操作系统第二次实验

学号：202200460104

姓名：密语

班级：网安 1 班

2024 年 6 月 10 日

目录

1 任务一

1.1 任务内容

为 μC/OS-II 加入时间片轮转调度器

1.2 任务分析

1.2.1 时间片轮转调度器

时间片轮转调度器是一种基于时间片的调度算法，每个任务被分配一个时间片，当时间片用完后，任务会被挂起，然后调度器会选择下一个任务执行。时间片轮转调度器是一种简单且高效的调度算法，适用于多任务系统。

1.2.2 OS_TCB 结构体

OS_TCB 结构体是 μC/OS-II 中任务控制块的结构体，包含了任务的所有信息，如任务 ID、任务优先级、任务状态等，同时也声明了维护就绪队列的指针。
OS_TCB 结构体的定义如下：

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;           /* Pointer to current top
       of stack                          */

#if OS_TASK_CREATE_EXT_EN > 0
    void            *OSTCBExtPtr;           /* Pointer to user
       definable data for TCB extension     */
    OS_STK          *OSTCBStkBottom;        /* Pointer to bottom of
       stack                              */
    INT32U           OSTCBStkSize;          /* Size of task stack (in
       number of stack elements)         */
    INT16U           OSTCBOpt;              /* Task options as passed
       by OSTaskCreateExt()              */
    INT16U           OSTCBId;               /* Task ID (0..65535)
                                            */
#endif

    struct os_tcb   *OSTCBNext;             /* Pointer to next     TCB
       in the TCB list                   */
    struct os_tcb   *OSTCBPrev;             /* Pointer to previous TCB
       in the TCB list                   */

#if (OS_EVENT_EN) || (OS_FLAG_EN > 0)
    OS_EVENT        *OSTCBEventPtr;         /* Pointer to
       event control block               */
#endif
```

```
18
19  #if (OS_EVENT_EN) && (OS_EVENT_MULTI_EN > 0)
20      OS_EVENT       **OSTCBEventMultiPtr;    /* Pointer to multiple
        event control blocks                 */
21  #endif
22
23  #if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
24      void            *OSTCBMsg;              /* Message received from
        OSMboxPost() or OSQPost()          */
25  #endif
26
27  #if (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
28  #if OS_TASK_DEL_EN > 0
29      OS_FLAG_NODE    *OSTCBFlagNode;         /* Pointer to event flag
        node                                 */
30  #endif
31      OS_FLAGS         OSTCBFlagsRdy;         /* Event flags that made
        task ready to run                    */
32  #endif
33
34      INT16U           OSTCBDly;              /* Nbr ticks to delay task
        or, timeout waiting for event    */
35      INT8U            OSTCBStat;             /* Task      status
                                                  */
36      INT8U            OSTCBStatPend;         /* Task PEND status
                                                  */
37      INT8U            OSTCBPrio;             /* Task priority (0 ==
        highest)                            */
38
39      INT8U            OSTCBX;                /* Bit position in group
        corresponding to task priority   */
40      INT8U            OSTCBY;                /* Index into ready table
        corresponding to task priority   */
41  #if OS_LOWEST_PRIO <= 63
42      INT8U            OSTCBBitX;             /* Bit mask to access bit
        position in ready table          */
43      INT8U            OSTCBBitY;             /* Bit mask to access bit
        position in ready group          */
44  #else
45      INT16U           OSTCBBitX;             /* Bit mask to access bit
        position in ready table          */
46      INT16U           OSTCBBitY;             /* Bit mask to access bit
```

```
        position in ready group          */
#endif

#if OS_TASK_DEL_EN > 0
    INT8U             OSTCBDelReq;           /* Indicates whether a task
         needs to delete itself         */
#endif

#if OS_TASK_PROFILE_EN > 0
    INT32U            OSTCBCtxSwCtr;         /* Number of time the task
        was switched in                 */
    INT32U            OSTCBCyclesTot;        /* Total number of clock
        cycles the task has been running  */
    INT32U            OSTCBCyclesStart;      /* Snapshot of cycle
        counter at start of task resumption   */
    OS_STK           *OSTCBStkBase;          /* Pointer to the beginning
         of the task stack              */
    INT32U            OSTCBStkUsed;          /* Number of bytes used
        from the stack                      */
#endif

#if OS_TASK_NAME_SIZE > 1
    INT8U             OSTCBTaskName[OS_TASK_NAME_SIZE];
#endif

/****************FOR ROUND ROBIN: queue of ready TCBs***************/
#if OS_SCHED_ROUND_ROBIN_EN > 0
    INT32U            quantum;
    struct os_tcb    *OSRdyTCBNext;
    struct os_tcb    *OSRdyTCBPrev;
#endif
/*********************************************************************/

} OS_TCB;
```

下面对一些重要的成员变量进行解释：

- OSTCBNext：指向就绪队列中下一个任务的指针
- OSTCBPrev：指向就绪队列中上一个任务的指针
- OSTCBPrio：任务的优先级
- quantum：时间片大小
- OSRdyTCBNext：指向时间片轮转调度器中下一个任务的指针

- OSRdyTCBPrev：指向时间片轮转调度器中上一个任务的指针

1.2.3 main.c 分析

```
int main(void)
{
    OSInit();
    systick_init();
    OSTaskCreate(task1, (void *)0,
                    &task1_stk[TASK1_STK_SIZE-1], TASK1_PRIO);
    OSTaskCreate(task2, (void *)0,
                    &task2_stk[TASK2_STK_SIZE-1], TASK2_PRIO);
    ASM_Switch_To_Unprivileged();
    OSStart();
    return 0;
}
static void task1(void *p_arg)
{
    int i;
    for (;;)
    {
        for (i=1;i<10;i++);
        syscall_print_str("Hello from Task 1!\n");
    }
}

static void task2(void *p_arg)
{
    int i;
    for (;;)
    {
        for (i=1;i<10;i++);
        syscall_print_str("Hello from Task 2!\n");
    }
}
```

main 函数中，首先调用 OSInit() 函数初始化操作系统，然后调用 systick_init() 函数初始化系统时钟，接着调用 OSTaskCreate() 函数创建任务 task1 和 task2，最后调用 OSStart() 函数启动操作系统。task1 和 task2 函数分别是任务 1 和任务 2 的入口函数，任务 1 和任务 2 的功能是循环打印字符串。在任务 1 和任务 2 的循环中，没有调用 OSTimeDly() 函数，因此任务 1 和任务 2 会一直运行，不会被挂起。

1.3 任务实现

首先在 os_core.c 中导入 stddef.h 头文件，用于定义 NULL 指针。我们使用双向链表来维护时间片轮转调度器，定义两个指针 OSRdyTCBQueueFront 和 OSRdyTCBQueueRear 分别指向时间片轮转调度器的头和尾，定义一个变量 OSRdyTCBQueueNum 来记录时间片轮转调度器中任务的数量。

1.3.1 编写 OSRdyQueueIn 函数

这个函数用于将任务插入到时间片轮转调度器中。使用了一个双向链表，将任务插入列表的尾部。每次插入任务时，需要维护队列头指针 OSRdyTCBQueueFront、队列尾指针 OSRdyTCBQueueRear 和队列中任务数 OSRdyTCBQueueNum。

```
1  void OSRdyQueueIn (OS_TCB *ptcb)
2  {
3      if (OSRdyTCBQueueNum == 0) { //如果就绪队列为空
4          OSRdyTCBQueueFront = ptcb; //将队列头指针指向ptcb
5          OSRdyTCBQueueRear = ptcb; //将队列尾指针指向ptcb
6          ptcb->OSRdyTCBNext = NULL; //ptcb的下一个指针为空
7          ptcb->OSRdyTCBPrev = NULL; //ptcb的上一个指针为空
8      }
9      else { //如果就绪队列不为空
10         OSRdyTCBQueueRear->OSRdyTCBNext = ptcb;//将ptcb插入到队列尾
11         ptcb->OSRdyTCBPrev = OSRdyTCBQueueRear; //将ptcb的上一个指针指
               向队列尾
12         ptcb->OSRdyTCBNext = NULL; //将ptcb的下一个指针置空
13         OSRdyTCBQueueRear = ptcb; //将队列尾指针指向ptcb
14     }
15     OSRdyTCBQueueNum++; //就绪队列中任务数加一
16 }
```

1.3.2 编写 OSRdyQueueOut 函数

这个函数用于将任务从时间片轮转调度器中取出。如果时间片轮转调度器为空，则返回 NULL；否则，将队列头指针指向下一个任务，将队列头的上一个指针置空，将 ptcb 的下一个指针和上一个指针置空，最后将就绪队列中任务数减一。

```
1  OS_TCB* OSRdyQueueOut ()
2  {
3      OS_TCB *ptcb;
4
5      if (OSRdyTCBQueueNum == 0) {//如果就绪队列为空
6          return NULL;
7      }
8
```

```
    ptcb = OSRdyTCBQueueFront;
    if (OSRdyTCBQueueNum == 1) { //只有一个任务在队列中
        OSRdyTCBQueueFront = NULL;
        OSRdyTCBQueueRear = NULL;
    }
    else { //多个任务在队列中
        OSRdyTCBQueueFront = ptcb->OSRdyTCBNext; //将队列头指针指向下一个任务
        OSRdyTCBQueueFront->OSRdyTCBPrev = NULL; //将队列头的上一个指针置空
    }

    ptcb->OSRdyTCBNext = NULL; //将ptcb的下一个指针置空
    ptcb->OSRdyTCBPrev = NULL; //将ptcb的上一个指针置空
    OSRdyTCBQueueNum--; //就绪队列中任务数减一

    return ptcb; //返回出队的任务
}
```

1.3.3 编写 OS_InitRdyQueue 函数

这个函数用于初始化时间片轮转调度器，将队列头指针、队列尾指针和队列中任务数都置空。

```
void OS_InitRdyQueue ()
{
    OSRdyTCBQueueFront = NULL; //将队列头指针置空
    OSRdyTCBQueueRear = NULL; //将队列尾指针置空
    OSRdyTCBQueueNum = 0; //将队列中任务数置零
}
```

1.3.4 编写减小任务的时间片的相关代码

```
#if OS_SCHED_ROUND_ROBIN_EN > 0
    //your code: decrese quantum time of current task (pointer of
       current tack's TCB: OSTCBCur):
    if (OSTCBCur != NULL && OSTCBCur->quantum > 0) {
    OSTCBCur->quantum--; //如果当前的任务不为空，且当前任务的时间片大于0，将当前任务的时间片减一
    }
#endif
```

1.3.5 编写时间片调度的相关代码

```
#if OS_SCHED_ROUND_ROBIN_EN > 0
    /* No, and if not in the ready queue, then queue in */
    if ((OSRdyGrp & ptcb->OSTCBBitY) == 0 || (OSRdyTbl[ptcb->OSTCBY] &
       ptcb->OSTCBBitX) == 0) {
        //your code: put ptcb into queue of ready tasks
        OSRdyQueueIn(ptcb);//将ptcb插入到就绪队列中
    }
#endif
```

```
#if OS_SCHED_ROUND_ROBIN_EN > 0
if (OSTCBCur == 0 | OSTCBCur->quantum == 0)
{
    if (OSTCBCur!=0)
        OSRdyQueueIn(OSTCBCur);
    //your code:
        OS_TCB *next_tcb = OSRdyQueueOut();//取出下一个任务
    if (next_tcb != NULL) {
            OSPrioHighRdy = next_tcb->OSTCBPrio;//将下一个任务的优先级
               赋值给OSPrioHighRdy
    }
}
#endif
```

```
#if OS_SCHED_ROUND_ROBIN_EN > 0
//your code:  put ptcb into queue of ready TCBs
        OSRdyQueueIn(ptcb);//将ptcb插入到就绪队列中
#endif
```

1.4 运行效果

将项目编译后运行，可以看到任务 1 和任务 2 交替执行，每个任务执行完一个时间片后就会切换到另一个任务，实现了时间片轮转调度器的功能。

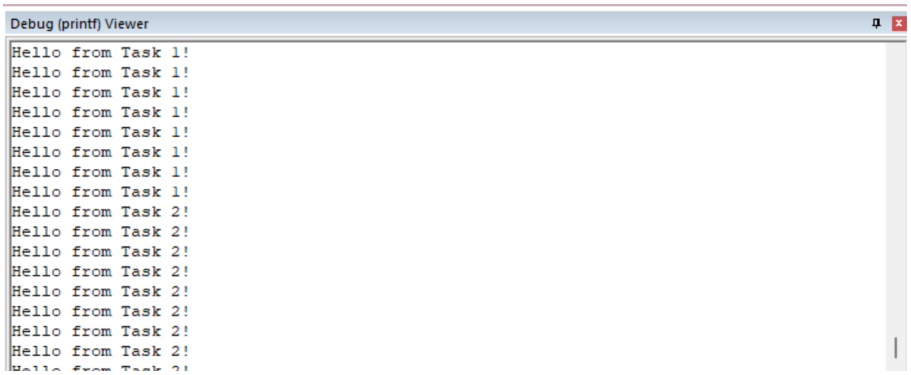

图 1: 运行界面

我们也可以调整 os_cfg.h 的时钟中断的间隔,来改变任务的切换速度。当 OS_TICKS_PER_SEC 的值为 140000 时，每个任务在一个时间片内只执行一次，如下图：

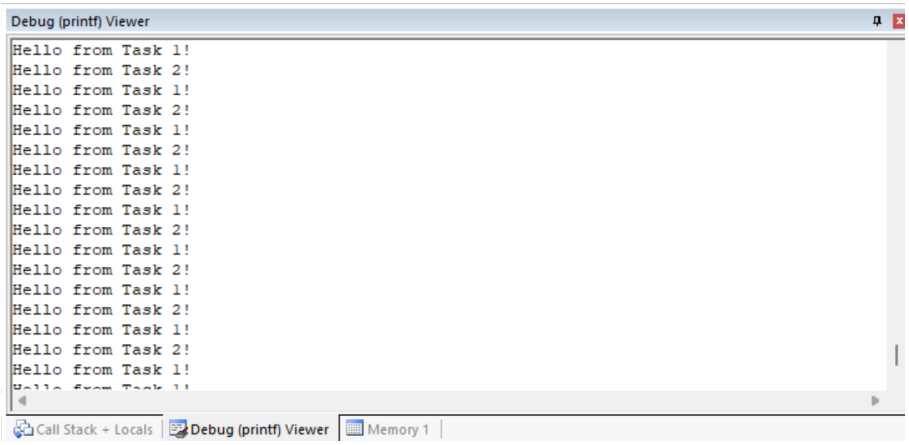

图 2: 运行界面

2 任务二

2.1 任务内容

每 8 个优先级分为一类，前 64 个优先级共分为 8 类，类内实施时间片轮转调度，类间实施优先级调度。

2.2 任务分析

我们可以根据任务的优先级，通过整除 8 的方式将任务分为 8 类，每一类实现基于队列的时间片轮转调度，类间实现优先级调度。

2.3 任务实现

2.3.1 定义就绪队列结构体

首先在 os_core.c 中定义一个就绪队列结构体，用于维护每一类任务的就绪队列。就绪队列结构体包含了队列头指针、队列尾指针和队列中任务数。并定义了 8 个就绪队列，用于存放 8 种不同优先级的任务。

```
1 typedef struct{
2     OS_TCB *front;
3     OS_TCB *rear;
4     INT8U num;
5 }OS_RdyQueue;
6 OS_RdyQueue OSRdyTCBQueue[8];
```

2.3.2 编写 OSRdyQueueIn 函数

定义一个 grp 变量，用于存放任务的优先级分组。将任务插入到对应的就绪队列中，维护队列头指针、队列尾指针和队列中任务数。

```
void OSRdyQueueIn (OS_TCB *ptcb)
{
    INT8U grp = ptcb->OSTCBPrio>>3;//左移3位，相当于整除以8
    OS_RdyQueue *queue = &OSRdyTCBQueue[grp];
    if (queue->num == 0) {
        queue->front = ptcb;
        queue->rear = ptcb;
        ptcb->OSRdyTCBNext = NULL;
        ptcb->OSRdyTCBPrev = NULL;
    }
    else {
        queue->rear->OSRdyTCBNext = ptcb;
        ptcb->OSRdyTCBPrev = queue->rear;
        ptcb->OSRdyTCBNext = NULL;
        queue->rear = ptcb;
    }
    queue->num++;
}
```

2.3.3 编写 OSRdyQueueOut 函数

函数需要设置一个 grp 参数，用于指定优先级分组。如果就绪队列为空，则返回 NULL；否则，将队列头指针指向下一个任务，将队列头的上一个指针置空，将 ptcb 的下一个指针和上一个指针置空，最后将就绪队列中任务数减一。

```
OS_TCB* OSRdyQueueOut (INT8U grp)
{
    OS_RdyQueue *queue=&OSRdyTCBQueue[grp];
    OS_TCB*ptcb;
    if (queue->num == 0) {
            return NULL;
        }

    ptcb = queue->front;
    if (queue->num == 1) {
            queue->front = NULL;
            queue->rear = NULL;
        }
    else {
            queue->front = ptcb->OSRdyTCBNext;
```

```
            queue->front->OSRdyTCBPrev = NULL;
        }
    ptcb->OSRdyTCBNext = NULL;
    ptcb->OSRdyTCBPrev = NULL;
    queue->num--;
    return ptcb;
}
```

2.3.4 编写 OS_InitRdyQueue 函数

初始化 8 个就绪队列，将队列头指针、队列尾指针和队列中任务数都置空。

```
void OS_InitRdyQueue ()
{
    for(int i=0;i<8;i++){
        OSRdyTCBQueue[i].front=NULL;
        OSRdyTCBQueue[i].rear=NULL;
        OSRdyTCBQueue[i].num=0;
    }
}
```

2.3.5 编写 OS_SchedNew 函数

在调度函数中实现优先级调度，首先找到最高优先级的就绪队列，然后从该队列中取出任务，如果队列为空，则继续查找下一个优先级的队列，直到找到非空队列。如果找到了非空队列，则将该队列中的任务取出，设置 OSPrioHighRdy 为该任务的优先级。

```
for(INT8U i=0;i<8;i++){
    if(OSRdyTCBQueue[i].num>0){
        OS_TCB *next_tcb = OSRdyQueueOut(i);
        if (next_tcb != NULL) {
        OSPrioHighRdy = next_tcb->OSTCBPrio;
        }
        break;
    }
}
```

2.4 运行效果

调整 OS_TICKS_PER_SEC 的值，可以改变任务的切换速度。当 OS_TICKS_PER_SEC 的值为 140000 时，每个任务在一个时间片内只执行一次，如下图：

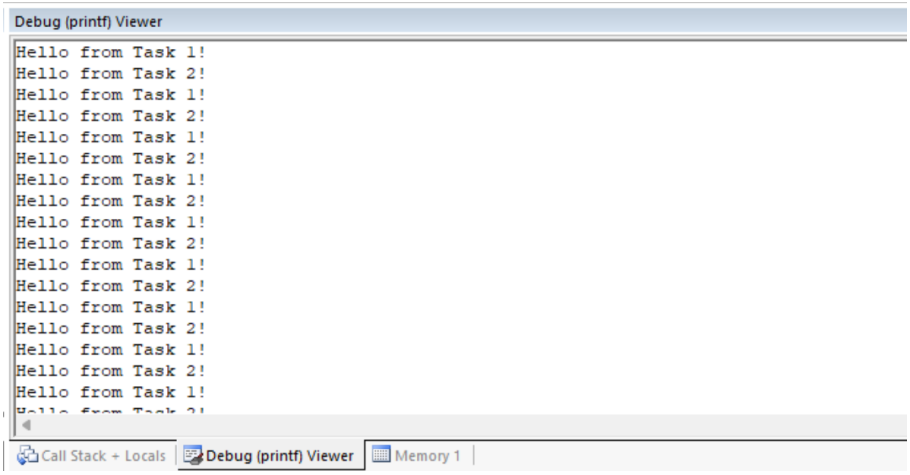

图 3: 运行界面

2.5 总结

本次实验通过链表和队列这两种数据结构，实现了时间片轮转调度器和基于优先级的时间片轮转调度器，时间片轮转调度器是一种简单且高效的调度算法，适用于多任务系统。优先级调度器可以根据任务的优先级来调度任务，优先级高的任务会先执行。通过实现这两种调度器，我们可以更好地管理任务，提高系统的效率。