上海交通大學

《操作系统》课程

学生实验报告

实验名称:Project4:Scheduling Algorithms

姓　　名：洪瑄锐

学　　号：517030910227

班　　级：F1703302

手　　机：15248246044

任课老师：吴晨涛

2019 年 5 月 26 日

目录

一、实验要求

该项目要求实施几种不同的流程调度算法，将为调度程序分配一组预定义的任务，并根据所选的调度算法调度任务。为每个任务分配优先级和 cpu burst。

- fcfs 按照请求 cpu 的顺序安排任务。
- sjf 按照 cpu burst 的长度，最短任务优先。
- priority 按照优先级安排任务。
- rr 每个任务运行一个时间片，或者 cpu burst 的剩余时间。
- priority with rr 按照优先级安排任务，优先级相同的每个任务运行宇哥时间片，或者 cpu burst 的剩余时间。

二、程序设计思想及代码解释

课本网站所提供的基础代码有以下几个重要部分：

- 结构体 task

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

- 存储任务的列表，注意 insert 是向列表头部插入

```
struct node {
    Task *task;
    struct node *next;
};

// insert and delete operations.
void insert(struct node **head, Task *task);
void delete(struct node **head, Task *task);
void traverse(struct node *head);
```

- “cpu.c”中的 run（Task*task,int slice）函数，用于执行各个任务

```
// 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);
}
```

- driver.c 函数

课本所提供的函数为我们写好了读取文件中的任务，并存入类型为 task 的变量当中。driver.c 要求我们自己写好 add（name，priority，burst）和 schedule（）函数。此外因为本项目要求输出 turnaround time,waiting time,response time，所以在 driver.c 中定义好变量用于记录时间，任务总数目，以及任务列表的头指针。因为需要为调度程序提供的每一个任务分配一个唯一的任务 tid，所以在 driver.c 中定义一个变量 value，在向列表中加入

任务时利用__sync_fetch_and_add(&value,1)为 value 自增 1 并赋值给该任务作为该任务的 tid。最后在调用 schedule（）函数后输出各个时间。

```
struct node **head;
float turnarround_time = 0;
float wait_time = 0;
float response_time = 0;
int task_num = 0;
int value = 0;
```

```
schedule();
printf("Average turnaround time = %f\n", turnarround_time / task_num);
printf("Average wait time = %f\n", wait_time / task_num);
printf("Average response time = %f\n", response_time / task_num);
```

2.1 FCFS

1.add（name，priority，burst）函数

定义一个 task 指针，为其动态分配内存，将 value 自增 1，赋值给 task->tid，将 name 赋值给 task->name，将 priority 赋值给 task->priority，将 burst 赋值给 task->burst，然后调用 insert 函数插入任务列表即可。

```
void add(char *name, int priority, int burst)
{
    Task *task;
    task = malloc(sizeof(Task));
    task->name = malloc(sizeof(char) * 20);
    strcpy(task->name, name);
    __sync_fetch_and_add(&value, 1);
    task->tid = value;
    task->priority = priority;
    task->burst = burst;
    insert(head, task);
}
```

2.schedule()函数

遍历任务列表，找到位于列表最后的任务即最先到达的任务（因为 insert 函数将 newnode 插在头部），调用 run 函数执行，响应时间即当前调用时刻的时间，等待时间对于 fcfs 来说也是当前调用时刻的时间，任务执行后当前时刻的时间要加上 burst，周转时间即改变后的当前时间。最后删除该任务。

```c
void schedule()
{
    struct node *p;
    while ((*head) != NULL)
    {
        p = *head;
        while (p->next != NULL)
            p = p->next;
        run(p->task, p->task->burst);
        wait_time += current_time;
        response_time += current_time;
        current_time += p->task->burst;
        turnarround_time += current_time;

        delete(head, p->task);
    }
}
```

2.2 SJF

1. add(name, priority, burst)函数，和 fcfs 相同。

2. schedule()函数

遍历任务列表，找到列表中 burst 最小的任务，调用 run 函数执行，响应时间即当前调用时刻的时间，等待时间对于 sjf 来说也是当前调用时刻的时间，任务执行后当前时刻的时间要加上 burst，周转时间即改变后的当前时间。最后删除该任务。

```c
void schedule()
{
    struct node *tmp;
    struct node *shortest;
    shortest = *head;
    while ((*head) != NULL)
    {
        tmp = *head;
        shortest = *head;
        while (tmp != NULL)
        {
            if (tmp->task->burst < shortest->task->burst)
            {
                shortest = tmp;
            }
            tmp = tmp->next;
        }
        run(shortest->task, shortest->task->burst);
        wait_time += current_time;
        response_time += current_time;
        current_time += shortest->task->burst;
        turnarround_time += current_time;
        delete(head, shortest->task);
    }

}
```

2.3 Priority

1. add(name, priority, burst)函数，和 fcfs 相同。

2. schedule()函数

遍历任务列表，找到列表中优先级最高的任务，priority 越大优先级越高。

调用 run 函数执行，响应时间即当前调用时刻的时间，等待时间对于 sjf 来说也是当前调用时刻的时间，任务执行后当前时刻的时间要加上 burst，周转时间即改变后的当前时间。最后删除该任务。

```
void schedule()
{
    struct node *tmp;
    struct node *highest;

    while ((*head) != NULL)
    {
        tmp = *head;
        highest = *head;
        while (tmp != NULL)
        {
            if (tmp->task->priority > highest->task->priority)
            {
                highest = tmp;
            }
            tmp = tmp->next;
        }
        run(highest->task, highest->task->burst);
        wait_time += current_time;
        response_time += current_time;
        current_time += highest->task->burst;
        turnarround_time += current_time;
        delete(head, highest->task);
    }

}
```

2.4 rr

1. add(name, priority, burst)函数， 和 fcfs 相同
2. add_to_tail(name, priority, burst， tid)函数

与 add(name, priority, burst)函数相比，它无需为 value 自增 1 赋值给 tid。

```
void add_to_tail(char* name, int priority, int burst, int tid)
{
    Task *task;
    task = malloc(sizeof(Task));
    task->name = malloc(sizeof(char) * 20);
    strcpy(task->name, name);
    task->priority = priority;
    task->burst = burst;
    task->tid = tid;

    insert(head, task);
}
```

3. schedule()函数

定义一个数组 int flag[50]，用于记录某个任务是否已经被调度过，供计算时间使用。如果被调度过则 flag[tid]=1， 否则 flag[tid]=0。

当任务列表不为空时重复执行 while 内部分。定位到任务列表最末，为当前要执行的任务。分为两种情况：

（1）该任务的 burst<=10，则可以一次性执行完毕，如果 flag[tid]==0，说明这是第一次调度，response time 为此刻的时间，然后将 flag[tid]修改为 1。调用 run()函数执行，因为该任务可以一次性执行，所以删除即可，waiting time-=burst（在任务列表清空后 waiting time+=turnaround time，因为每个任务的

等待时间为其总周转时间-总执行时间)，current time += burst，turnaround time += current time。

（2）该任务的 burst>10，则需要分时间片执行。如果 flag[tid]==0，说明这是第一次调度，response time 为此刻的时间，然后将 flag[tid]修改为 1。调用 run()函数执行，执行时间为 10，删除该任务，将该任务 burst-10 后重新添加到任务列表中。waiting time -= 10，current_time += 10。

```
while ((*head) != NULL)
{
    tmp = *head;
    while (tmp->next != NULL)
        tmp = tmp->next;
    //分为burst<=10和burst>10
    if (tmp->task->burst <= 10)//可以一次执行完毕
    {
        if (flag[tmp->task->tid] == 0)//设置为调度过
        {
            response_time += current_time;
            flag[tmp->task->tid] = 1;
        }
        run(tmp->task, tmp->task->burst);
        wait_time -= tmp->task->burst;
        current_time += tmp->task->burst;
        turnarround_time += current_time;//每个任务都是在0时刻发布的，在此时此刻完成
        delete(head, tmp->task);
    }
    else
    {
        if (flag[tmp->task->tid] == 0)
        {
            response_time += current_time;
            flag[tmp->task->tid] = 1;
        }

        run(tmp->task, 10);
        wait_time -= 10;
        current_time += 10;
        tmp->task->burst -= 10;

        delete(head, tmp->task);
        add_to_tail(tmp->task->name, tmp->task->priority, tmp->task->burst, tmp->task->tid);
    }
}
wait_time += turnarround_time;
```

2.5 Priority with rr

1. add(name, priority, burst)

与 fcfs 中的 add 不同的是，该函数添加了 pri[priority]++，pri[i]记录了优先级为 i 的任务的个数。

```
void add(char *name, int priority, int burst) {
    Task *task;
    task = malloc(sizeof(Task));
    task->name = malloc(sizeof(char) * 20);
    strcpy(task->name, name);
    task->priority = priority;
    task->burst = burst;
    __sync_fetch_and_add(&value, 1);
    task->tid = value;
    pri[priority]++;
    insert(head, task);
}
```

2. schedule()函数

1. 将任务列表调换顺序，使得先到达的任务在头部

```
struct node* tmp;
struct node* newnode;
//将head反过来存，使得T1在头部
newhead = malloc(sizeof(struct node*));
*newhead = NULL;
while ((*head) != NULL)
{
    newnode = malloc(sizeof(struct node));
    newnode->task = (*head)->task;
    newnode->next = *newhead;
    *newhead = newnode;
    *head = (*head)->next;
}
```

2. 对每个优先级进行遍历，分为三种情况

（1）pri[i]==0，意味着该优先级没有任务，continue。

（2）pri[i]==1，意味着该优先级只有 1 个任务，遍历任务列表，直接执行即可。

```
for (int i = 10; i > 0; i--)
{
    if (pri[i] == 0) continue;
    //printf("pri[i]=%d", pri[i]);
    if (pri[i] == 1)
    {
        tmp = *newhead;
        while (tmp != NULL)
        {
            if (tmp->task->priority == i)
            {
                run(tmp->task, tmp->task->burst);
                wait_time -= tmp->task->burst;
                response_time += current_time;
                current_time += tmp->task->burst;
                turnaround_time += current_time;
                delete(newhead, tmp->task);
                pri[i]--;
                break;
            }
            else tmp = tmp->next;
        }
    }
```

（3）pri[i]>1，说明该优先级有很多个任务，需要利用 rr 算法调度任务。首先初始化用于记录任务是否被调度过的 flag 数组。

While pri[i]不为 0，说明 rr 没有结束，则一直执行 while 中的内容。遍历任务列表，当优先级为 i 时判断 burst 是否<=10：

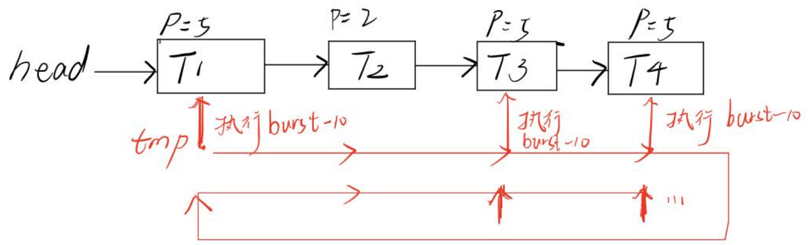
- 如果是则可以一次性执行完毕，如果 flag[tid]==0，说明这是第一次调度，response time 为此刻的时间，然后将 flag[tid]修改为 1。调用 run()函数执行，因为该任务可以一次性执行，所以删除即可，注意将 pri[i]减 1，waiting time-=burst(在任务列表清空后 waiting time+=turnaround time，因为每个任务的等待时间为其总周转时间-总执行时间)，current time += burst，turnaround time += current time。
- 如果 burst>10 则需要执行时间片的时间，如果 flag[tid]==0，说明这是

第一次调度，response time 为此刻的时间，然后将 flag[tid]修改为 1。

调用 run()函数执行，执行时间为 10，删除该任务，将该任务 burst-10。

waiting time -= 10， current_time += 10。

最后遍历下一个任务，如果已经走到任务列表的尾部，则回到头部，直至 pri[i]=0。




```
else
{
    tmp = *newhead;
    for (int k = 0; k < 50; k++)
    {
        flag[k] = 0;//未被调度过
    }
    while (pri[i] > 0)
    {
        if (tmp->task->priority == i)
        {
            //分为burst<=10和burst>10
            if (tmp->task->burst <= 10)//可以一次执行完毕
            {
                if (flag[tmp->task->tid] == 0)//设置为调度过
                {
                    response_time += current_time;
                    flag[tmp->task->tid] = 1;
                }
                run(tmp->task, tmp->task->burst);
                wait_time -= tmp->task->burst;
                current_time += tmp->task->burst;
                turnarround_time += current_time;//每个任务都是在0时刻发布的，在此时此刻完成
                delete(newhead, tmp->task);
                pri[i]--;
            }
```

```
                else
                {
                    if (flag[tmp->task->tid] == 0)
                    {
                        response_time += current_time;
                        flag[tmp->task->tid] = 1;
                    }
                    run(tmp->task, 10);
                    wait_time -= 10;
                    current_time += 10;
                    tmp->task->burst -= 10;
                }
            }
            if (tmp->next == NULL)
                tmp = *newhead;
            else
                tmp = tmp->next;

        }
    }
}


wait_time += turnarround_time;
```

三、运行结果截图

1. FCFS

```
osc@ubuntu:~/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 turnaround time = 94.375000
Average wait time = 73.125000
Average response time = 73.125000
```

2. SJF

```
osc@ubuntu:~/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 = [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 = 82.500000
Average wait time = 61.250000
Average response time = 61.250000
```

3.Priority

```
osc@ubuntu:~/final-src-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.
Average turnaround time = 98.125000
Average wait time = 76.875000
Average response time = 76.875000
```

4.rr

```
osc@ubuntu:~/final-src-osc10e/ch5/project/posix$ ./rr rr-schedule.txt
Running task = [T1] [40] [50] for 10 units.
Running task = [T2] [40] [50] for 10 units.
Running task = [T3] [40] [50] for 10 units.
Running task = [T4] [40] [50] for 10 units.
Running task = [T5] [40] [50] for 10 units.
Running task = [T6] [40] [50] for 10 units.
Running task = [T1] [40] [40] for 10 units.
Running task = [T2] [40] [40] for 10 units.
Running task = [T3] [40] [40] for 10 units.
Running task = [T4] [40] [40] for 10 units.
Running task = [T5] [40] [40] for 10 units.
Running task = [T6] [40] [40] for 10 units.
Running task = [T1] [40] [30] for 10 units.
Running task = [T2] [40] [30] for 10 units.
Running task = [T3] [40] [30] for 10 units.
Running task = [T4] [40] [30] for 10 units.
Running task = [T5] [40] [30] for 10 units.
Running task = [T6] [40] [30] for 10 units.
Running task = [T1] [40] [20] for 10 units.
Running task = [T2] [40] [20] for 10 units.
Running task = [T3] [40] [20] for 10 units.
Running task = [T4] [40] [20] for 10 units.
Running task = [T5] [40] [20] for 10 units.
Running task = [T6] [40] [20] for 10 units.
Running task = [T1] [40] [10] for 10 units.
Running task = [T2] [40] [10] for 10 units.
Running task = [T3] [40] [10] for 10 units.
Running task = [T4] [40] [10] for 10 units.
Running task = [T5] [40] [10] for 10 units.
Running task = [T6] [40] [10] for 10 units.
Average turnaround time = 275.000000
Average wait time = 225.000000
Average response time = 25.000000
```

5.Priority with rr

```
osc@ubuntu:~/final-src-osc10e/ch5/project/posix$ ./priority_rr schedule.txt
Running task = [T8] [10] [25] for 25 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 20 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.
Average turnaround time = 105.000000
Average wait time = 83.750000
Average response time = 68.750000
```