**北京科技大学实验报告**

学院： 计通学院 专业： 计算机科学与技术 班级： 计172

姓名： 金玉卿 学号： 41724051 实验日期： 2019 年 12 月 3 日

**实验名称：操作系统实验4 线程调度（4分）**

**实验目的：**以一个教学型操作系统EOS为例，深入理解线程（进程）调度的执行时机、过程以及调度程序实现的基本方法；能对核心源代码进行分析和修改；训练分析问题、解决问题以及自主学习能力，逐步达到能独立对小型操作系统的功能进行分析、设计和实现。

**实验环境：**EOS操作系统及其实验环境。

**实验内容：**

跟踪EOS的线程调度程序，分析EOS基于优先级的抢占式调度的核心源代码，阐述其实现方法；修改EOS的调度程序，添加时间片轮转调度功能。

**实验步骤：**

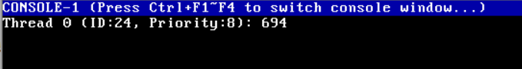
**1）EOS基于优先级的抢占式调度工作过程的跟踪与源代码分析**

（分析EOS基于优先级的抢占式调度的核心源代码，阐述其实现方法，包括相关数据结构和算法等；简要说明在本部分实验过程中完成的主要工作，包括对EOS调度程序的跟踪等）

* 1. **基于优先级抢占式调度实现方法**

当这种线程调度方式运行时，如有比正在执行的线程优先级高的线程处于“就绪”状态，这种调度方式会停止正在执行的低优先级的线程，然后将处理器分配给高优先级的线程，使之执行，而低优先级的线程会进入“就绪”状态，直到再也没有比它优先级高的“就绪”线程时，它才能重新获得处理器。同时要注意调度的时机及终端结束后返回的线程判定。

EOS中为线程定义了0到31的32个优先级，其中 0 优先级最低，31 优先级最高。线程控制块结构体 THREAD（在文件 ps/psp.h 第68行定义）中的 Priority 域就是用来记录线程优先级的。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，控制台对于新建线程来说是一种临界资源，由于所有10个新建线程优先级都是8，所以至始至终只有第0个线程在运行，而其它具有相同优先级的线程只能处于“就绪”状态。



线程调度执行的时机：在ke/i386/int.asm 文件中，定义了一个统一的中断处理函数 Interrupt，由外部设备触发的各种硬中断都会进入此函数进行处理。在 Interrupt 函数中首先会调用 IntEnter 函数将被中断执行的线程的 CPU 现场保存到线程控制块的 KernelContext 域中，然后调用 KiDispatchInterrupt 函数（在文件ke/i386/dispatch.c 中定义）将中断派遣到对应的中断服务程序中进行处理，最后 调用 IntExit 函数恢复被中断执行的线程的 CPU 现场（在不考虑线程调度的情况下），使之继续运行。为实现线程调度中断处理，在 IntExit 函数中调用了 PspSelectNextThread 函数（在文件ps/sched.c 中定义），该函数按照调度策略从所有处于“就绪”状态的线程和当前被中断执行的线程中选择中断返回后继续执行的线程，最后 IntExit 函数恢复被 PspSelectNextThread 函数选中的线程的 CPU 现场，使之继续执行。

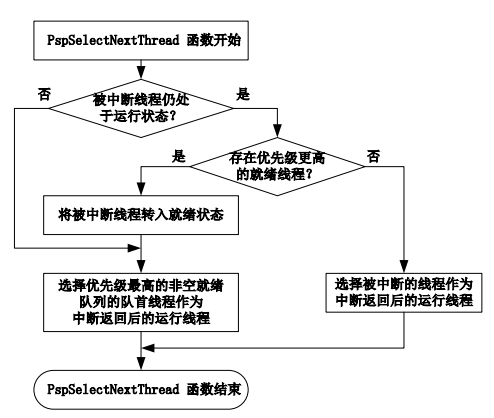
* 1. **核心源代码分析**

核心函数：线程调度函数。

功能描述：线程调度函数。当最外层中断服务程序执行完成后，并不是立刻返回被中断运行的线程，而是调用这个函数选择一个合适的线程继续运行（被中断的线程可能继续运行）。

返回值：应执行线程的 CPU 环境块指针。

函数流程图：



源代码实现：

PCONTEXT

PspSelectNextThread(

VOID

)

{

ULONG HighestPriority;

SIZE\_T StackSize;

// 扫描就绪位图，获得当前最高优先级。注意：就绪位图可能为空。

BitScanReverse(&HighestPriority, PspReadyBitmap);

if (NULL != PspCurrentThread && Running == PspCurrentThread->State) {

if (0 != PspReadyBitmap && HighestPriority > PspCurrentThread->Priority) {

// 如果存在比当前运行线程优先级更高的就绪线程，当前线程应被抢先。

// 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插入其

// 优先级对应的就绪队列的队首。注意，不能调用 PspReadyThread。

ListInsertHead( &PspReadyListHeads[PspCurrentThread->Priority],

&PspCurrentThread->StateListEntry );

BIT\_SET(PspReadyBitmap, PspCurrentThread->Priority);

PspCurrentThread->State = Ready;

} else {

// 当前线程继续运行。

// 注意：中断程序执行时可能换出了当前线程绑定运行的地址空间。

MmSwapProcessAddressSpace(PspCurrentThread->AttachedPas);

return &PspCurrentThread->KernelContext;

}

} else if(0 == PspReadyBitmap) {

// 被中断运行线程处于非运行状态，必须存在一个可运行的就绪线程。 ASSERT(FALSE);

KeBugCheck("No ready thread to run!");

}

if (NULL != PspCurrentThread) {

// 如果当前线程结束了自己则在这里释放线程的内核栈，因为线程在执行时不能

// 释放自己正在占用的栈。

if (Terminated == PspCurrentThread->State) {

StackSize = 0;

MmFreeVirtualMemory( &PspCurrentThread->KernelStack,

&StackSize,

MEM\_RELEASE,

TRUE );

}

// 取消指针 PspCurrentThread 对线程对象的引用。

ObDerefObject(PspCurrentThread);

}

// 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。

PspCurrentThread = CONTAINING\_RECORD(PspReadyListHeads[HighestPriority].Next, THREAD, StateListEntry);

ObRefObject(PspCurrentThread);

PspUnreadyThread(PspCurrentThread);

PspCurrentThread->State = Running;

// 换入线程绑定运行的地址空间。

MmSwapProcessAddressSpace(PspCurrentThread->AttachedPas);

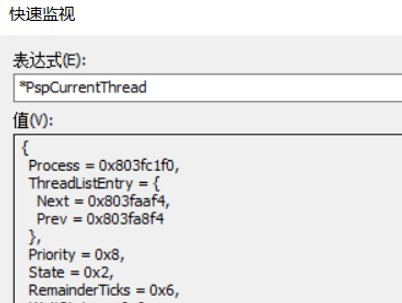
// 返回线程的上下文环境块，恢复线程运行。

return &PspCurrentThread->KernelContext;

}

* 1. **EOS调度程序的跟踪**
     1. 跟踪当前线程不被抢先的情况

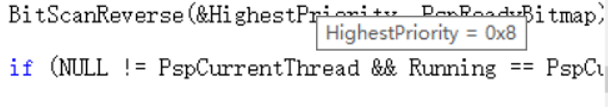
1. 在PspSelectNextThread中每当有定时计数器中断发生时（每 10ms 一次）都会触发线程调度函数PspSelectNextThread，都会扫描就绪位图获得当前最高优先级。
2. 在“监视”窗口以二进制格式查看就绪位图变量的值，此时就绪位图的值为100000001，即优先级为8和0的两个就绪队列中存在就绪线程。
3. “快速监视”中查看当前正在运行的线程





综合以上信息可以知道当前正在运行的线程就是“rr”命令新建的第 0 个线程。

1. 继续调试，BitScanReverse 函数执行完后，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。



1. 通过上述跟踪调试线程调度函数 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：

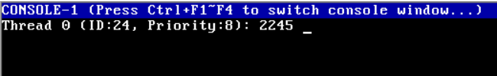
(1) 第 0 个线程仍然处于“运行”状态。

(2) 没有比其优先级更高的处于就绪状态的线程。

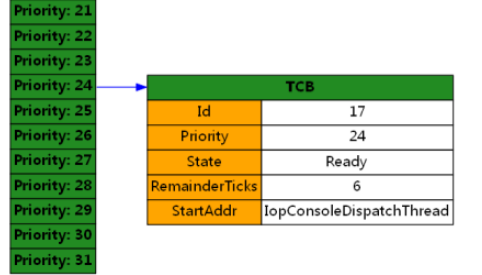
* + 1. 跟踪当前线程被抢先的情况

当有比第 0 个新建的线程优先级更高的线程进入就绪状态时，第0个新建的线程就会被抢先，可以通过在第 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。

1. 在第0个新建的线程正在执行时在虚拟机窗口按下一次空格键，此时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。
2. 此时控制台上第0个线程执行停止。



1. 查看“就绪线程队列”窗口，优先级为24的就绪队列中存在就绪线程。由StartAddr可知是控制台派遣线程。



1. 继续调试，BitScanReverse 函数执行完后，从就绪位图中扫描最高优先级24，并保存在变量 HighestPriority 中。由于其优先级高于正在运行的第 0 个新建的线程，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。

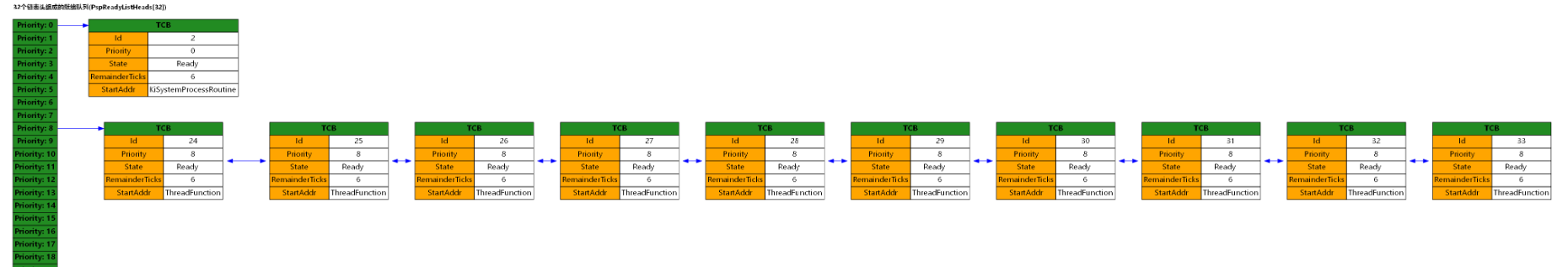
ListInsertHead( &PspReadyListHeads[PspCurrentThread->Priority],

&PspCurrentThread->StateListEntry );

BIT\_SET(PspReadyBitmap, PspCurrentThread->Priority);

PspCurrentThread->State = Ready;

1. 此时查看“就绪线程队列”，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。



1. 继续执行，正在执行的第0个新建的线程已经进入了就绪状态。

if (NULL != PspCurrentThread) {

// 如果当前线程结束了自己则在这里释放线程的内核栈，因为线程在执行时不能

// 释放自己正在占用的栈。

if (Terminated == PspCurrentThread->State) {

StackSize = 0;

MmFreeVirtualMemory( &PspCurrentThread->KernelStack,

&StackSize,

MEM\_RELEASE,

TRUE );

}

// 取消指针 PspCurrentThread 对线程对象的引用。

ObDerefObject(PspCurrentThread);

}

1. 使优先级为24的控制台派遣线程在CPU上执行。

PspCurrentThread = CONTAINING\_RECORD(PspReadyListHeads[HighestPriority].Next, THREAD, StateListEntry);

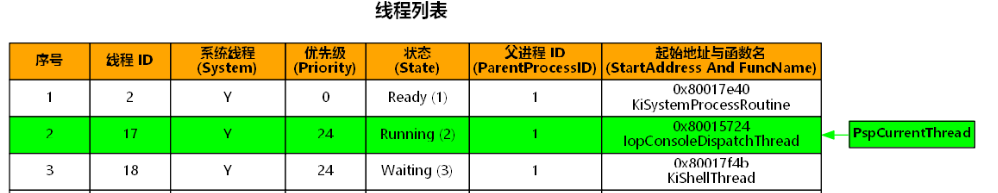
ObRefObject(PspCurrentThread);

PspUnreadyThread(PspCurrentThread);

PspCurrentThread->State = Running;

1. 此时刷新“就绪线程队列”，看到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。





**2）为EOS添加时间片轮转调度**

（给出实现方法的简要描述、源代码、测试及结果等）

**2.1 实现方法简要描述**

修改 ps/sched.c 文件中的 PspRoundRobin 函数。被定时计数器中断服务程序 KiIsrTimer 调用。无输入参数无返回值。

* 在进行时间片轮转调度前，先判断被中断线程是否仍处于“运行”状态。只有当被中断线程仍处于“运行”状态时，才需要进行时间片轮转调度。所以在 PspRoundRobin 函数中的第一行代码可以如下（线程状态的定义可以参见 ps/psp.h 的第 103 行）：

if (NULL != PspCurrentThread && Running == PspCurrentThread->State)

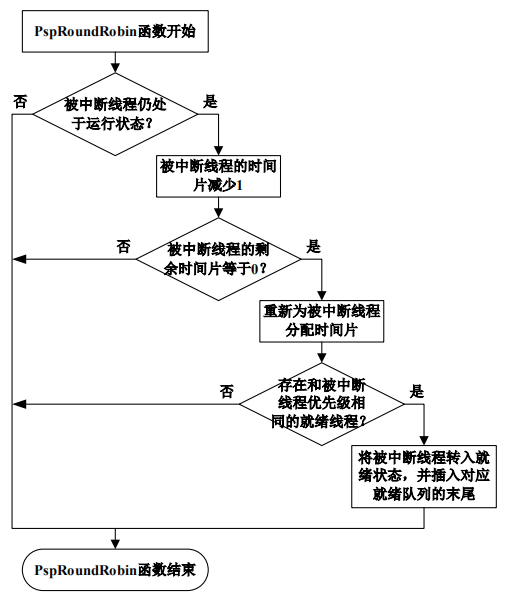
* 被中断线程所拥有的时间片保存在 PspCurrentThread 的 RemainderTicks 域中。因此修改线程的 时 间 片 通 过“PspCurrentThread->RemainderTick--”实现。当线程时间片减到 0 后，应为其重新分配时间片，重新为被中断线程分配时间片时，应该使用宏定义 TICKS\_OF\_TIME\_SLICE 为PspCurrentThread 变量的 RemainderTicks 域赋值
* 在判断就绪队列中是否存这样的就绪线程，即其优先级与被中断线程优先级相同时，只需要扫描就绪位图即可（这样速度更快）。若存在，将该被中断线程转入就绪状态插入对应就绪队列队尾。此时，可以用下面的代码作为条件语句中的布尔表达式：

BIT\_TEST(PspReadyBitmap, PspCurrentThread->Priority)

* 使用下面的代码将被中断线程转入就绪状态，并将其插入对应优先级就绪队列的末尾：

PspReadyThread(PspCurrentThread);

函数流程图：



**2.2 源代码**

VOID

PspRoundRobin(

VOID

)

{

// 在此添加代码，实现时间片轮转调度算法。

// 先判断被中断线程是否仍处于“运行”状态

if( NULL != PspCurrentThread && Running == PspCurrentThread->State) {

// 在此实现时间片轮转调度算法

PspCurrentThread->RemainderTicks--;

// 被中断线程的剩余时间片等于0

if(PspCurrentThread->RemainderTicks==0){

// 重新为被中断线程分配时间片

PspCurrentThread->RemainderTicks = TICKS\_OF\_TIME\_SLICE;

// 存在和被中断线程优先级相同的就绪线程？

if(BIT\_TEST(PspReadyBitmap, PspCurrentThread->Priority)!=0){

PspReadyThread(PspCurrentThread);

}

}

}

return;

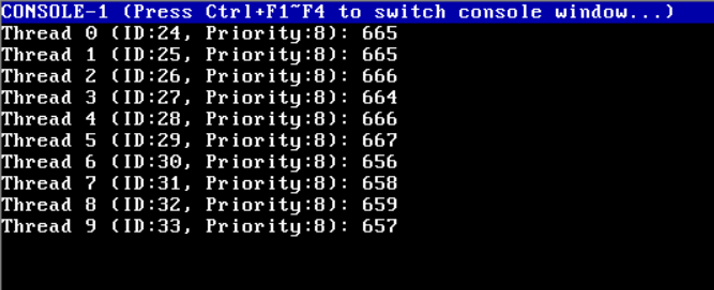
}

* 1. **测试及结果**

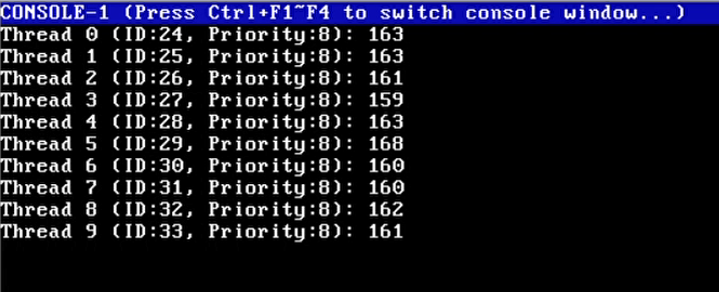
1. 代码修改完毕后，按 F7 生成 EOS 内核项目。

2. 按 F5 启动调试。

3. 在 EOS 控制台中输入命令“rr”后按回车。看到 10 个线程轮转执行的效果.



4. 修改ps/psp.h 第 120 行定义的 TICKS\_OF\_TIME\_SLICE 的值修改为 1。



**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS线程调度的特点、不足及改进意见；结合EOS对线程调度相关问题提出自己的思考，分析线程调度的执行时机和过程；分析为EOS添加时间片轮转调度实现方法的有效性、不足和改进意见，如果同时采用了多种实现方法，则进行对比分析；其他需要说明的问题）

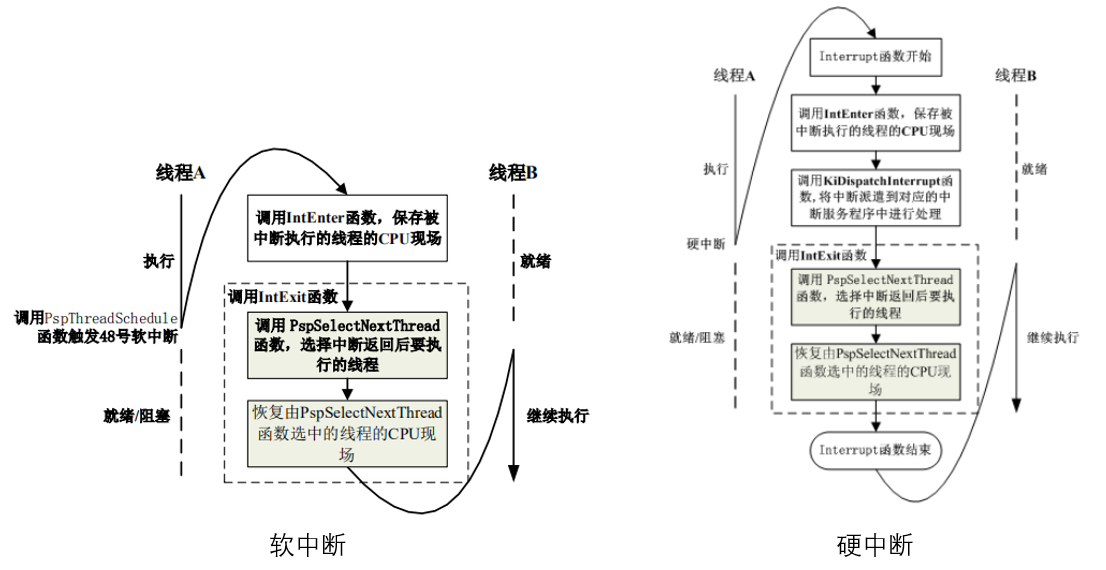
1. EOS线程调度的特点

EOS 的线程调度是由PspSelectNextThread函数和PspRoundRobin函数两个函数协作完成的。PspSelectNextThread函数最终决定哪个线程可以使用处理器，PspRoundRobin函数中实现时间片轮转调度，PspSelectNextThread选择一个合适的线程继续运行。

1. 线程调度的执行时机和过程

执行时机：当遇到中断后线程执行调度。中断可以是软中断或硬中断，不论哪一种，最终都是通过调用 PspThreadSchedule 函数来主动执行线程调度的。在 PspThreadSchedule 函数中使用了一个宏定义 KeThreadSchedule，该宏定义是通过“int48”指令触发了一个中断号为48 的软中断，专门用于正在运行的线程主动中断后触发线程调度，从而让出处理器的情况。

执行过程如下图，分别为考虑线程调度情况下， EOS48 号软中断与硬中断处理过程的流程图：



1. 添加时间片轮转调度实现方法的有效性

编写时间片轮转调度函数需要注意时间片长度需要设置合理，时间片设得太短会导致过多的进程切换，降低了 CPU 效率；而设得太长又可能引起对短的交互请求的响应变差。

1. 思考题

（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，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。但是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，无法进行实验。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：由 PspRoundRobin 函数和 PspSelectNextThread 函数的流程可知，当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，PspRoundRobin 函数会直接结束，所以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，那么比该中断线程更高的就绪进程就无法运行。

（3）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，20 个线程轮流执行一次需要 60×20＝1200ms，也就是需要 1 秒多的时间，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，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。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。