

合肥工业大学

操作系统实验报告

实验题目	实验 4 线程的状态和转换
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

1. 实验目的和任务要求

- 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。
- 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

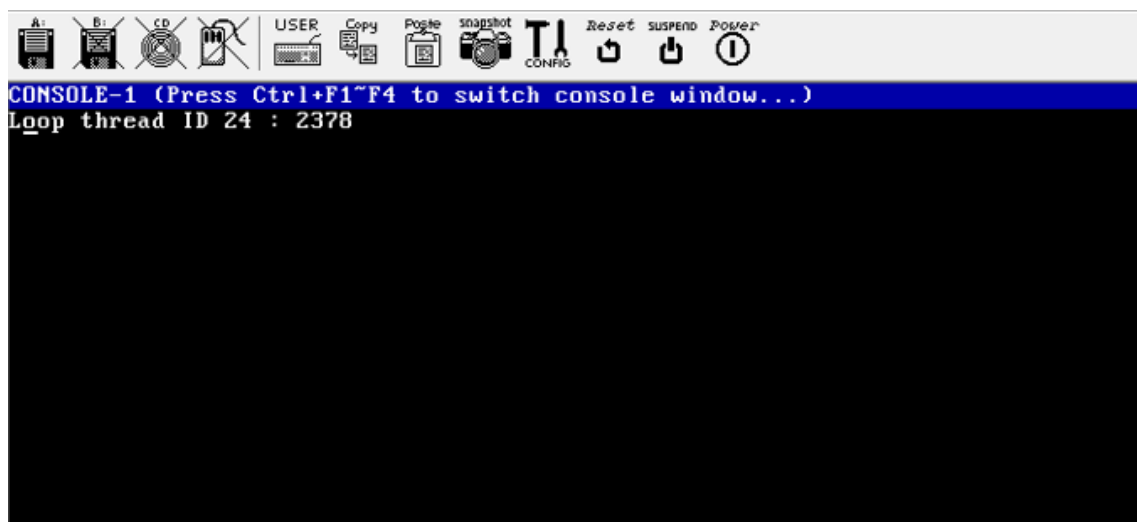
在 EOS 中，线程是处理器调度的基本单位。当一个进程被创建时，系统首先会为该进程分配一些资源（包括内存，内核对象，以及指令和数据等），然后系统会为该进程创建一个默认线程，做为该进程的主线程。进程的主线程开始执行后，就可以认为是进程开始执行了。多数情况下，进程只需要在主线程运行的过程中就可以完成工作。但是，随着单个处理器中内核数量的增加，越来越多的软件要求使用多线程进行并行处理，从而提高硬件资源的利用率，以及软件执行的效率。EOS 支持多线程并发执行，除了在前一节提到的多个进程（每个进程都有一个主线程）并发执行的情况外，还可以在一个进程中创建多个线程。例如，在一个进程的主线程中，可以调用 API 函数 `CreateThread` 来创建一个新线程，这个新线程与主线程共享该进程的所有资源，例如访问进程的地址空间、执行进程的代码、读写进程打开的文件等。而且，EOS 中的线程是属于内核级的，所有线程会一起竞争处理机的使用权，不会区分线程属于哪个进程。

3. 实验内容

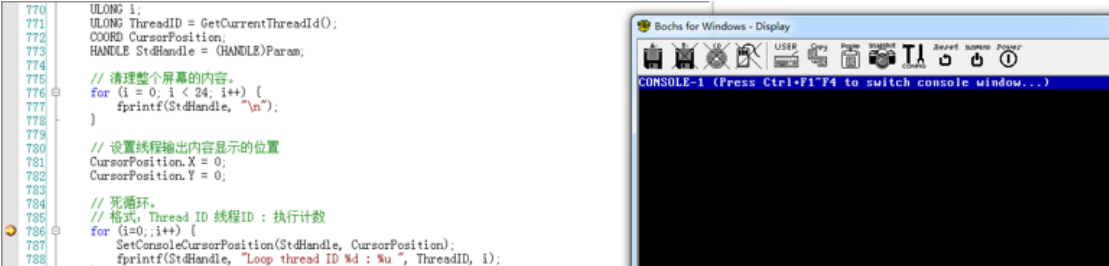
为了完成这个练习，对 EOS 中的下列线程状态转换过程有一个全面的认识：

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。

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 797 行），在此函数中使用 `LoopThreadFunction` 函数（第 755 行）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：



调试线程状态转换的过程：在ke/sysproc.c文件的LoopThreadFunction函数中，开始死循环的代码行（第787行）添加一个断点。



EOS会在断点处中断执行，表明loop线程已经开始死循环了。此时，EOS中所有的系统线程要么处于就绪状态（其优先级一定小于8，例如系统空闲线程），要么就处于阻塞状态（例如控制台派遣线程或控制台线程），所以，只有优先级为8的loop线程能够在处理器上执行。

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	0x80018a5c LoopThreadFunction

按照下面的步骤对断点进行一些调整：

1. 删除所有断点。
2. 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130行）。
 - 在PspUnreadyThread函数体中添加一个断点（第158行）。
 - 在PspWait函数体中添加一个断点（第223行）。
 - 在PspUnwaitThread函数体中添加一个断点（第282行）。
 - 在PspSelectNextThread函数体中添加一个断点（第395行）。
3. 按F5继续执行，然后激活虚拟机窗口。

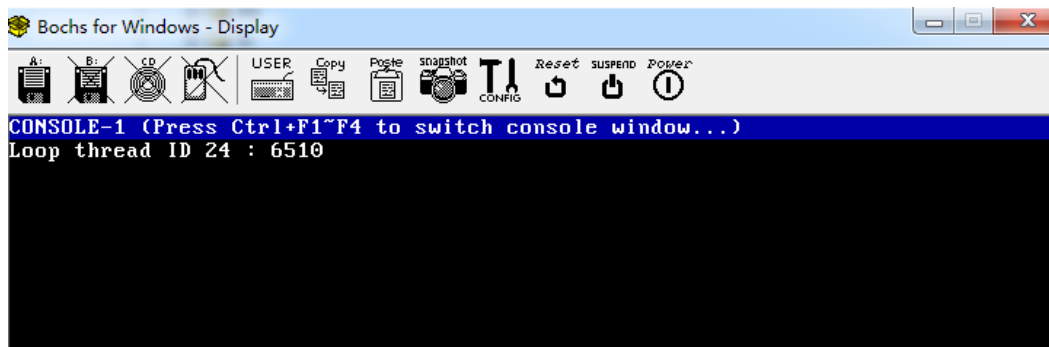
此时在虚拟机窗口中会看到loop线程在不停执行，而之前添加的断点都没有被命中，说明此时还没有任何线程的状态发生改变。具体实验效果如图所示。在开始观察线程状态转换过程之前还有必要做一个说明。在后面的练习中，会在loop线程执行的过程中按一次空格键，这会导致EOS依次执行下面的操作：

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，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。

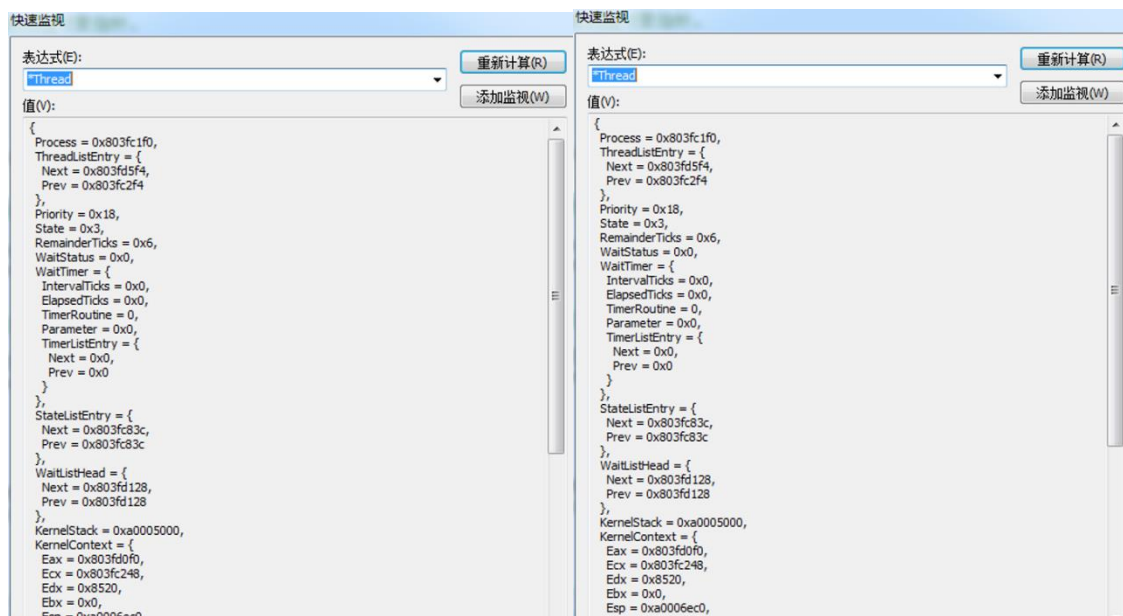
（一）线程由阻塞状态进入就绪状态

按照下面的步骤调试线程状态转换的过程：

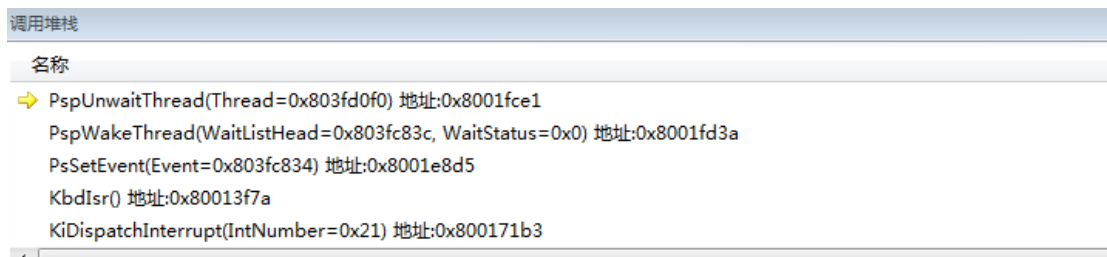
1. 在虚拟机窗口中按下一次空格键。



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IopConsoleDispatchThread，说明这个线程就是控制台派遣线程。监视器如图3-4所示。



关闭快速监视对话框，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）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。在该服务程序的最后中会唤醒控制台派遣线程，将键盘事件派遣到活动的控制台。堆栈执行结果如图3-5所示。



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进入就绪状态。

在“调用堆栈”窗口中双击PspUnwaitThread函数对应的堆栈项，先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。监视器执行结果如图所示。

The image shows a debugger interface with two main windows. The top window displays the source code of the `PspUnwaitThread` function, with line numbers 84 through 103. The code includes comments in Chinese and several function calls like `KeEnableInterrupts` and `PspRoundRobin`. The bottom window is titled "快速监视" (Quick Watch) and shows the value of the `*Thread` expression. The value is a struct containing fields for `Process`, `ThreadListEntry` (with `Next` and `Prev` pointers), `Priority`, `State`, `RemainderTicks`, `WaitStatus`, `WaitTimer`, and `TimerListEntry` (with `Next` and `Prev` pointers). All pointer values are 0x0, and the `State` field is 0x0.

```
84 |         Timer->ElapsedTicks = 0;
85 |         Timer->TimerRoutine(Timer->Parameter);
86 |     }
87 | }
88 | KiNextTimerListEntry = NULL;
89 |
90 | //
91 | // 时间片轮转调度。
92 | //
93 | IntState = KeEnableInterrupts(FALSE);
94 | PspRoundRobin();
95 | KeEnableInterrupts(IntState);
96 | }
97 |
98 | VOID
99 | KeInitializeTimer(
100 |     IN PKTIMER Timer,
101 |     IN ULONG Milliseconds,
102 |     IN PKTIMER_ROUTINE TimerRoutine,
103 |     IN ULONG_PTR Parameter
```

调用堆栈

名称
→ PspUnwaitThread(Thread=0x803fc3f0) 地址:0x8001fce1
PspOnWaitTimeout(Param=0x803fc3f0) 地址:0x8001fb5b
→ KiIsrTimer() 地址:0x80017bdc
KiDispatchInterrupt(IntNumber=0x20) 地址:0x8001719a
Interrupt() 地址:0x80017a36 (无调试信息)

快速监视

表达式(E):
*Thread

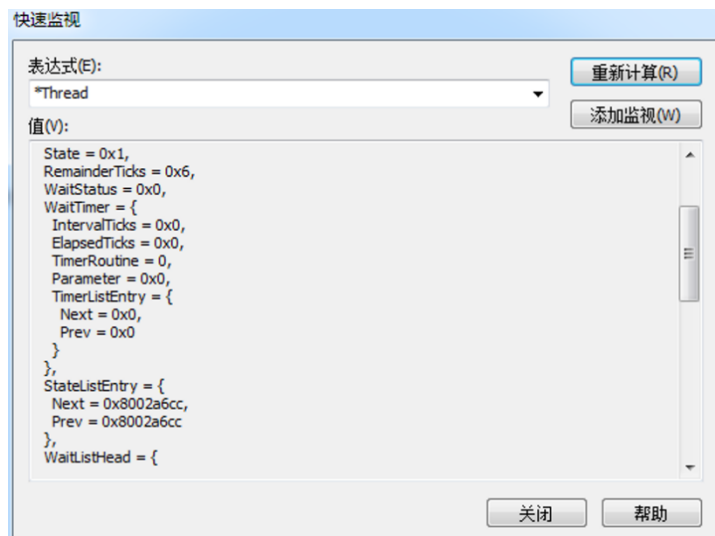
值(V):

```
{
  Process = 0x803fc1f0,
  ThreadListEntry = {
    Next = 0x803fd5f4,
    Prev = 0x803fc2f4
  },
  Priority = 0x18,
  State = 0x0,
  RemainderTicks = 0x6,
  WaitStatus = 0x0,
  WaitTimer = {
    IntervalTicks = 0x0,
    ElapsedTicks = 0x0,
    TimerRoutine = 0,
    Parameter = 0x0,
    TimerListEntry = {
      Next = 0x0,
      Prev = 0x0
    }
  }
}
```

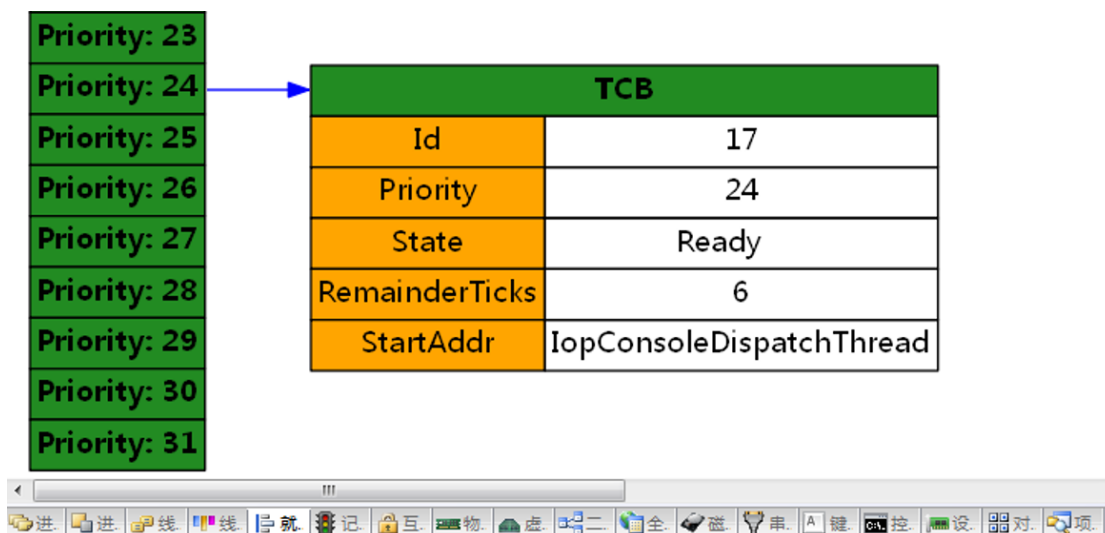
重新计算(R)
添加监视(W)
关闭 帮助

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。



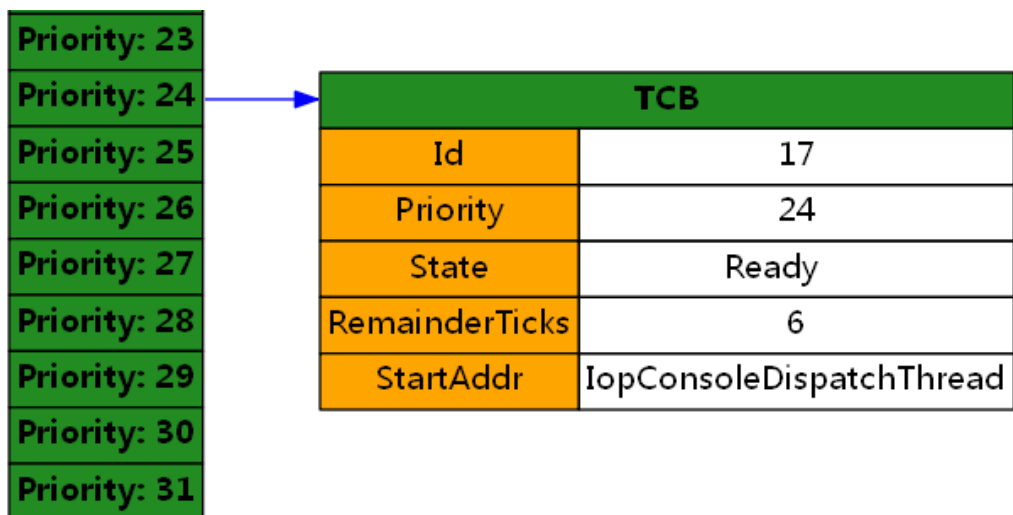
选择“调试”菜单“窗口”菜单中的“就绪线程队列”菜单项，打开“就绪线程队列”窗口，在该窗口的工具栏上点击“刷新”按钮，可以看到在优先级为 24 的就绪队列中已插入线程 ID 为 17 的线程控制块，如图 12-3 所示。然后，在“线程运行轨迹”窗口，查看 ID 为 17 的线程的运行轨迹，可以看到该线程已由“阻塞”状态转化为“就绪”状态了。



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。

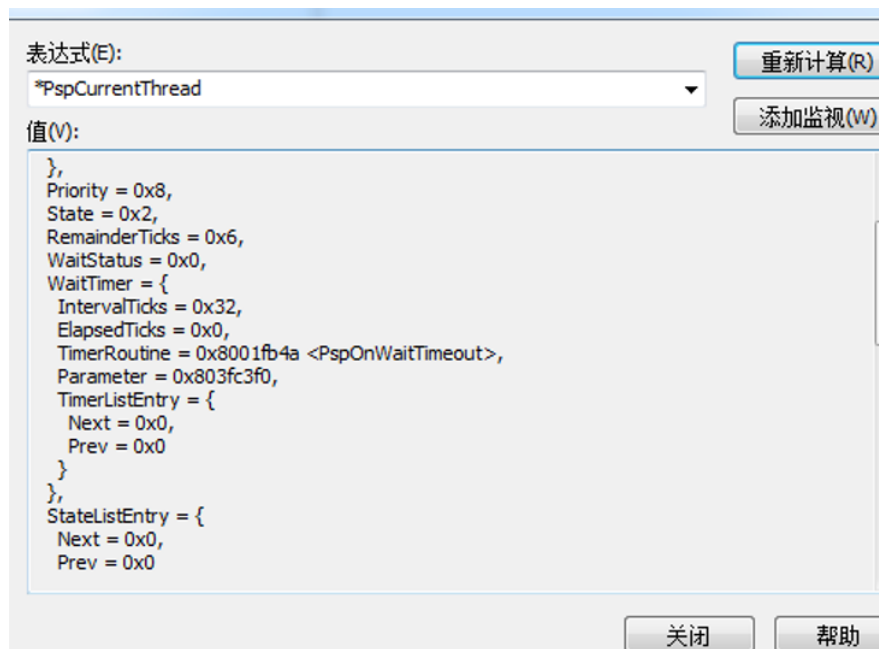
至此，控制台派遣线程已经进入就绪状态了，因为其优先级（24）比当前处于运行状态的loop线程的优先级（8）要高，根据EOS已实现的基于优先级的抢先式调度算法，loop线程会进入就绪状态，控制台派遣线程会抢占处理器从而进入运行状态。接下来调试这两个转换过程。



线程由运行状态进入就绪状态

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，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，观察当前占用处理器的线程的情况。其中State域的值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，由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中。



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）。再从快速监视对话框中查看“*PspCurrentThread”表达式的值。其中State域值为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。具体执行效果如图3-7所示，仔细阅读PspSelectNextThread函数这个

花括号中的源代码，理解这些源代码是如何改变线程状态的，并与 PspReadyThread 函数中的源代码进行比较，说明这两段源代码的异同，体会为什么在这里不能直接调用 PspReadyThread 函数。

快速监视

表达式(E):

*PspCurrentThread

重新计算(R)

添加监视(W)

值(V):

Priority = 0x8,
State = 0x1,
RemainderTicks = 0x6,
WaitStatus = 0x0,
WaitTimer = {
IntervalTicks = 0x32,
ElapsedTicks = 0x0,
TimerRoutine = 0x8001fb4a <PspOnWaitTimeout>,
Parameter = 0x803fc3f0,
TimerListEntry = {
Next = 0x0,
Prev = 0x0
}
},
StateListEntry = {
Next = 0x8002a64c,
Prev = 0x8002a64c
},
},

关闭 帮助

就绪线程队列

Priority: 3

Priority: 4

Priority: 5

Priority: 6

Priority: 7

Priority: 8

Priority: 9

Priority: 10

Priority: 11

Priority: 12

Priority: 13

Priority: 14

Priority: 15

Priority: 16

Priority: 17

Priority: 18

State

Ready

RemainderTicks

6

StartAddr

KiSystemProcessRoutine

TCB

Id

24

Priority

8

State

Ready

RemainderTicks

6

StartAddr

LoopThreadFunction

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，将线程中断运行时的上下文保存到线程控制块中。
- (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。

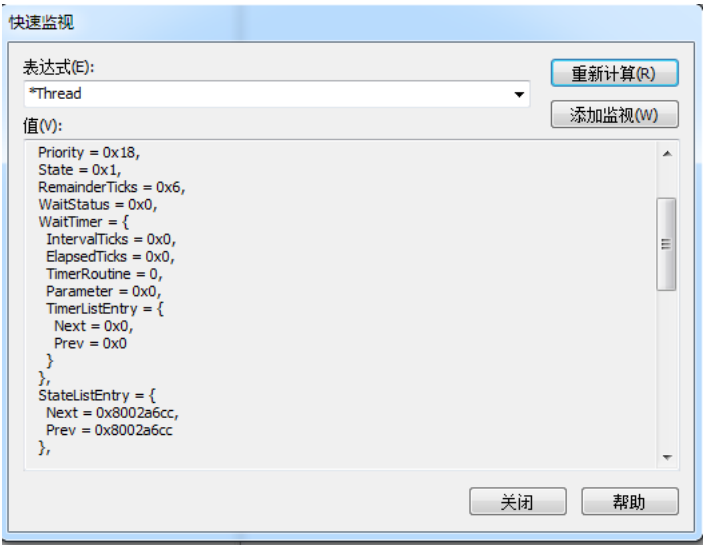
至此，loop线程已经进入就绪状态了，接下来调试控制台派遣线程会得到处理

器进入运行状态的过程。

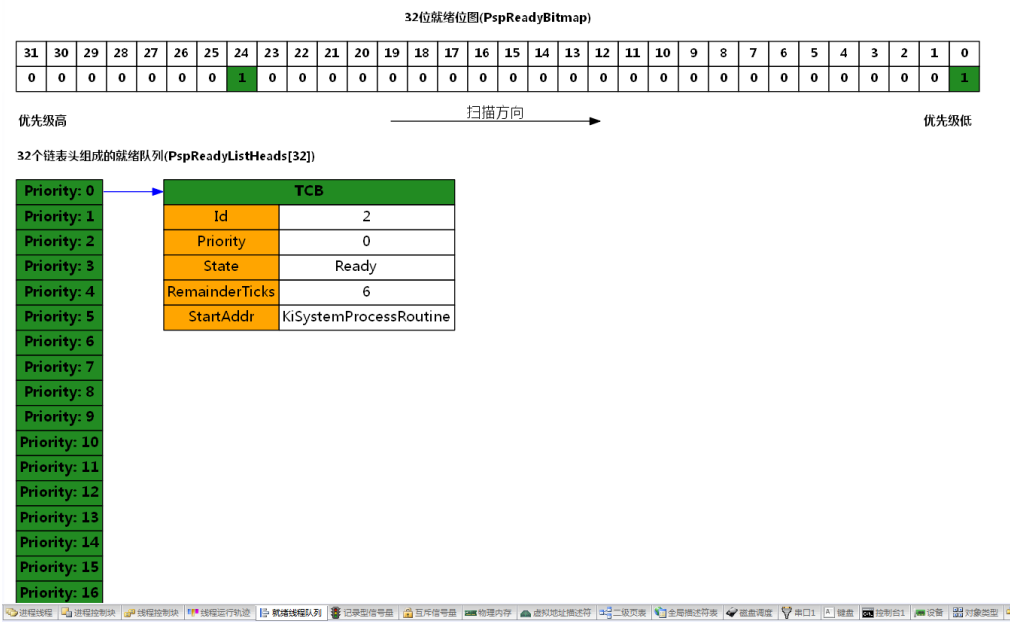
线程由就绪状态进入运行状态

按照下面的步骤调试线程状态转换的过程：

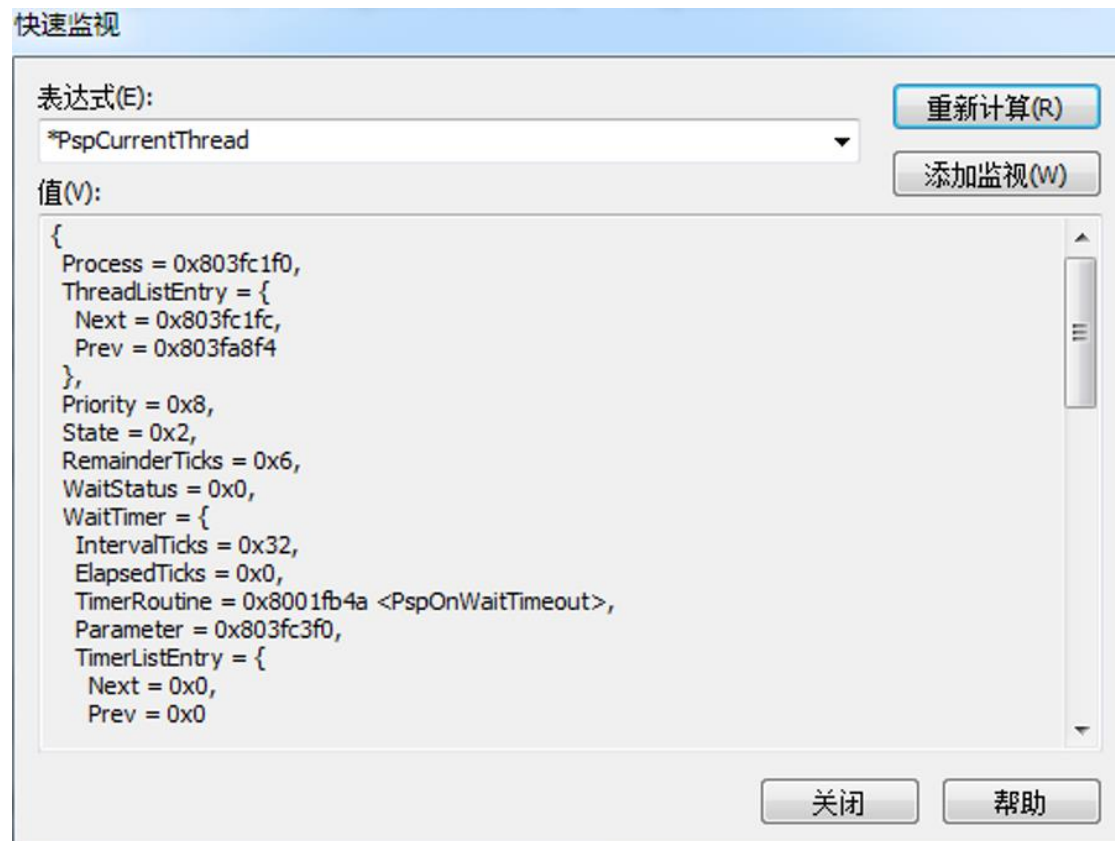
1. 按F5继续执行，在PspUnreadyThread函数中的断点处中断。在快速监视对话框中查看“*Thread”表达式的值。其中State域的值1（Ready），双向链表项StateListEntry的Next和Prev指针的值都不为0，说明这个线程处于就绪状态，并在优先级为24的就绪队列中；StartAddr域的值为IopConsoleDispatchThread，说明这个线程就是控制台派遣线程。仔细阅读PspUnreadyThread函数中的源代码，理解这些源代码是如何改变线程状态的。



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。具体执行效果如图所示。



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。监视器具体执行效果如图3-9所示。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。



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。

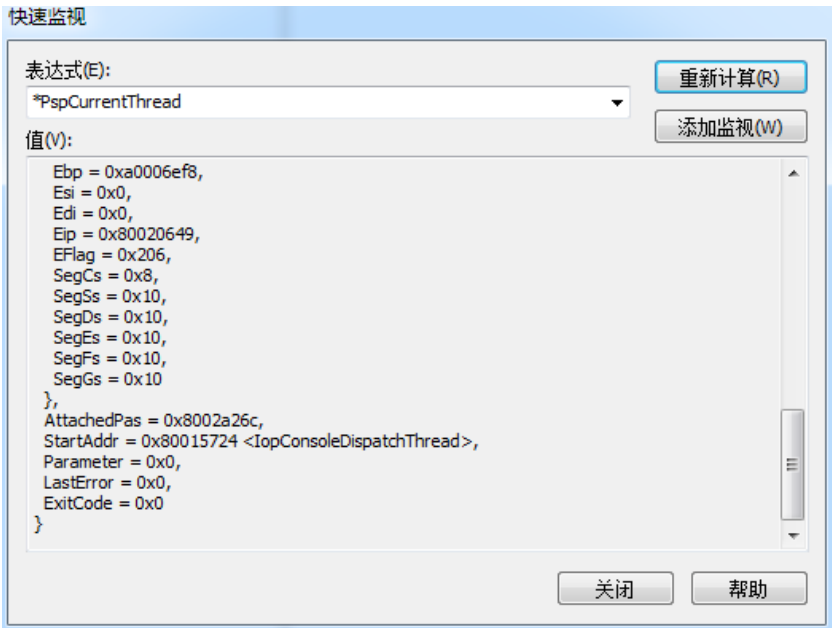
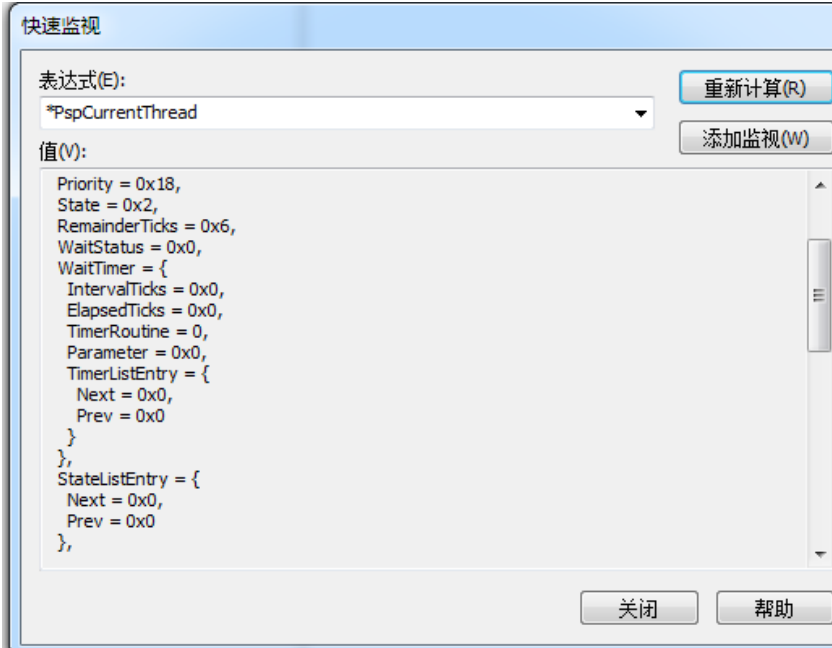
至此，控制台派遣线程已经开始运行了。因为此时没有比控制台派遣线程优先级更高的线程来抢占处理器，所以控制台派遣线程可以一直运行，直到将此次由于空格键被按下而产生的键盘事件处理完毕，然后控制台派遣线程会由运行状态重新进入阻塞状态，开始等待下一个键盘事件到来。

（四）线程由运行状态进入阻塞状态

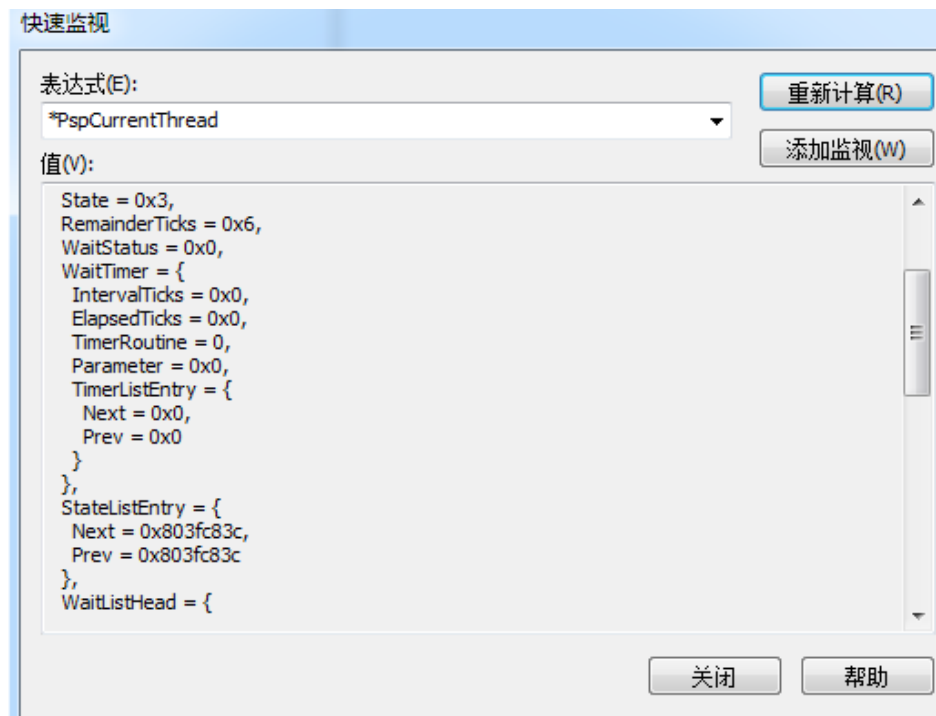
按照下面的步骤调试线程状态转换的过程：

1. 按F5继续执行，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

IopConsoleDispatchThread，说明这个线程就是控制台派遣线程。监视器执行效果如图所示。



2. 按 F10 单步调试，直到左侧的黄色箭头指向代码第 248 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。监视器具体执行效果如图 3-11。第 248 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。



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。

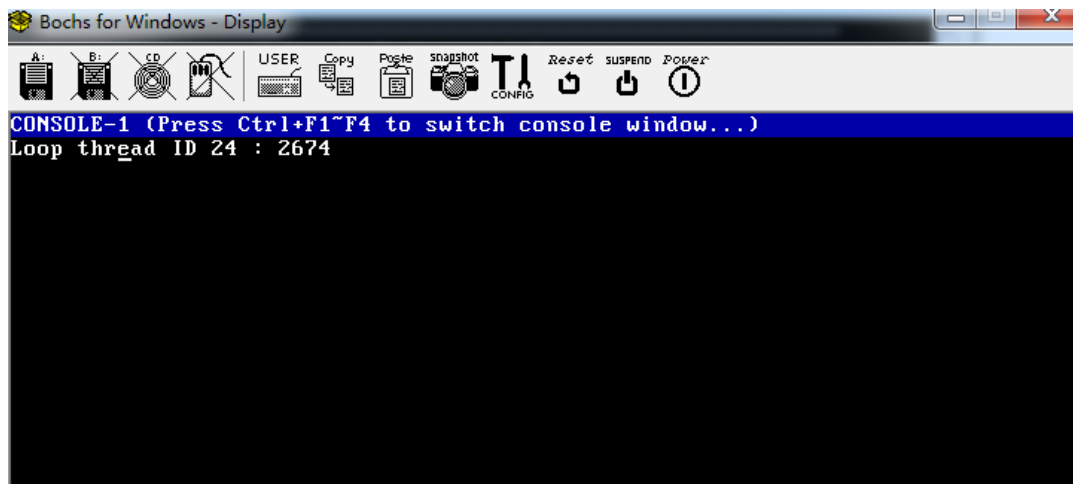
至此，控制台派遣线程已经进入阻塞状态了。因为此时loop线程是就绪队列中优先级最高的线程，线程调度功能会选择让loop线程继续执行。按照下面的步骤调试线程状态转换的过程：

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。
3. 可以再次按空格键，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。

为线程增加挂起状态

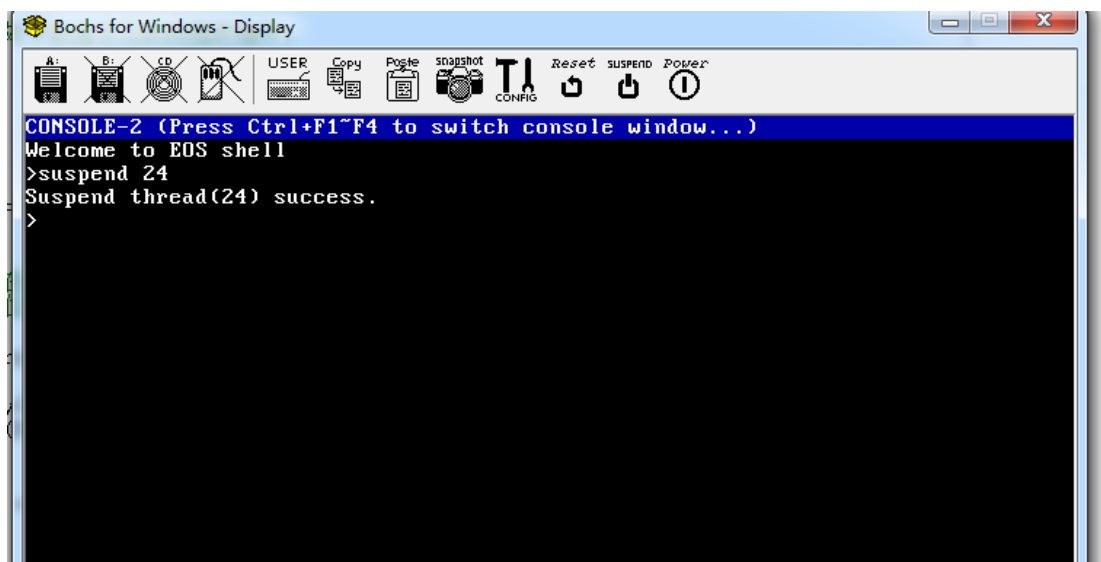
EOS已经实现了一个suspend命令，其命令函数为ConsoleCmdSuspendThread（在ke/sysproc.c文件的第843行）。在这个命令中调用了Suspend原语（在ps/psspend.c文件第27行的PsSuspendThread函数中实现）。Suspend原语可以将一个处于就绪状态的线程挂起。以loop线程为例，当使用suspend命令将其挂起时，loop线程的执行计数就会停止增长。按照下面的步骤观察loop线程被挂起的情况：

1. 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 31”（如果loop线程的ID是31）后按回车。命令执行成功的结果如图3-12所示。

5. 按Ctrl+F1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。



(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898行）。在这个命令中调用了Resume原语（在ps/psspend.c文件第87行的PsResumeThread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PsResumeThread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。

代码：/*

提供该示例代码是为了阐释一个概念，或者进行一个测试，并不代表着最安全的编码实践，因此不应在应用程序或网站中使用该示例代码。

*/

STATUS

```
PsResumeThread(  
    IN HANDLE hThread  
)
```

功能描述：

恢复指定的线程。

参数：

hThread - 需要被恢复的线程的句柄。

返回值：

如果成功则返回 STATUS_SUCCESS。

--*/

```
{  
    STATUS Status;  
    BOOL IntState;  
    PTHREAD Thread;  
    //  
    // 根据线程句柄获得线程对象的指针  
    //  
    Status = ObRefObjectByHandle(hThread, PspThreadType,  
(PVOID*)&Thread);  
    if (EOS_SUCCESS(Status)) {  
        IntState = KeEnableInterrupts(FALSE); // 关中断  
        if (Zero == Thread->State) {  
            //  
            // 将线程从挂起线程队列中移除。  
            //  
            ListRemoveEntry(&Thread->StateListEntry);  
  
            //  
            // 将挂起的线程恢复为就绪状态。  
            // 线程由静止就绪状态 (Static Ready) 进入活动就绪状态  
            (Active Ready)。  
            //  
            PspReadyThread(Thread);  
  
            //  
            // 执行线程调度，让刚刚恢复的线程有机会执行（如果真的能够  
            // 调度到它的话）。  
            //  
            PspThreadSchedule();  
        }  
    }  
}
```



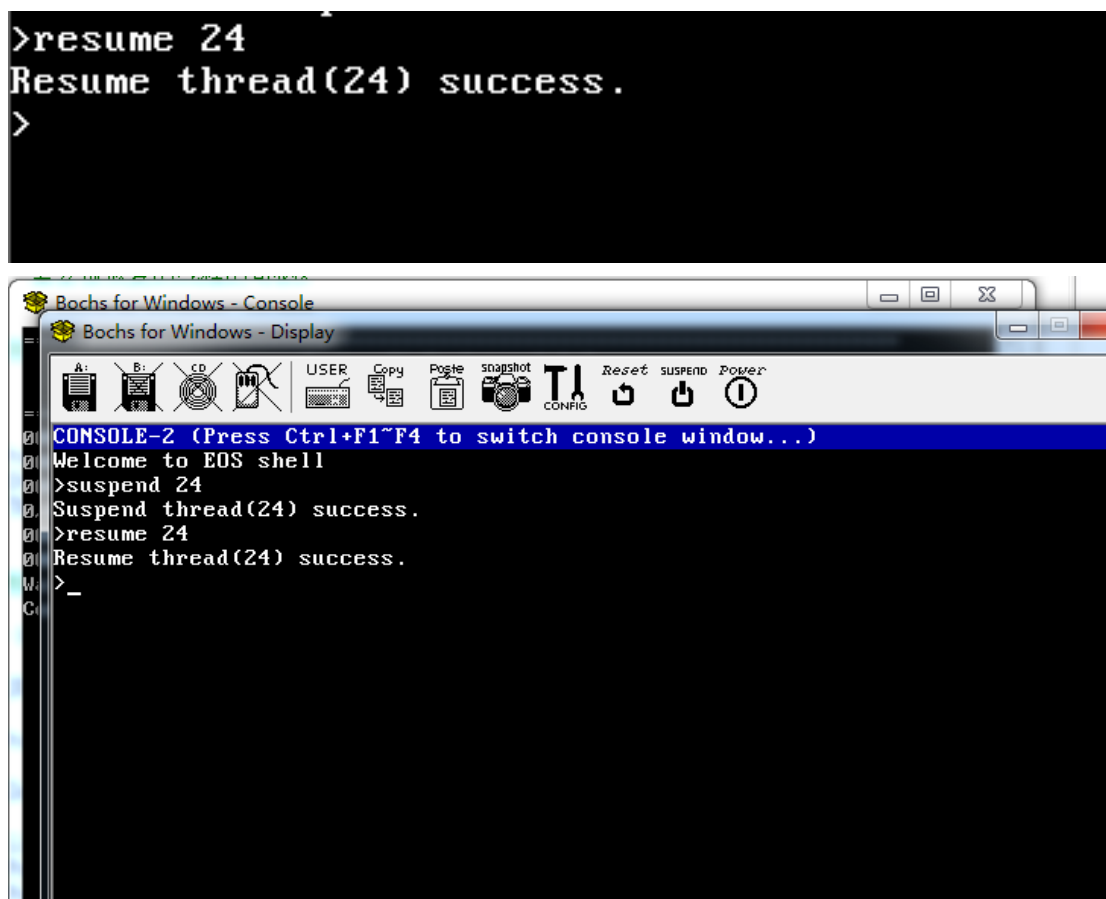
```

        Status = STATUS_SUCCESS;
    } else {
        Status = STATUS_NOT_SUPPORTED;
    }
    KeEnableInterrupts(IntState);    // 开中断
    ObDerefObject(Thread);
}
return Status;
}

```

(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 31”（如果loop线程的ID是31）后按回车。命令执行成功的结果如图3-13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。

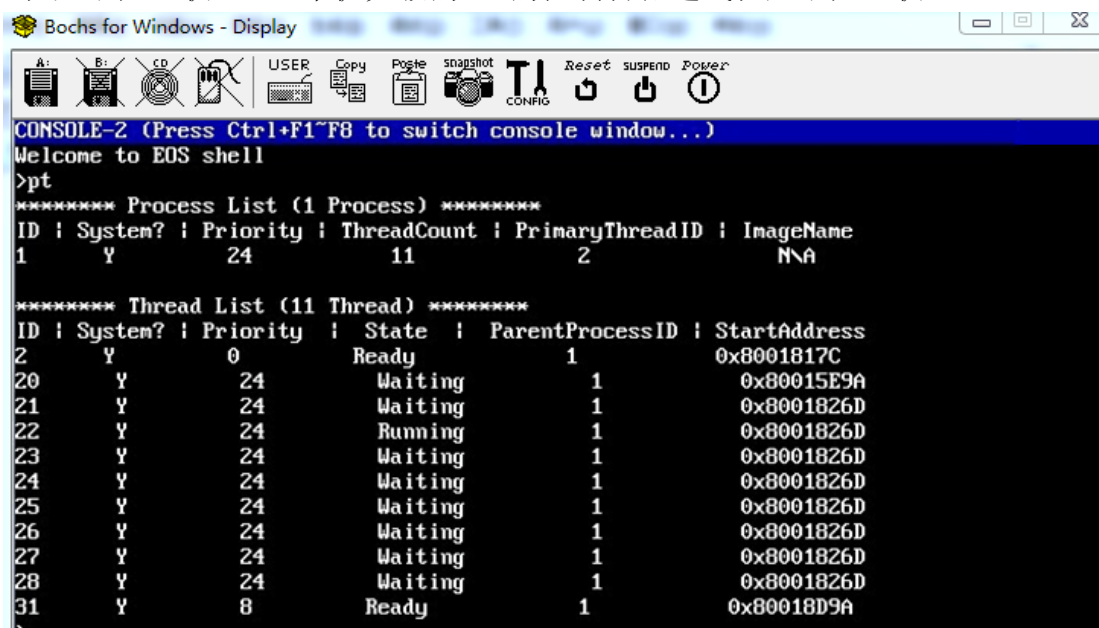


4. 实验的思考与问题分析

(1) 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：

有一个优先级为 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，1 个优先级的 24 的控制台派遣线程处于阻塞状态。



```
Bochs for Windows - Display
A: B: CD USER Copy Paste Snapshot CONFIG Reset Suspend Power
CONSOLE-2 (Press Ctrl+F1~F8 to switch console window...)
Welcome to EOS shell
>pt
***** Process List (1 Process) *****
ID | System? | Priority | ThreadCount | PrimaryThreadID | ImageName
1   | Y        | 24      | 11          | 2               | N/A

***** Thread List (11 Thread) *****
ID | System? | Priority | State | ParentProcessID | StartAddress
2   | Y        | 0       | Ready | 1               | 0x8001817C
20  | Y        | 24      | Waiting | 1               | 0x80015E9A
21  | Y        | 24      | Waiting | 1               | 0x8001826D
22  | Y        | 24      | Running | 1               | 0x8001826D
23  | Y        | 24      | Waiting | 1               | 0x8001826D
24  | Y        | 24      | Waiting | 1               | 0x8001826D
25  | Y        | 24      | Waiting | 1               | 0x8001826D
26  | Y        | 24      | Waiting | 1               | 0x8001826D
27  | Y        | 24      | Waiting | 1               | 0x8001826D
28  | Y        | 24      | Waiting | 1               | 0x8001826D
31  | Y        | 8       | Ready | 1               | 0x80018D9A
```

(2) 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：

在控制台 2 中执行 suspend 命令时，优先级为 24 的控制台 2 线程抢占处理器，即控制台 2 线程处于运行状态，因此此时 loop 处于就绪状态。

(3) 实验 3.2 节中只调试了图 5-3 中显示的最主要的四种转换过程，对于线程由新建进入就绪状态，或者由任意状态进入结束状态的转换过程还没有调试，请读者找到这两个转换过程执行的源代码，自己练习调试。

答；

(4) 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：

就绪→运行，运行→就绪，运行→阻塞都要使用 TCB 因为这些过程有线程调进或调出处理机的过程，新建→就绪，阻塞→就绪不需要使用 TCB 上下文，因为没有占用处理机资源。

(5) 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

(6) EOS 源代码, 对已经实现的线程的挂起状态进行改进。首先, 不再使用 Zero 状态表示静止就绪状态, 在枚举类型 THREAD_STATE 中定义一个新的项用来表示静止就绪状态, 并对 PsSuspendThread 函数进行适当修改。其次, 处于阻塞状态和运行状态的线程也应该可以被挂起并被恢复, 读者可以参考第 5.2.4 节中的内容以及图 5-5 来完成此项改进。注意, 处于运行状态的线程可以将自己挂起。要设计一些方案对所修改的代码进行全面的测试, 保证所做的改进是正确的。如果完成了以上改进, 请思考一下控制台命令 pt 需要进行哪些相应的修改?

答:

STATUS

```
PsResumeThread(  
    IN HANDLE hThread  
)  
{  
    STATUS Status;  
    BOOL IntState;  
    PTHREAD Thread; .  
    Status = ObRefObjectByHandle(hThread, PspThreadType,  
(PVOID*)&Thread); .  
    if (EOS_SUCCESS(Status)) {  
        IntState = KeEnableInterrupts(FALSE); // 关中断  
        if (Zero == Thread->State) {  
            ListRemoveEntry(&Thread->StateListEntry);  
            PspReadyThread(Thread);  
            PspThreadSchedule();  
            Status = STATUS_SUCCESS; .  
        } else {  
            Status = STATUS_NOT_SUPPORTED;  
        }  
        KeEnableInterrupts(IntState); //开中断  
        ObDerefObject(Thread);  
    }  
    return Status;  
}
```

4. 总结和感想体会

这个实验让我深刻的认识到一个进程从创建而产生至撤销而消亡的整个生命周期, 可以用一组状态加以刻画, 根据三态模型, 进程的生命周期可分为如下三种进程状态:

1. 运行态(running): 占有处理器正在运行
2. 就绪态(ready): 具备运行条件, 等待系统分配处理器以便运行
3. 等待态(blocked): 不具备运行条件, 正在等待某个事件的完成

到目前为止，我们或多或少总是假设所有的进程都在内存中。事实上，可能出现这样一些情况，例如由于进程的不断创建，系统的资源已经不能满足进程运行的要求，这个时候就必须把某些进程挂起（suspend），对换到磁盘镜像区中，暂时不参与进程调度，起到平滑系统操作负荷的目的。

参考文献

- [1]北京英真时代科技有限公司[DB/CD].<http://www.engintime.com>.
- [2]汤子瀛，哲凤屏，汤小丹。计算机操作系统。西安：西安电子科技大学出版社，1996.