

Project 4-- Scheduling Algorithms

陈思远 518021910567

Project 4-- Scheduling Algorithms

- 1.实验内容和目标
2. 调度算法实现
 - 2.1 FCFS
 - 2.2 SJF
 - 2.3 Priority Scheduling
 - 2.4 RR
 - 2.5 Priority with RR
3. 总结与反思

1.实验内容和目标

1. 实现FCFS、SJF、priority scheduling、RR 和 priority with RR五种调度算法。
2. 使用原子操作增加tid，防止竞态。
3. 计算各个调度算法的平均周转时间、平均等待时间、平均响应时间。

2. 调度算法实现

driver.c读入各个进程的信息，然后用 add 函数将进程逐一插入链表。之后，调用 schedule 函数调度各个进程，每种算法的 add 和 schedule 函数都有所不同。此外，给每个进程分配tid时，可以使用原子操作对tid进行自增，防止竞态。

```
tmp->tid=__sync_fetch_and_add(&task_id,1); //分配tid
```

2.1 FCFS

FCFS算法采用先来先服务的原则，按照进程到达的顺序进行调度。由于给出的链表类在插入时会把最先插入的进程放在最末，因此在schedule函数中，每次选择需要执行的进程都要遍历链表，找到最末的那个，执行后删除。

此外，还要计算该算法的平均周转时间、平均等待时间、平均响应时间。因此，每个任务完成和开始时，都要记录当前的时间，以便计算。由于假设所有任务一同到达，所以对FCFS算法来说，平均等待时间等于平均响应时间。具体代码如下。

```
struct node *head;
int num=0;
int task_id=0;

int curtime=0;
int pretime=0;
int turnaround=0;
int wait=0;
int response=0;

// add a task to the list
void add(char *name, int priority, int burst){
```

```

    struct task* tmp=malloc(sizeof(struct task));
    tmp->name=name;
    tmp->priority=priority;
    tmp->burst=burst;
    tmp->tid=__sync_fetch_and_add(&task_id,1); //分配tid
    insert(&head,tmp);
    num++;
}

// invoke the scheduler
void schedule(){
    int tsk_num=num;
    while(num){
        struct node *cur=head;
        while(cur->next!=NULL){
            cur=cur->next;
        }
        run(cur->task,cur->task->burst);

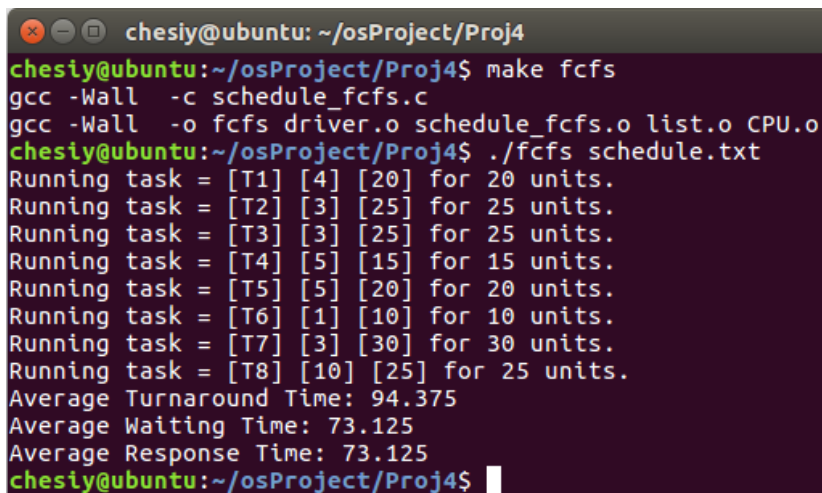
        curtime+=cur->task->burst;
        turnaround+=curtime;
        wait+=pretime;
        pretime+=cur->task->burst;

        delete(&head,cur->task);
        free(cur->task);
        num--;
    }
    double avg_turnaround = 1.0 * turnaround / tsk_num;
    double avg_wait = 1.0 * wait / tsk_num;
    double avg_response = avg_wait;

    printf("Average Turnaround Time: %.3lf\n", avg_turnaround);
    printf("Average Waiting Time: %.3lf\n", avg_wait);
    printf("Average Response Time: %.3lf\n", avg_response);
}

```

运行结果如下。



```

chesiy@ubuntu: ~/osProject/Proj4
chesiy@ubuntu:~/osProject/Proj4$ make fcfs
gcc -Wall -c schedule_fcfs.c
gcc -Wall -o fcfs driver.o schedule_fcfs.o list.o CPU.o
chesiy@ubuntu:~/osProject/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.
Average Turnaround Time: 94.375
Average Waiting Time: 73.125
Average Response Time: 73.125
chesiy@ubuntu:~/osProject/Proj4$

```

2.2 SJF

SJF算法每次执行CPU burst最小的进程。因此在 `schedule` 函数中，每次选择要执行的进程都会遍历链表，找到CPU burst最小的那个，执行后删除。此外，SJF算法计算各个时间的方法与FCFS几乎一致，故不赘述。具体代码如下。

```
struct node *head;
int num=0;
int task_id=0;

int curtime=0;
int pretime=0;
int turnaround=0;
int wait=0;
int response=0;

// add a task to the list
void add(char *name, int priority, int burst){
    struct task* tmp=malloc(sizeof(struct task));
    tmp->name=name;
    tmp->priority=priority;
    tmp->burst=burst;
    tmp->tid=__sync_fetch_and_add(&task_id,1); //分配tid
    insert(&head,tmp);
    num++;
}

// invoke the scheduler
void schedule(){
    int tsk_num=num;
    while(num){
        struct node *mini=head;
        struct node *cur=head;
        while(cur!=NULL){
            if(cur->task->burst<mini->task->burst){
                mini=cur;
            }
            cur=cur->next;
        }
        run(mini->task,mini->task->burst);

        curtime+=mini->task->burst;
        turnaround+=curtime;
        wait+=pretime;
        pretime+=mini->task->burst;

        delete(&head,mini->task);
        free(mini->task);
        num--;
    }

    double avg_turnaround = 1.0 * turnaround / tsk_num;
    double avg_wait = 1.0 * wait / tsk_num;
    double avg_response = avg_wait;

    printf("Average Turnaround Time: %.3lf\n", avg_turnaround);
    printf("Average Waiting Time: %.3lf\n", avg_wait);
    printf("Average Response Time: %.3lf\n", avg_response);
}
```

运行结果如下。

```
chesiy@ubuntu:~/osProject/Proj4$ make sjf
gcc -Wall -c schedule_sjf.c
gcc -Wall -o sjf driver.o schedule_sjf.o list.o CPU.o
chesiy@ubuntu:~/osProject/Proj4$ ./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: 82.500
Average Waiting Time: 61.250
Average Response Time: 61.250
chesiy@ubuntu:~/osProject/Proj4$
```

2.3 Priority Scheduling

Priority算法每次执行优先级最高的进程。因此在 `schedule` 函数中，每次选择要执行的进程都会遍历链表，找到优先级最高的那个，执行后删除。此外，Priority算法计算各个时间的方法与前两种几乎一致，故不赘述。由于Priority算法与SJF算法在实现上的区别仅为判断条件一个是优先级最高，一个是耗时最短，故仅展示部分代码。

```
// invoke the scheduler
void schedule(){
    int tsk_num=num;
    while(num){
        struct node *maxi=head;
        struct node *cur=head;
        while(cur!=NULL){
            if(cur->task->priority>=maxi->task->priority){
                maxi=cur;
            }
            cur=cur->next;
        }
        run(maxi->task,maxi->task->burst);

        curtime+=maxi->task->burst;
        turnaround+=curtime;
        wait+=pretime;
        pretime+=maxi->task->burst;

        delete(&head,maxi->task);
        free(maxi->task);
        num--;
    }
}
```

运行结果如下。

```

chesiy@ubuntu:~/osProject/Proj4$ make priority
gcc -Wall -c schedule_priority.c
gcc -Wall -o priority driver.o schedule_priority.o list.o CPU.o
chesiy@ubuntu:~/osProject/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.
Average Turnaround Time: 96.250
Average Waiting Time: 75.000
Average Response Time: 75.000
chesiy@ubuntu:~/osProject/Proj4$

```

2.4 RR

RR算法与之前的三个算法差别较大。此处设定时间片为10ms，故每个进程每次最多只能执行10ms。10ms后，即使未执行完，也会轮到之后的进程，如此循环往复。因此在 `schedule` 函数中，每次选择要执行的进程，若未执行完则不能删除。此外，还需要每次修改CPU burst。由于最早进入的进程在链表中被放在最后，因此需要先把当前进程指向链表的末尾，每次执行完当前进程后选定下一进程（在链表中位于当前进程的前一个）。

为了计算RR算法的平均响应时间，还需在 `task` 结构体中增加两项——`response_time` 用于记录该进程的响应时间，`first` 则用于表示该进程是否是第一次被执行（若非第一次，不可更新 `response_time`）。

```

typedef struct task {
    char *name;
    int priority;
    int burst;
    int tid;
    int first; //whether the task is executed the first time;
    int response_time;
} Task;

```

每执行一次时间片，都更新当前时间，假如当前任务执行完，则更新总周转时间。总等待时间可以通过总周转时间减去所有任务的执行时间得到。

```

struct node *head;
int num=0;
int task_id=0;

int curtime=0;
int turnaround=0;
int wait=0;
int response=0;
int burst_sum=0;

// add a task to the list
void add(char *name, int priority, int burst){
    struct task* tmp=malloc(sizeof(struct task));
    tmp->name=name;
    tmp->priority=priority;
    tmp->burst=burst;
    tmp->tid=__sync_fetch_and_add(&task_id,1); //分配tid
}

```

```

tmp->response_time=0;
tmp->first=1;
insert(&head,tmp);
num++;
burst_sum+=burst;
}

// invoke the scheduler
void schedule(){
    int tsk_num=num;

    struct node *cur=head;
    struct node *pre;
    while(cur->next!=NULL){
        cur=cur->next;
    }
    pre=cur;
    while(num){
        if(cur->task->burst-10>0){
            run(cur->task,10);

            if(cur->task->first==1){
                response+=curtime;
                cur->task->first=0;
            }
            curtime+=10;

            cur->task->burst-=10;
            cur=head;
            while(cur->next!=pre&&cur->next!=NULL){
                cur=cur->next;
            }
            pre=cur;
            if(cur==head){
                pre=NULL;
            }
        }else{
            run(cur->task,cur->task->burst);

            if(cur->task->first==1){
                response+=curtime;
                cur->task->first=0;
            }
            curtime+=cur->task->burst;
            turnaround+=curtime;

            cur->task->burst=0;
            pre=cur;

            while(pre->next!=cur&&pre->next!=NULL){
                pre=pre->next;
            }
            delete(&head,cur->task);
            free(cur->task);
            num--;
            cur=pre;
        }
    }
}

```

```

wait=turnaround-burst_sum;
double avg_turnaround = 1.0 * turnaround / tsk_num;
double avg_wait = 1.0 * wait / tsk_num;
double avg_response = 1.0 * response / tsk_num;

printf("Average Turnaround Time: %.3lf\n", avg_turnaround);
printf("Average Waiting Time: %.3lf\n", avg_wait);
printf("Average Response Time: %.3lf\n", avg_response);
}

```

运行结果如下。

```

chesiy@ubuntu:~/osProject/Proj4$ make rr
gcc -Wall -c schedule_rr.c
gcc -Wall -o rr driver.o schedule_rr.o list.o CPU.o
chesiy@ubuntu:~/osProject/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.
Average Turnaround Time: 128.750
Average Waiting Time: 107.500
Average Response Time: 35.000

```

2.5 Priority with RR

Priority with RR算法与RR算法较为类似，但Priority with RR只在相同优先级的进程之间轮转。为了简化 `schedule` 函数，我定义了一个node类型的指针数组（即10个链表，分别储存不同优先级的进程），因此在 `add` 函数中，也是按照进程优先级存入不同的链表中。

相同优先级的调度算法实现与之前的RR算法一样，都是每10ms轮换一次。不同优先级之间则按从高到低先后进行。

计算该算法的各项时间方法与RR算法非常相似，故不再赘述。具体代码如下。

```

struct node *head[10];
int num[10]={0};
int task_id=0;
int tsk_num=0;

int curtime=0;
int turnaround=0;
int wait=0;
int response=0;

```

```

int burst_sum=0;

// add a task to the list
void add(char *name, int priority, int burst){
    struct task* tmp=malloc(sizeof(struct task));
    tmp->name=name;
    tmp->priority=priority;
    tmp->burst=burst;
    tmp->tid=__sync_fetch_and_add(&task_id,1); //分配tid
    tmp->response_time=0;
    tmp->first=1;
    insert(&head[priority-1],tmp);
    num[priority-1]++;
    burst_sum+=burst;
    tsk_num++;
}

// invoke the scheduler
void schedule(){
    for(int i=9;i>=0;i--){
        struct node *cur=head[i];
        struct node *pre;
        if(cur==NULL){
            continue;
        }
        while(cur->next!=NULL){
            cur=cur->next;
        }
        pre=cur;
        while(num[i]){
            if(cur->task->burst-10>0){
                run(cur->task,10);

                if(cur->task->first==1){
                    response+=curtime;
                    cur->task->first=0;
                }
                curtime+=10;

                cur->task->burst-=10;
                cur=head[i];
                while(cur->next!=pre&&cur->next!=NULL){
                    cur=cur->next;
                }
                pre=cur;
                if(cur==head[i]){
                    pre=NULL;
                }
            }else{
                run(cur->task,cur->task->burst);

                if(cur->task->first==1){
                    response+=curtime;
                    cur->task->first=0;
                }
                curtime+=cur->task->burst;
                turnaround+=curtime;
            }
        }
    }
}

```



```

        cur->task->burst=0;
        pre=head[i];

        while(pre->next!=cur&&pre->next!=NULL){
            pre=pre->next;
        }
        delete(&head[i], cur->task);
        free(cur->task);
        num[i]--;
        if(num[i]){
            cur=pre;
        }
    }
}

wait=turnaround-burst_sum;
double avg_turnaround = 1.0 * turnaround / tsk_num;
double avg_wait = 1.0 * wait / tsk_num;
double avg_response = 1.0 * response / tsk_num;

printf("Average Turnaround Time: %.3lf\n", avg_turnaround);
printf("Average Waiting Time: %.3lf\n", avg_wait);
printf("Average Response Time: %.3lf\n", avg_response);
}

```

运行结果如下。

```

chesiy@ubuntu:~/osProject/Proj4$ make priority_rr
gcc -Wall -c -o schedule_priority_rr.o schedule_priority_rr.c
gcc -Wall -o priority_rr driver.o schedule_priority_rr.o list.o CPU.o
chesiy@ubuntu:~/osProject/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.
Average Turnaround Time: 105.000
Average Waiting Time: 83.750
Average Response Time: 68.750
chesiy@ubuntu:~/osProject/Proj4$

```

3. 总结与反思

这次project的前三种算法实现较为容易，后两种则相对复杂，但只要熟悉各个算法的原理，都可以顺利完成。在做project的过程中，我花了较长的时间去读懂给出的代码和makefile，理顺这些代码文件之间的关系，这使得我真正开始写代码后逻辑顺畅。总之，通过这个project，我对进程调度的几种算法掌握更加熟练，也复习了如何计算平均周转时间、平均等待时间和平均响应时间。