**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 11 月 17 日

**实验名称：操作系统实验1 操作系统启动（2分）**

**实验目的：**以一个教学型操作系统EOS为例，了解操作系统的启动过程，理解操作系统启动后的工作方式；能对核心源代码进行分析；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成功启动的全过程，分析相关源代码；查看EOS启动后的状态和行为。

**实验步骤：**

**1）EOS操作系统启动过程的跟踪与源代码分析**

（分析从引导到EOS内核加载的相关源代码；简要说明在本部分实验过程中完成的主要工作，包括BIOS、引导程序、EOS内核加载程序的跟踪等）

* 1. **调试BIOS程序**

CPU加电后首先执行BIOS程序，主要是对CPU、内存等硬件设备进行检测和初始化。启动调试后，Bochs会在CPU要执行的第一条指令处中断，即BIOS的第一条指令处中断，Console窗口显示将要执行的BIOS第一条指令的相关信息。



通过Console窗口显示的内容，可以获得关于BIOS第一条指令的如下信息：

* 行首的[0xfffffff0]表示此条指令所在的物理地址。
* f000:fff0表示此条指令所在的逻辑地址。
* jmp far f000:e05b是此条指令的反汇编代码。
* 行尾的ea5be000f0是此条指令的十六进制字节码。

此时查看寄存器和内存中的数据可以发现CS段寄存器的值为0xf000，IP通用寄存器的值为0xfff0，与BIOS第一条指令的信息一致，且内存值为0，未执行任何指令。

* 1. **调试软盘引导扇区程序**

（1）BIOS在执行完自检和初始化工作后，将软盘引导扇区（512字节）加载到物理地址0x7c00-0x7dff位置，并从0x7c00处的指令开始执行引导程序。

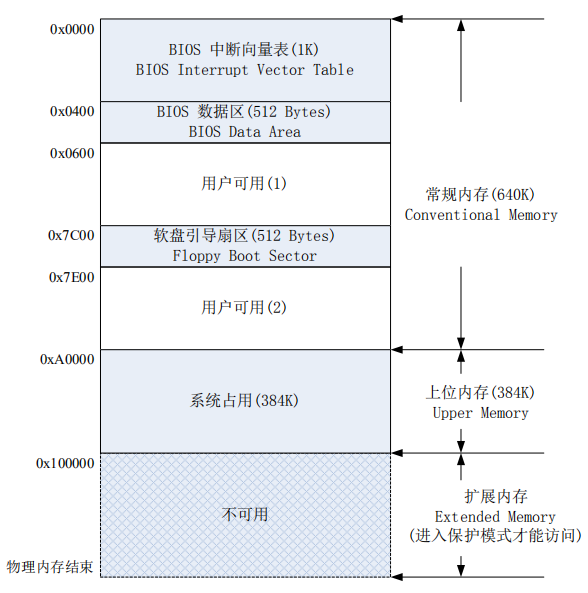
该部分说明代码可以在boot.lst中查看，在文件的开始（第72行）就将指令指针的ip值设为0x7C00，目标程序的第一个字节放在0x7C00处，后面的内存顺序存放。

org 0x7C00

73 00000000 EB6D jmp short Start

74 00000002 90 nop

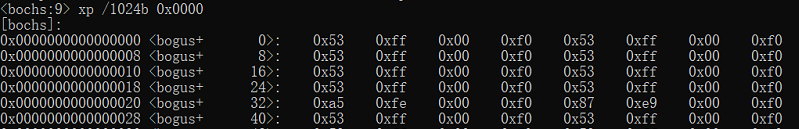
软盘引导扇区被加载后的物理内存布局如图：



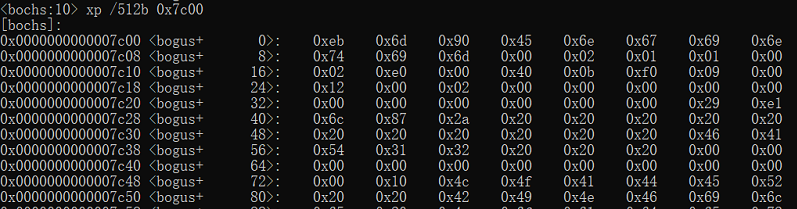
输入调试命令vb 0x0000:0x7c00，在逻辑地址0x0000:0x7c00处添加断点，输入c继续执行，在0x7c00处的断点中断，中断后在Console窗口输出下一个要执行的指令。字节码为eb6d，要跳转执行的下一条指令地址为0x00007c6f。



此时查看CS寄存器的值为0x0000，IP寄存器的值为0x7c00，输入调试命令xp /1024b 0x0000输出BIOS中断向量表中的内容。



输入调试命令xp /512b 0x7c00显示软盘引导扇区程序的字节码，输出内容如下。



此内存最开始的两个字节为0xeb和0x6d，和引导程序第一条指令（eb6d）相同，同时观察后面的内存内容，可以发现和boot.lst中红色虚框中内容相同，且这块内存最后的两个字节为0x55和0xaa，表示引导扇区是激活的，可以用来引导操作系统。



可以输入命令查看从0x0600开始的512字节内存即第一个用户可用区域是空白的，从0x7e00开始的512b字节内存即第二个用户可用区域也是空白的。

软盘引导扇区程序的主要任务就是将软盘中的loader.bin文件加载到物理内存的0x1000处，然后跳转到loader程序的第一条指令继续执行loader程序。

在boot.lst中第107行将LOADER\_ORG赋值为0x1000，即loader.bin开始的地址



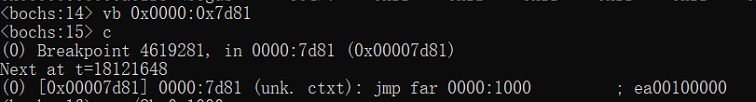
第278行有加载完loader.bin文件后跳转到loader程序中执行的指令行



分析boot.asm中的源代码，可知读取Loader.bin的流程大致如下：

先将FAT1读入FAT缓冲区，将根目录读入缓冲区，再在根目录中执行CompareNextDirEntry和CheckFileSize函数，对每一个文件比较文件名和文件大小查找Loader.bin文件，找到文件后开始加载文件：初始化ax为文件的第一个簇号，一个簇一个簇地读，通过查找FAT表获得下一个要读取的簇；对于每一个簇，需要ReadSector函数，从第ax个Sector开始，将c1个Sector循环读入es:bx中。

（2）由指令可知其相对程序0x7c00偏移0x0181即逻辑地址为0x0000:7d81，输入调试命令vb 0x0000:0x7d81添加断点，输入c执行，到断点处中断。在Console窗口显示



此条指令会跳转到物理内存0x1000处（即Loader程序的第一条指令）继续执行。打开loader.lst，查找到loader程序的第一条指令

33 00000000 E91801 jmp Start

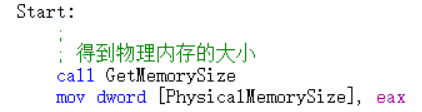
此时查看内存0x1000处的数据，可以看到此块内存的前三个字节和loader.lst文件中的第一条指令的字节码相同，即loader程序已经加载到从地址0x1000开始的内存中。



* 1. **调试加载程序**

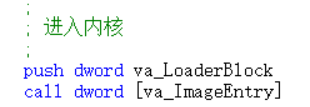
Loader程序的主要任务是将操作系统内核文件（kernel.dll 文件）加载到内存中，然后让 CPU 进入保护模式并且启用分页机制，最后进入操作系统内核开始执行（调用 kernel.dll 的入口点函数）。

（1）阅读Loader.asm文件代码，首先利用BIOS的int 0x15中断服务程序检测物理内存的大小，记录在内存变量PhysicalMemorySize中，以便传递给内核的内存管理器。



初始化FAT12文件系统相关的变量后，在根目录中查找文件名为szKernelFileName的文件，返回文件的目录项指针，加载内核文件到其虚拟基址对应的物理内存0x10000处，读取kernel.dll文件的过程与读取Loader.bin文件类似；当内核加载完毕后，准备跳入保护模式， Loader程序会打开A20地址线，加载全局描述符表（GDT），并通过一个长跳转进入80386的保护模式，此时可以访问32位的物理地址。

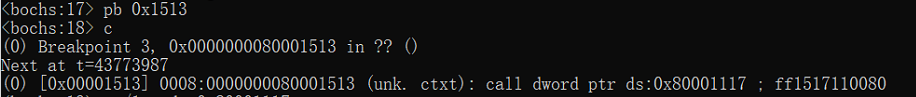
在ProtectionMode保护模式代码中，InitKernelImage函数将文件对齐的映像展开为节对齐，在loader.sam文件第714行可以找到准备进入EOS操作系统内核执行的指令：



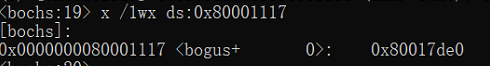
同时也可以在loader.lst文件中找到准备进入EOS操作系统内核执行的指令（第755行）

755 0000014F FF15[17010080] call dword [va\_ImageEntry]

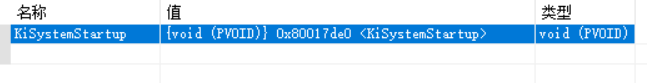
经计算知道此条指令物理地址为0x1513（0x13c4+0x14f），在该物理地址添加断点进行调试，在Console窗口显示要执行的下一条指令：



根据反汇编指令可以确定内核入口点函数的地址保存在逻辑地址ds:0x8000117处的四个字节中，查看内存中保存的32位函数入口地址，在Console窗口输出如下内容：



即此块内存中保存的函数地址为0x80017de0，即内核入口点函数的地址。在start.c源代码文件中的KiSystemStartup函数添加监视以看到此函数地址为{void (PVOID)} 0x800\*\*\*\*\* <KiSystemStartup> 与在虚拟内存 x80001117 处保存的函数入口地址相同，说明的确是由 Loader 程序进入了操作系统内核。



该函数会首先执行内核的初始化操作，后调用 PsCreateSystemProcess 函数创建一个系统进程，然后调用 KeThreadSchedule 函数调度到系统进程的主线程中执行。

PsCreateSystemProcess(KiSystemProcessRoutine);

KeThreadSchedule();

主线程的线程函数是文件ke/sysproc.c 中的 KiSystemProcessRoutine 函数，创建用于执行系统第二步初始化的初始化线程并退化为优先级位0的空闲线程。

Status = PsCreateThread( 0,

KiInitializationThread,

NULL,

FALSE,

&ThreadHandle,

NULL );

初始化线程创建了一个优先级为 24 的控制台派遣线程（线程函数为 io/console.c 文件中的 IopConsoleDispatchThread 函数），用于将键盘事件派遣到活动的控制台线程中，后初始化线程创建优先级为24的控制台线程（线程函数均为 ke/sysproc.c 文件中的 KiShellThread 函数）服务于各自的控制台窗口与用户交互。

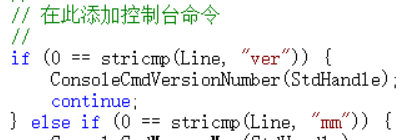
**2）查看EOS启动后的状态和行为**

（给出在本部分实验过程中完成的主要工作）

在EOS启动完毕进入内核后，会进一步完成一系列的内核初始化工作，直至EOS操作系统可以接收用户输入的命令为止。

1. **EOS运行内核命令”ver”，查看进程线程信息**

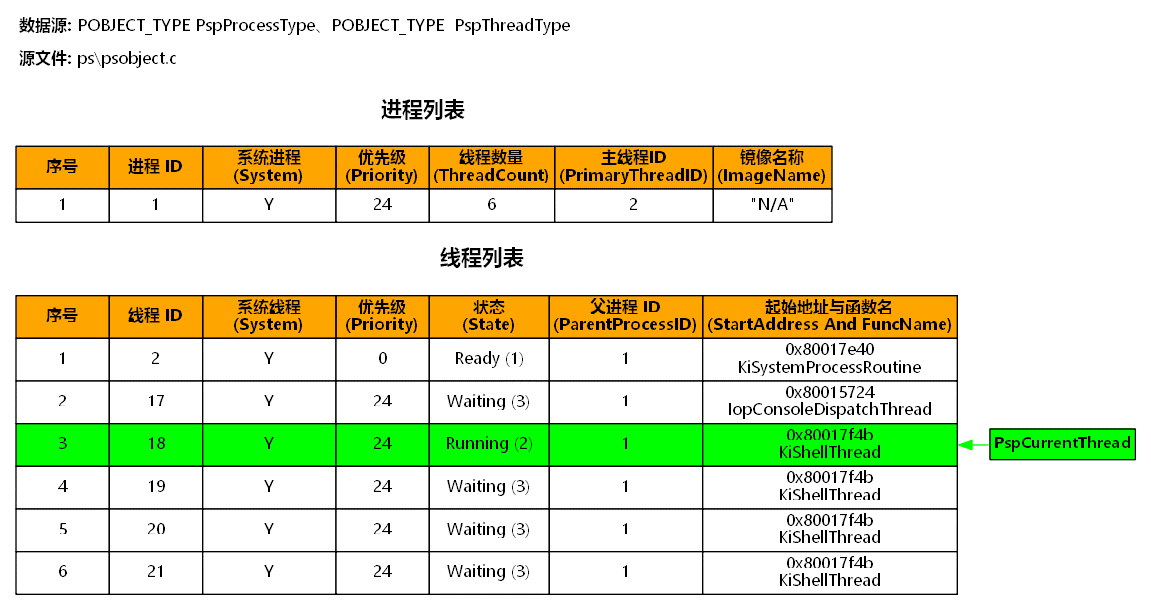
在ke/sysproc.c的KiShellThread中定义如果输入命令为“ver”则执行ConsoleCmdVersionNumber函数



该函数实现的功能为

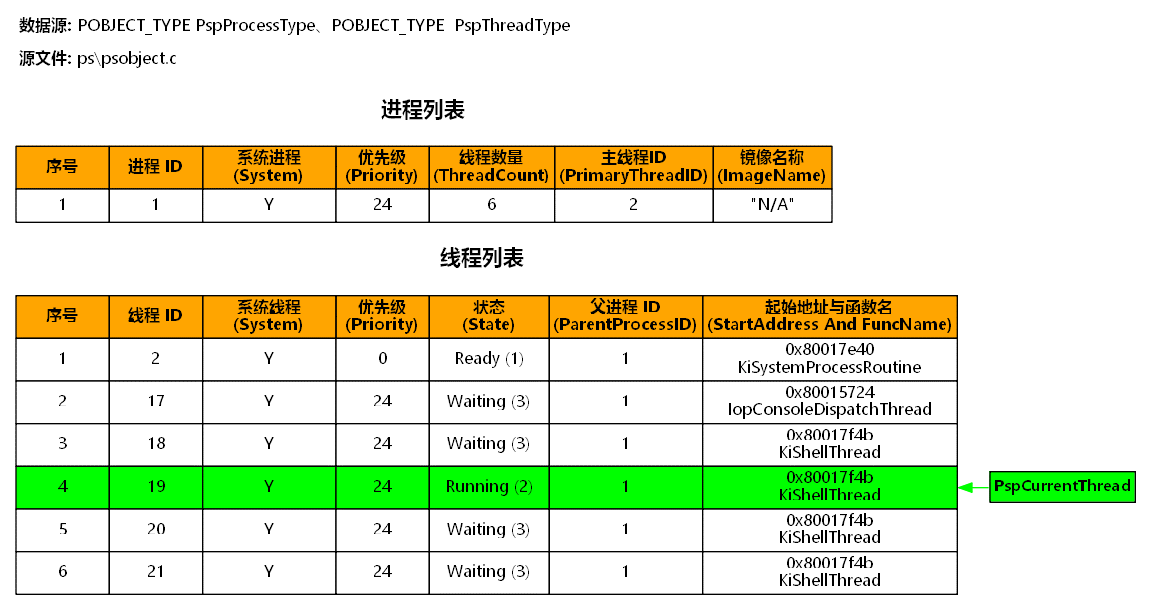
fprintf(StdHandle, "\nEngintime EOS [Version Number 1.2]\n\n"); }

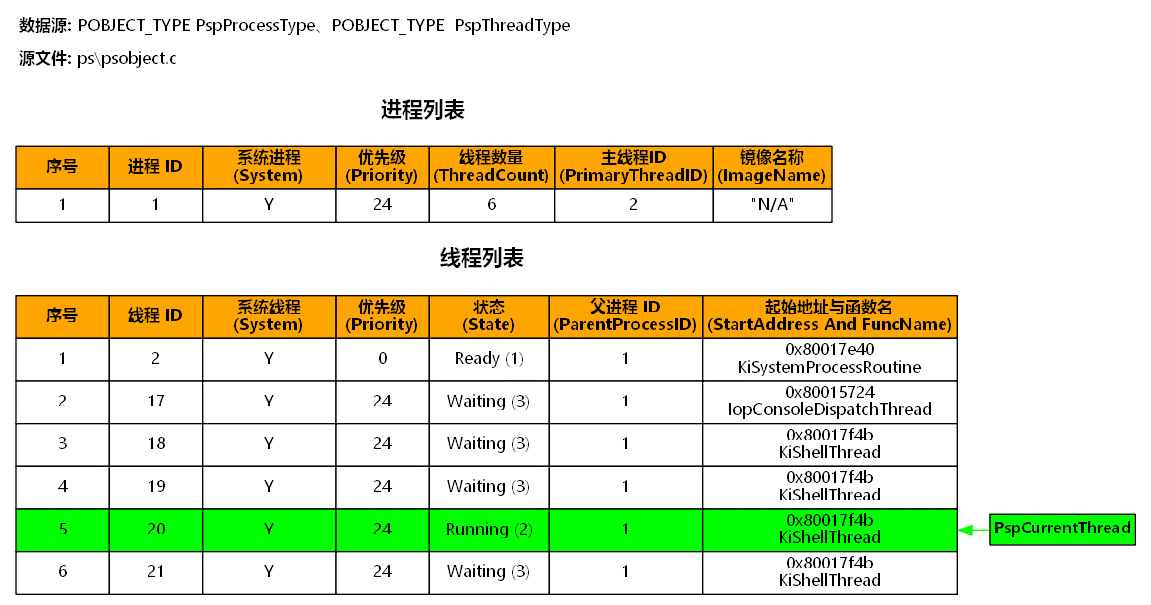
在”ver”命令函数添加断点进行调试，控制台输入”ver”后发生中断，刷新进程线程窗口，得如图内容。

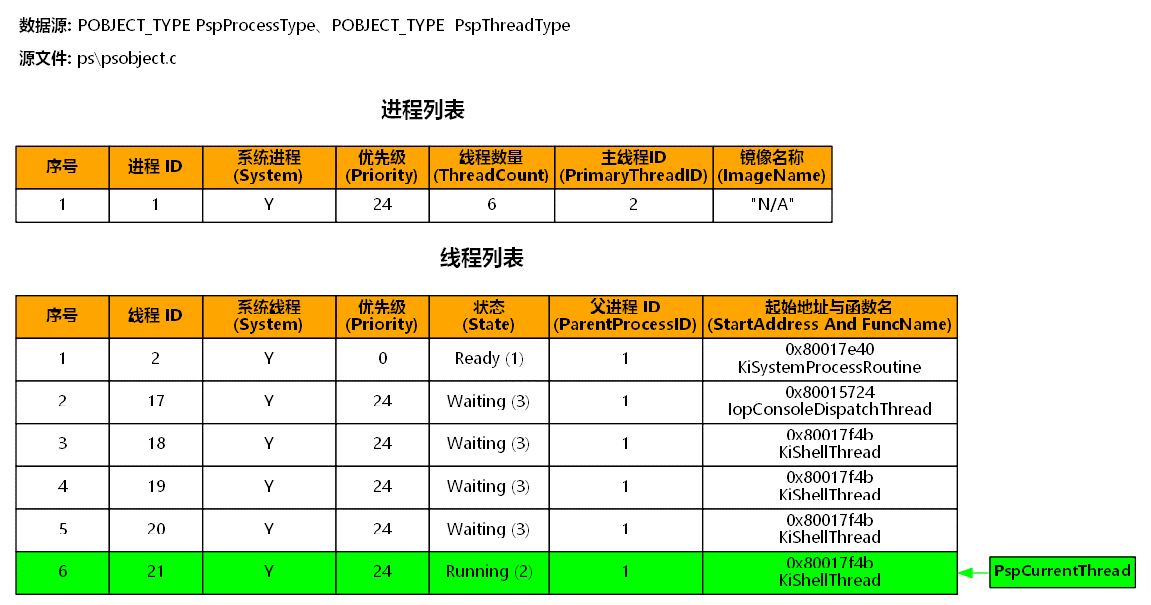


对图中显示的进程和线程信息进一步分析可以知道，进程列表中只有一个ID为1的系统进程，优先级为24，包含有6个线程，其中ID为2的线程是该进程的主线程，线程列表中的6个线程都是系统线程，父进程的ID都是1.其中ID为2的线程优先级为0，是空闲线程，在无线程占有处理器时运行，其余时间处于就绪状态；ID为17的线程是控制台派遣线程，用于将键盘事件派遣到活动的控制台线程，在无键盘事件发生时处于阻塞状态；余下的四个线程是控制台线程，对应四个控制台。

按F5继续运行，然后分别按 Ctrl+F2，Ctrl+F3，Ctrl+F4 切换到另外的 3 个控制台，在每个控制台中执行 ver 命令，命中断点后刷新进程线程窗口查看其内容，验证对应的控制台线程是否处于运行状态。运行结果如下图：







当EOS没有任何程序或命令时，空闲线程（ID=2）处于运行状态，即执行死循环，刷新进程线程窗口，可以得到此时进程线程情况。

//

// 将当前线程优先级降至最低，当前线程做为空闲线程进入空闲循环。

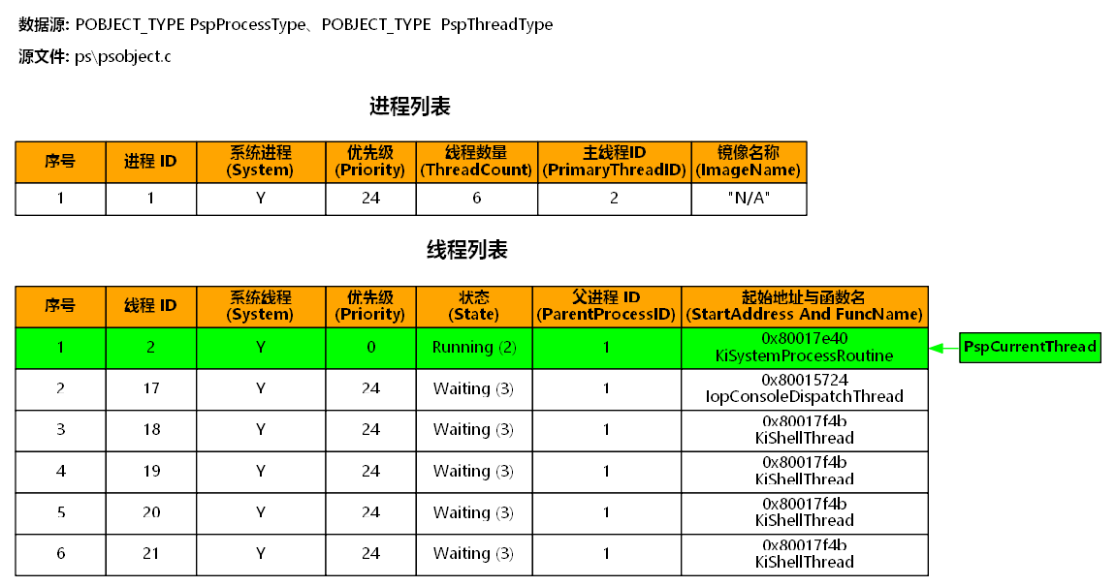
//

PsSetThreadPriority(CURRENT\_THREAD\_HANDLE, 0);

for(;;) {

i++;

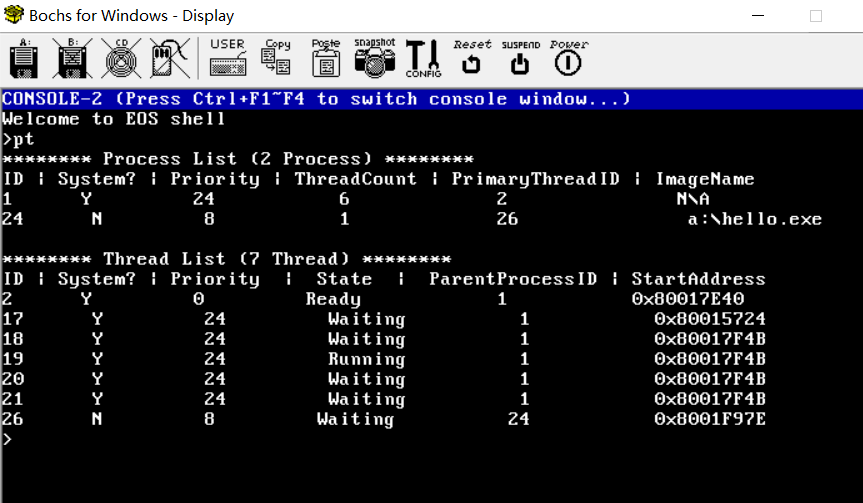
}



即当没有其他线程运行时，空闲线程总是不停的执行死循环，直到有中断发生或有更高优先级的线程抢占了处理器，其他进程处于Waiting状态。

**（2）EOS应用程序正在运行时，进程和线程信息**

添加应用程序Hello.exe文件至软盘镜像文件，在EOS控制台中输入命令“hello”后回车，切换到控制台2，输入“pt”后回车，输出的进程和线程信息如图所示。



上图中ID为24的进程为应用程序进程。ID为1的系统进程优先级为24，包含6个线程，其中ID为2的线程是该进程的主线程，没有映像名称。ID为24的应用程序，优先级为8，包含1个线程，其中ID为26的线程是该进程的主线程。

在线程列表里，除空闲线程外，系统线程的优先级都为24，其父进程的ID都为1，地址基本在同一内存块中，而应用程序线程的优先级为8，其父进程ID为31，地址与系统线程不在同一内存块中。

**结果分析：**

（对本实验所做工作及结果进行分析，包括结合EOS，在理解的基础上总结操作系统的启动过程以及启动后的工作方式；对EOS启动过程的相关问题提出自己的思考；其他需要说明的问题）

1. EOS的启动过程

通过实验，可以将EOS的启动过程总结如下：

EOS内核完成初始化后，用户即可与之进行交互。至此，EOS启动完毕。

Loader程序将软盘根目录中的操作系统内核Kernel.dll文件加载到物理内存中，然后启动CPU的保护模式和分页机制，最后跳转到Kernel.dll的入口点函数中执行。

Boot程序将软盘根目录中的Loader程序Loader.bin文件加载到物理内存的0x1000处，并跳转到Loader程序中执行。

CPU从默认位置执行BIOS的开机自检和初始化程序，然后BIOS会将软盘引导扇区加载到物理内存的0x7c00处，并跳转到引导扇区的Boot程序中执行。

CPU加电瞬间各个寄存器会自动初始化为默认值，首先执行BIOS程序，进行自检及初始化，根据在CMOS中保存的配置信息判断使用哪种设备启动操作系统，若是从软盘启动操作系统，则会将软盘的引导扇区加载到物理内存的0x7C00处，让CPU执行软盘引导扇区中的引导程序，然后启动操作系统。

Boot程序被限制在一个软盘扇区（512字节）内，除加载并执行Loader程序之外不做任何其它事情。

Loader程序检测物理内存的大小，并将内核文件kernel.dll载到物理内存0x10000处，打开A20地址线，加载GDT，为内核准备保护模式执行环境。启动分页机制，对内核文件进行节对齐后跳转到kernel.dll入口点继续执行，将控制权交给内核。

内核文件的入口点是源文件ke/start.c中的Ki SystemStartup 函数。完成初始化后，即创建了一个系统进程，系统进程的主线程执行后创建初始化线程，初始化线程创建控制台派遣线程和控制台线程用于和用户进行交互，至此EOS操作系统的启动过程完成。

1. EOS启动后的工作方式

通过进程线程列表，可以发现当EOS没有运行任何程序或命令时，空闲线程（ID=2）处于运行状态，进程列表中只有一个系统进程，包含6个线程，其中ID为17的线程是控制台派遣线程，用于将键盘事件派遣到活动的控制台线程，在无键盘事件时，总是处于阻塞状态，其他四个线程是控制台线程对应四个控制台用于用户交互。

1. 思考题

（1）为什么 EOS 操作系统从软盘启动时要使用 boot.bin 和 loader.bin 两个程序？使用一个可以吗？它们各自的主要功能是什么？如果将 loader.bin 的功能移动到 boot.bin 文件中，则boot.bin 文件的大小是否仍然能保持小于 512 字节？

答：在 IDE 环境启动执行 EOS 操作系统的时候，会把 boot.bin、loader.bin 和 kernel.dll 三个二进制文件写到软盘镜像文件中，然后让虚拟机来执行软盘里的 EOS 操作系统。仅使用其中的一个是不能运行的。

Boot.bin 程序的功能是：在 Boot.bin 程序执行的过程中，CPU 始终处于实模式状态。Boot.bin 程序利用 BIOS 提供的 int 0x13 中断服务程序读取软盘 FAT12 文件系统的根目录，在根目录中搜寻 loader.bin 文件。

Loader.bin 程序的功能是：Loader.bin 程序的任务和 Boot.bin 程序很相似，同样是将其它的程序加载到物理内存中，但这次加载的是 EOS 内核。除此之外，Loader.bin 程序还负责检测内存大小，为内核准备保护模式执行环境等工作。 如果把 loader.bin 功能移动到 boot.bin程序中，就会导致程序规模的扩大，可能使其大小大于 512 字节。

（2）软盘引导扇区加载完毕后内存中有两个用户可用的区域，为什么软盘引导扇区程序选择将 loader.bin 加载到第一个可用区域的 0x1000 处呢？这样做有什么好处？这样做会对loader.bin 文件的大小有哪些限制。

答：第一个用户可用区域是低地址区，并且空间是比较小的，适合容纳较小的文件，所以选择把占用空间小的loder.bin 加载到第一用户区。

好处：由低地址开始，方便检索查找。小文件占用较小的空间，能够节约资源。 限制：loader.bin 文件必须小于 1c00k 才能放到第一用户区。