实验 4 线程的状态和转换

1. 实验目的和任务要求

（1） 调试线程在各种状态间的转换过程，熟悉线程的状态和转换。

（2） 通过为线程增加挂起状态，加深对线程状态的理解。

2. 实验原理

⑴ 线程的状态和 EOS 是如何定义这些状态的，线程是如何在这些状态之间转换的。

⑵ EOS 为线程添加的挂起状态，Suspend 和 Resume 原语操作。

⑶ 线程状态的转换是与线程的同步和线程的调度是密不可分的。

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 调试线程状态的转换过程

在本练习中，会在与线程状态转换相关的函数中添加若干个断点，并引导读者单步调试这些函数，使读者对 EOS 中的下列线程状态转换过程有一个全面的认识：

（1） 线程由阻塞状态进入就绪状态。

（2） 线程由运行状态进入就绪状态。

（3） 线程由就绪状态进入运行状态。

（4） 线程由运行状态进入阻塞状态。

为了完成这个练习，EOS 准备了一个控制台命令“loop”，这个命令的命令函数是 ke/sysproc.c 文件中的 ConsoleCmdLoop 函数（第 796 行），在此函数中使用 LoopThreadFunction 函数（第 754 行）作为线程函数创建了一个优先级为 8 的线程（后面简称为“loop 线程”），该线程会在控制台中不停的（死循环）输出该线程的 ID 和执行计数，执行计数会不停的增长以表示该线程在不停的运行。可以按照下面的步骤查看一下 loop 命令执行的效果：

（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“loop”后按回车，观察执行的效果。

（4） 结束此次调试。

调试的初始情况如下：

在计数一定时间后，情况如下：

接下来按照下面的步骤查看一下当 loop 线程正在运行时，系统中各个线程的状态：

1. 在 ke/sysproc.c 文件的 LoopThreadFunction 函数中，开始死循环的代码行（第 786 行）添加一个断点。

断点如图所示：

2. 按 F5 启动调试。

3. 待 EOS 启动完毕，在 EOS 控制台中输入命令“loop”后按回车。

EOS 会在断点处中断执行，表明 loop 线程已经开始死循环了。选择“调试”菜单“窗口”中的“进程线程”，打开“进程线程”窗口，在该窗口工具栏上点击“刷新”按钮，可以查看当前系统中所有的线程信息。其中，系统空闲线程处于就绪状态，其优先级为 0；控制台派遣线程和所有控制台线程处于阻塞状态，其优先级为 24；只有优先级为 8 的 loop 线程处于运行状态，能够在处理器上执行。

在 EOS 控制台中输入命令“loop”，按回车：

```
sysproc.c  起始页  start.c
759  功能描述：
760      死循环线程函数。
761
762  参数：
763      Param — 未使用。
764
765  返回值：
766      返回 0 表示线程执行成功。
767
768  --*/
769  {
770      ULONG i;
771      ULONG ThreadID = GetCurrentThreadId();
772      COORD CursorPosition;
773      HANDLE StdHandle = (HANDLE)Param;
774
775      // 清理整个屏幕的内容。
776      for (i = 0; i < 24; i++) {
777          fprintf(StdHandle, "\n");
778      }
779
780      // 设置线程输出内容显示的位置
781      CursorPosition.X = 0;
782      CursorPosition.Y = 0;
783
784      // 死循环。
785      // 格式：Thread ID 线程ID : 执行计数
786      for (i=0;;i++) {
787          SetConsoleCursorPosition(StdHandle, CursorPosition);
788          fprintf(StdHandle, "Loop thread ID %d : %u ", ThreadID,
789      }
790
791      return 0;
792  }
```

刷新完成后，当前的进程线程状态如下：

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)	映像名称 (ImageName)
1	1	Y	24	7	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 KiSheellThread
4	19	Y	24	Waiting (3)	1	0x80017f4b KiSheellThread
5	20	Y	24	Waiting (3)	1	0x80017f4b KiSheellThread
6	21	Y	24	Waiting (3)	1	0x80017f4b KiSheellThread
7	24	Y	8	Running (2)	1	0x80018a5c LoopThreadFunction ← PspCurrentThread

接下来按照下面的步骤对断点进行一些调整，为后面的调试过程做准备。注意，不要停止之前的调试，

下面的步骤要在之前调试的基础上继续进行调试：

1. 删除所有断点。
2. 选择“视图”菜单中的“项目管理器”，打开“项目管理器”窗口。
3. 在“项目管理器”窗口中双击打开 ps/sched.c 文件，在与线程状态转换相关的函数中添加断点，这样，一旦有线程的状态发生改变，EOS 会中断执行，就可以观察线程状态转换的详细过程了。

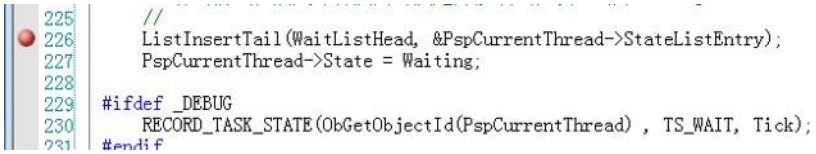
需要添加的断点有：

在 PspReadyThread 函数体中添加一个断点（第 129 行）。

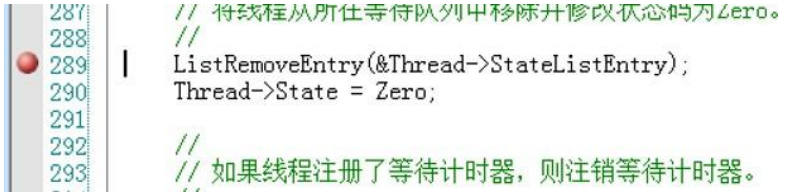
```
127      // 最后将线程的状态修改为就绪状态。
128      //
129      ListInsertTail(&PspReadyListHeads[Thread->Priority], &Thread->StateListEntry);
130      BIT_SET(PspReadyBitmap, Thread->Priority);
131      Thread->State = Ready;
132
133  #ifdef _DEBUG
134      RECORD_TASK_STATE(ObGetObjectId(Thread) , TS_READY, Tick);
135  #endif
136  }
```

在 PspUnreadyThread 函数体中添加一个断点（第 161 行）。

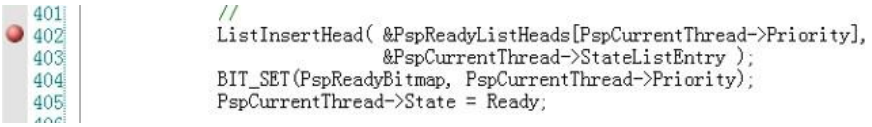
在 PspWait 函数体中添加一个断点（第 226 行）。



在 PspUnwaitThread 函数体中添加一个断点（第 289 行）。



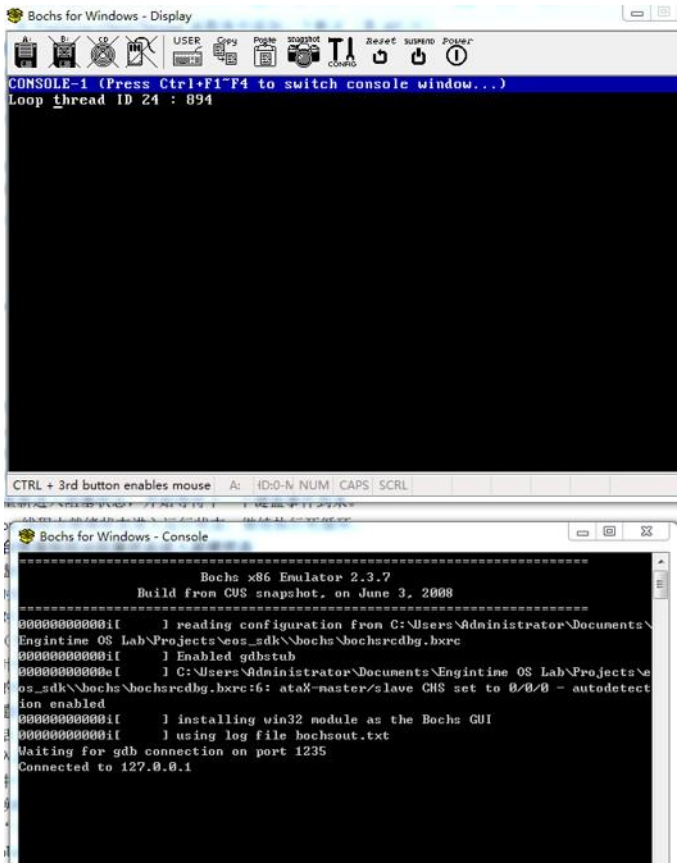
在 PspSelectNextThread 函数体中添加一个断点（第 402 行）。



4. 按 F5 继续执行，然后激活虚拟机窗口。

此时在虚拟机窗口中会看到 loop 线程在不停执行，而之前添加的断点都没有被命中，说明此时还没有任何线程的状态发生改变。（注意，如果命中了断点，可能是由于之前存在未处理的键盘事件导致的，此时只需继续按 F5，直到不再命中断点为止。）

虚拟机的执行情况如下（持续执行，断点均没有命中）：



在开始观察线程状态转换过程之前还有必要做一个说明。在后面的练习中，会在 loop 线程执行的过程中按一次空格键，这会导致 EOS 依次执行下面的操作：

1. 控制台派遣线程被唤醒，由阻塞状态进入就绪状态。
2. loop 线程由运行状态进入就绪状态。
3. 控制台派遣线程由就绪状态进入运行状态。
4. 待控制台派遣线程处理完毕由于空格键被按下而产生的键盘事件后，控制台派遣线程会由运行状态重新进入阻塞状态，开始等待下一个键盘事件到来。
5. loop 线程由就绪状态进入运行状态，继续执行死循环。

3.2.1 控制台派遣线程由阻塞状态进入就绪状态

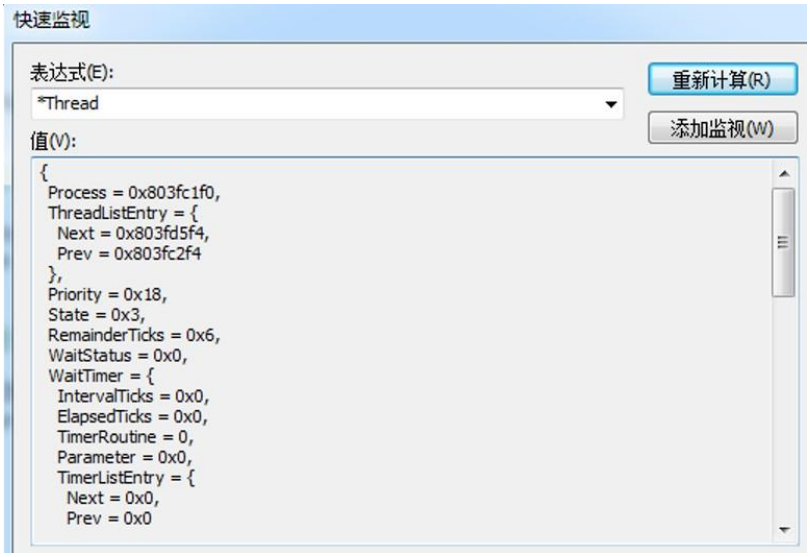
1. 在虚拟机窗口中按下一次空格键。

结果如下图：

2. 此时 EOS 会在 PspUnwaitThread 函数中的断点处中断。在“调试”菜单中选择“快速监视”，在快速监视对话框的表达式编辑框中输入表达式“*Thread”，

然后点击“重新计算”按钮，即可查看线程控制块（TCB）中的信息。其中 State 域的值为 3（Waiting），双向链表项 StateListEntry 的Next 和Prev 指针的值都不为 0，说明这个线程还处于阻塞状态，并在某个同步对象的等待队列中；StartAddr 域的值为 IopConsoleDispatchThread，说明在 PspUnwaitThread 函数中要处理的线程就是控制台派遣线程。

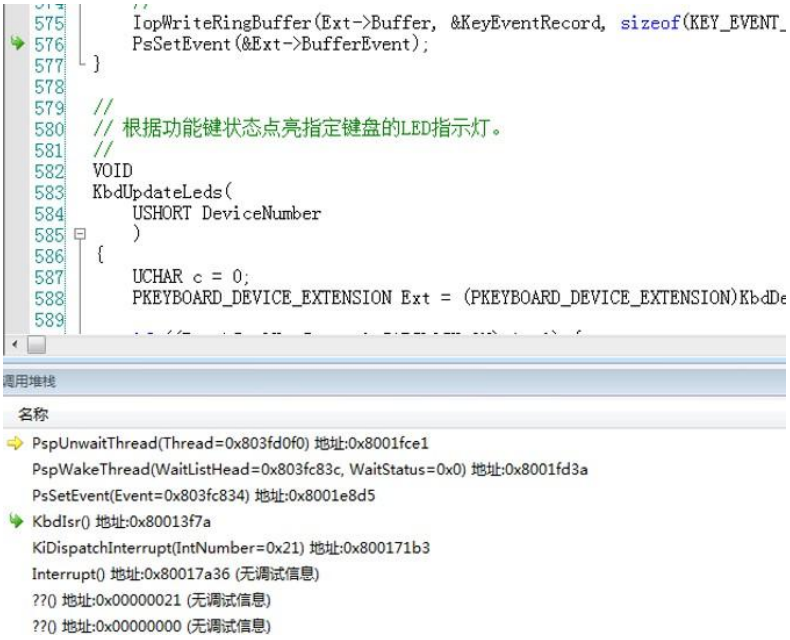
查看*Thread 相关信息：



3. 激活“调用堆栈”窗口。根据当前的调用堆栈，可以看到是由键盘中断服务程序（KdbIsr 函数）进入的。当按下空格键后，就会发生键盘中断，从而触发键盘中断服务程序。在“调用堆栈”窗口中双击 KdbIsr 函数对应的堆栈项，可以看到在该服务程序的最后会将键盘事件放入缓冲区，然后唤醒控制台派遣线程，由控制台派遣线程将键盘事件派遣到活动的控制台。

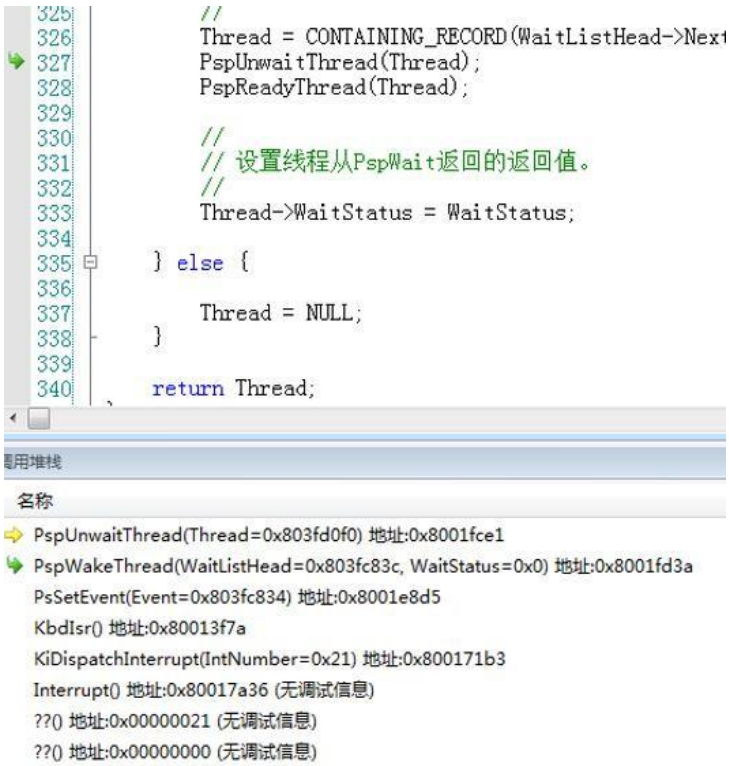
调用堆栈窗口如下：

执行双击操作后：



4. 在“调用堆栈”窗口中双击 PspWakeThread 函数对应的堆栈项。可以看到在此函数中连续调用了 PspUnwaitThread 函数和 PspReadyThread 函数，从而使处于阻塞状态的控制台派遣线程先退出阻塞状态，然后再进入就绪状态。

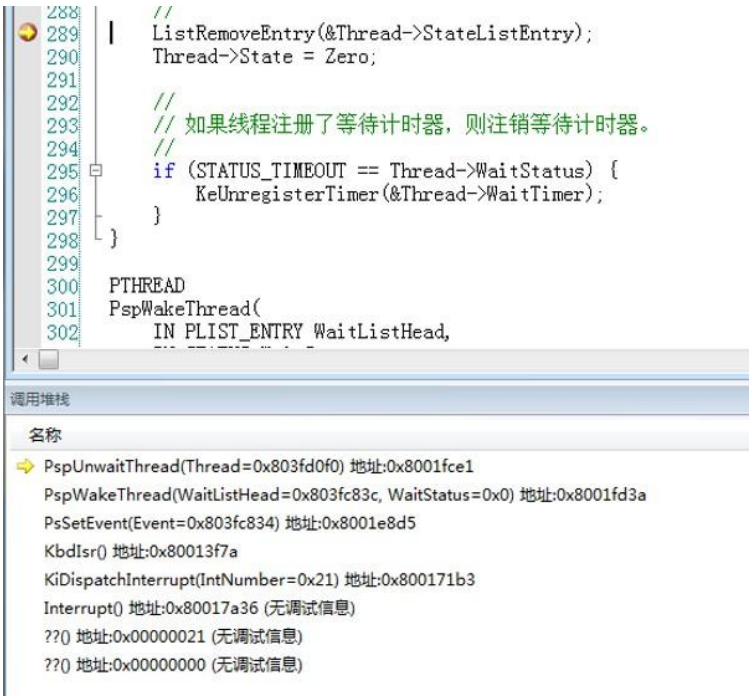
执行双击操作后：



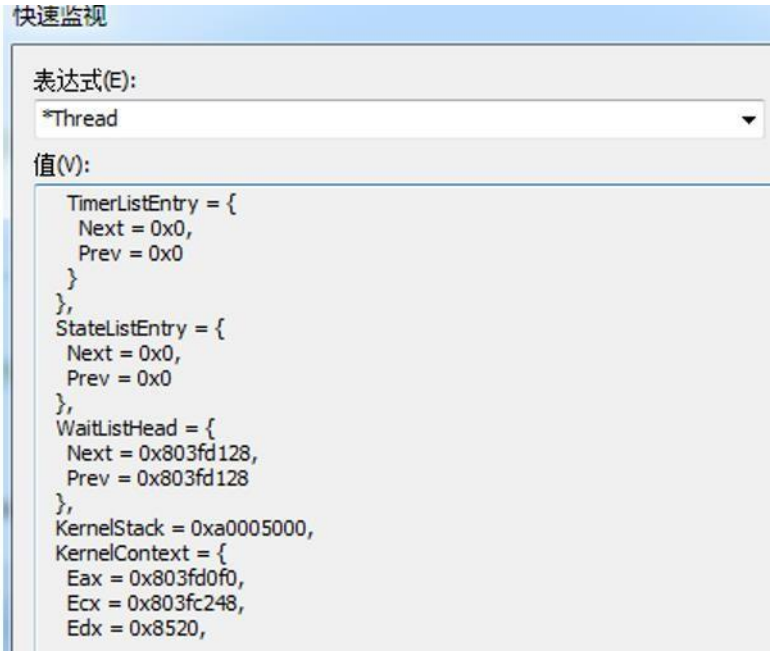
5. 在“调用堆栈”窗口中双击 PspUnwaitThread 函数对应的堆栈项，再来看看此函数是如何改变线程状态的。按 F10 单步调试直到此函数的最后，然后再从快速监视对话框中观察“*Thread”表达式的值。此时 State 域的值为 0（Zero），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都为 0，说

明这个线程已经处于游离状态，并已不在任何线程状态的队列中。仔细阅读 PspUnwaitThread 函数中的源代码，理解这些源代码是如何改变线程状态的。

执行双击操作后：

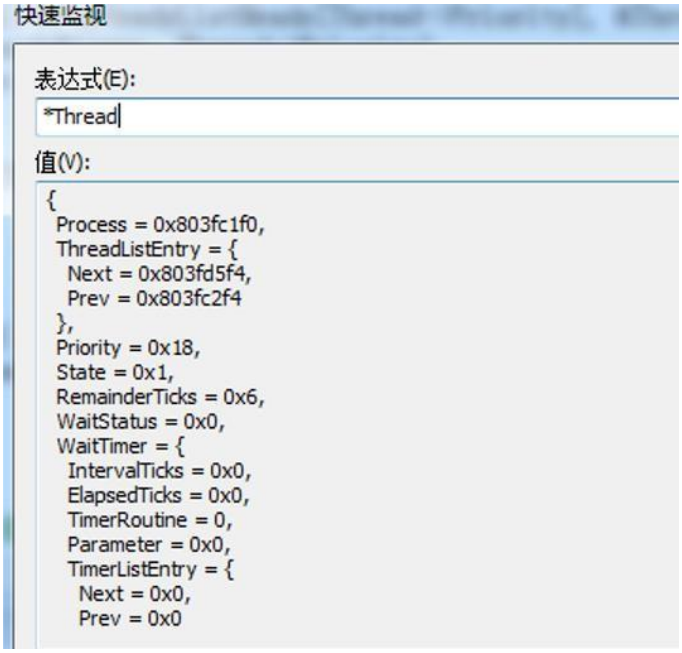


单步调试至末尾后：



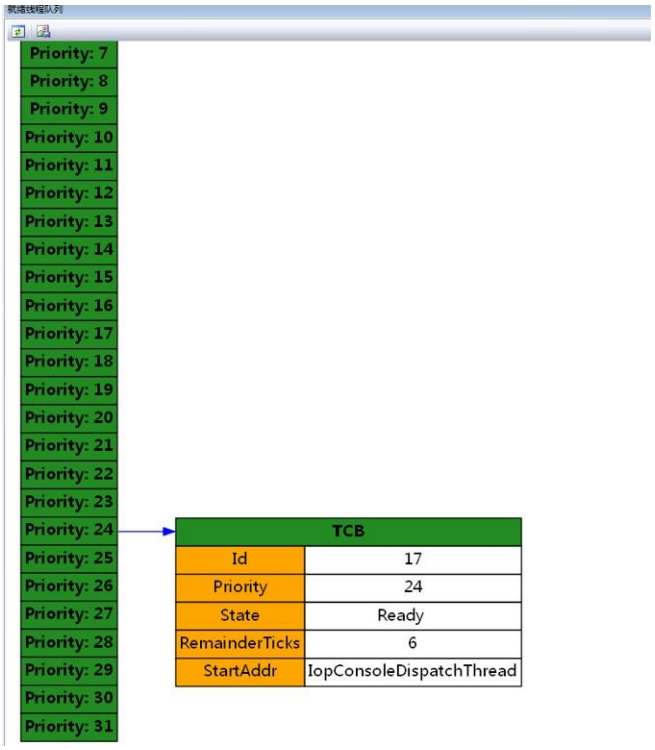
6. 按 F5 继续执行，在 PspReadyThread 函数中的断点处中断。按 F10 单步调试直到此函数的最后，然后再从快速监视对话框中观察“*Thread”表达式的值。此时 State 域的值为 1（Ready），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都不为 0，说明这个线程已经处于就绪状态，并已经被放入优先级为 24 的就绪队列中。仔细阅读 PspReadyThread 函数中的源代码，理解这些源代码是如何改变线程状态的。

“*Thread”表达式的值如下：



7. 选择“调试”菜单“窗口”菜单中的“就绪线程队列”菜单项，打开“就绪线程队列”窗口，在该窗口的工具栏上点击“刷新”按钮，可以看到在优先级为 24 的就绪队列中已插入线程 ID 为 17 的线程控制块。然后，在“线程运行轨迹”窗口，查看 ID 为 17 的线程的运行轨迹，可以看到该线程已由“阻塞” 状态转化为“就绪”状态了。

刷新队列后：



运行轨迹如下：

81	运行							PspSelectNextThread	462
82				就绪				PspReadyThread	134
82	就绪							PspSelectNextThread	408
82				运行				PspSelectNextThread	462
82				阻塞				PspWait	230
82	运行							PspSelectNextThread	462
83				就绪				PspReadyThread	134
83	就绪							PspSelectNextThread	408
83				运行				PspSelectNextThread	462
Tick	TID=2	TID=3	TID=17	TID=18	TID=19	TID=20	TID=21	函数	行号

通过以上的调试，可以将线程由阻塞状态进入就绪状态的步骤总结如下：

（1）将线程从等待队列中移除。

（2）将线程的状态由 Waiting 修改为 Zero。

（3）将线程插入其优先级对应的就绪队列的队尾。

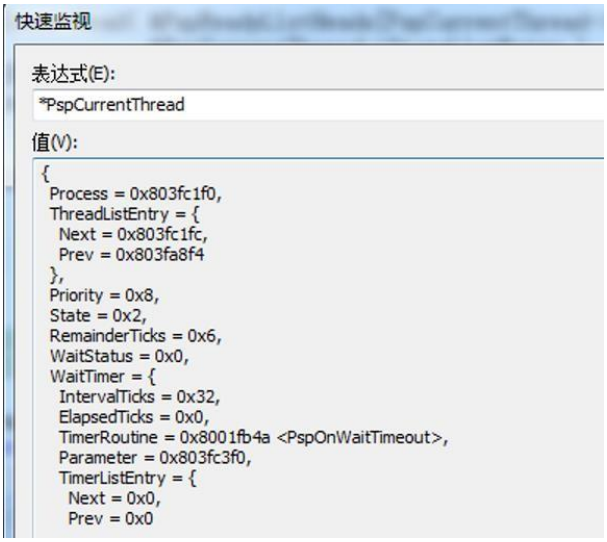
（4）将线程的状态由 Zero 修改为 Ready。

至此，控制台派遣线程已经进入就绪状态了，因为其优先级（24）比当前处于运行状态的 loop 线程的优先级（8）要高，根据 EOS 已实现的基于优先级的抢先式调度算法，loop 线程会由运行状态进入就绪状态，控制台派遣线程会抢占处理器由就绪状态进入运行状态。接下来调试这两个转换过程。

3.2.2 loop 线程由运行状态进入就绪状态

1. 按 F5 继续执行，在 PspSelectNextThread 函数中的断点处中断。在“快速监视”对话框中查看“*PspCurrentThread”表达式的值，观察当前线程的信息。其中 State 域的值为 2（Running），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都为 0，说明这个线程仍然处于运行状态，
于只能有一个处于运行状态的线程，所以这个线程不在任何线程状态的队列中；StartAddr 域的值为 LoopThreadFunction，说明这个线程就是 loop 线程。注意，在本次断点被命中之前，loop 线程就已经被中断执行了，并且其上下文已经保存在线程控制块的 KernelContext 中。

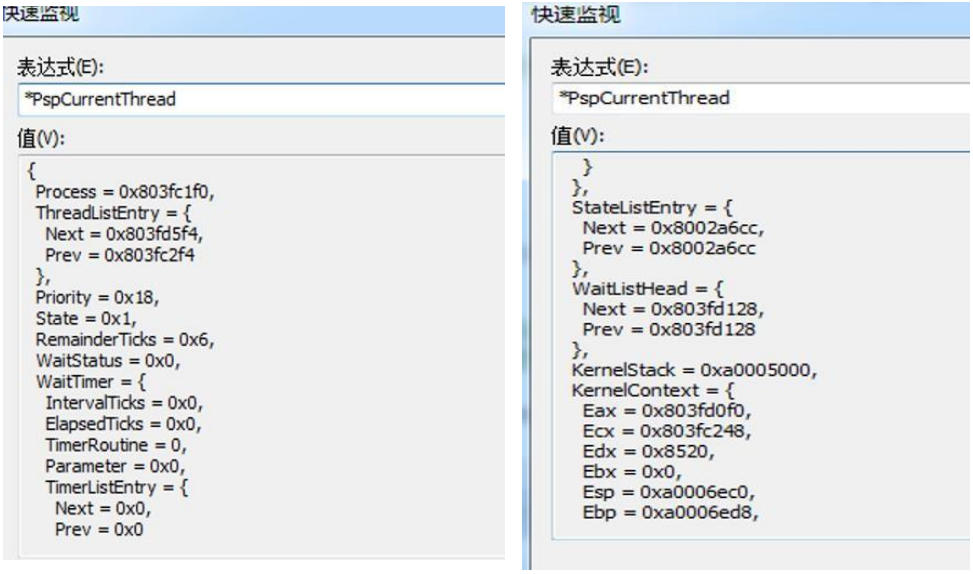
查看*PspCurrentThread 结果如下：



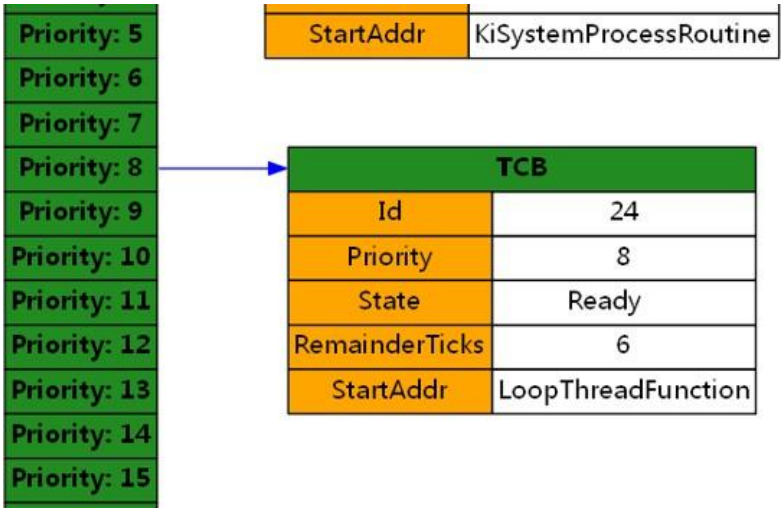
2. 按 F10 单步调试，直到对当前线程的操作完成（也就是花括号中的操作完成）。在“快速监视”对话框中查看“*PspCurrentThread”表达式的值。其中 State 域的值为 1（Ready），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都不为 0，说明 loop 线程已经进入了就绪状态，并已经被放入优先级为

8 的就绪队列中。刷新“就绪线程队列”窗口中，可以看到在优先级为 8 的就绪队列中已插入线程 ID 为 24 的线程控制块。

查看“*PspCurrentThread”的结果如下：



刷新“就绪线程队列”窗口后，结果如下（优先级为 8 的就绪队列中已插入线程 ID 为 24 的线程控制块）：



通过以上的调试，可以将线程由运行状态进入就绪状态的步骤总结如下：

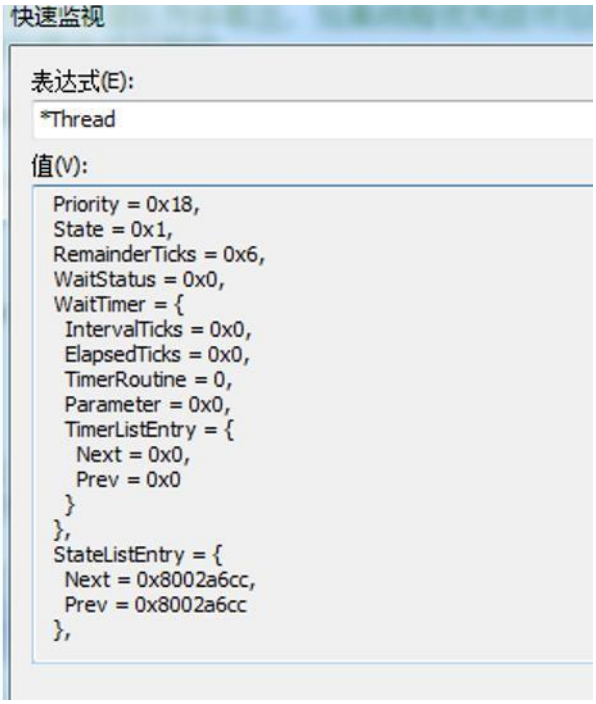
1. 线程中断运行，中断处理函数会将线程中断运行时的上下文保存到线程控制块的 KernelContext 中。当线程恢复运行时，需要将其上下文恢复到处理器中，使其从之前中断的位置继续运行。
2. 如果处于运行状态的线程被更高优先级的线程抢先，就需要将该线程插入其优先级对应的就绪队列的队首。（注意，如果处于运行状态的线程主动让出处理器，例如时间片用完，就需要将线程插入其优先级对应的就绪队列的队尾。）
3. 将线程的状态由 Running 修改为 Ready。

至此，loop 线程已经让出处理器并有运行状态进入就绪状态了，接下来调试控制台派遣线程得到处理器由就绪状态进入运行状态的过程。

3.2.3 控制台派遣线程由就绪状态进入运行状态

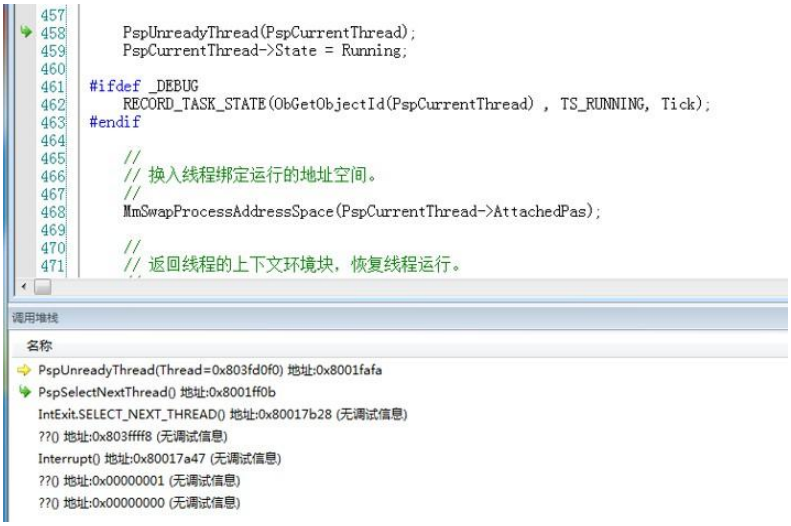
1. 按 F5 继续执行，在 PspUnreadyThread 函数中的断点处中断。在快速监视对话框中查看“*Thread”表达式的值。其中 State 域的值为 1（Ready），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都不为 0，说明这个线程处于就绪状态，并在优先级为 24 的就绪队列中；StartAddr 域的值为 IopConsoleDispatchThread，说明这个线程就是控制台派遣线程。

查看*Thread 结果如下：



2. 关闭快速监视对话框后，在“调用堆栈”窗口中激活 PspSelectNextThread 函数对应的堆栈项，可以看到在 PspSelectNextThread 函数中已经将 PspCurrentThread 全局指针指向了控制台派遣线程，并在调用 PspUnreadyThread 函数后，将当前线程的状态改成了 Running。

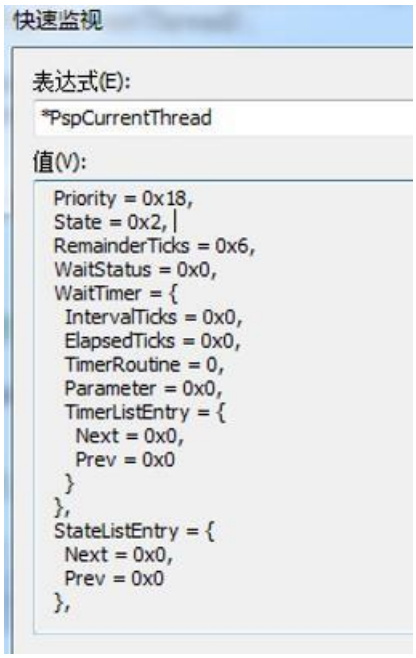
查看“调用堆栈”窗口结果如下：



3. 在“调用堆栈”窗口中激活 PspUnreadyThread 函数对应的堆栈项，然后按 F10 单步调试，直到返回 PspSelectNextThread 函数并将线程状态修改

为 Running。再从快速监视对话框中查看“*PspCurrentThread”表达式的值，观察当前占用处理器的线程的情况。其中 State 域的值为 2（Running），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都为 0，说明控制台派遣线程已经处于运行状态了。接下来，会将该线程的上下文从线程控制块（TCB）复制到处理器的各个寄存器中，处理器就可以从该线程上次停止运行的位置继续运行了。

查看“*PspCurrentThread”结果如下：



通过以上的调试，可以将线程由就绪状态进入运行状态的步骤总结如下：

（1）将线程从其优先级对应的就绪队列中移除。

（2）将线程的状态由 Ready 修改为 Zero。

（3）将线程的状态由 Zero 修改为 Running。

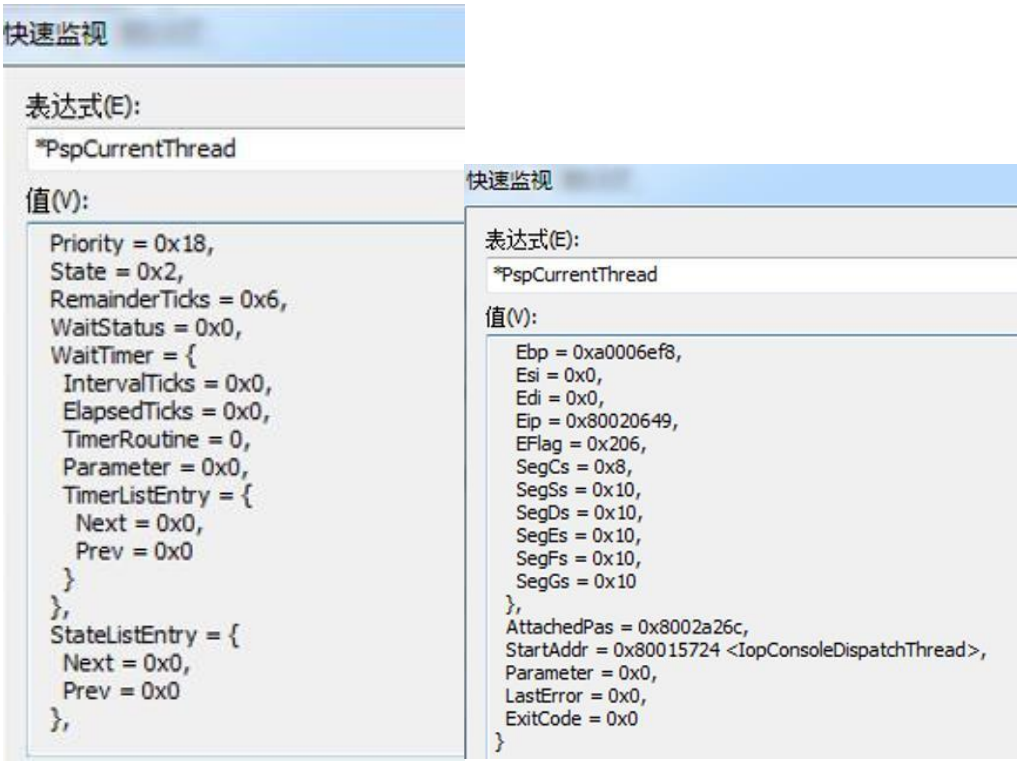
（4）将线程的上下文从线程控制块（TCB）复制到处理器的各个寄存器中，让线程从上次停止运行的位置继续运行。

至此，控制台派遣线程已经开始运行了。因为此时没有比控制台派遣线程优先级更高的线程来抢占处理器，所以控制台派遣线程可以一直运行，直到将此次由于回车键被按下而产生的键盘事件处理完毕，然后控制台派遣线程会由运行状态重新进入阻塞状态，开始等待下一个键盘事件到来。

3.2.4 控制台派遣线程由运行状态进入阻塞状态

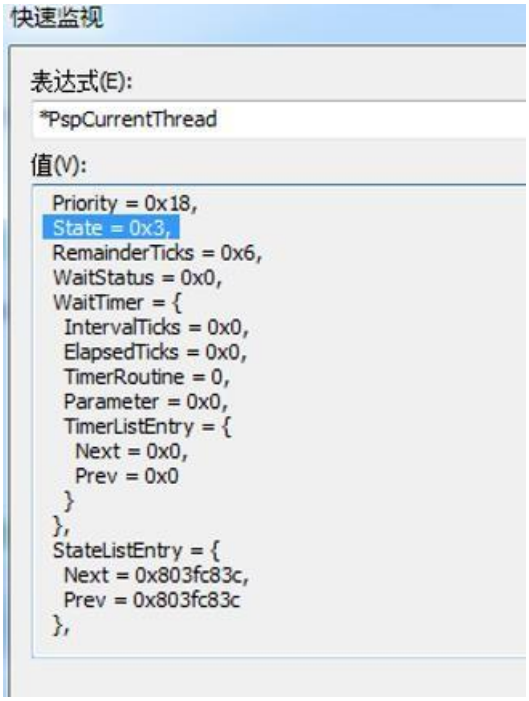
1. 按 F5 继续执行，在 PspWait 函数中的断点处中断。在快速监视对话框中查看“*PspCurrentThread”表达式的值，观察当前占用处理器的线程的情况。其中 State 域的值为 2（Running），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都为 0，说明这个线程仍然处于运行状态；StartAddr 域的值为 IopConsoleDispatchThread，说明这个线程就是控制台派遣线程。

查看“*PspCurrentThread”结果如下：



2. 按 F10 单步调试，直到左侧的黄色箭头指向代码第 255 行。再从快速监视对话框中查看“*PspCurrentThread”表达式的值。其中 State 域的值为 3（Waiting），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都不为 0，说明控制台派遣线程已经处于阻塞状态了，并在某个同步对象的等待队列中。第 255 行代码可以触发线程调度功能，会中断执行当前已经处于阻塞状态的控制台派遣线程，并将处理器上下文保存到该线程的线程控制块中。

查看“*PspCurrentThread”结果如下：



通过以上的调试，可以将线程由运行状态进入阻塞状态的步骤总结如下：

(1) 将线程插入等待队列的队尾。

(2) 将线程的状态由 Running 修改为 Waiting。

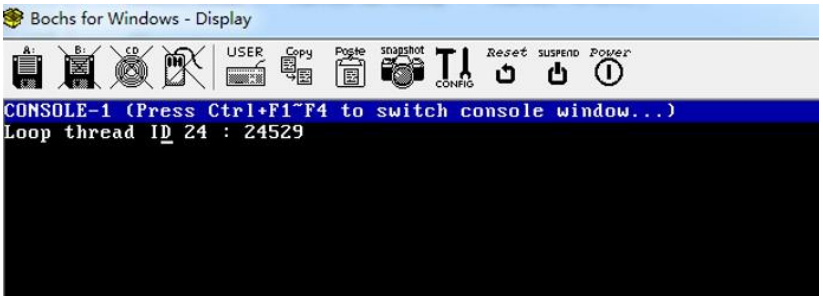
(3) 将线程中断执行，并将处理器上下文保存到该线程的线程控制块中。

至此，控制台派遣线程已经进入阻塞状态了。因为此时 loop 线程是就绪队列中优先级最高的线程，线程调度功能会选择让 loop 线程继续执行。按照下面的步骤调试线程状态转换的过程：

1. 按 F5 继续执行，与本实验 3.2.3 节中的情况相同，只不过这次变为 loop 线程由就绪状态进入运行状态。

2. 再按 F5 继续执行，EOS 不会再被断点中断。激活虚拟机窗口，可以看到 loop 线程又开始不停的执行死循环了。

虚拟机窗口状态如下：



3. 可以按空格键，将以上的调试步骤重复一遍。这次调试的速度可以快一些，仔细体会线程状态转换的过程。

3.3 为线程增加挂起状态

EOS 已经实现了一个 suspend 命令，其命令函数为：ConsoleCmdSuspendThread（在 ke/sysproc.c 文件的第 842 行）。在这个命令中调用了 Suspend 原语（在 ps/psspnd.c 文件第 27 行的 PsSuspendThread 函数中实现）。Suspend 原语可以将一个处于就绪状态的线程挂起。以 loop 线程为例，当使用 suspend 命令将其挂起时，loop 线程的执行计数就会停止增长。按照下面的步骤观察 loop 线程被挂起的情况：

1. 停止之前的调试过程，并删除之前添加的所有断点。

2. 按 F5 启动调试。

3. 待 EOS 启动完毕，在 EOS 控制台中输入命令“loop”后按回车。此时可以看到 loop 线程的执行

计数在不停增长，说明 loop 线程正在执行。记录下 loop 线程的 ID。

4. 按 Ctrl+F2 切换到控制台 2，输入命令“suspend 24”（如果 loop 线程的 ID 是 24）后按回车。

5. 按 Ctrl+1 切换回控制台 1，可以看到由于 loop 线程已经成功被挂起，其执行计数已经停止增长了。此时占用处理器的是 EOS 中的空闲线程。

当前 ID 如下：

切换控制台，并输入 suspend 24 命令：

此时切换回最初的控制台（计时已经停止）：

3.3.1 要求

EOS 已经实现了一个 resume 命令，其命令函数为：ConsoleCmdResumeThread（在 ke/sysproc.c 文件的第 897 行定义）。在这个命令函数中调用了 Resume 原语（在 ps/pssp nd.c 文件第 87 行的 PsResumThread 函数中实现）。Resume 原语可以将一个被 Suspend 原语挂起的线程（处于静止就绪状态）恢复为就绪状态。但是 PsResumThread 函数中的这部分代码（第 119 行）还没有实现，要求读者在这个练习中完成这部分代码。

完善后结果如下：

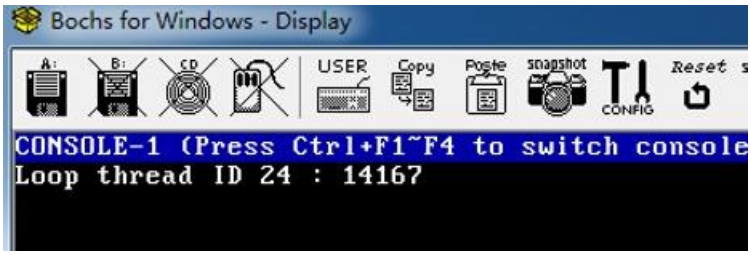
```
112     if (EOS_SUCCESS(Status)) {
113
114         IntState = KeEnableInterrupts(FALSE);   // 关中断
115
116         if (Zero == Thread->State) {
117
118             //
119             // 在此添加代码将线程恢复为就绪状态
120             //
121             ListRemoveEntry(&Thread->StateListEntry);
122             PspReadyThread(Thread);
123             PspThreadSchedule();
124
125             Status = STATUS_SUCCESS;
126
127         } else {
128
129             Status = STATUS_NOT_SUPPORTED;
130
131         }
```

3.3.2 测试方法

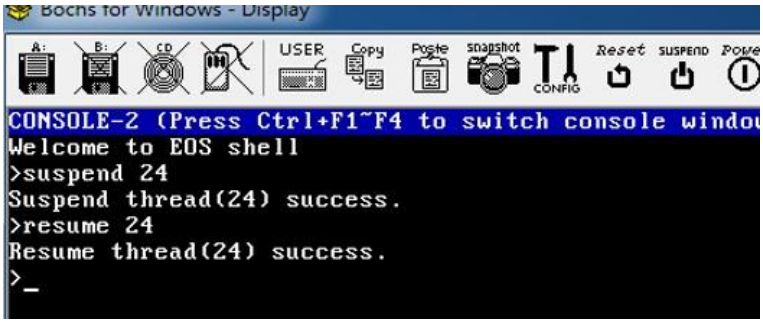
待读者完成 Resume 原语后，可以先使用 suspend 命令挂起 loop 线程，

然后在控制台 2 中输入命令“Resume 24”（如果 loop 线程的 ID 是 24）后按回车。命令执行成功的结果如图 12-5 所示。如果切换回控制台 1 后，发现 loop 线程的执行计数恢复增长就说明 Resume 原语可以正常工作了。当然，也可以在控制台 2 中反复交替使用 suspend 和 resume 命令来进行测试。

执行命令前状态如下：



执行命令后状态如下：



再次开始计时：

4. 实验的思考与问题分析

1. 思考一下，在本实验中，当 loop 线程处于运行状态时，EOS 中还有哪些线程，它们分别处于什么状态。可以使用控制台命令 pt 查看线程的状态或者在“进程线程”窗口中查看线程的状态。

答：还剩下 10 个线程：1 个优先级为 24 的控制台派遣线程处于阻塞状态，1 个优先级为 0 的空闲线程处于就绪状态，8 个优先级为 24 的控制台线程处于阻塞状态。

2. 当 loop 线程在控制台 1 中执行，并且在控制台 2 中执行 suspend 命令时，为什么控制台 1 中的 loop 线程处于就绪状态而不是运行状态？

答：当在控制台 2 中执行 suspend 命令时，控制台 2 会抢占处理器，此时 loop 线程会处于就绪状态。

3. 总结一下在图 5-3 中显示的转换过程，哪些需要使用线程控制块中的上下文（将线程控制块中的上下文恢复到处理器中，或者将处理器的状态复制到

线程控制块的上下文中），哪些不需要使用，并说明原因。

答：状态首先是就绪状态，然后转换成运行状态，再转换为阻塞状态，需要使用进程控制块，因为在转换过程中有线程调进或调出处理机的过程。由新建到就绪，和由阻塞到就绪的过程不占用处理机资源，是不需要使用线程控制块的。

4. 在本实验 3.2 节中总结的所有转换过程都是分步骤进行的，为了确保完整性，显然这些转换过程是不应该被打断的，也就是说这些转换过程都是原语操作（参见本书第 2.6 节）。请读者找出这些转换过程的原语操作（关中断和开中断）是在哪些代码中完成的。（提示，重新调试这些转换过程，可以在调用堆栈窗口列出的各个函数中逐级查找关中断和开中断的代码。）

答：IntState=KeEnableInterrupts(FALSE);//关中断

KeEnableInterrupts(IntState);//开中断

5. 总结和感想体会

⑴ 通过本次实验，我对于线程四种状态的变化更加熟悉，在不断调试状态的过程中，通过每一次转换和尝试，我了解了转换时需要做的相关操作，对于内核代码的理解能力也得到了提升。

⑵ 线程由阻塞状态转换为就绪状态时，先将线程从等待队列中移除，然后将线程的状态由 Waiting 修改为 Zero，将线程插入其优先级对应的就绪队列的队尾，最终将线程的状态由 Zero 修改为 Ready。

⑶ 当线程由运行状态进入就绪状态时，线程会中断运行，中断处理函数会将线程中断运行时的上下文保存到线程控制块的 KernelContext 中。当线程恢复运行时，需要将其上下文恢复到处理器中，使其从之前中断的位置继续运行。

⑷ 当线程由运行状态进入就绪状态时，如果处于运行状态的线程被更高优先级的线程抢先，就需要将该线程插入其优先级对应的就绪队列的队首。但是，如果处于运行状态的线程主动让出处理器，例如时间片用完，就需要将线程插入其优先级对应的就绪队列的队尾。