合肥工業大學

操作系统实验报告

实验题目	实验 6 时间片轮转调度
学生姓名	孙淼
学　　号	2018211958
专业班级	计算机科学与技术 18-2 班
指导教师	田卫东
完成日期	12.01

合肥工业大学 计算机与信息学院

1. 实验目的和任务要求

- 调试 EOS 的线程调度程序，熟悉基于优先级的抢先式调度。
- 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

重点理解 EOS 当前使用的基于优先级的抢先式调度，调度程序执行的 时机和流程，以及实现时间片轮转调度的细节。阅读本书第 5 章中的第 5.4 节。重点理解 EOS 当前使用的基于优先级的抢先式调度，调度程序执行的时机和流程，以及实现时间片轮转调度的细节。

3. 实验内容

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。

```
688   VOID
689   ConsoleCmdRoundRobin(
690       IN HANDLE StdHandle
691       )
692   /*++
```

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。

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）。

```
638   //
639   // 线程参数结构体
640   //
641   typedef struct _THREAD_PARAMETER {
642       SHORT Y;                // 线程输出内容所在行
643       HANDLE StdHandle;       // 线程用来输出内容的标准句柄
644   }THREAD_PARAMETER, *PTHREAD_PARAMETER;
645
646   PRIVATE
647   ULONG
648   ThreadFunction(
649       PVOID Param
650       )
651   /*++
```

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。

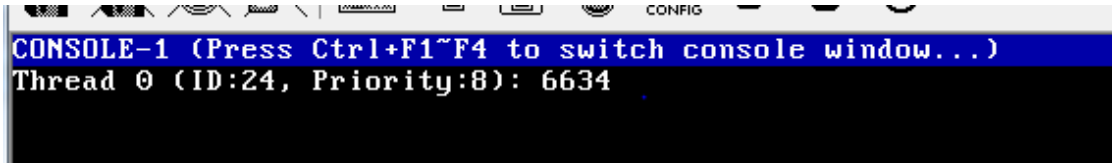
新建的线程会一直运行，原因是在 ThreadFunction 函数中使用了死循环，所以只能在 ConsoleCmdRoundRobin 函数的最后调用 TerminateThread 函数来强制结束这些新建的线程。

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。

命令开始执行后，观察其执行效果（如图 14-1），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。

结果如下：



调试线程调度程序，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。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

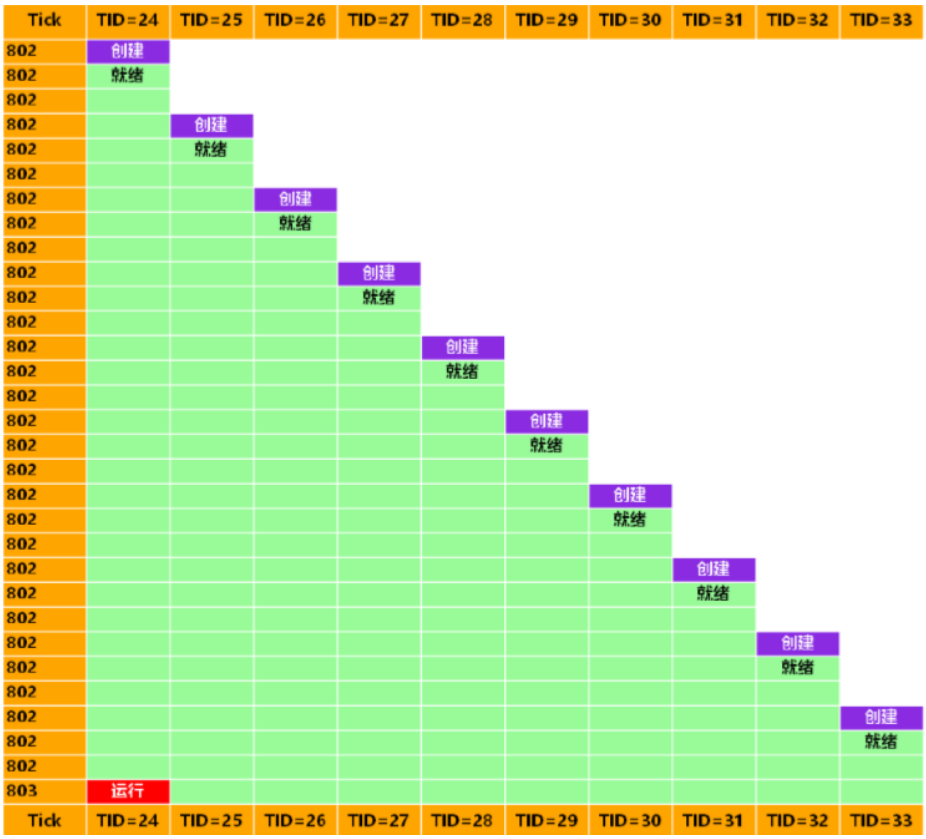
序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	16	2	"N/A"

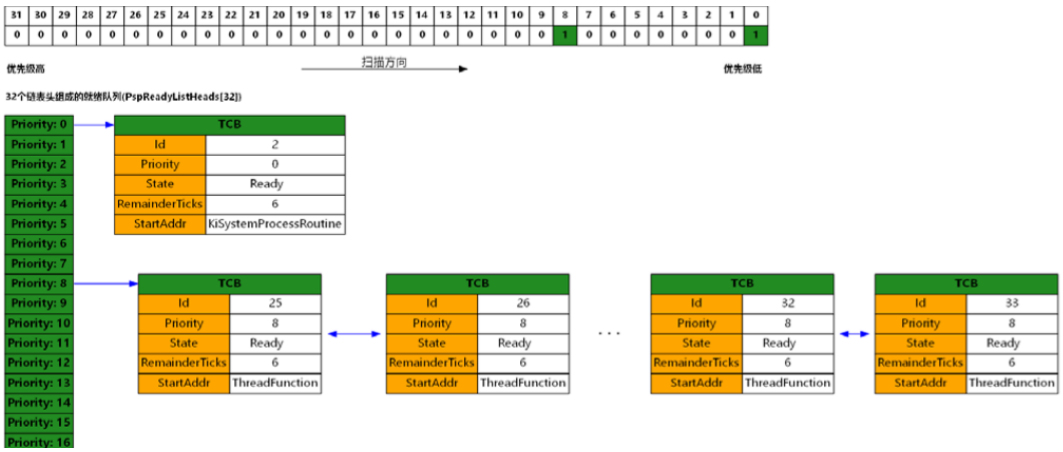
线程列表

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 ← PspCurrentThread
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

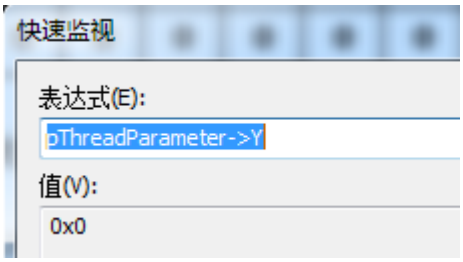
在 EOS 控制台中输入命令“rr”后按回车。“rr”命令开始执行后，会在断点处中断。刷新“进程线程”窗口，可以看到如图 14-2 所示的内容。其中，从 ID 为 24 到 ID 为 33 的线程是“rr”命令创建的 10 个优先级为 8 的线程，ID 为 24 的线程处于运行状态，其它的 9 个线程处于就绪状态。

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。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。

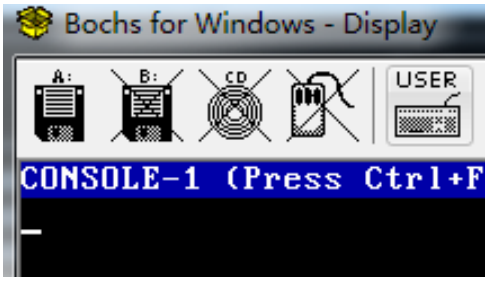




查看 ThreadFunction 函数中变量 pThreadParameter->Y 的值应该为 0，说明正在调试的是第 0 个新建的线程。



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。



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。

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。

图例： 创建 就绪 运行 阻塞 结束

默认最多显示10个线程的120行数据，可使用本窗口工具栏上的“绘制指定范围的线程”按钮，查看需要显示的内容

Tick	TID=24	TID=25	TID=26	TID=27	TID=28	TID=29	TID=30	TID=31	TID=32	TID=33	函数	行号
2143	创建										PspCreateThread	572
2143	就绪										PspReadyThread	134
2143											PspReadyThread	134
2143		创建									PspCreateThread	572
2143		就绪									PspReadyThread	134
2143											PspReadyThread	134
2143			创建								PspCreateThread	572
2143			就绪								PspReadyThread	134
2143											PspReadyThread	134
2143				创建							PspCreateThread	572
2143				就绪							PspReadyThread	134
2143											PspReadyThread	134
2143					创建						PspCreateThread	572
2143					就绪						PspReadyThread	134
2143											PspReadyThread	134
2143						创建					PspCreateThread	572
2143						就绪					PspReadyThread	134
2143											PspReadyThread	134
2143							创建				PspCreateThread	572
2143							就绪				PspReadyThread	134
2143											PspReadyThread	134
2143								创建			PspCreateThread	572
2143								就绪			PspReadyThread	134
2143											PspReadyThread	134
2143									创建		PspCreateThread	572
2143									就绪		PspReadyThread	134
2143											PspReadyThread	134
2143										创建	PspCreateThread	572
2143										就绪	PspReadyThread	134
2143											PspReadyThread	134
2144	运行										PspSelectNextThread	462
2145	就绪										PspSelectNextThread	408
2145	运行										PspSelectNextThread	462
2146	就绪										PspSelectNextThread	408
2146	运行										PspSelectNextThread	462
Tick	TID=24	TID=25	TID=26	TID=27	TID=28	TID=29	TID=30	TID=31	TID=32	TID=33	函数	行号

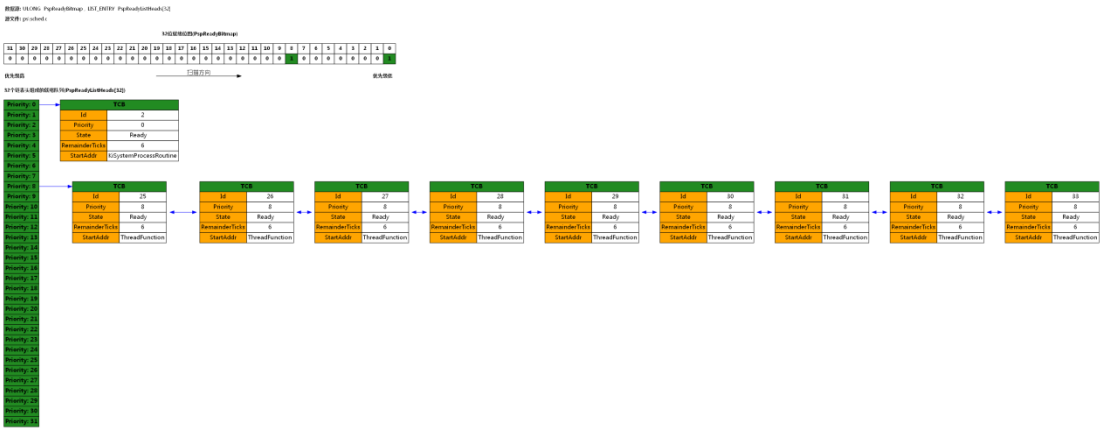
通过之前的调试，可以观察到“rr”命令新建的第 0 个线程始终处于运行状态，而不会被其它具有相 同优先级的线程抢先，那么在 EOS 内核中是如何实现这种调度算法的呢？读者可以按照下面的步骤进行调 试，查看当有中断发生从而触发线程调度时，第 0 个新建的线程不会被抢先的情况。

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。

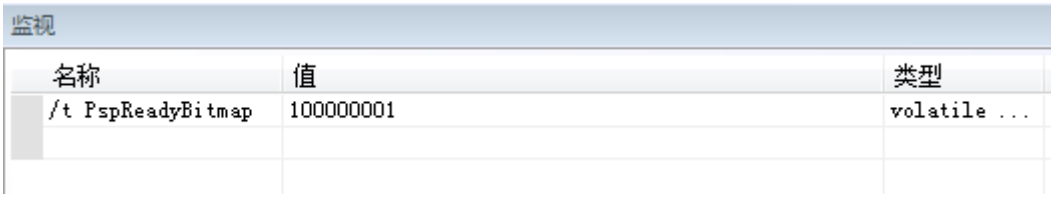
3. 按 F5 继续执行。因为每当有定时计数器中断发生时（每 10ms 一次）都会触发线程调度函数 PspSelectNextThread，所以很快就会在刚刚添加的断点处中断。

4. 此时，刷新“就绪线程队列”窗口，仍然会显示如图 14-4 所示的内容。

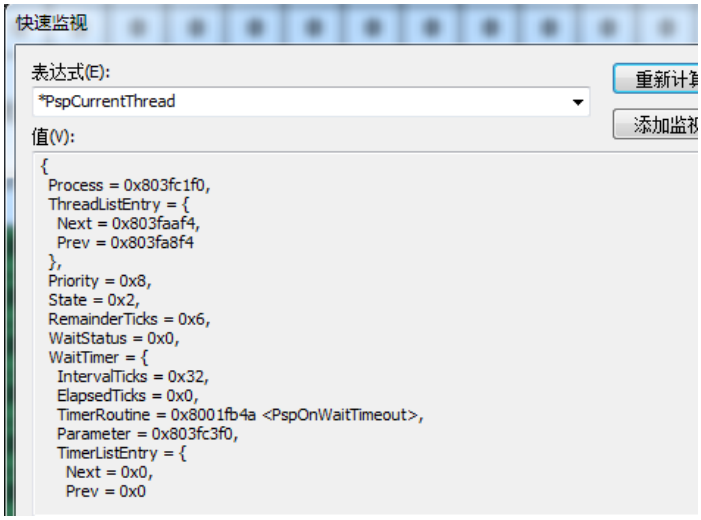


5. 还可以在“调试”菜单“窗口”中选择“监视”，激活“监视”窗口（此时按 F1 可以获得关于“监 视”窗口的帮助）。在“监视”窗口中添加表达式“/t PspReadyBitmap”，以二进制格式查看就 绪位图变量的值。此时就绪位图的值应该为 100000001，表示优先级为 8 和 0 的两个就绪队列中 存在就绪线程。（注意，如果就绪位图的值不是 100000001，就继续按 F5，直到就绪位图变为此值）。

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。



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。

名称	值	类型
/t PspReadyBitmap	100000001	volatile ...
HighestPriority	0x8	ULONG

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。
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。
5. 刷新“就绪线程队列”窗口，会显示如图 14-5 所示的内容。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，通过其线程函数名称可以确认其

为控制台派遣线程。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

优先级高　　扫描方向　　优先级低

图 14-5：控制台派遣线程进入就绪状态后的就绪位图

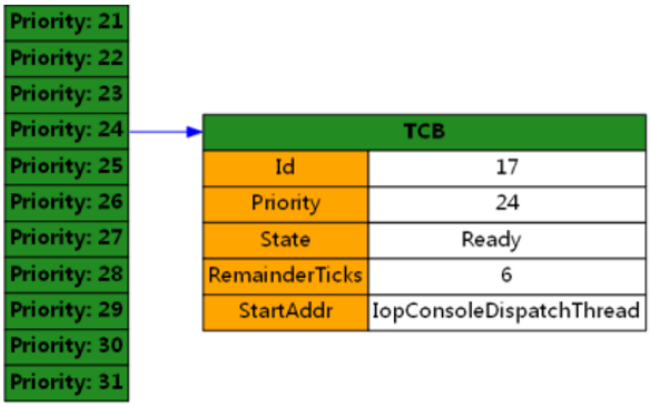
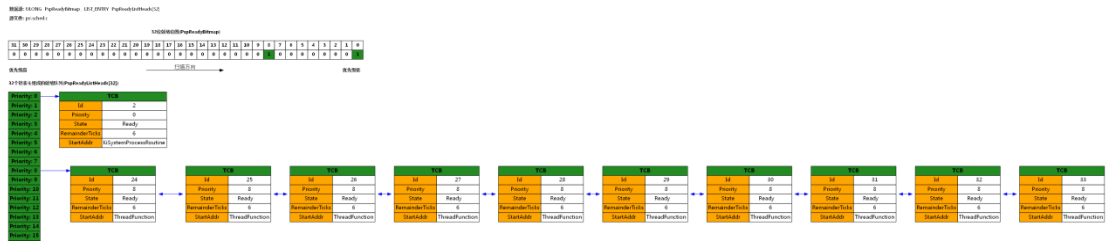


图 14-6：控制台派遣线程进入就绪队列

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。

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。



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。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	16	2	"N/A"

线程列表

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	Running (2)	1	0x80015724 IopConsoleDispatchThread ← PspCurrentThread
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	Ready (1)	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

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。读者可以结合 PspSelectNextThread 函数中的源代码和实验中提供的详细步骤多做一些练习，加深对线程调度算法的理解。

为 EOS 添加时间片轮转调度算法

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。

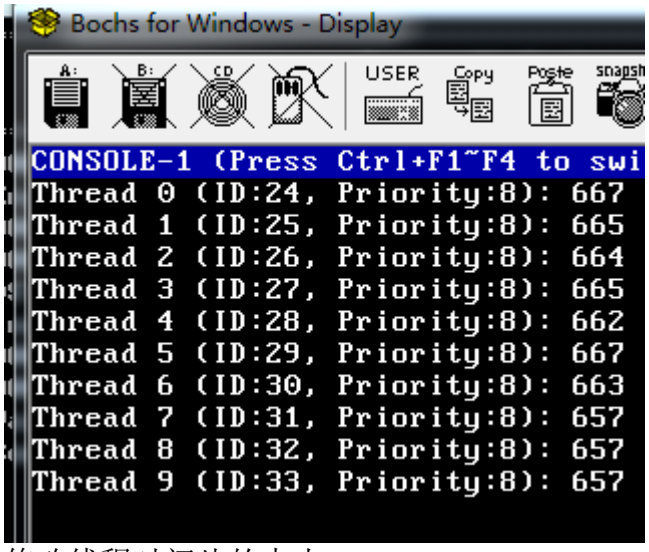
修改部分代码：

```
VOID PspRoundRobin( VOID ){
    if(NULL!=PspCurrentThread&&Running==PspCurrentThread->State) {
    PspCurrentThread->RemainderTicks--;
    if (0 == PspCurrentThread->RemainderTicks) {
    PspCurrentThread->RemainderTicks = TICKS_OF_TIME_SLICE;
```

```
    if(BIT_TEST(PspReadyBitmap, PspCurrentThread->Priority)) {
     PspReadyThread(PspCurrentThread); } } }}
```

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。
4. 读者也可以尝试通过“线程运行轨迹”窗口查看线程轮转执行的效果。

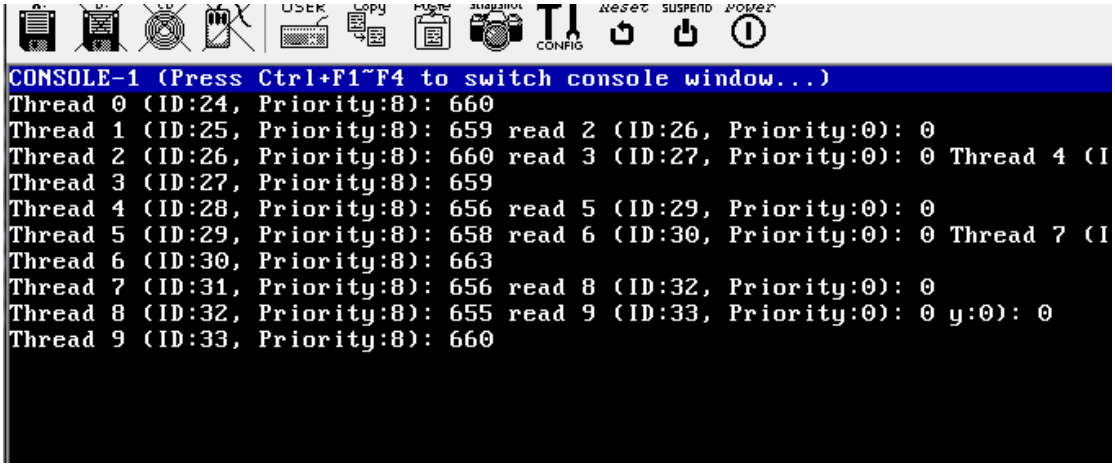


修改线程时间片的大小

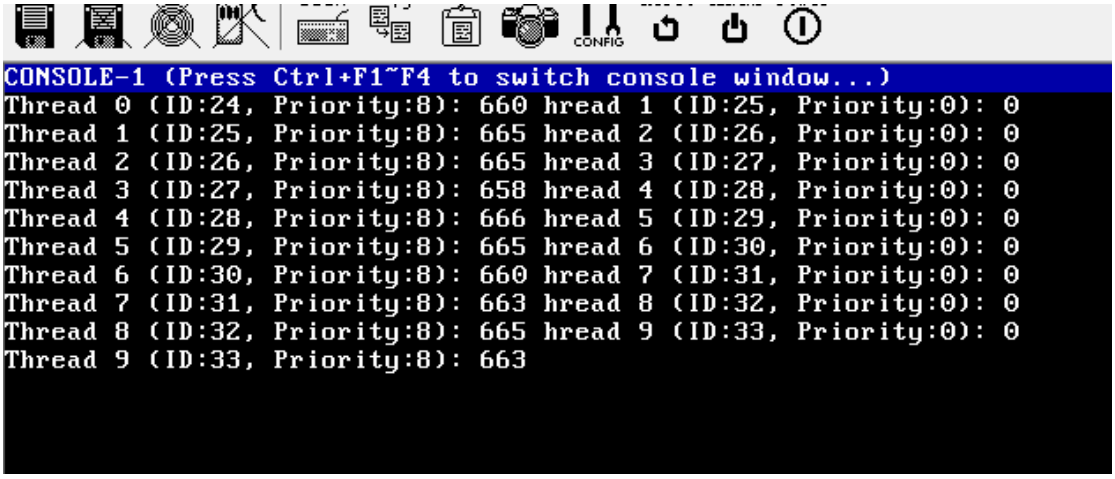
在成功为 EOS 添加了时间片轮转调度后，可以按照下面的步骤修改时间片的大小：

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。

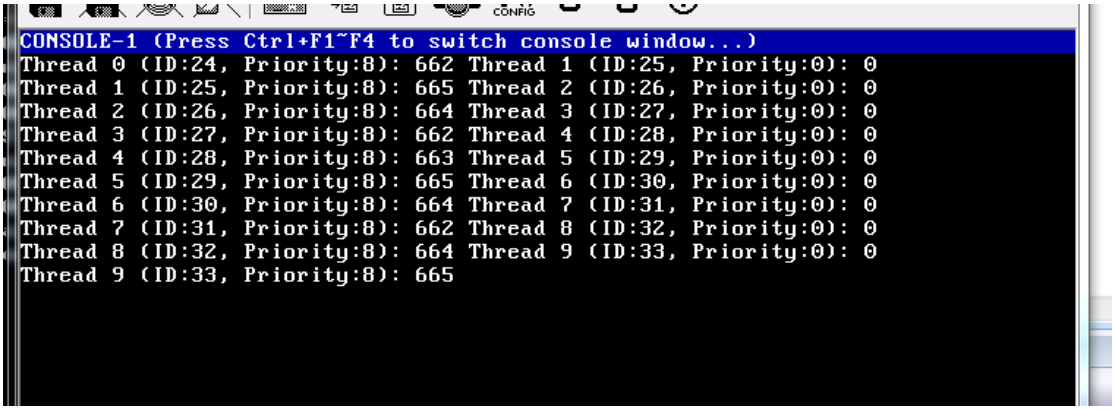
TICKS_OF_TIME_SLICE



TICKS_OF_TIME_SLICE



TICKS_OF_TIME_SLICE



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：

EOS 会设置 CPU 停止响应外部设备产生的硬中断，也就不会在由硬中断触发线程调度。开中断和关中断使处理机在这段时间屏蔽掉了外界所有中断，

使他线程无法占用资源。使用开中断和关中断进程同步不会改变线程状态，可以保证那些没有获得处理器的资源都在就绪队列中。关中断后 CPU 就不会响应任何由外部设备发出的硬中断（包括定时计数器中断和键盘中断等）了，也就不会发生线程调度了，从而保证各个线程可以互斥的访问控制台。这里绝对不能使用互斥信号量（mutex）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列中了，而时间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）时间片轮转调度发现被中断线程的时间片用完后，而且在就绪队列中没有与被中断线程优先级相同的就绪线程时，为什么不需要将被中断线程转入“就绪”状态？如果此时将被中断线程转入了“就绪” 状态又会怎么样？可以结合 PspRoundRobin 函数和 PspSelectNextThread 函数的流程进行思考，并使用抢先和不抢先两种情况进行说明。
答：
（1）因为其他优先队列的线程等待时间不能过长。
（2）若将中断线程转入就绪队列，只有当此线程执行完毕之后，其他队列的线程才有机会进入就绪队列，尤其是当其他就绪队列中的线程关于人机交互的时候，会严重影响用户体验。

（3）在 EOS 只实现了基于优先级的抢先式调度时，同优先级的线程只能有一个被执行。当实现了时间片轮转调度算法后，同优先级的线程就能够轮流执行从而获得均等的执行机会。但是，如果有高优先级的线程一直占用 CPU，低优先级的线程就永远不会被执行。尝试修改 ke/sysproc.c 文件中的 ConsoleCmdRoundRobin 函数来演示这种情况（例如在 10 个优先级为 8 的线程执行时，创建一个优先级为 9 的线程）。设计一种调度算法来解决此问题，让低优先级的线程也能获得被执行的机会。
答：
解决该问题的最简单方法是实现动态优先级算法。动态优先级是指在创建进程是所赋予的优先级，可随线程的推进而改变，以便获得良好的性能调度。例如，可用规定，在就绪队列中的线程，随着其等待时间的增长，其优先级以速率 X 增加，并且正在执行的线程，其优先级以速率 y 下降。这样，在各个线程具有不同优先级的情况下，对于优先级低的线程，在等待足够的时间后，其优先级便可能升为最高，从而获得被执行的机会。此时，在基于优先级的抢占式调度算法、时间片轮转调度算法和动态优先级算法的共同作用下，可防止一个高优先级的长作业长期的垄断处理器。

（4）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 Windows、linux 等操作系统中，虽然都提供了时间片轮转调度算法却很少真正被派上用场，下面解释原因，在 Windows 任务管理器中，即使系统中已经运行了数百个线程，但 CPU 的利用率仍然很低，甚至为 0.因为这些线程在大部分时间都处于阻塞状态，阻塞的原因是各种各样的，最主要的原因是等待 I/O 完成或者等待命令消息的到达。例如，在编辑 Word 文档时，每敲击一次键盘，Word 就会立即作出反应，并且文档中插入字符。此时会感觉 Word 运行的非常流畅。事实上，并非如此，Word 主线程大部分时间都处于阻塞等待状态，等待用户敲击键盘。在用户没有敲击键盘或没有使用鼠标点击时，Word 主线程处于阻塞状态，它将让出处理器给其它需要的线程。当用户敲击一个按键后，Word 主线程将会立刻被操作系统唤醒，此时 Word 开始处理请求。Word 在处理输入请求时所用的 CPU 时间是非常短的（因为 CPU 非常快），是微秒级的，远远低于时间片轮转调度的时间片大小（Windows 下是 60 毫秒），处理完毕后 Word 又立刻进入阻塞状态，等待用户下一次敲击键盘。或者拿音乐播放器来分析，表面上感觉播放器在不停地播放音乐，但是 CPU 的利用率仍然会很低。这是由于播放器将一段声音编码交给声卡，由声卡来播放，在声卡播放完这段声音之前，播放器都是处于阻塞等待状态的。当声卡播放完片段后，播放器将被唤醒，然后它将下一个声音片段交给声卡继续播放。掌握了上面的知识后，就可以很容易解释为什么这么多线程同时在运行而一点都感觉不到轮替现象。

5. 总结和感想体会

经过这次实验，我对时间片调度的理解更深了，时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。如果在时间片结束时进程还在运行，则 CPU 将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则 CPU 当即进行切换。调度程序所要做的就是维护一张就绪进程列表，当进程用完它的时间片后，它被移到队列的末尾。