

# Lab 6 实验报告

## 实验内容

本实验实现了两种调度算法：

1. RR（时间片轮转）调度算法
2. 优先级（非抢占）调度算法

## 实验原理

### 1. RR 调度算法

RR 调度算法的基本思想是，设置一个时间片，每个任务每次最多能获得一个时间片的时间，若该任务运行时间少于时间片，则运行完后直接从就绪队列调入下个任务；若该任务运行时间大于一个时间片，则当时间片用完时，该任务会被抢占，并插入到就绪队列队尾，就绪队列的下个任务会被调入运行。

从 RR 调度算法内容可以看到，RR 调度是抢占调度。所以，当某个任务被抢占时，我们需要通过**中断**的方式实现调度。由于之前的实验中，我们采用过时钟中断，即每隔相同的时间发出一次中断。所以我们可以利用时钟中断的方法，每隔一段时间进行一次 RR 调度，这样即可实现抢占式调度。

具体实现方式为，在 `tick()` 函数内添加中断函数表，每次时钟中断时会把函数表里的函数依次运行一遍，具体代码如下：

```
void tick(void){
    tick_number++;

    oneTickUpdateWallClock();

    if(tick_hook) tick_hook(); //user defined

    //每次时钟中断会把中断函数表里的函数执行一遍
    //这样可以使得每次 tick 中断能够运行调度函数
    int i=0;
    for(i=0;i<func_num;i++){
        func_list[i]();
    }
}
```

同时我们还应提供维护中断函数表的函数，具体如下：

```
#define MAX_FUNC 16
```

```
void (*func_list[MAX_FUNC])(void);
```

```
int func_num=0;
```

```
void append_funclist(void(*func)(void)){  
    func_list[func_num++]=func;  
}
```

```
void clear_funclist(void){  
    func_num=0;  
}
```

append\_funclist 函数能增加中断函数表项，而 clear\_funclist 可以直接清空中断函数表。

由此我们实现了如何进行中断。接下来，我们利用时钟中断的方法实现 RR 调度：

```
void RRSchedule(void){  
    //需加进 func_list  
    disable_interrupt();  
    myTCB* nextTsk;  
    //myPrintk(0x7, "succeed\n");  
    if(tick_number%TIME_SLICE==0){  
        //时间片已到  
        //showReadyQueue();  
        tick_number++;  
        //myPrintk(0x7, "succeed\n");  
        if(currentTsk->TSK_State==TSK_DONE){  
            //如果已经完成现在的任务  
            //则无需将其入队  
            nextTsk = RRDequeue(&RR_Ready_queue);  
            if(nextTsk==NULL){  
                enable_interrupt();  
                return;  
            }  
        }  
        else{  
            context_switch(currentTsk,nextTsk);  
            enable_interrupt();  
            return;  
        }  
    }  
}
```

```

    }
}
else{
    //没有完成任务
    currentTsk->TSK_State=TSK_WAIT;
    RREnqueue(&RR_Ready_queue,currentTsk);
    nextTsk = NextRRTsk(&RR_Ready_queue);
    //myPrintk(0x9,"yes\n");
    if(nextTsk==NULL){
        enable_interrupt();
        return;
    }
    else{
        //myPrintk(0x9,"yes\n");
        //showReadyQueue();
        RRDequeue(&RR_Ready_queue);
        if(currentTsk==nextTsk){
            enable_interrupt();
            return;
        }
        else
            context_switch(currentTsk,nextTsk);
        return;
    }
}
}
else if(currentTsk->TSK_State==TSK_DONE){
    //showReadyQueue();
    //myPrintk(0x7,"\nFinish!\n");
    nextTsk = RRDequeue(&RR_Ready_queue);
    if(nextTsk==NULL){
        //myPrintk(0x7,"\nWin!\n");
        if(RRqueue_empty(&RR_Arriv_queue)){
            clear_funclist();
            context_switch(currentTsk,initial_task);
        }
        enable_interrupt();
        return;
    }
}

```

```

    }
    else{
        //myPrintk(0x7, "\nFail!\n");
        context_switch(currentTsk,nextTsk);
        enable_interrupt();
        return;
    }
}
}
}

```

上述代码大致可分为两种情况：

- 1.时间片已用完，如果当前任务已经结束，则从就绪队列调入新任务；否则，将当前任务入队，并抢占调入新任务；
- 2.时间片未用完，如果当前任务已经结束，则调入新任务，在调入过程中如果发现就绪队列为空且即将到达任务的队列也为空，此时证明所有任务已经结束，故切换上下文到最初任务（initial\_task，即 shell 任务），此时可输入其他 shell 命令。

这里我们利用开中断和关中断避免调度过程中计时和抢占调度，context\_switch 中也有开关中断，因为我们修改了 CTX\_SW 的汇编代码：

CTX\_SW:

```

    call disable_interrupt #stop timer

    pushf
    pusha

    movl prevTSK_StackPtr, %eax
    movl %esp, (%eax)
    movl nextTSK_StackPtr, %esp

    popa
    popf

    call enable_interrupt #begin timer

    ret

```

我们除了需要 RRSchedule 中断函数外，还需要根据每个任务的到达时间将任务插入就绪队列的中断函数 RRArrivSchedule，代码如下：

```

void RRArrivSchedule(void){
    //需加进 func_list
    disable_interrupt();
    myTCB * arrivTsk;

```

```

arrivTsk=NextRRTsk(&RR_Arriv_queue);
if(arrivTsk==NULL){
    enable_interrupt();
    return;
}
else{
    while((arrivTsk!=NULL)&&(arrivTsk->TskPara.arrTime<=tick_number)){
        arrivTsk=RRDequeue(&RR_Arriv_queue);
        RREnqueue(&RR_Ready_queue, arrivTsk);
        arrivTsk=arrivTsk->arriv_nextTCB;
    }
}
enable_interrupt();
}

```

这里的即将到达队列 `RR_Arriv_queue` 是根据到达时间从低到高的顺序来维护的链表（维护方法见下面优先级调度），所以我们只看队头是否符合到达时间要求即可。

## 2. 优先级调度（非抢占）

优先级调度，即优先调度就绪队列中优先级较高的任务的算法。

优先级调度与 RR 调度非常类似，但不同的一点是，由于这里我们实现的是非抢占型优先级调度算法，所以应该等到任务结束才开始调入下个任务。

我们仍需和 RR 调度一样，添加两个中断函数。其中将任务从即将到达队列加入至就绪队列的中断函数 `PrioArrivSchedule` 和 RR 调度几乎一模一样，这里不再赘述。

重点是优先级调度函数，即 `PrioSchedule`，其代码如下：

```

void PrioSchedule(){
    //非抢占式调度
    //如果 arriv queue 无内容则返回 shell
    disable_interrupt();
    myTCB *nextTsk;
    if(currentTsk->TSK_State==TSK_DONE){
        //任务完成才调度，非抢占
        nextTsk=PrioDequeue(&Prio_Ready_queue);
        if(nextTsk==NULL){
            if(Prioqueue_empty(&Prio_Arriv_queue)){
                //myPrintk(0x7,"yes");
                clear_funclist();
                context_switch(currentTsk,initTsk);
            }
        }
        enable_interrupt();
    }
}

```

```

        return;
    }
    else{
        context_switch(currentTsk,nextTsk);
        enable_interrupt();
        return;
    }
}
else{
    enable_interrupt();
    return;
}
}

```

可以看到只有在当前任务完成时才会调入新的任务, 同样地如果就绪队列和即将到达任务队列皆为空, 则返回初始任务, 继续等待 shell 指令的输入。

这里就绪队列和即将到达队列一样, 是通过优先级数/到达时间从低到高顺序来维护的。而维护方法则是插入时维护。下面是就绪队列插入函数的代码:

```

void PrioEnqueue(Prioqueue *queue,myTCB *newtsk){
    myTCB *p;
    int priority,arrrtime;
    if(queue->queueType==READY_QUEUE){
        //showArrivQueue();
        //按照优先级顺序入队
        if(Prioqueue_empty(queue)){
            queue->head=newtsk;
            queue->tail=newtsk;
            newtsk->nextTCB=NULL;
            newtsk->TSK_State=TSK_WAIT;
            return;
        }
        else{
            p=queue->head;
            priority=newtsk->TSkPara.priority;
            //myPrintk(0x7, "arrrtime=%d\n", arrrtime);
            if(priority<=p->TSkPara.priority){
                //前插
                //myPrintk(0x7, "forwarding:%d\n", newtsk->TSK_ID);
                newtsk->nextTCB=p;
                queue->head=newtsk;
            }
        }
    }
}

```

```

        return;
    }
    else if(queue->head==queue->tail){
        //后插
        //myPrintk(0x7, "backing:%d\n", newtsk->TSK_ID);
        p->nextTCB=newtsk;
        queue->tail=newtsk;
        newtsk->nextTCB=NULL;
    }
    else{
        while((p!=queue->tail)&&((p->nextTCB)->TSkPara.priority<=priority))
            p=p->nextTCB;
        if(p==queue->tail){
            queue->tail=newtsk;
            newtsk->nextTCB=NULL;
        }
        newtsk->nextTCB=p->nextTCB;
        p->nextTCB=newtsk;
        return;
    }
}
}
else{
    //showArrivQueue();
    //按照到达时间顺序入队
    if(Prioqueue_empty(queue)){
        queue->head=newtsk;
        queue->tail=newtsk;
        newtsk->arriv_nextTCB=NULL;
        newtsk->TSK_State=TSK_WAIT;
        return;
    }
    else{
        p=queue->head;
        arrtime=newtsk->TSkPara.arrTime;
        //myPrintk(0x7, "arrtime=%d\n", arrtime);
        if(arrtime<=p->TSkPara.arrTime){
            //前插

```

```

        //myPrintk(0x7,"forwarding:%d\n",newtsk->TSK_ID);

        newtsk->arriv_nextTCB=p;

        queue->head=newtsk;

        return;
    }

    else if(queue->head==queue->tail){

        //后插

        //myPrintk(0x7,"backing:%d\n",newtsk->TSK_ID);

        p->arriv_nextTCB=newtsk;

        queue->tail=newtsk;

        newtsk->arriv_nextTCB=NULL;

    }

    else{

        while((p!=queue->tail)&&((p->arriv_nextTCB)->TSkPara.arrTime<=arrtime))

            p=p->arriv_nextTCB;

        if(p==queue->tail){

            queue->tail=newtsk;

            newtsk->arriv_nextTCB=NULL;

        }

        newtsk->arriv_nextTCB=p->arriv_nextTCB;

        p->arriv_nextTCB=newtsk;

        return;

    }

}

}

}

```

# 测试用例

## 1.RR 调度

这里 RR 调度测试使用的是三个任务函数：

```

void RRTsk1(void){
    unsigned long i,j,k;
    for(i=0;i<50;i++){
        myPrintf(0x5," tsk1:%d",i);
        for(j=0;j<500000;j++){

```



```

        k+=j*j;
    }//delay
}
RRTskEnd();
}

void RRTsk2(void){
    unsigned long i,j,k;
    for(i=0;i<26;i++){
        myPrintf(0x7," tsk2:%c",'a'+i);
        for(j=0;j<500000;j++){
            k+=j*j;
        }//delay
    }
    RRTskEnd();
}

void RRTsk3(void){
    unsigned long i,j,k;
    for(i=0;i<26;i++){
        myPrintf(0x8," tsk3:%c",'A'+i);
        for(j=0;j<500000;j++){
            k+=j*j;
        }//delay
    }
    RRTskEnd();
}

```

其中每个函数依次输出字符串，不同任务输出的字符串颜色不同，方便识别。同时每次输出字符串时，会有个循环来延迟，增加每次输出字符串的间隔时间和每个任务的总时间，使得每个任务实际运行时间应大于时间片，方便观察 RR 抢占调度的效果。

## 2. 优先级调度（非抢占）

与 RR 调度类似，优先级调度的任务函数如下：

```

void PrioTsk1(void){
    unsigned long i,j,k;
    for(i=0;i<50;i++){
        myPrintf(0x5," tsk1:%d",i);
        for(j=0;j<500000;j++){
            k+=j*j;
        }//delay
    }
}

```

```

    }
    PrioTskEnd();
}

void PrioTsk2(void){
    unsigned long i,j,k;
    for(i=0;i<26;i++){
        myPrintf(0x7," tsk2:%c",'a'+i);
        for(j=0;j<500000;j++){
            k+=j*j;
            k-=j*j;
        }//delay
    }
    PrioTskEnd();
}

void PrioTsk3(void){
    unsigned long i,j,k;
    for(i=0;i<26;i++){
        myPrintf(0x8," tsk3:%c",'A'+i);
        for(j=0;j<500000;j++){
            k+=j*j;
            k-=j*j;
        }//delay
    }
    PrioTskEnd();
}

void PrioTsk4(void){
    unsigned long i,j,k;
    for(i=50;i<100;i++){
        myPrintf(0x4," tsk4:%d",i);
        for(j=0;j<500000;j++){
            k+=j*j;
        }//delay
    }
    PrioTskEnd();
}

```

# 实验结果与分析

注：测试结果建议看 QUME 界面（因为有颜色区分）。本实验环境为 vlab 虚拟机，性能可能不如助教的电脑，所以助教测试 RR 调度时如果没有显示出抢占的效果，可以修改 `src/myOS/kernel/RR_schedule.c` 文件中：

```
#define TIME_SLICE 6
```

将时间片调小。

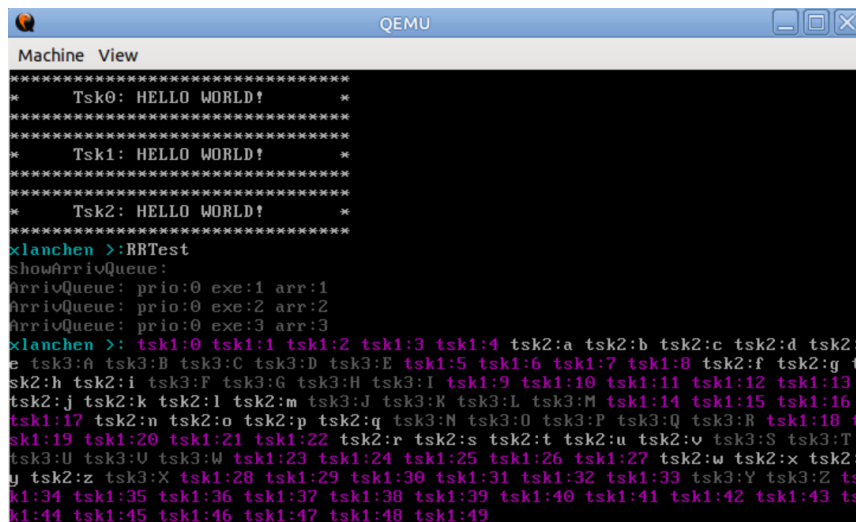
## 1. RR 调度测试

我们添加指令"RRTest"来测试 RR 调度，每个测试任务设置如下：

```
RRCreateTsk(tskBody: RRTsk1, priority: 0, exetime: 1, arftime: 1);
RRCreateTsk(tskBody: RRTsk2, priority: 0, exetime: 2, arftime: 2);
RRCreateTsk(tskBody: RRTsk3, priority: 0, exetime: 3, arftime: 3);
```

这里只有 `arftime` 重要，其他参数并不会用上，执行时间我们以各任务的实际运行时间为准。

测试结果：



```
Machine View
*****
*   Tsk0: HELLO WORLD!   *
*****
*   Tsk1: HELLO WORLD!   *
*****
*   Tsk2: HELLO WORLD!   *
*****
xlanchen >:RRTest
showArriQueue:
ArriQueue: prio:0 exe:1 arr:1
ArriQueue: prio:0 exe:2 arr:2
ArriQueue: prio:0 exe:3 arr:3
xlanchen >: tsk1:0 tsk1:1 tsk1:2 tsk1:3 tsk1:4 tsk2:a tsk2:b tsk2:c tsk2:d tsk2:
e tsk3:A tsk3:B tsk3:C tsk3:D tsk3:E tsk1:5 tsk1:6 tsk1:7 tsk1:8 tsk2:f tsk2:g t
sk2:h tsk2:i tsk3:F tsk3:G tsk3:H tsk3:I tsk1:9 tsk1:10 tsk1:11 tsk1:12 tsk1:13
tsk2:j tsk2:k tsk2:l tsk2:m tsk3:J tsk3:K tsk3:L tsk3:M tsk1:14 tsk1:15 tsk1:16
tsk1:17 tsk2:n tsk2:o tsk2:p tsk2:q tsk3:N tsk3:O tsk3:P tsk3:Q tsk3:R tsk1:18 t
sk1:19 tsk1:20 tsk1:21 tsk1:22 tsk2:r tsk2:s tsk2:t tsk2:u tsk2:v tsk3:S tsk3:T
tsk3:U tsk3:V tsk3:W tsk1:23 tsk1:24 tsk1:25 tsk1:26 tsk1:27 tsk2:w tsk2:x tsk2:
y tsk2:z tsk3:X tsk1:28 tsk1:29 tsk1:30 tsk1:31 tsk1:32 tsk1:33 tsk3:Y tsk3:Z ts
k1:34 tsk1:35 tsk1:36 tsk1:37 tsk1:38 tsk1:39 tsk1:40 tsk1:41 tsk1:42 tsk1:43 ts
k1:44 tsk1:45 tsk1:46 tsk1:47 tsk1:48 tsk1:49_
```

开始时，`tsk1` 入队并调度，在 `tsk1` 执行时，由于一个时间片大小为 6，所以 `tsk2` 和 `tsk3` 都同时进入就绪队列。当 `tsk1` 的时间片用完后，`tsk1` 插入就绪队列队尾，然后调入 `tsk2` 运行。同样地，`tsk2` 时间片用完后插入就绪队列队尾，然后调入并运行 `tsk3`。当 `tsk3` 的时间片用完后，`tsk3` 插入就绪队列，此时队头是 `tsk1`，所以调入 `tsk1` 并运行，开始新一轮时间片轮转。

上面过程可用下面流程表示：

开始就绪队列：

`tsk1`

`tsk1` 运行时：

`tsk2->tsk3`

`tsk1` 被调出，`tsk2` 被调入后就绪队列：

`tsk3->tsk1`

tsk2 被调出,tsk3 被调入后就绪队列:

tsk1->tsk2

. . . . . (进入新一轮时间片轮转)

从测试结果也可以看到字符串颜色相互轮转,直到所有任务运行完成。实验结果符合 RR 调度预期。

## 2. 优先级调度

我们添加指令"PrioTest"来测试 RR 调度,每个测试任务设置如下:

```
PrioCreateTsk(tskBody: PrioTsk1,priority: 3,exetime: 3,arrrtime: 0);
PrioCreateTsk(tskBody: PrioTsk2,priority: 2,exetime: 3,arrrtime: 9);
PrioCreateTsk(tskBody: PrioTsk3,priority: 1,exetime: 1,arrrtime: 9);
PrioCreateTsk(tskBody: PrioTsk4,priority: 0,exetime: 1,arrrtime: 9);
```

这里规定优先级数越低,优先级越高,所以优先级数为 0 的任务优先级最高。

测试结果如下:

最开始,就绪队列只有 tsk1 到达,所以只能调度运行 tsk1。

随后 tsk2,tsk3 和 tsk4 同时到达就绪队列,由于 tsk4 优先级最高,所以先运行 tsk4。并且由于是非抢占,所以要等 tsk4 运行完后再调度。tsk4 运行完后,tsk3 优先级比 tsk2 高,所以先运行 tsk3,最后运行完 tsk2。测试结果符合预期。