->response_time = time;
            cur->task->flag = 1;
            total_response += cur->task->response_time;
        }
        if (cur->task->burst_left <= QUANTUM)
        {
            onetime = cur->task->burst_left;
            run(cur->task, onetime);
            cur->task->turnaround_time = time + onetime;  //终止时间
            cur->task->wait_time = cur->task->turnaround_time - cur->task->burst;
            total_wait += cur->task->wait_time;
            total_turnaround += cur->task->turnaround_time;
            delete(&head[priority], cur->task);
        }
        else
        {
            run(cur->task, onetime);
            cur->task->burst_left -= onetime;
        }
        time += onetime;
        cur = cur->next;
    }
    return 1;
}
```

实现结果:

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch5/project/posix$ ./priority_rr schedule.txt
Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [25] for 5 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 5 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T1] [4] [20] for 10 units.
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 = [T7] [3] [30] 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] [25] for 5 units.
Running task = [T3] [3] [25] for 5 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Average wait time = 83.750000
Average turnaround time = 105.000000
Average response time = 68.750000
```

Bonus

Fix race condition

在`SMP`环境中，对变量`tid`的赋值可能存在竞争条件。为解决竞争条件，我们可以使用原子整数:

定义全局变量`tid_value`

```
1   extern int tid_value; //cpu.h
2   tid_value = 0;        //schedule_*.c
```

并在`add()`函数中为进程`tid`赋值时使用原子操作

```
1   newTask->tid = __sync_fetch_and_add(&tid_value,1);
```

计算算法平均响应时间，平均等待时间和平均周转时间

见前文五种调度算法实现结果图(计算样例均为`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
```