



SHANDONG UNIVERSITY

操作系统第 2 次实验报告

谢子洋 202100460116

2023 年 4 月 8 日

目录

1	实验内容	2
1.1	任务 1: 为 $\mu C/OS-II$ 加入时间片轮转调度器	2
1.1.1	实现双向队列	2
1.1.2	实现轮转调度	3
1.2	任务 2: 实现多级队列调度	4
1.2.1	所需数据结构	5
1.2.2	实现多级队列调度算法	5
2	实验结果	8
3	实验心得	8
3.1	轮转调度	8
3.2	多级队列调度	8
	参考文献	9

1 实验内容

1.1 任务 1: 为 $\mu C/OS-II$ 加入时间片轮转调度器

轮转调度:

为每个进程分配一个时间片, 允许进程在其分配的时间片内运行. 时间片结束前进程阻塞或者结束, 则 CPU 立即切换进程; 时间片结束前进程仍在运行, 则将 CPU 剥夺分配给其他进程; 时间片长度设置应合理, 过短导致切换上下文等操作消耗大量资源, 过长会导致部分进程时钟得不到执行.

1.1.1 实现双向队列

(1) 队列初始化: 令队首指针 **Front** 和队尾指针 **Rear** 指向地址 0x0, 并设置队列长度为 0.

```
1 //os_core.c文件OS_InitRdyQueue函数添加语句.
2 void OS_InitRdyQueue ()
3 {
4     OSRdyTCBQueueNum=0;
5     OSRdyTCBQueueFront=(OS_TCB*) 0;
6     OSRdyTCBQueueRear=(OS_TCB*) 0;
7 }
```

(2) 队列 (在队尾) 添加元素:

重新设置进程的时间片, 其后分情况讨论,

若是队列长度为 0, 则令队首指针和队尾指针都指向新的元素.

若是队列长度非 0, 则先链接新元素再令队尾指针指向新元素.

过程中保证队首元素的 **Prev** 指针指向地址 0x0, 并且队尾元素的 **Next** 指针指地址 0x0, 最后队列长度自增 1.

```
1 //os_core.c文件OSRdyQueueIn函数添加语句.
2 void OSRdyQueueIn (OS_TCB *ptcb){
3     //your codes
4     ptcb->quantum=OS_SCHED_QUANTUM_MAX;
5     if (OSRdyTCBQueueNum==0) {
6         OSRdyTCBQueueFront=ptcb;
7         OSRdyTCBQueueRear=ptcb;
8         OSRdyTCBQueueRear->OSRdyTCBPrev=(OS_TCB*) (0);
9         OSRdyTCBQueueRear->OSRdyTCBNext=(OS_TCB*) (0);
10    }
11    else{
12        OSRdyTCBQueueRear->OSRdyTCBNext=ptcb;
13        ptcb->OSRdyTCBPrev=OSRdyTCBQueueRear;
14        OSRdyTCBQueueRear=ptcb;
15        OSRdyTCBQueueRear->OSRdyTCBNext=(OS_TCB*) (0);
16    }
17    OSRdyTCBQueueNum++;
18 }
```

(3) 队首元素出队: 同样分情况讨论:

当队列长度为 1 时, 另存首元素地址, 并令指针 **Front** 和 **Rear** 指向地址 0x0.

当队列元素非 1 时, 另存首元素地址, 并令指针 **Front** 指向次首元素.

过程中保证队首元素的 **Prev** 指针指向地址 0x0, 并且队尾元素的 **Next** 指针指地址 0x0.

最后令队列长度自减 1 并返回出队元素的地址.

```
1 //os_core.c文件OSRdyQueueOut函数添加语句.
2 OS_TCB* OSRdyQueueOut ()
3 {
4     OS_TCB*temp=OSRdyTCBQueueFront;
5     if (OSRdyTCBQueueNum==1) {
6         OSRdyTCBQueueRear->OSRdyTCBPrev= (OS_TCB*) (0);
7         OSRdyTCBQueueRear->OSRdyTCBNext= (OS_TCB*) (0);
8     }
9     else{
10         OSRdyTCBQueueFront=OSRdyTCBQueueFront->OSRdyTCBNext;
11         OSRdyTCBQueueFront->OSRdyTCBPrev= (OS_TCB*) (0);
12     }
13     OSRdyTCBQueueNum--;
14     return temp;
15 }
```

1.1.2 实现轮转调度

(1) 加入新进程

新进程需要先加入队列尾部, 并等待队列中所有位于其之前的进程时间片消耗完才能执行.

```
1 //os_core.c文件OS_TCBInit函数,添加语句: 1. 新进程入队列尾等待.
2 INT8U OS_TCBInit (...)
3 {
4     #if OS_SCHED_ROUND_ROBIN_EN > 0
5         //your code:
6         OSRdyQueueIn(ptcb);
7         //end code
8     #endif
9 }
```

(2) 进程时间片消耗

当前执行进程每经过一次系统时钟周期, 则令其时间片自减, 代表时间片消耗.

```
1 //os_core文件OSTimeTick函数, 添加语句: 1.当前执行进程的时间片自减
2 #if OS_SCHED_ROUND_ROBIN_EN > 0
3     //your code:
4     OSTCBCur->quantum--;
5     //end code
6 #endif
```

(3) 执行进程切换

当执行中进程时间片消耗完全后, 令该进程进入队尾. 从队首获取下一个要执行的进程并设置为执行状态.

```
1 //os_core.c文件OS_SchedNew函数, 添加语句:
2 //1.当前进程时间片结束时, 原进程入队列尾, 队首元素出队列
3 //2.将队首元素设置为正执行进程
4 static void OS_SchedNew (void) {
5     //.....
6     #if OS_SCHED_ROUND_ROBIN_EN > 0
7         if (OSTCBCur == 0 | OSTCBCur->quantum == 0)
8         { //运行态转成就绪态的任务入队
9             if (OSTCBCur!=0) {OSRdyQueueIn(OSTCBCur); }
10            //your code:
11            OS_TCB* newTask=OSRdyQueueOut();
12            OSPrioHighRdy = newTask->OSTCBPrio;
13            //end code
14        }
15    #endif
16    //.....
17 }
```

阻塞态转到就绪态的任务入就绪队列等待.

```
1 //os_core文件OSTimeTick函数, 添加语句如下:
2 void OSTimeTick (void)
3 {
4     //.....
5     #if OS_SCHED_ROUND_ROBIN_EN > 0
6         if ((OSRdyGrp&ptcb->OSTCBBitY) == 0 || (OSRdyTbl[ptcb->OSTCBY]&ptcb->OSTCBBitX)==0)
7             OSRdyQueueIn(ptcb);
8     #endif
9     //.....
10 }
```

1.2 任务 2: 实现多级队列调度

操作系统支持 64 个进程, 令每 8 个优先级分为一类, 则前 64 个优先级可共分为 8 类. 类内实施时间片轮转调度, 类间实施优先级调度, 即多级队列策略.

因为 $\mu s/OS$ 系统是嵌入式系统, 系统中所有进程是从开始便确定的并且不断执行, 因此优先级类之间不能使用非抢占式调度.

仿照上文时间片轮转调度, 逻辑上设置 8 个链表分别保存 8 个类内的进程. 每次时钟中断时系统检查包含进程数不为 0 的最高优先级类, 并按顺序执行其队列上进程. 为每个类分配一个总时间片, 当高优先级类的总时间片数耗尽时, 将类内进程移动到低优先级类中, 恢复总时间片数. 每个类拥有各自优先级, 优先级高越高分配的总时间片更少.

1.2.1 所需数据结构

(1) 优先级类结构体

定义结构体, 保存一个优先级类的所有信息, 包括: 队尾指针, 队首指针, 队列进程个数, 总时间片数.

```
1 //ucos_ii.h文件中添加结构体:
2 typedef struct multi_level_queue {
3     INT8U quantum;
4     INT8U QueueNum;
5     OS_TCB *FRONT;
6     OS_TCB *REAR;
7 }MultiLevelQueue;
```

(2) 信息变量

定义每个优先级类对应的总时间片数, 优先级越高的类对应总时间片越小.

申请长度为 8 的优先级类结构体数组, 分别对应 8 个优先级类, 索引越小类优先级越高.

申请整数变量记录当前所使用的优先级类序号 (0-7), 序号越小代表优先级越高.

```
1 //os_core.c文件中添加如下变量定义:
2 INT8U classQuntunmList[8]={10,25,35,40,50,60,70,80};
3 MultiLevelQueue MultiClass[8]; //eight classes
4 INT8U CurClass; // curent selected class
```

1.2.2 实现多级队列调度算法

(1) 初始化

初始化 8 个优先级类的链表信息, 并设定由最高优先级 0 开始执行.

```
1 //os_core.c文件OSInit函数中增加如下语句:
2 void OSInit (void)
3 {
4     //.....
5     #if OS_SCHED_MULTILEVEL_QUEUE_EN > 0
6         CurClass=0; //指定当前优先级类为类0(最高优先级类)
7         for(int i=7;i≥0;i--) //初始化8个优先级类的信息.
8         {
9             OS_InitRdyQueue();
10            MultiClass[i].quantum=classQuntunmList[i];
11            MultiClass[i].QueueNum=OSRdyTCBQueueNum;
12            MultiClass[i].FRONT=OSRdyTCBQueueFront;
13            MultiClass[i].REAR=OSRdyTCBQueueRear;
14        }
15    #endif
16    //.....
17 }
```

(2) 加入新进程

新进程进入优先级为 0 的进程队列并等待执行.

```
1 //os_core.c文件OS_TCBInit()函数添加语句: 1.令新进程入优先级0的就绪进程队列
2 INT8U OS_TCBInit (.....)
3 {
4     //.....
5     #if OS_SCHED_MULTILEVEL_QUEUE_EN > 0
6     OSRdyQueueIn(ptcb);
7     #endif
8     //.....
9 }
```

(3) 时间片消耗

每经过一次系统时钟中断, 缩减当前进程的时间片, 同时减小其优先级类的总时间片数.

```
1 //os_core文件OSTimeTick函数, 添加语句: 1.令进程时间片减小 2.令所在类总时间片减小
2 void OSTimeTick (void)
3 {
4     //.....
5     #if OS_SCHED_MULTILEVEL_QUEUE_EN > 0
6     OSTCBCur->quantum--;
7     MultiClass[CurClass].quantum--;
8     #endif
9     //.....
10    //.....
11    OSRdyQueueIn(ptcb);
12    //.....
13 }
```

(4) 切换进程/切换优先级类

当执行中进程的时间片长度为 0 时:

若进程所处优先级类的总时间片数不为 0 时, 只切换进程.

若进程所处优先级类的总时间片数为 0 时:

若优先级类是最低优先级 7, 则只切换进程并恢复总时间片数到最大值.

若优先级不是 7, 则将所有进程移动到低优先级队列尾部, 并切换到低优先级类.

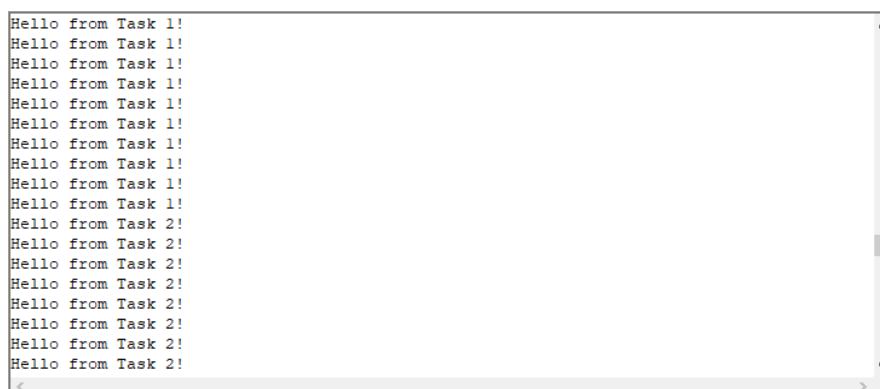
令当前优先级的进程队列出进程并执行.

代码如下:

```
1 static void OS_SchedNew (void)
2 {
3     //.....
4     #if OS_SCHED_MULTILEVEL_QUEUE_EN > 0
5         if (OSTCBCur == 0 | OSTCBCur->quantum == 0){
6             if(MultiClass[CurClass].quantum==0){
7                 if(CurClass!=7){
8                     //switch class
9                     CurClass+=1;
10                    OS_TCB*tempPoi;
11                    OS_TCB*tempPoi_;
12                    tempPoi=OSTCBCur;
13                    OSRdyTCBQueueNum=MultiClass[CurClass].QueueNum;
14                    OSRdyTCBQueueFront=MultiClass[CurClass].FRONT;
15                    OSRdyTCBQueueRear=MultiClass[CurClass].REAR;
16                    //push processes in original class to new lower level class
17                    while(tempPoi!=0)
18                    {
19                        tempPoi_=tempPoi->OSRdyTCBNext;
20                        OSRdyQueueIn(tempPoi);
21                        tempPoi=tempPoi_;
22                    }
23                }
24                else{
25                    MultiClass[CurClass].quantum=classQuntunmList[7];
26                }
27            }
28            else{
29                if (OSTCBCur!=0){OSRdyQueueIn(OSTCBCur);}
30            }
31            OS_TCB* newTask=OSRdyQueueOut();
32            OSPrioHighRdy = newTask->OSTCBPrio;
33        }
34    #endif
35    //.....
36 }
```


2 实验结果

task1 和 task2 交替执行, 每经过 5 个系统时钟中断便切换一次进程.



```
Hello from Task 1!
Hello from Task 1!
Hello from Task 1!
Hello from Task 1!
Hello from Task 1!
Hello from Task 1!
Hello from Task 1!
Hello from Task 1!
Hello from Task 1!
Hello from Task 1!
Hello from Task 2!
Hello from Task 2!
Hello from Task 2!
Hello from Task 2!
Hello from Task 2!
Hello from Task 2!
Hello from Task 2!
Hello from Task 2!
Hello from Task 2!
Hello from Task 2!
```

图 1: 实验结果

3 实验心得

3.1 轮转调度

设计队列数据结构, 每个元素代表一个进程, 队列中元素的顺序即代表着进程执行的顺序. 添加新的待执行的进程就是将新进程加入队列尾部, 切换进程就是将队列首进程出队列再入队列.

轮转调度进程切换的时机完全依赖于时间片长度的设计, 当一个进程要么在时间片内完成, 要么在时间片结束时被切换. 因此时间片的长度设定对该调度算法至关重要, 若是过长则会导致部分进程等待很长时间得到不执行, 若是过短则会增大进程切换导致的开销.

3.2 多级队列调度

设计多个队列分别代表不同的优先级类, 类之间使用优先级调度, 优先执行不为空并且优先级最高的队列中的进程. 同一队列中不同进程之间使用轮转调度算法, 同上.

同一进程所处优先级类不是固定的. 若进程经过某一优先级的总时间片后仍未完成, 则降低该进程优先级, 令其进入低优先级队列.

多级队列调度同样需要着重考虑时间片长度, 并且除了进程执行的时间片长度, 还需考虑优先级类的总时间片长度, 即该优先级内进程在该优先级下执行的总时间. 当不同优先级设置的总时间片长度比例合适时, 开销小的进程总能在高优先级下就完成, 而开销大的进程最后会变为低优先级. 这样资源能更合理的分配给不同进程.

参考文献

- [1] Andrew S.Tanenbaum,Herbert Bos. 现代操作系统. 北京: 机械工业出版社,2017.