实验 6 时间片轮转调度

1. 实验目的和任务要求

（1） 调试 EOS 的线程调度程序，熟悉基于优先级的抢先式调度。
（2） 为 EOS 添加时间片轮转调度，了解其它常用的调度算法。

2. 实验原理

（1） 基于优先级的抢占调度
（2） 调度程序执行的时机和流程
（3） 实现时间片轮转调度的细节

3. 实验内容

3.1 准备实验

按照下面的步骤准备实验：

1. 启动 OS Lab。
2. 新建一个 EOS Kernel 项目。

3.2 阅读控制台命令“ rr ”相关的源代码

阅读 ke/sysproc.c 文件中第 689 行的 ConsoleCmdRoundRobin 函数，及该函数用到的第 648 行的 ThreadFunction 函数和第 641 行的 THREAD_PARAMETER 结构体，学习“rr”命令是如何测试时间片轮转调度的。在阅读的过程中需要特别注意下面几点：

在 ConsoleCmdRoundRobin 函数中使用 ThreadFunction 函数做为线程函数，新建了 10 个优先级为 8 的线程，做为测试时间片轮转调度用的线程。

在新建的线程中，只有正在执行的线程才会在控制台的对应行（第 0 个线程对应第 0 行，第 1 个线程对应第 1 行，依此类推）增加其计数，这样就可以很方便的观察到各个线程执行的情况。

时间片轮转调度的相关算法如下：

```
PRIVATE
VOID
ConsoleCmdRoundRobin(
    IN HANDLE StdHandle
    )
/*++

功能描述:
    测试时间片轮转调度。控制台命令“rr”。

参数:
    StdHandle -- 标准输入、输出句柄。

返回值:
    无。

--*/
{
    ULONG i;
    COORD CursorPosition;
    HANDLE ThreadHandleArray[10];
    THREAD_PARAMETER ThreadParameterArray[10];

    // 清理整个屏幕的内容。
    for (i = 0; i < 24; i++) {
        fprintf(StdHandle, "\n");
    }

    // 新建 10 个优先级为 8 的线程。关闭中断从而保证新建的线程不会执行。
    __asm("cli");
    for (i = 0; i < 10; i++) {

        ThreadParameterArray[i].Y = i;
        ThreadParameterArray[i].StdHandle = StdHandle;

        ThreadHandleArray[i] = (HANDLE)CreateThread(
            0, ThreadFunction, (PVOID)&ThreadParameterArray[i], 0, NULL);

        // 设置线程的优先级。
        PsSetThreadPriority(ThreadHandleArray[i], 8);
    }
    __asm("sti");

    // 当前线程等待一段时间。由于当前线程优先级 24 高于新建线程的优先级 8，
    // 所以只有在当前线程进入“阻塞”状态后，新建的线程才能执行。
    Sleep(40 * 1000);

    // 当前线程被唤醒后，会抢占处理器。强制结束所有新建的线程。
    for (i = 0; i < 10; i++) {
        TerminateThread(ThreadHandleArray[i], 0);
        CloseHandle(ThreadHandleArray[i]);
    }

    // 设置字符在屏幕上输出的位置。
    CursorPosition.X = 0;
    CursorPosition.Y = 23;
    SetConsoleCursorPosition(StdHandle, CursorPosition);
}
```

控制台对于新建的线程来说是一种临界资源，所以，新建的线程在向控制台输出时，必须使用“关中断”和“开中断”进行互斥（参见 ThreadFunction 函数的源代码）。

由于控制台线程的优先级是 24，高于新建线程的优先级 8，所以只有在控制台线程进入“阻塞”状态后，新建的线程才能执行。

新建的线程会一直运行，原因是在 ThreadFunction 函数中使用了死循环，所以只能在 ConsoleCmdRoundRobin 函数的最后调用 TerminateThread 函数来强制结束这些新建的线程。

按照下面的步骤执行控制台命令“rr”，查看其在没有时间片轮转调度时的

执行效果：

1. 按 F7 生成在本实验 3.1 中创建的 EOS Kernel 项目。
2. 按 F5 启动调试。
3. 待 EOS 启动完毕，在 EOS 控制台中输入命令“rr”后按回车。

执行命令如下：

命令开始执行后，观察其执行效果，会发现并没有体现“rr”命令相关源代码的设计意图。通过之前对这些源代码的学习，10 个新建的线程应该在控制台对应的行中轮流地显示它们的计数在增加，而现在只有第 0 个新建的线程在第 0 行显示其计数在增加，说明只有第 0 个新建的线程在运行，其它线程都没有运行。造成上述现象的原因是：所有 10 个新建线程的优先级都是 8，而此时 EOS 只实现了基于优先级的抢先式调度，还没有实现时间片轮转调度，所以至始至终都只有第 0 个线程在运行，而其它具有相同优先级的线程都没有机会运行， 只能处于“就绪”状态。

执行结果如下：

3.3 调试线程调度程序

在为 EOS 添加时间片轮转调度之前，先调试一下 EOS 的线程调度程序 PspSelectNextThread 函数，学习就绪队列、就绪位图以及线程的优先级是如何在线程调度程序中协同工作的，从而加深对 EOS 已经实现的基于优先级的抢先式调度的理解。

3.3.1 调试当前线程不被抢先的情况

新建的第 0 个线程会一直运行，而不会被其它同优先级的新建线程或者低优先级的线程抢先，这是由当前实现的基于优先级的抢先式调度算法决定的。按照下面的步骤调试这种情况，在 PspSelectNextThread 函数中处理的过程。

1. 结束之前的调试。
2. 在 ke/sysproc.c 文件的 ThreadFunction 函数中，调用 fprintf 函数的代码行（第 679 行）添加一个断点。

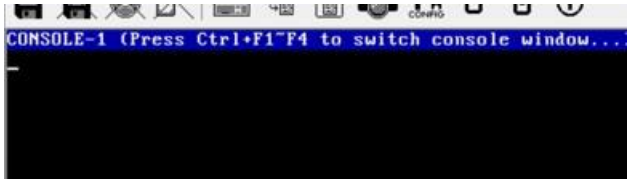
添加断点如下：

```
672
673     // 在线程参数指定的行循环显示线程执行的状态。死循环。通过
674     // 格式: Thread 序号 (优先级): 执行计数
675     for (i = 0; ; i++) {
676         __asm("cli");
677         PsGetThreadPriority(CURRENT_THREAD_HANDLE, &Priority)
678         SetConsoleCursorPosition(pThreadParameter->StdHandle,
679         fprintf(pThreadParameter->StdHandle, "Thread %d (ID:%
680             pThreadParameter->Y, ObGetObjectId(PspCurrentThre
681         __asm("sti");
682     }
683
684     return 0;
685 }
686
```

3. 按 F5 启动调试。

4. 待 EOS 启动完毕，在 EOS 控制台中输入命令“rr”后按回车。“rr”命令开始执行后，会在断点处中断。

虚拟机窗口状态如下：



5. 刷新“进程线程”窗口，可以看到如图 14-2 所示的内容。其中，从 ID 为 24 到 ID 为 33 的线程是“rr”命令创建的 10 个优先级为 8 的线程，ID 为 24 的线程处于运行状态，其它的 9 个线程处于就绪状态。

查看线程状态如下：

进程线程

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	17	Y	24	Waiting (3)	1	0x80015724 IopConsoleDispatchThread
3	18	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
4	19	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
5	20	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
6	21	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
7	24	Y	8	Running (2)	1	0x800188a2 ThreadFunction
8	25	Y	8	Ready (1)	1	0x800188a2 ThreadFunction
9	26	Y	8	Ready (1)	1	0x800188a2 ThreadFunction
10	27	Y	8	Ready (1)	1	0x800188a2 ThreadFunction
11	28	Y	8	Ready (1)	1	0x800188a2 ThreadFunction
12	29	Y	8	Ready (1)	1	0x800188a2 ThreadFunction
13	30	Y	8	Ready (1)	1	0x800188a2 ThreadFunction
14	31	Y	8	Ready (1)	1	0x800188a2 ThreadFunction
15	32	Y	8	Ready (1)	1	0x800188a2 ThreadFunction
16	33	Y	8	Ready (1)	1	0x800188a2 ThreadFunction

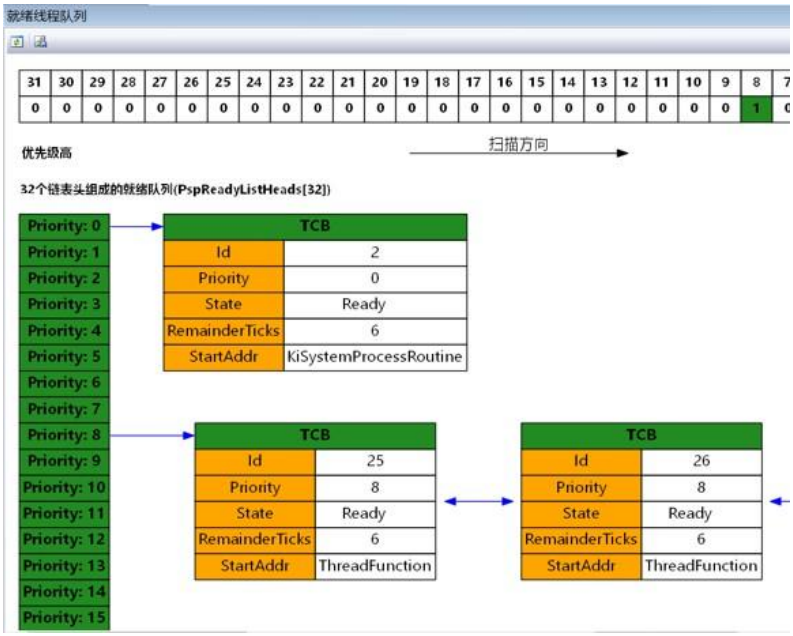
6. 在“线程运行轨迹”窗口点击其工具栏上的“绘制指定范围线程”按钮，输入起始线程 ID 为 24 和结束线程 ID 为 33（需根据当前实际创建的线程 ID 进行调整），点击“绘制”按钮，可以查看这 10 个线程的运行状态和调度情况。

指定绘制范围如下：

指定线程绘制结果：

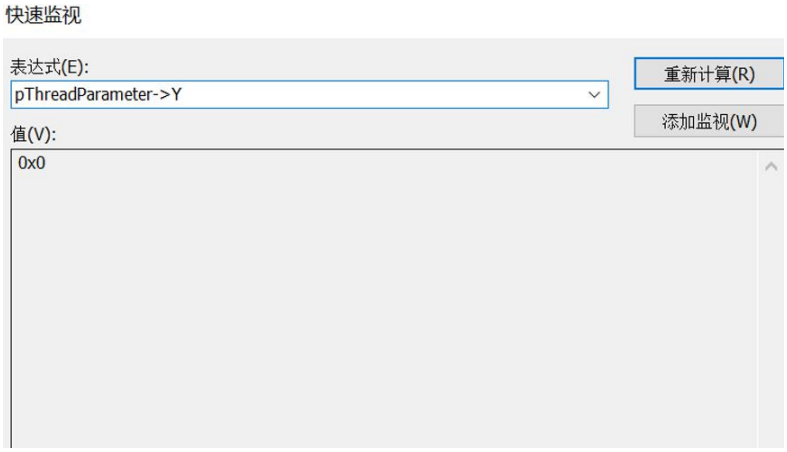
7. 刷新“就绪线程队列”窗口，可以看到在就绪位图中，第 0 位和第 8 位为 1，其它位都为 0，相对应的，在就绪队列中只有优先级为 0 和优先级为 8 的就绪队列中挂接了处于就绪状态的线程，其它优先级的就绪队列都为空，如图14-4 所示。其中，优先级为 0 的线程是空闲线程，9 个优先级为 8 的线程是从 ID 为 25 到 ID 为 33 的线程，而 ID 为 24 的线程此时正处于运行状态， 所以不在优先级为 8 的就绪队列中。

“就绪线程队列”窗口结果如下：



8. 查看 ThreadFunction 函数中变量 pThreadParameter->Y 的值应该为 0，说明正在调试的是第 0 个新建的线程。

通过快速监视窗口查看如下：



9. 激活虚拟机窗口，可以看到第 0 个新建的线程还没有在控制台中输出任何内容，原因是 fprintf 函数还没有执行。

10. 激活 OS Lab 窗口后按 F5 使第 0 个新建的线程继续执行，又会在断点处中断。再次激活虚拟机窗口，可以看到第 0 个新建的线程已经在控制台中输出了第一轮循环的内容。可以多按几次 F5 查看每轮循环输出的内容。

第一次按下 F5（计数值为 0）：

第二次按下 F5（计数值+1）：

第三次按下 F5（计数值再次+1）：

11. 再次在“线程运行轨迹”窗口中点击工具栏上的“绘制指定范围线程”按钮，输入起始线程 ID 为 24 和结束线程 ID 为 33,点击“绘制”按钮，可以看到仍然是只有 ID 为 24 的线程处于运行状态，其它 9 个线程处于就绪状态。

通过之前的调试，可以观察到“rr”命令新建的第 0 个线程始终处于运行状态，而不会被其它具有相同优先级的线程抢先，那么在 EOS 内核中是如何实现这种调度算法的呢？读者可以按照下面的步骤进行调试，查看当有中断发生从而触发线程调度时，第 0 个新建的线程不会被抢先的情况。

1. 请继续之前的调试。

2. 在 ps/sched.c 文件的 PspSelectNextThread 函数中，调用 BitScanReverse 函数扫描就绪位图的代码行（第 391 行）添加一个断点。

添加断点如下：

3. 按 F5 继续执行。因为每当有定时计数器中断发生时（每 10ms 一次）都会触发线程调度函数 PspSelectNextThread，所以很快就会在刚刚添加的断点处中断。

此时状态如下：

```
390     //
391     BitScanReverse(&HighestPriority, PspReadyBitmap);
392
393     if (NULL != PspCurrentThread && Running == PspCurrentThread->Stat
394
395         if (0 != PspReadyBitmap && HighestPriority > PspCurrentThread
396
397             //
398             // 如果存在比当前运行线程优先级更高的就绪线程，当前线程应
399             // 因为当前线程仍处于运行状态，所以被高优先级线程抢先后应
400             // 优先级对应的就绪队列的队首。注意，不能调用 PspReadyThr
401             //
402             ListInsertHead( &PspReadyListHeads[PspCurrentThread->Prio
403                             &PspCurrentThread->StateListEntry );
404             BIT_SET(PspReadyBitmap, PspCurrentThread->Priority);
405             PspCurrentThread->State = Ready;
406
```

4. 此时，刷新“就绪线程队列”窗口。

5. 还可以在“调试”菜单“窗口”中选择“监视”，激活“监视”窗口（此时按 F1 可以获得关于“监视”窗口的帮助）。在“监视”窗口中添加表达式“/t PspReadyBitmap”，以二进制格式查看就绪位图变量的值。此时就绪位图的值应该为 100000001，表示优先级为 8 和 0 的两个就绪队列中存在就绪线程。（注意，如果就绪位图的值不是 100000001，就继续按 F5，直到就绪位图变为此值）。

查看监视窗口帮助如下：

变量窗口

- **“监视”窗口**

  “监视”窗口用于显示、计算和编辑变量与表达式。“监视”窗口是网格窗口，其中包含三列：“名称”、“值”和“类型”。“名称”列包含变量名称或表达式。“值”和“类型”列显示变量或表达式的值和数据类型。

  在“监视”窗口中可以添加要监视其值的变量。此外，还可以添加变量以外的其他内容，您可以添加调试器所能识别的任何有效表达式。

  在中断模式中时，右击源代码编辑窗口中的变量名，然后从快捷菜单中选择“添加监视”，这样，就将变量自动放置到“监视”窗口中。

  在“调试”菜单上指向“窗口”，然后选择“监视”即可打开“监视”窗口。

- **“快速监视”对话框**

  “快速监视”对话框在概念上类似于“监视”窗口，但是“快速监视”每次只能显示一个变量或表达式。如果需要快速查看变量或表达式而不想打开“监视”窗口，则可以使用“快速监视”。

  “快速监视”对话框的另一个好处是它是可变大小的。如果要查看一个较大对象的成员，在“快速监视”对话框中通常比在“监视”窗口中更方便查看。

  在中断模式中时，右击源代码编辑窗口中的变量名，然后从快捷菜单中选择“快速监视”，这样，就将变量自动放置到“快速监视”对话框中。

- **更改数字格式**

  可以将调试器窗口中用于显示数值的格式设置为十进制或十六进制。选择调试工具栏上的十六进制按钮可以在十进制和十六进制间切换。

  也可以在变量窗口中为表达式添加格式选项来指定用于显示数值的格式，语法为：

  /f expression

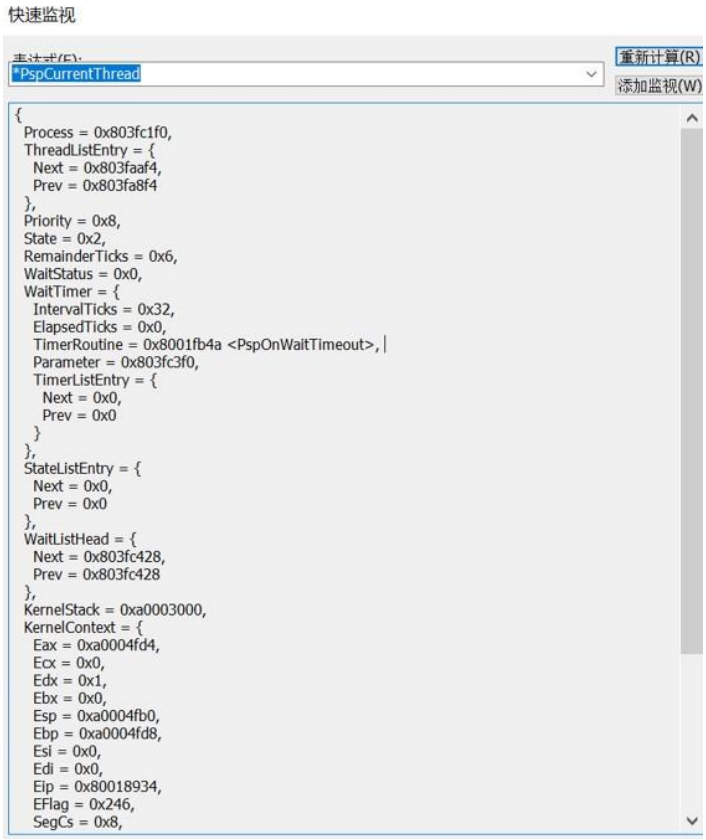
  其中 f 代表格式选项，可用的格式选项有：

选项	格式
/x	十六进制。
/d	有符号的十进制。
/u	无符号的十进制。
/o	八进制。
/t	二进制。
/f	浮点格式。
/s	字符串。

注意，格式选项比调试工具栏上的十六进制按钮的优先级高，用格式选项显示的表达式的值，将不会受十六进制按钮状态的影响。

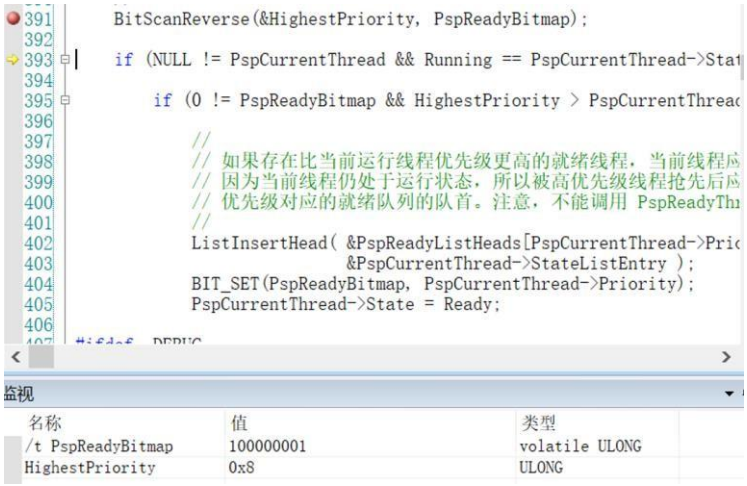
6. 在“调试”菜单中选择“快速监视”，在“快速监视”对话框的“表达式” 中输入表达式“*PspCurrentThread”后，点击“重新计算”按钮，可以查看当前正在运行的线程（即被中断的线程）的线程控制块中各个域的值。其中优先级（Priority 域）的值为 8；状态（State 域）的值为 2（运行状态）；时间片（RemainderTicks 域)的值为 6；线程函数（StartAddr 域)为 ThreadFunction。综合这些信息即可确定当前正在运行的线程就是“rr”命令新建的第 0 个线程。当然也可以通过“线程控制块”窗口查看这些内容。

查看“快速监视”窗口如下：



7. 关闭“快速监视”对话框。按 F10 单步调试，BitScanReverse 函数会从就绪位图中扫描最高优先级，并保存在变量 HighestPriority 中。查看变量 HighestPriority 的值为 8。

查看变量 HighestPriority 的值如下：



8. 继续按 F10 单步调试，直到在 PspSelectNextThread 函数返回前停止（第 473 行），注意观察线程调度执行的每一个步骤。通过调试线程调度函数 PspSelectNextThread 的执行过程，“rr”命令新建的第 0 个线程在执行线程调度时没有被抢先的原因可以归纳为两点：

(1) 第 0 个线程仍然处于“运行”状态。

(2) 没有比其优先级更高的处于就绪状态的线程。

调试结果如下：

```
463 #endif
464
465     //
466     // 换入线程绑定运行的地址空间。
467     //
468     MmSwapProcessAddressSpace(PspCurrentThread->AttachedPas);
469
470     //
471     // 返回线程的上下文环境块，恢复线程运行。
472     //
473     return &PspCurrentThread->KernelContext;
474 }
475
476 VOID
477 PspThreadSchedule(
478     VOID
479     )
480 /*++
```

3.3.2 调试当前线程被抢先的情况

如果有比第 0 个新建的线程优先级更高的线程进入就绪状态，则第 0 个新建的线程就会被抢先，例如在第 0 个线程运行的过程中，按下空格键，就会让之前处于阻塞状态的控制台派遣线程进入就绪状态，而控制台派遣线程的优先级为 24，高于优先级为 8 的第 0 个新建的线程，线程调度函数就会让控制台派遣线程抢占处理器。

读者可以按照下面的步骤调试这种情况在 PspSelectNextThread 函数中的处理过程（注意，接下来的调试要从本实验 3.3.1 调试的状态继续调试，所以不要结束之前的调试）。

1. 选择“调试”菜单中的“删除所有断点”，删除之前添加的所有断点。

2. 在 ps/sched.c 文件的 PspSelectNextThread 函数的第 402 行添加一个断点。

添加断点操作如下：

```
398             // 如果存在比当前运行线程优先级更高的就绪线程，当前线程应
399             // 因为当前线程仍处于运行状态，所以被高优先级线程抢先后应
400             // 优先级对应的就绪队列的队首。注意，不能调用 PspReadyThr
401             //
402             ListInsertHead( &PspReadyListHeads[PspCurrentThread->Pric
403                             &PspCurrentThread->StateListEntry );
404             BIT_SET(PspReadyBitmap, PspCurrentThread->Priority);
405             PspCurrentThread->State = Ready;
406
407 #ifdef _DEBUG
408             RECORD_TASK_STATE(ObGetObjectId(PspCurrentThread) , TS_RE
409 #endif
```

3. 按 F5 继续执行，激活虚拟机窗口，可以看到第 0 个新建的线程正在执行。

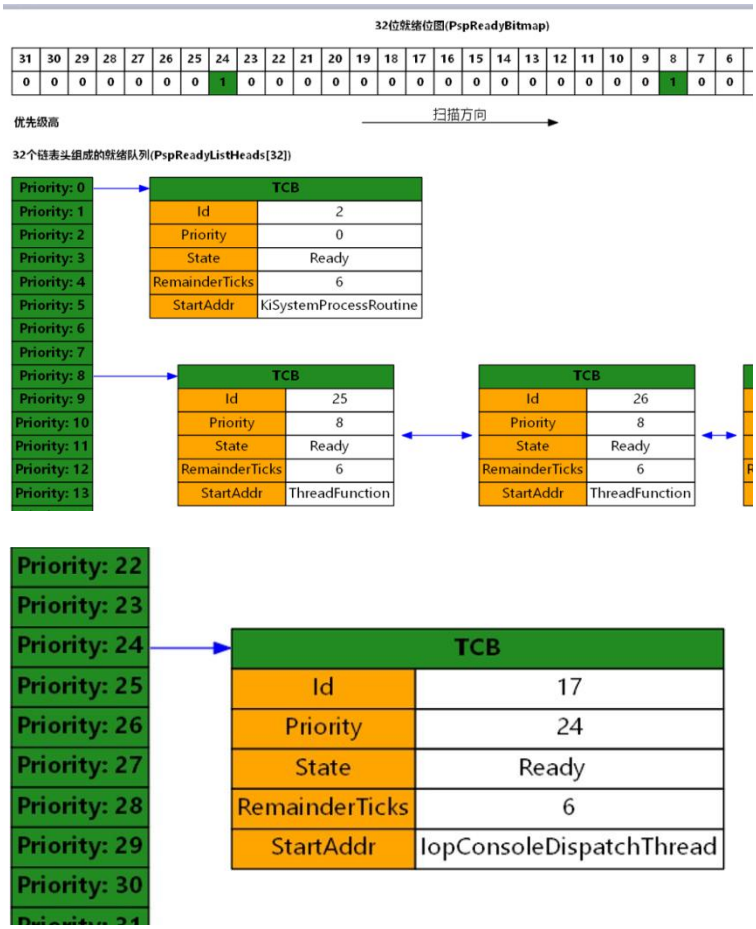
4. 在虚拟机窗口中按下一次空格键，使之前处于阻塞状态的控制台派遣线程进入就绪状态，并触发线程调度函数 PspSelectNextThread，就会在刚刚添加的断点处中断。

当前状态如下：

```
394
395         if (0 != PspReadyBitmap && HighestPriority > PspCurrentThread
396
397             //
398             // 如果存在比当前运行线程优先级更高的就绪线程，当前线程应
399             // 因为当前线程仍处于运行状态，所以被高优先级线程抢先后应
400             // 优先级对应的就绪队列的队首。注意，不能调用 PspReadyThr
401             //
402             ListInsertHead( &PspReadyListHeads[PspCurrentThread->Pric
403                             &PspCurrentThread->StateListEntry );
404             BIT_SET(PspReadyBitmap, PspCurrentThread->Priority);
405             PspCurrentThread->State = Ready;
406
407 #ifdef  DEBUG
```

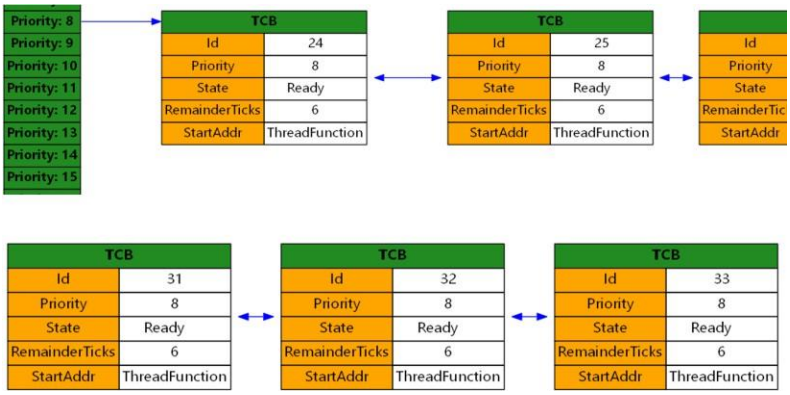
5. 刷新“就绪线程队列”窗口。可以看到，在 32 位就绪位图中第 24 位用绿色高亮显示且值为 1，说明优先级为 24 的就绪队列中存在就绪线程。在“32 个链表头组成的就绪队列”中可以查看优先级为 24 的就绪队列中挂接了一个处于就绪状态的线程，如图 14-6 所示，通过其线程函数名称可以确认其为控制台派遣线程。

“就绪线程队列”窗口如下：



6. 按 F10 单步调试到第 408 行。由于线程调度函数 PspSelectNextThread 在前面扫描就绪位图时已经发现了存在优先级为 24 的就绪线程，其优先级高于正在运行的第 0 个新建的线程，所以在刚刚执行的语句中将当前正在运行的第 0 个新建的线程放入优先级为 8 的就绪队列的队首，并将其状态设置为就绪状态。此时刷新“就绪线程队列”窗口，可以看到新建的第 0 个线程已经挂接在了优先级为 8 的就绪队列的队首，优先级为 8 的就绪队列中一共挂接了 10 个线程。

“就绪线程队列”窗口如下：



7. 继续按 F10 单步调试，直到在第 455 行中断执行，注意观察线程调度执行的每一个步骤。此时，正在执行的第 0 个新建的线程已经进入了就绪状态，让出了 CPU。线程调度程序接下来的工作就是选择优先级最高的非空就绪队列的队首线程作为当前运行线程，也就是让优先级为 24 的控制台派遣线程在 CPU 上执行。

执行状态如下：

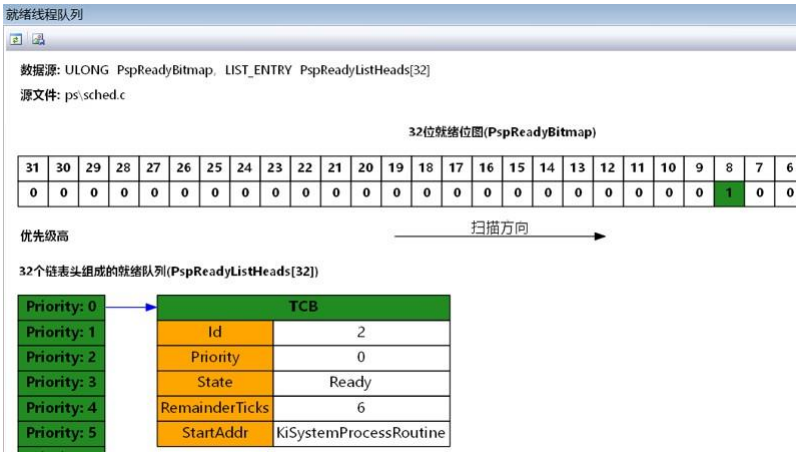
```
454     //
455     PspCurrentThread = CONTAINING_RECORD(PspReadyListHeads[Hig
456     ObRefObject(PspCurrentThread);
457
458     PspUnreadyThread(PspCurrentThread);
459     PspCurrentThread->State = Running;
460
461 #ifdef _DEBUG
```

8. 继续按 F10 单步调试，直到在 PspSelectNextThread 函数返回前（第 473 行）中断执行，注意观察线程调度执行的每一个步骤。此时，优先级为 24 的控制台派遣线程已经进入了运行状态，在中断返回后，就可以开始执行了。刷新“就绪线程队列”窗口，可以看到线程调度函数已经将控制台派遣线程移出了就绪队列。刷新“进程线程列表”窗口，可以看到当前线程指针 PspCurrentThread 已经指向了控制台派遣线程。

函数执行状态如下：

名称	值	类型
/t PspReadyBitmap	100000001	volatile ULONG
HighestPriority	0x18	ULONG
PspCurrentThread	(volatile PTHREAD) 0x803fd0f0	volatile PTHREAD

“就绪线程队列”窗口如下：



9. 删除所有的断点后，结束调试。

通过以上的调试过程，读者已经观察到了基于优先级的抢先式调度算法中高优先级线程抢占处理器的完成过程和源代码实现。

3.4 为 EOS 添加时间片轮转调度算法

3.4.1 要求

修改 ps/sched.c 文件中的 PspRoundRobin 函数(第 344 行)，在其中实现时间片轮转调度算法。

修改结果如下：

```
348
349  功能描述:
350      时间片轮转调度函数，被定时计数器中断服务程序 KiIsrTimer 调用。
351
352  参数:
353      无。
354
355  返回值:
356      无。
357
358  --*/
359  {
360      if (NULL != PspCurrentThread && Running == PspCurrentThread->State)
361      {
362          // 在此实现时间片轮转调度算法
363          PspCurrentThread->RemainderTicks--;
364          if (PspCurrentThread->RemainderTicks == 0)
365          {
366              PspCurrentThread->RemainderTicks = TICKS_OF_TIME_SLICE;
367
368              if (BIT_TEST(PspReadyBitmap, PspCurrentThread->Priority))
369              {
370                  PspReadyThread(PspCurrentThread);
371              }
372          }
373      }
374      return;
375  }
```

3.4.2 测试方法

1. 代码修改完毕后，按 F7 生成 EOS 内核项目。
2. 按 F5 启动调试。
3. 在 EOS 控制台中输入命令“rr”后按回车。应能看到 10 个线程轮转执行的效果。

轮转执行最初结果如下：

```
CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
Thread 0 (ID:24, Priority:8): 28
Thread 1 (ID:25, Priority:8): 29
Thread 2 (ID:26, Priority:8): 20
Thread 3 (ID:27, Priority:8): 20
Thread 4 (ID:28, Priority:8): 19
Thread 5 (ID:29, Priority:8): 19
Thread 6 (ID:30, Priority:8): 19
Thread 7 (ID:31, Priority:8): 19
Thread 8 (ID:32, Priority:8): 18
Thread 9 (ID:33, Priority:8): 19
```

轮转执行了一定时间后：

```
Thread 0 (ID:24, Priority:8): 178
Thread 1 (ID:25, Priority:8): 177
Thread 2 (ID:26, Priority:8): 177
Thread 3 (ID:27, Priority:8): 180
Thread 4 (ID:28, Priority:8): 179
Thread 5 (ID:29, Priority:8): 179
Thread 6 (ID:30, Priority:8): 179
Thread 7 (ID:31, Priority:8): 169
Thread 8 (ID:32, Priority:8): 168
Thread 9 (ID:33, Priority:8): 169
```

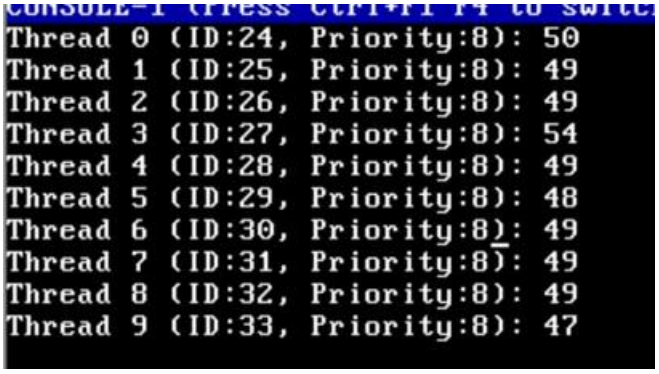
3.5 修改线程时间片的大小

在成功为 EOS 添加了时间片轮转调度后，可以按照下面的步骤修改时间片的大小：

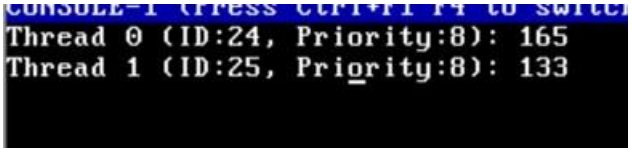
1. 在 OS Lab 的“项目管理器”窗口中找到 ps/psp.h 文件，双击打开此文件。
2. 将 ps/psp.h 第 120 行定义的 TICKS_OF_TIME_SLICE 的值修改为 1。
3. 按 F7 生成 EOS 内核项目。
4. 按 F5 启动调试。
5. 在 EOS 控制台中输入命令“rr”后按回车。观察执行的效果。

还可以按照上面的步骤为 TICKS_OF_TIME_SLICE 取一些其它的极端值，例如 20 或 100 等，分别观察“rr”命令执行的效果。通过分析造成执行效果不同的原因，理解时间片的大小对时间片轮转调度造成的影响。

时间片为 1 时的执行（每一个都在不停变化，速度较快）：



时间片为 100 时的执行（肉眼可见地一个个执行，速度放慢）：



4. 实验的思考与问题分析

1. 结合线程调度执行的时机，说明在 ThreadFunction 函数中，为什么可以使用“关中断”和“开中断”的方法来保护控制台这种临界资源。一般情况下，应该使用互斥信号量（MUTEX）来保护临界资源，但是在 ThreadFunction 函数中却不能使用互斥信号量，而只能使用“关中断”和“开中断”的方法，结合线程调度的对象说明这样做的原因。

答：关中断后 CPU 就不会响应任何由外部设备发出的硬中断（包括定时计数器和键盘中断等）了，也就不会发生线程调度了，从而保证各个线程可以互斥地访问控制台。

不能是要互斥信号量保护临界资源的原因：如果使用互斥信号量，则那些由于访问临界区而被阻塞的线程，就会被放入互斥信号量的等待队列，就不会在相应优先级的就绪队列中了，而时间片轮转调度算法是对就绪队列的线程进行轮转调度，而不是对这些被阻塞的线程进行调度，也就无法进行实验了。使用“关中断”和“开中断”进行同步就不会改变线程的状态，可以保证那些没有获得处理

器的线程都在处于就绪队列中。

2. EOS 内核时间片大小取 60ms（和 Windows 操作系统完全相同），在线程比较多时，就可以观察出线程轮流执行的情况（因为此时一次轮转需要 60ms，10 个线程轮流执行一次需要 60×10＝600ms，所以 EOS 的控制台上可以清楚地观察到线程轮流执行的情况）。但是在 Windows、Linux 等操作系统启动后，正常情况下都有上百个线程在并发执行，为什么觉察不到它们被轮流执行，并且每个程序都运行的很顺利呢？

答：在系统中存在着许多的线程，但大部分的线程还是处于阻塞状态，而且 CPU 的处理速度很快，处理完其中一个马上又会转到另一个，每一个线程基本都会轮转到，在用户的感知下，就会觉得每个操作都很顺利。

5. 总结和感想体会

⑴ 熟悉了轮转调度算法和优先级抢占机制，对于该机制的原理、过程、具体实现都有了较深刻的体验。

⑵ 对于时间轮转算法的应用也有了一定了解，并尝试了不同的时间片，当时间片较小时体会不到进程之间的切换，当时间片较大时会很明显，时间片大小的确定非常重要。

⑶ 优先级抢占算法在某些场合可能并不适用，具体选择哪个机制还要结合实际的应用环境。