

# OS Lab6 实验报告

## 实验目标

1. 在Lab5的基础上，实现FCFS、SJF、优先级、RR四种调度方法，并将其模块化封装

## 源代码说明

由于本次实验没有提供框架，所以源代码比较混乱，在报告中仅展示关键部分

`task.c` 文件中，原Lab5中实现的 FCFS 函数，被替代为 `scheduler` 类型（与文档中大致相同）的调度器函数，而对应算法的实现则位于 `taskScheduler` 文件夹中，实现了模块化封装，如下所示

```
1  typedef struct scheduler {
2      unsigned int type;
3      int preemptiveOrNot;    //if True, the scheduler is preemptive
4      myTCB* (*nextTaskFunc)(void);
5      void(*enqueueTaskFunc)(myTCB* task);
6      void(*dequeueTaskFunc)();
7      void(*schedulerInitFunc)(myTCB* task);
8      void(*createTaskHook)(myTCB* task, void (*taskBody)(void));    //if
// set, will be called in createTask (before taskStart)
9      void(*tickHook)(void);    //if set, tickHook will be called every tick
10 }scheduler;
11
12 //进程池中一个未就绪队列中的TCB的开始
13 void taskStart(myTCB* task) {
14     task->TASK_State = TASK_READY;
15     //将一个未就绪队列中的TCB加入到就绪队列
16     sch.enqueueTaskFunc(task);
17 }
```

调度方法的设置位于 `osStart.c` 文件中，在 `TaskManagerInit()` 函数前进行

```
1  void setScheduler(int scheduleMethod) {
2      switch (scheduleMethod) {
3          case FCFS:
4              setSchedulerFCFS(&sch);
5              break;
6          case PRIIO:
7              setSchedulerPrio(&sch);
8              break;
9          case SJF:
10             setSchedulerSJF(&sch);
11             break;
12          case RR:
13             setSchedulerRR(&sch);
14          default:
15             myPrintk(0x02, "Invalid scheduler type.");
16             break;
17     }
```

在 `myTCB` 类型中增加优先级、执行时间和到达时间参数

```

1 // struct for taskPara
2 typedef struct taskPara {
3     unsigned int priority;
4     unsigned int exeTime;
5     unsigned int arrTime;
6 } taskPara;
7
8 // #error "TODO: 为 myTCB 增补合适的字段"
9 typedef struct myTCB {
10     unsigned long* stackTop;          /* 栈顶指针 */
11     unsigned long stack[STACK_SIZE]; /* 开辟了一个大小为STACK_SIZE的栈空间 */
12     unsigned long TASK_State;         /* 进程状态 */
13     unsigned long TASK_ID;           /* 进程ID */
14     taskPara paras;
15     void (*task_entrance)(void);     /* 进程的入口地址 */
16     struct myTCB* rqNextTCB;         /* 就绪队列中下一个TCB */
17     struct myTCB* wqNextTCB;         /* 等待队列中下一个TCB */
18 } myTCB;

```

为实现到达时间和执行时间，利用 `tickHook()` 函数，每 tick 运行一次，将等待队列中的任务的到达时间减一，将当前执行的任务的执行时间减一；当等待队列中任意任务的到达时间为零时，将其移出等待队列，移进就绪队列

另外创建任务时，若到达时间为0，则直接进入就绪队列

**对于RR调度，当一个时间片结束时，将当前就绪队列队头任务移到队尾，并进行一次调度**

```

1 void tickHookFCFS() {
2     myTCB* task = wq.head;
3     while (task) {
4         if (task->TASK_State == TASK_WAIT) {
5             task->paras.arrTime--;
6             if (task->paras.arrTime == 0) {
7                 task->TASK_State = TASK_READY;
8                 taskDequeueWqFCFS(task);
9                 taskEnqueueRqFCFS(task);
10            }
11        }
12        task = task->wqNextTCB;
13    }
14    if (currentTask->paras.exeTime) {
15        currentTask->paras.exeTime--;
16    }
17 }
18
19 // 创建任务并加入等待队列，若到达时间为0则加入就绪队列
20 void createTaskFCFS(myTCB* task, void (*taskBody)(void)) {
21     task->task_entrance = taskBody;
22     stack_init(&(task->stackTop), taskBody);
23     task->TASK_State = TASK_WAIT;

```

```

24     if (task->paras.arrTime == 0) {
25         task->TASK_State = TASK_READY;
26         taskEnqueueRqFCFS(task);
27     }
28     else {
29         taskEnqueueWqFCFS(task);
30     }
31 }
32
33 //对于RR调度，当一个时间片结束时，将当前就绪队列队头任务移到队尾，并进行一次调度
34 void tickHookRR() {
35     myTCB* task = wq.head;
36     while (task) {
37         if (task->TASK_State == TASK_WAIT) {
38             task->paras.arrTime--;
39             if (task->paras.arrTime == 0) {
40                 task->TASK_State = TASK_READY;
41                 taskDequeueWqRR(task);
42                 taskEnqueueRqRR(task);
43             }
44         }
45         task = task->wqNextTCB;
46     }
47     if (currentTask->paras.exeTime) {
48         currentTask->paras.exeTime--;
49     }
50     if (rq.head != rq.tail) {
51         if (tick_number % TIME_SLICE == 0) {
52             task = rq.head;
53             taskDequeueRqRR();
54             taskEnqueueRqRR(task);
55             schedule();
56         }
57     }
58 }

```

在优先级和SJF调度中，我选择在任务进入就绪队列时就已优先级进行排序，这样调度时只需要将下一个任务设为当前就绪队列头即可

**当入队的任务的优先级比当前任务的优先级更高（执行时间比当前任务更短）时，抢占当前任务**

```

1 //将一个未在就绪队列中的TCB按优先数升序加入到就绪队列中
2 void taskEnqueueRqPrio(myTCB* task) { //将task入队rq
3     if (rqIsEmptyPrio()) {
4         rq.head = task;
5         rq.tail = task;
6     }
7     else if (rq.head == rq.tail) {
8         if (task->paras.priority < rq.head->paras.priority) {
9             task->rqNextTCB = rq.head;
10            rq.head = task;
11            schedule(); //当入队的任务的优先级比当前任务的优先级更高时，抢占当前任务
12            return;
13        }
14        else {

```

```

15         rq.tail->rqNextTCB = task;
16         task->rqNextTCB = NULL;
17         rq.tail = task;
18     }
19 }
20 else {
21     if (task->paras.priority < rq.head->paras.priority) {
22         task->rqNextTCB = rq.head;
23         rq.head = task;
24         schedule(); //当入队的任务的优先级比当前任务的优先级更高时，抢占当前任务
25         return;
26     }
27     myTCB* temp = rq.head;
28     while (temp->rqNextTCB) {
29         if (task->paras.priority < temp->rqNextTCB->paras.priority) {
30             task->rqNextTCB = temp->rqNextTCB;
31             temp->rqNextTCB = task;
32             break;
33         }
34         temp = temp->rqNextTCB;
35     }
36     if (!(temp->rqNextTCB)) {
37         temp->rqNextTCB = task;
38         task->rqNextTCB = NULL;
39         rq.tail = task;
40     }
41 }
42 }

```

## 实验运行结果

为方便展示，删去了shell任务

创建的任务如图所示，每当 `tick_number` 是10的倍数时，输出当前 `tick_number`，共输出10次。可能是由于 `tick_number` 变化太快，程序在两次 `tick` 之间无法运行完一次循环，所以不是每一次 `tick_number` 是10的倍数时，都能够输出当前 `tick_number`，当输出10次时，能够保证任务执行时间在1s与2s之间

三个任务输出的颜色分别为绿 (0x2)、紫 (0x5)、蓝 (0x3)

```

1 void myTask0(void) {
2     myPrintf(WHITE, message1);
3     myPrintf(WHITE, "*      Task0: TASK BEGIN!      *\n");
4     int times = 0, temp = 0;
5     while (times < 10) {
6         if (temp != tick_number) {
7             if (tick_number % 10 == 0) {
8                 myPrintf(COLOR0, "%d ", tick_number);
9                 times++;
10            }
11        }
12        temp = tick_number;
13    }
14    myPrintf(WHITE, "\n");
15    myPrintf(RED, "*      Task0: TASK END!      *\n");

```

```

16     myPrintf(RED, message1);
17
18     taskEnd();    //the task is end
19 }

```

设置三个任务的执行时间均为200（2s），到达时间如图所示

```

1 createTask(myTask0, (taskPara) { 3, 190, 30 }); //priority, exeTime, arrTime
2 createTask(myTask1, (taskPara) { 2, 200, 0 });
3 createTask(myTask2, (taskPara) { 1, 210, 60 });

```

## FCFS

符合到达时间的顺序

```

Machine View
*****
*          INIT   INIT !          *
*****
*          Task1: TASK BEGIN!      *
10 20 50 80 100 110 120 130 160 170
*          Task1: TASK END!        *
*****
*          Task0: TASK BEGIN!      *
210 270 280 290 310 320 330 340 370 400
*          Task0: TASK END!        *
*****
*          Task2: TASK BEGIN!      *
400 420 430 460 470 490 500 510 520 530
*          Task2: TASK END!        *
*****

```

19:00:05

## SJF

30时间单位时，Task0进入就绪队列，执行时间为190，而此时Task1执行时间为200-30=170，因此无抢占。同理，Task2也无抢占

```
QEMU
Machine View
*****
*      INIT    INIT !      *
*****
*      Task1: TASK BEGIN!   *
10 50 60 70 80 110 120 130 140 150
*      Task1: TASK END!     *
*****
*      Task0: TASK BEGIN!   *
220 230 240 250 280 300 310 350 360 370
*      Task0: TASK END!     *
*****
*      Task2: TASK BEGIN!   *
400 410 430 440 450 460 470 500 510 520
*      Task2: TASK END!     *
*****
19:00:04
```

## 优先级

首先执行最先进入就绪队列的Task1，Task2在60时间单位时抢占Task1开始执行。200时间单位后，Task2执行完毕，Task1剩余部分开始执行。140时间单位后，优先级最低的Task0开始执行

```
QEMU
Machine View
*****
*      INIT    INIT !      *
*****
*      Task1: TASK BEGIN!   *
10 20 30 40 50 60 *****
*      Task2: TASK BEGIN!   *
70 100 110 120 140 150 160 170 180 190
*      Task2: TASK END!     *
*****
280 290 300 320
*      Task1: TASK END!     *
*****
*      Task0: TASK BEGIN!   *
420 430 440 450 460 490 500 510 520 530
*      Task0: TASK END!     *
*****
19:00:07
```

## RR

时间片设置为60时间单位

```
QEMU
Machine View
*****
*          INIT    INIT !          *
*****
*****
*      Task1: TASK BEGIN!          *
10 40 *****
*      Task0: TASK BEGIN!          *
60 70 80 90 130 140 160 *****
*      Task2: TASK BEGIN!          *
180 200 220 240 250 280 310 320 330 340 360 370 390 400 410 420 430 450
*      Task0: TASK END!            *
*****
480 490
*      Task2: TASK END!            *
*****
570
*      Task1: TASK END!            *
*****
19:00:06
```

将测试任务改为

```
1 | createTask(myTask0, (taskPara) { 3, 180, 10 }); //priority, exeTime, arrTime
2 | createTask(myTask1, (taskPara) { 2, 200, 0 });
3 | createTask(myTask2, (taskPara) { 1, 220, 20 });
```

使用SJF调度，即可体现出抢占（Task0抢占Task1）

```
QEMU
Machine View
*****
*          INIT    INIT !          *
*****
*****
*      Task1: TASK BEGIN!          *
10 *****
*      Task0: TASK BEGIN!          *
20 30 60 70 80 90 120 130 140 160
*      Task0: TASK END!            *
*****
200 220 240 250 260 270 290 300 310
*      Task1: TASK END!            *
*****
*      Task2: TASK BEGIN!          *
390 400 410 420 480 490 500 510 520 530
*      Task2: TASK END!            *
*****
19:00:05
```

## 遇到的问题解决方法

在设置调度器之前就调用了调度器中的函数，导致程序卡死