**北京科技大学实验报告**

学院：计算机与通信工程学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期： 2023年 10月 16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。

**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中完成的主要工作）

**1、分析从引导到EOS内核加载的相关源代码**

EOS操作系统启动的基本过程如下：a、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。b、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。c、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。d、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。下面，通过分析从引导到EOS内核加载的相关源代码，对各步骤的具体情况进行分析。

**（1）BIOS程序的执行过程**

CPU在加电瞬间，其各个寄存器会自动初始化为默认值，其中CS和IP寄存器的默认值指向了BIOS程序的第一条指令，从而使得CPU开始执行BIOS程序。源码表明BIOS程序的第一条指令的地址为[0xfffffff0]，符合CS和IP寄存器的默认值。BIOS程序在执行完自检和初始化后，会将软盘引导扇区（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。

**（2）软盘引导扇区BOOT程序的执行过程**

Boot程序的执行过程中，CPU处于实模式，Boot程序将Loader.bin文件读入地址0x1000起始的物理内存，最后跳转到0x1000处开始执行Loader程序。阅读boot.asm，进行代码分析。

a、代码开始于org 0x7C00，这是引导扇区程序的起始地址。jmp short Start 跳转到 Start 标签，并有一个 nop 指令，通常用于填充空间。

b、代码定义了引导扇区的文件系统参数，包括扇区大小、每簇扇区数、保留扇区数量、FAT 表数、根目录文件数最大值等。这些参数用于描述文件系统和存储有关引导过程的信息。

c、在start标签处，初始化了CPU的段寄存器，将DS、ES、SS设置为0（即物理地址0），SP设置为0，以确保堆栈操作正常。接着初始化屏幕，设置显示模式和颜色。

d、代码计算根目录的起始扇区号、根目录占用的扇区数量、数据区的起始扇区号等文件系统相关信息，并分配了缓冲区用于FAT表和根目录。

e、在FindFile部分查找Loader.bin文件，并检查文件是否符合要求。

f、加载文件，读取完文件后关闭软驱马达并通过jmp 0:LOADER\_ROG跳转到Loader.bin的执行点。

**（3）Loader程序的执行过程**

Loader程序首先利用 BIOS 的 int 0x15 中断服务程序检测物理内存的大小，然后将内存大小记录在内存变量中，以便传递给内核的内存管理器。接下来Loader程序从软盘的根目录中将内核文件kernel.dll 载入物理内存0x10000 处。加载内核文件完毕后，Loader程序进入保护模式并启动分页机制，对内核文件进行节对齐操作后，通过调用kernel.dll的入口点函数，Loader程序便跳转到kernel.dll的入口点继续执行。阅读loader.asm，进行代码分析。

a、结构体 LOADER\_PARAMETER\_BLOCK用于描述加载器的参数。

b、在 Start 标签下开始实模式代码的执行，它首先获取物理内存大小，初始化 FAT12 文件系统相关的变量，然后在根目录中查找文件名为 kernel.dll 的文件。

c、查找到kernel.dll文件后，检查该文件，准备进入保护模式，加载全局描述符表，打开地址线 A20，设置 CR0 的保护标志位，通过jmp dword CS\_SELECTOR:ProtectionMode跳转到保护模式的代码。

d、在ProtectionMode代码块中，会对页表进行初始化，然后启动分页机制。

e、call InitKernelImage初始化内核镜像，然后通过call dword [va\_ImageEntry]进入内核。

**（4）内核的初始化过程**

a、内核文件 kernel.dll 的入口点是源文件 ke/start.c 中的 KiSystemStartup 函数。该函数会首先执行内核的初始化操作，使内核具备基本的中断管理和时钟管理功能。KiSystemStartup函数会执行各个管理器的初始化。

b、调用 PsCreateSystemProcess 函数创建一个系统进程（也是唯一的），然后调用 KeThreadSchedule 函数调度到系统进程的主线程中执行，其线程函数是文件 ke/sysproc.c中的KiSystemProcessRoutine 函数。KiSystemProcessRoutine先调用PsCreateThread，创建初始化线程KiInitializationThread。

c、KiSystemProcessRoutine将主线程优先级降至0，并且死循环。创建控制台派遣函数IopConsoleDispatchThread，用于将键盘事件派遣到活动的控制台线程中。

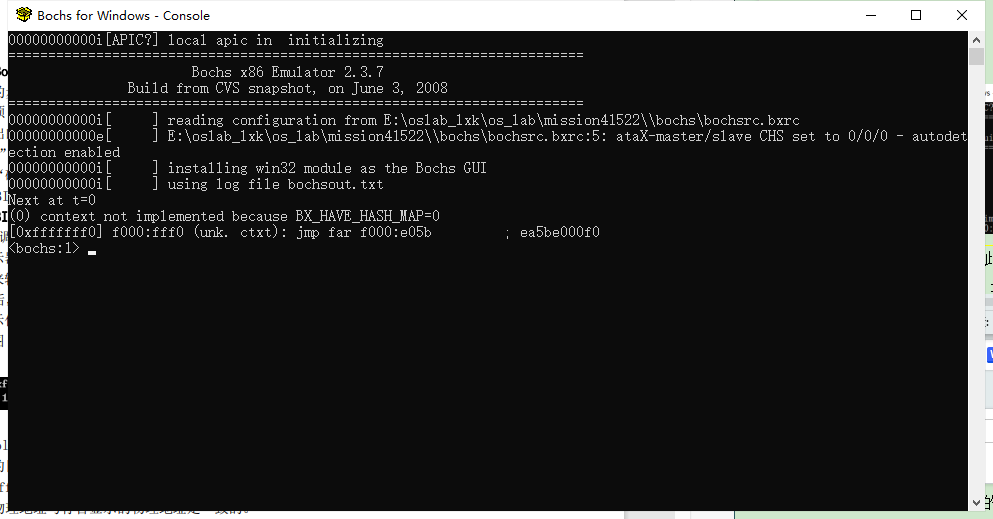
d、创建控制台线程KiShellThread，至此，操作系统有了一个 ID 为1 的系统进程，其优先级为 24。

e、至此，操作系统有了一个 ID 为1 的系统进程，其优先级为24。包含有 6 个线程，其中 ID 为 2 的线程是系统进程的主线程，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，以及4个线程都是控制台线程，分别对应于四个控制台。

**2、本部分实验中完成的主要工作**

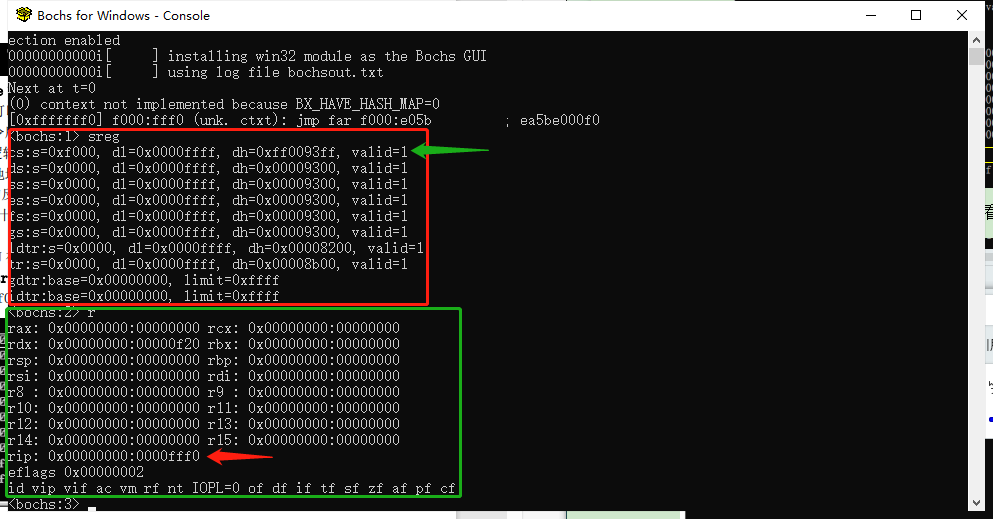
**（1）调试BIOS程序**

a、按F7生成项目，F5进行调试，Bochs会在CPU要执行的第一条指令处中断，Console窗口如下。可以看到，指令的物理地址是0xfffffff0，逻辑地址是f000:fff0，指令的反汇编代码是jmp far f000:e05b，指令的十六进制字节码是ea5be000f0。

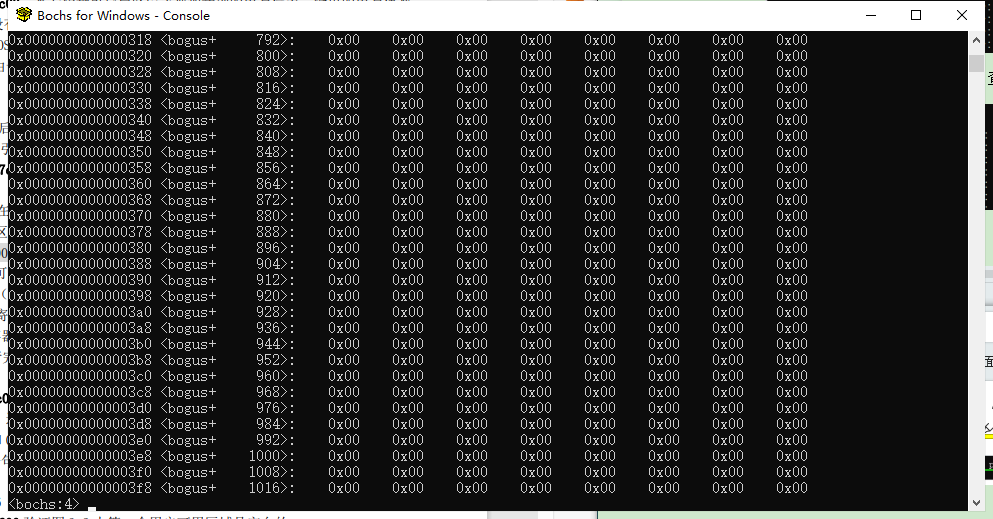


b、输入sreg，查看寄存器的值，CS寄存器的值为0xf000。

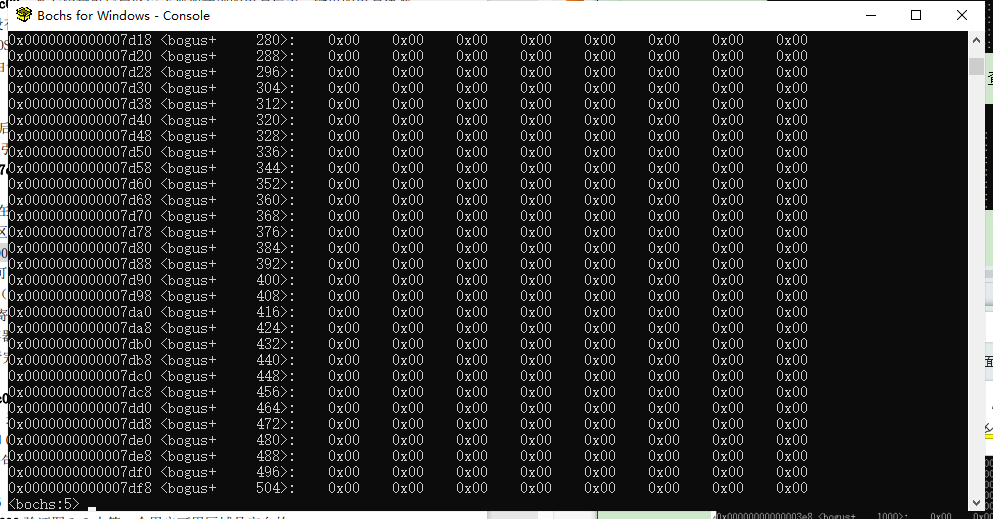
c、输入r，查看通用寄存器的值，IP寄存器的值为0xfff0。



d、输入xp /1024b 0x0000，查看开始的1024个字节的物理内存。



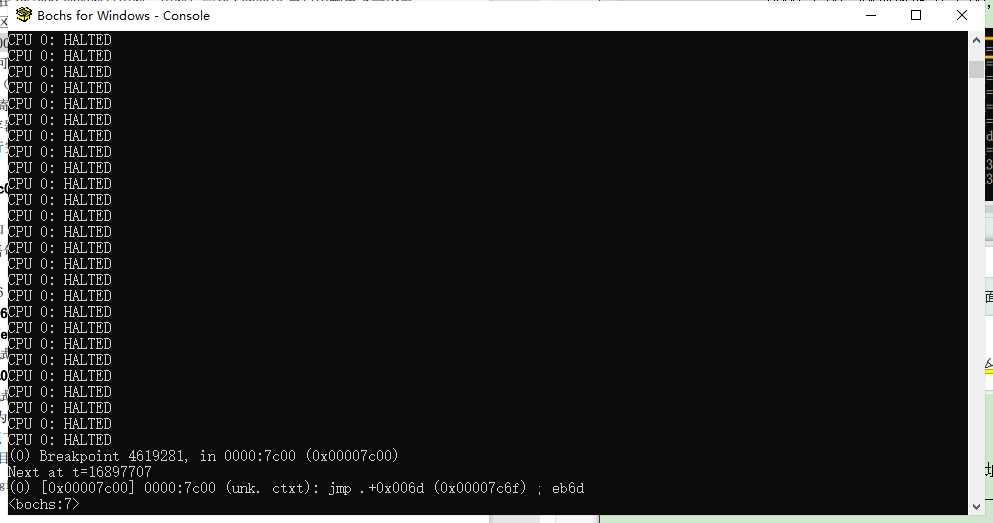
e、输入xp /512b 0x7c00，查看引导扇区应该被加载到的内存位置。



**（2）调试软盘引导扇区程序**

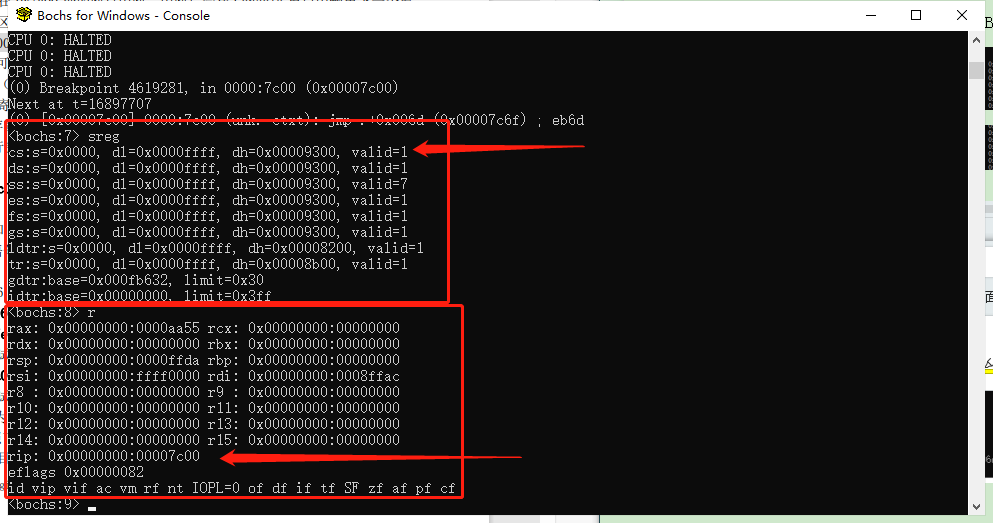
a、输入vb 0x0000:0x7c00，在物理地址0x7c00处设置断点。

b、输入c，执行到0x7c00处中断，下一条要执行的指令为：(0) [0x00007c00] 0000:7c00 (unk. ctxt): jmp .+0x006d (0x00007c6f) ; eb6d

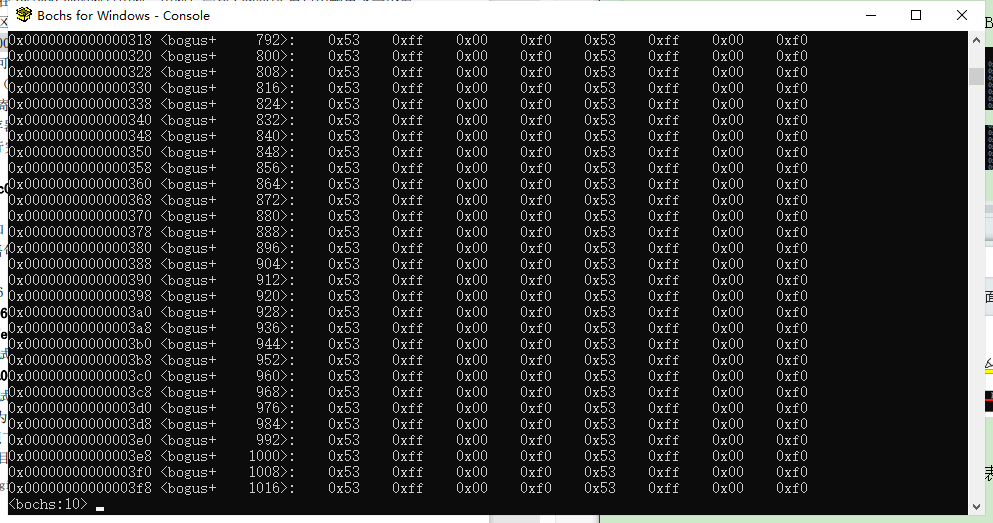


c、输入sreg，查看寄存器CS的值为0x0000。

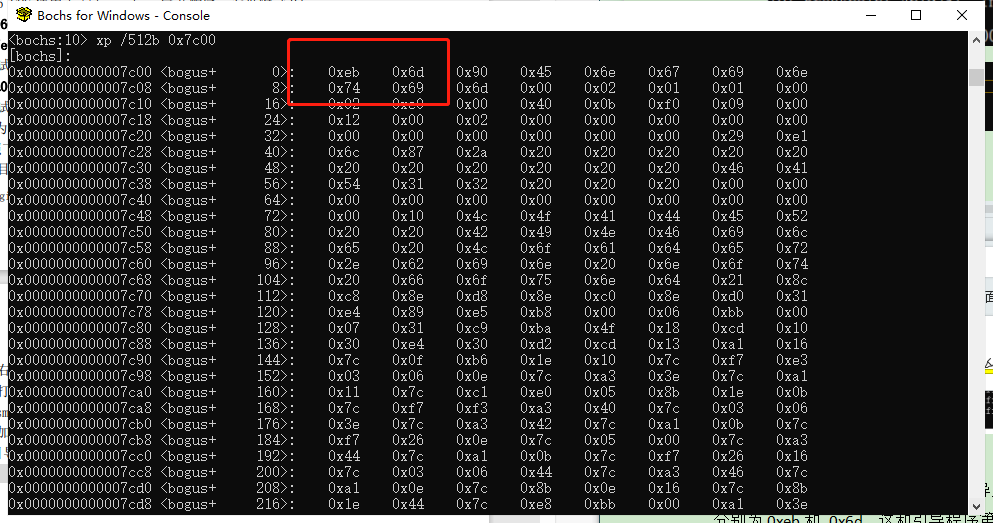
d、输入r，查看寄存器IP的值为0x7c00。

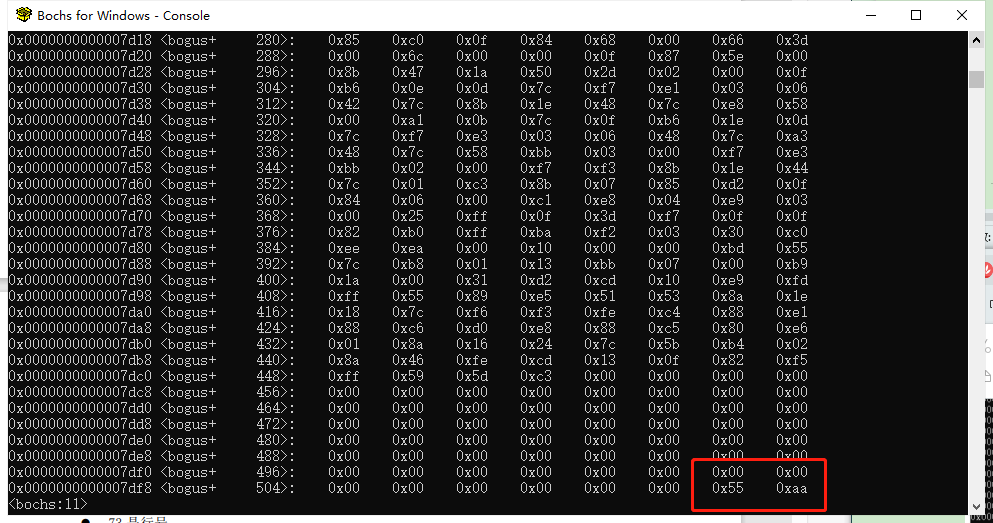


e、输入xp /1024b 0x0000，中断向量表被载入。

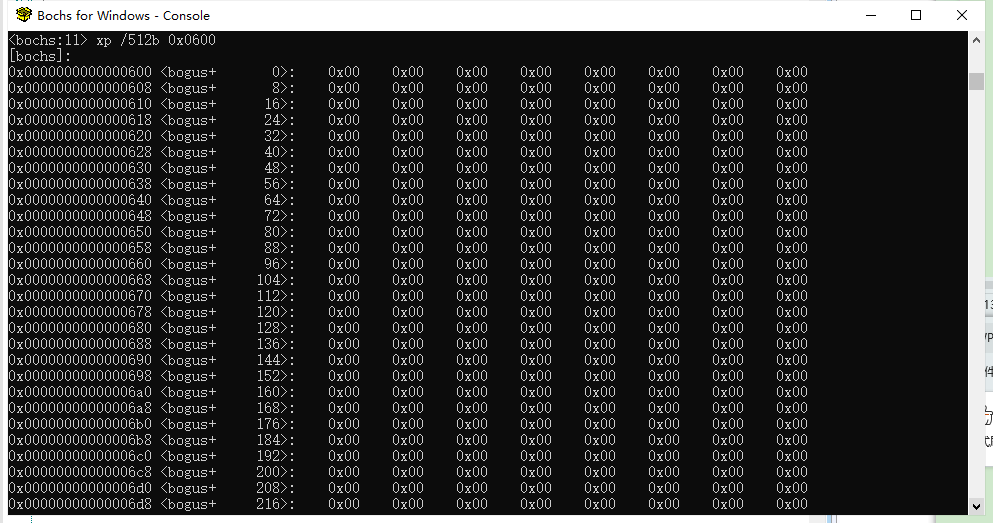


f、输入xp /512b 0x7c00显示软盘引导扇区程序的所有字节码。此块内存最开始的两个字节分别为0xeb和 0x6d，这和引导程序第一条指令的字节码（eb6d）是相同的。此块内存最后的两个字节分别为0x55 和0xaa，表示引导扇区是激活的，可以用来引导操作系统，这两个字节是 boot.asm 中最后一行语句 dw 0xaa55定义的。

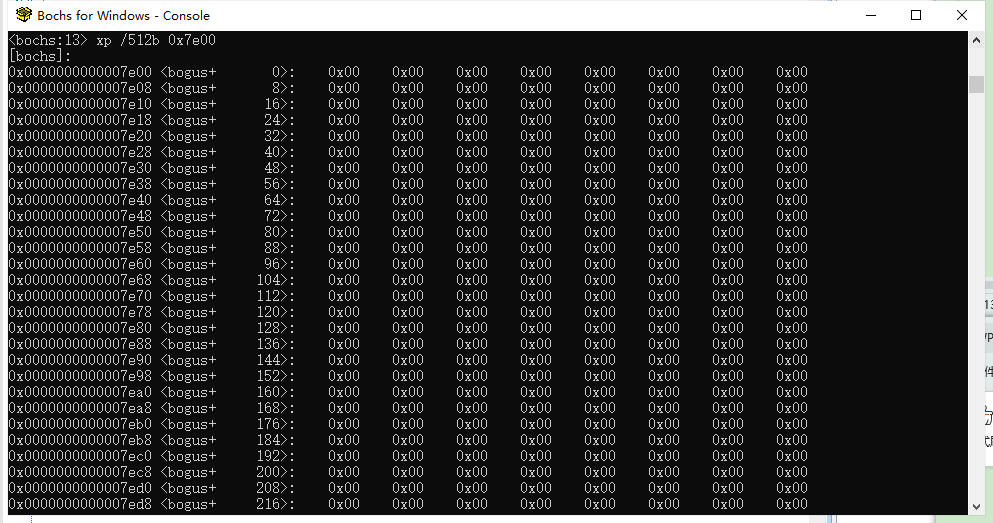




g、输入xp /512b 0x0600，验证第一个用户可用区域是空白的。

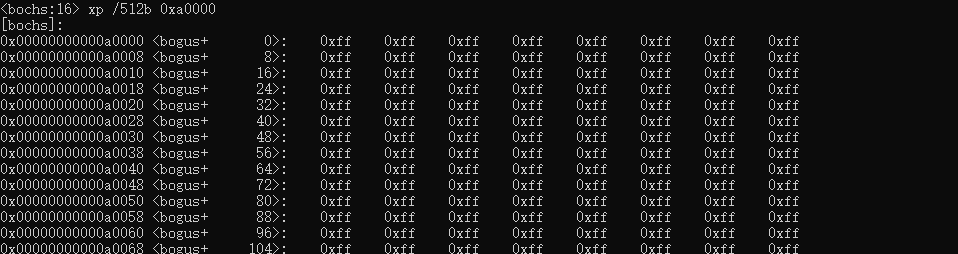


h、输入xp /512b 0x7e00，验证第二个用户可用区域是空白的。



|  |
| --- |
| **自行设计**  自己设计两个调试命令，验证这两个用户可用区域的高地址端也是空白的。  已知用户可用区域1的物理地址范围是0x0600~0x7bff，现在已经用xp /512b 0x0600验证第一个用户可用区域的0x0600~0x07ff是空白的，用户可用区域1的低地址端是空白的。但0x0800~0x7bff没有验证。用户可用区域2也是同理。  xp /512b 0x7a00验证用户可用区域1的高地址端也是空白的  xp/ 512b 0x9e00 验证用户可用区域2的高地址端也是空白的 |

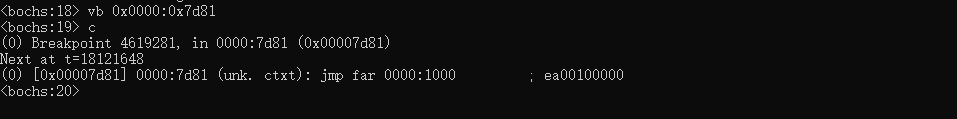
i、输入xp /512b 0xa0000，验证尚未内存已经被系统占用



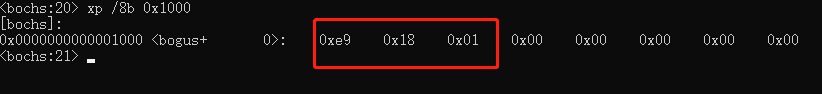
|  |
| --- |
| **自行设计**  自己设计一个查看内存的调试命令，验证上位内存的高地址端已经被系统占用。  上位内存物理地址范围是0xa0000-0xfffff，所以输入xp /512b 0xffdff，验证上位内存的高地址端已经被占用。 |

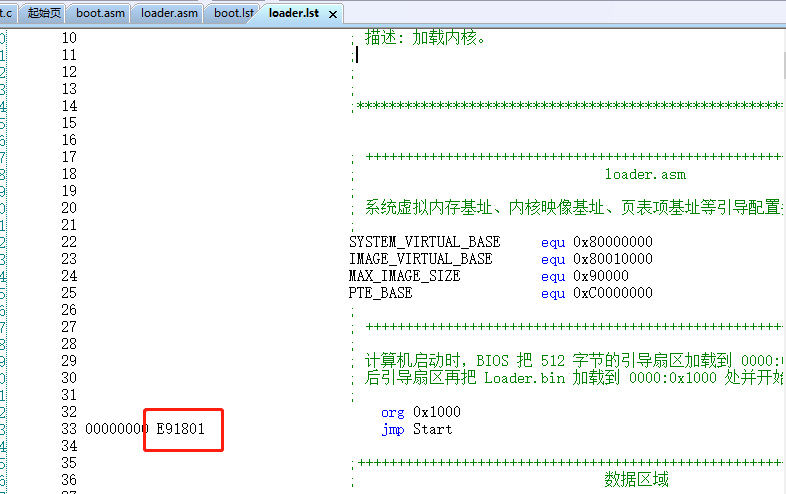
j、查看boot.lst文件，引导扇区程序将loader.bin文件加载到物理内存0x1000处，然后从0x1000处执行loader程序。输入vb 0x0000:0x7d81，添加断点。

k、输入c，执行至断点，显示的指令会跳转到物理内存0x1000处，执行loader程序。



l、输入xp /8b 0x1000，查看0x1000处的数据，的前三个字节和 loader.lst 文件中的第一条指令的字节码是相同的。



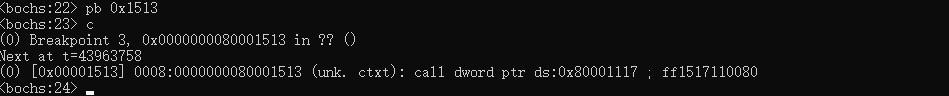


|  |
| --- |
| **自行设计**  查看内存中 loader 程序结束位置的字节码，并与 loader.lst 文件中最后指令的字节码比较。  loader.bin文件大小为1566b，  输入xp /1566b 0x1000，查看最后几个字节与loader.lst最后的指令字节码一致。 |

**（3）调试加载程序**

a、在 loader.lst 文件中查找进入内核的指令，计算出其物理地址0x1513，输入pb 0x1513在0x1513加入断点。

b、输入c，继续执行到断点处。



c、查看断点，以确定内核入口点函数的地址就保存在逻辑地址 ds:0x8000117 处的四个字节中。

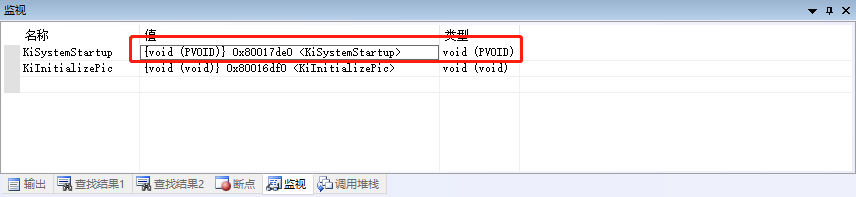
d、输入x /1wx ds:0x80001117，查看内存中保存的 32 位函数入口地址。



**（4）调试内核**

a、查看start.c文件，为LIInitializePic函数添加断点，开始调试。

b、执行至断点处，监视该函数，得到函数的地址，与（3）得到的地址比较，结果相同。

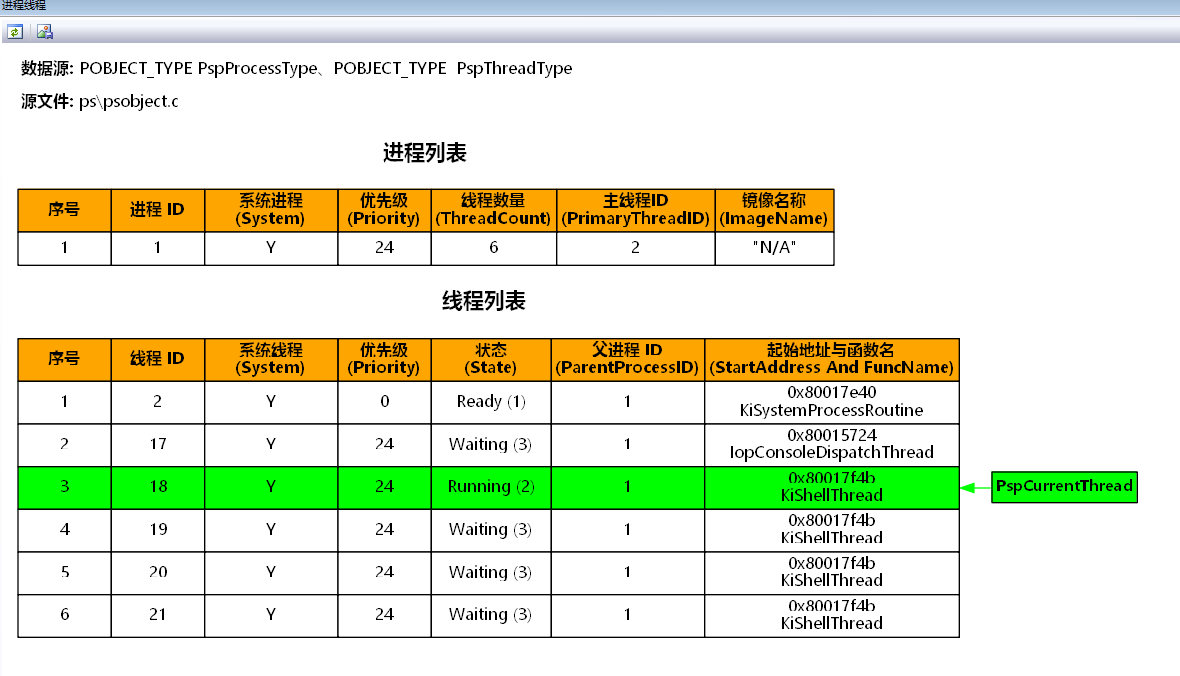


**2）查看EOS启动后的状态和行为**

（简要说明在本部分实验中完成的主要工作）

**（1）查看EOS启动后进程与线程的状态**

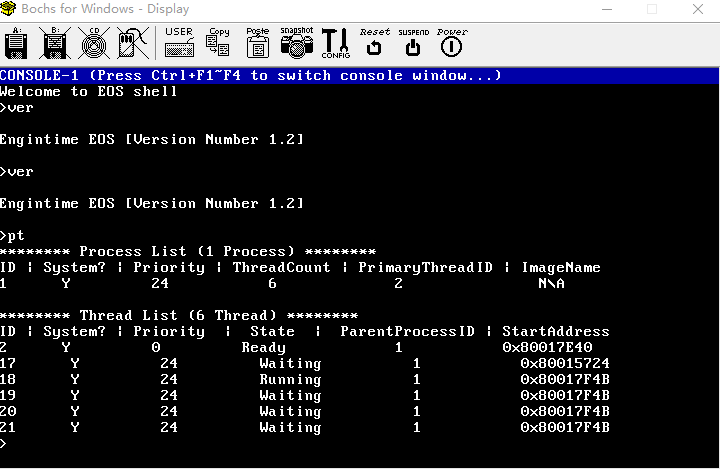
a、在 ke/sysproc.c 文件的第 372 行，“ver”命令函数中添加一个断点，启动调试后输入“ver”，在断点处中断，查看进程线程。



b、ke/sysproc.c 文件的第 143 行空闲线程的死循环中添加了一个断点，按F5来到断点。可以看到空闲线程（ID=2）此时处于运行状态：



c、控制台输入pt，查看所有进程和线程的情况。

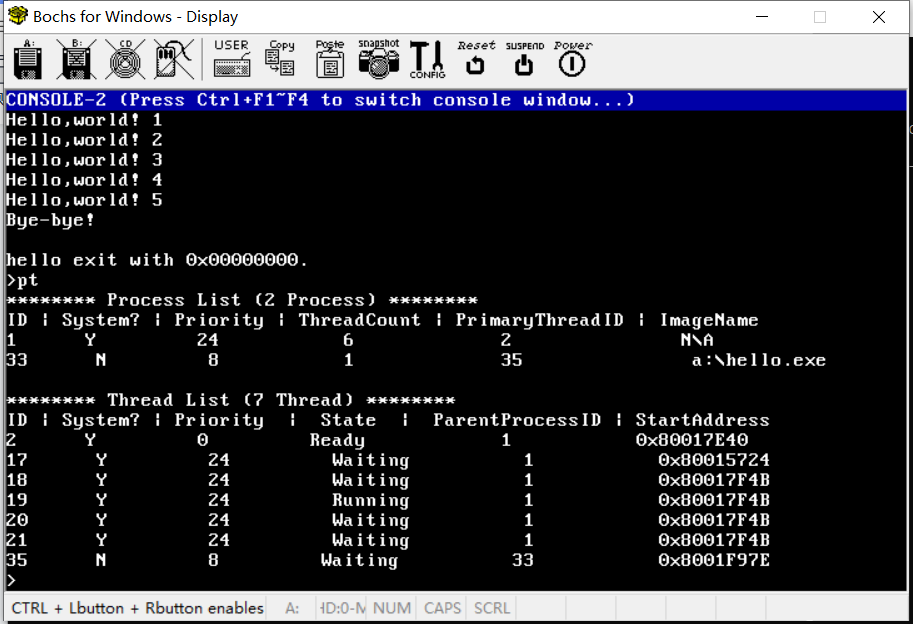


**（2）当一个EOS应用程序正在运行时的进程和线程信息**

a、将hello.exe添加到软盘镜像文件，启动EOS，输入hello，hello.exe便开始执行。



b、输入hello后切换至控制台2，输入pt，查看进程线程信息。

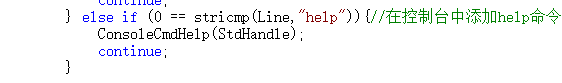


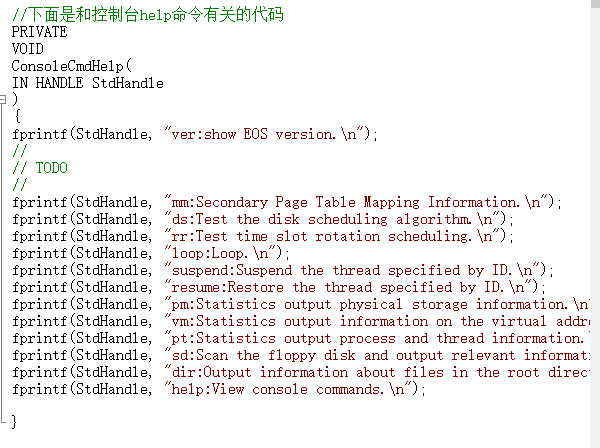
对比发现，33号进程是hello.exe进程，是应用程序进程；它有一个35号线程，是应用程序线程。它在system?一栏是N，在priority一栏优先级是8，比较低。

**（3）为EOS内核添加help命令**

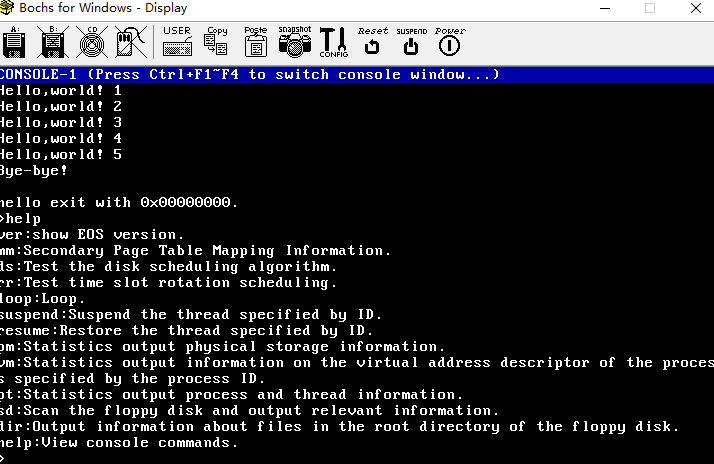
a、在ke/sysproc.c文件中，添加函数声明和相应的函数实现。







b、控制台输入help，查看结果。



**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、操作系统的启动过程**

EOS操作系统启动的基本过程如下：a、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。b、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。c、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。d、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。下面，通过分析从引导到EOS内核加载的相关源代码，对各步骤的具体情况进行分析。

**（1）BIOS程序的执行过程**

CPU在加电瞬间，其各个寄存器会自动初始化为默认值，其中CS和IP寄存器的默认值指向了BIOS程序的第一条指令，从而使得CPU开始执行BIOS程序。BIOS程序在执行完自检和初始化后，会将软盘引导扇区（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。

**（2）软盘引导扇区BOOT程序的执行过程**

Boot程序的执行过程中，首先读取文件系统的根目录，CPU处于实模式，Boot程序将Loader.bin文件读入地址0x1000起始的物理内存，最后跳转到0x1000处开始执行Loader程序。

**（3）Loader程序的执行过程**

Loader程序首先利用 BIOS 的 int 0x15 中断服务程序检测物理内存的大小，然后将内存大小记录在内存变量中，以便传递给内核的内存管理器。接下来Loader程序从软盘的根目录中将内核文件kernel.dll 载入物理内存0x10000 处。加载内核文件完毕后，Loader程序进入保护模式并启动分页机制，对内核文件进行节对齐操作后，通过调用kernel.dll的入口点函数，Loader程序便跳转到kernel.dll的入口点继续执行。

**（4）内核的初始化过程**

内核文件 kernel.dll 的入口点是源文件 ke/start.c 中的 KiSystemStartup 函数。该函数会首先执行内核的初始化操作，使内核具备基本的中断管理和时钟管理功能。KiSystemStartup函数会执行各个管理器的初始化。

操作系统有一个 ID 为1 的系统进程，其优先级为24。包含有 6 个线程，其中 ID 为 2 的线程是系统进程的主线程，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，以及4个线程都是控制台线程，分别对应于四个控制台。

**2、EOS启动后的工作方式**

KiSystemStartup函数中，会创建系统启动进程（ID1，优先级24）和主线程（ID2）。主线程中先调用第二步初始化函数，然后创建了1个控制台派遣线程（ID17，优先级24），创建了4个控制台线程（ID18、19、20、21，优先级24），然后主线程优先级降为0，进入死循环。

**3、对EOS启动过程中的相关问题思考**

在操作系统启动前，会进行硬件自检和初始化，我对这个过程中系统引导的安全性感到好奇，想知道EOS系统在启动过程中是如何保障计算机系统的安全性，防止恶意软件在启动过程中注入计算机系统的。

通过阅读实验指导书并查阅相关资料，我知道了操作系统在启动时会进行硬件自检和初始化，许多系统支持UEFI（统一可扩展固件接口），它提供了更强大的引导和安全功能。UEFI引入了"安全引导"，可以确保只有数字签名的引导加载程序才能运行。这可以防止未经授权的引导加载程序的执行。

引导加载程序可以执行启动前的完整性检查。这些检查可能包括验证引导加载程序的数字签名，检查引导加载程序的哈希以确保其完整性等。

除此之外，也可以从启动介质的角度考虑安全性，引导设备，通常是硬盘或固态硬盘，通常设置为只读，以防止在运行时被修改。这可以防止恶意软件将自身或修改引导加载程序注入到引导设备上。

操作系统和引导加载程序通常支持故障安全机制。如果启动过程中检测到异常或攻击，系统可能会采取措施，如自动恢复到安全状态，锁定关键操作或要求管理员干预。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期：2023 年10月17日

**实验名称：操作系统实验2 线程状态及转换（2分）**

**实验目的：**以一个教学型操作系统EOS为例，熟悉线程状态及其转换，理解线程状态转换与线程调度的关系；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。

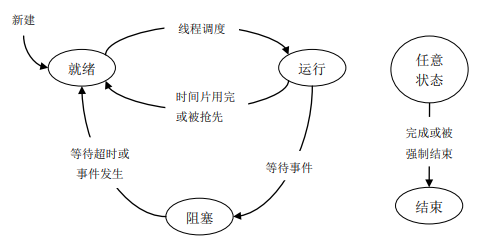
**实验内容：**

跟踪EOS线程在各种状态间的转换过程，分析EOS中线程状态及其转换的相关源代码，描述EOS定义的线程状态以及状态转换的实现方法；修改EOS的源代码，为线程增加挂起状态。

**1）EOS线程状态转换过程的跟踪与源代码分析**

（分析EOS中线程状态及其转换的核心源代码，总结EOS定义的线程状态以及状态转换的实现方法，包括数据结构和算法等；简要说明在本部分实验中完成的主要工作）

线程在其整个生命周期中会在多个不同的状态之间进行转换。EOS线程的状态由线程控制块TCB中的State域保存。EOX线程的状态和其转换过程示意图如下：



下面对线程的转换过程简单介绍：

a、新建->就绪：当创建一个进程或线程时，新进程的主线程或者新线程都会被初始化为就绪状态，并被放入就绪队列中。

b、就绪->运行：当调度程序认为某个处于就绪状态的线程应当执行时，便使其成为当前运行的线程，该线程就会从就绪状态进入运行状态。

c、运行->就绪：当前运行线程因时间片用完或被更高优先级线程抢先时，当前运行线程就会由运行状态转入就绪状态。

d、运行->阻塞：当前运行线程可能因调用 API 函数 WaitForSingleObject 等待事件或者执行 I/O 请求而被阻塞，从而由运行状态转入阻塞状态。

e、阻塞->就绪：

f、任意状态->结束状态：线程可以由任意一个状态转入结束状态。例如，线程执行完毕会由运行状态转入结束状态，就绪线程或者阻塞线程如果被强制结束，也会转入结束状态。

**1、EOS中线程状态及转换的核心源代码分析：数据结构**

**（1）线程对象结构体TCB**

该结构定义在ps/psp.h文件中如下：

|  |
| --- |
| typedef struct \_THREAD {  PPROCESS Process; // 线程所属进程指针  LIST\_ENTRY ThreadListEntry; // 进程的线程链表项  UCHAR Priority; // 线程优先级  UCHAR State; // 线程当前状态  ULONG RemainderTicks; // 剩余时间片，用于时间片轮转调度  STATUS WaitStatus; // 阻塞等待的结果状态  KTIMER WaitTimer; // 用于有限等待唤醒的计时器  LIST\_ENTRY StateListEntry; // 所在状态队列的链表项  LIST\_ENTRY WaitListHead; // 等待队列，所有等待线程结束的线程都在此队列等待。  //  // 为了结构简单，EOS没有对内核进行隔离保护，所有线程都运行在内核状态，所以目  // 前线程没有用户空间的栈。  //  PVOID KernelStack; // 线程位于内核空间的栈  CONTEXT KernelContext; // 线程执行在内核状态的上下文环境状态  //  // 线程必须在所属进程的地址空间中执行用户代码，但可在任何进程的地址空间中执行  // 内核代码，因为内核代码位于所有进程地址空间共享的系统地址空间中。  //  PMMPAS AttachedPas; // 线程在执行内核代码时绑定进程地址空间。  PTHREAD\_START\_ROUTINE StartAddr; // 线程的入口函数地址  PVOID Parameter; // 传递给入口函数的参数  ULONG LastError; // 线程最近一次的错误码  ULONG ExitCode; // 线程的退出码  } THREAD; |

本实验中常用的包括：Process域指向线程所依附的进程，KernelStack域指向栈在内存中的位置，KernelContext域中保留线程被中断执行时的处理器的上下文，Priority是线程优先级，State是线程当前状态，RemainderTicks是剩余时间片，用于时间片轮转调度 。

**（2）线程的状态**

该结构定义在ps/psp.h文件中如下：

|  |
| --- |
| typedef enum \_THREAD\_STATE {  Zero, // 0  Ready, // 1  Running, // 2  Waiting, // 3  Terminated // 4  } THREAD\_STATE; |

其中，zero为线程状态转换过程中的中间状态，ready为就绪状态，running为运行状态，waiting为等待状态，terminated为结束状态。

**（3）记录处于某个状态的所有线程的数据结构**

该结构定义在sched.c文件中内容包括：

a、LIST\_ENTRY PspReadyListHeads[32];

32个链表头组成的数组，分别对应了0~31的32个优先级的就绪队列。下标为n的链表对应优先级为n的就绪队列。

b、volatile ULONG PspReadyBitmap = 0;

32位就绪图，位图的第n位为1，说明优先级为n的就绪队列非空。

c、LIST\_ENTRY PspSleepingListHead;

睡眠等待线程队列，线程调用Sleep后，在这个队列中进行等待。

d、LIST\_ENTRY PspTerminatedListHead;

已结束线程队列。

e、volatile PTHREAD PspCurrentThread = NULL;

当前运行线程。

**（4）等待计数器**

定义在ke.h文件中，用于在等待超时后调用回调函数。

|  |
| --- |
| typedef struct \_KTIMER  {  ULONG IntervalTicks;  ULONG ElapsedTicks;  PKTIMER\_ROUTINE TimerRoutine;  ULONG\_PTR Parameter;  LIST\_ENTRY TimerListEntry;  }KTIMER, \* PKTIMER; |

相关操作包括：

a、初始化计时器KeInitializeTimer函数。

设置内核计时器结构体指针、计时时间间隔、计时时间到达后要调用的回调函数、要传递给回调函数的参数。

b、注册计时器KeRegisterTimer函数。

注册已初始化的内核计时器，仅仅将计时器插入计时器链表的头部。

c、注销计时器KeUnregisterTimer函数。

仅仅将计时器从计时器链表中移除。

**（5）静止就绪状态的线程队列**

定义在psspnd.c文件中，LIST\_ENTRY SuspendListHead = {&SuspendListHead, &SuspendListHead};

**2、EOS中线程状态及转换的核心源代码：算法**

**（1）PspReadyThread函数**

该函数定义在sched.c文件中，使zero状态或者运行状态的线程转入就绪状态。从上文写的数据结构可以看出，与就绪线程有关的数据结构有3个——就绪队列、就绪位图、TCB的状态项。

a、调用ListInsertTail函数，在优先级为Thread->Priority的就绪队列PspReadyListHeads[Thread->Priority]的末尾插入Thread的链表项。

b、BIT\_SET(PspReadyBitmap, Thread->Priority);设置该优先级在32位就绪位图PspReadyBitmap的对应位为1.

c、Thread->State = Ready;更新TCB的状态项为Ready。

**（2）PspUnreadyThread函数**

该函数定义在sched.c文件中，函数用于取消线程的就绪状态，使线程转入 Zero 状态，传入参数线程指针Thread。与PspReadyThread类似，与就绪线程有关的数据结构有3个——就绪队列、就绪位图、TCB的状态项。

a、ListRemoveEntry(&Thread->StateListEntry);把Thread的链表项从就绪队列移除。

b、BIT\_CLEAR(PspReadyBitmap, Thread->Priority);如果线程优先级对应的就绪队列变为空，则清除就绪位图中对应的位。

c、Thread->State = Zero;更新TCB的状态项为Zero。

**（3）PspWait函数**

该函数定义在sched.c文件中，当前线程按照FCFS的原则插入指定的等待队列的队尾，线程阻塞等待直到等待超时或者PspWakeThread被调用。

a、转换为等待状态

ListInsertTail(WaitListHead, &PspCurrentThread->StateListEntry);当前运行线程的链表项进入等待队列。

PspCurrentThread->State = Waiting;TCB的状态项变等待。

b、如果不是永久等待：先调用KeInitializeTimer函数（在ktimer.c文件定义）初始化内核计时器结构体，计时时间到达后产生中断，调用回调函数PspOnWaitTimeout。其中，回调函数PspOnWaitTimeout（定义在sched.c）调用PspUnwaitThread、PspReadyThread，将该线程从等待状态变为就绪状态。然后调用KeRegisterTimer，注册已初始化的内核计时器，将计时器插入计时器链表的头部。

c、最后需要改变PspCurrentThread->WaitStatus。

d、PspThreadSchedule();线程调度。定义在ke.h，表示指令int $48。

**（4）PspUnwaitThread函数**

该函数定义在sched.c文件中，使处于等待状态的线程脱离等待队列并转入zero状态，涉及的数据结构包括：等待队列、TCB的状态项、等待计时器。

1. 利用ListRemoveEntry函数将线程从等待队列中移除，并修改状态为zero
2. 如果线程注册了等待计时器，则通过KeUnregisterTimer函数注销等待计时器。

**（5）PspWakeThread函数**

该函数定义在sched.c文件中，用于唤醒指定等待队列的队首线程，

a、调用 PspUnwaitThread 函数脱离阻塞状态:PspUnwaitThread(Thread);

b、调用PspReadyThread 函数进入就绪状态:PspReadyThread(Thread);

c、获取被唤醒线程从PspWait返回的返回值：Thread->WaitStatus = WaitStatus;

**（6）PspSelectNextThread函数**

该函数定义在sched.c文件中，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）。

a、PspSelectNextThread被调用的场合有2个：由外部设备触发的各种硬中断（定时计数器的定时中断、键盘中断）

b、KeThreadSchedule函数

该函数定义在ke/i386/int.asm的Int\_48 标号所在行。处理 48 号软中断，没有进入 Interrupt 函数，而是调用了IntEnter 、 IntExit 函数，执行函数 PspSelectNextThread，从而选择一个“就绪”线程开始运行。

c、BitScanReverse(&HighestPriority, PspReadyBitmap);扫描就绪位图，获得当前最高优先级。

d、如果存在比当前运行线程优先级更高的就绪线程，当前运行线程变为就绪。

**（7）PsSuspendThread函数**

该函数定义在ps/psspnd.c文件中，挂起指定的线程。

a、关中断：IntState = KeEnableInterrupts(FALSE);

b、将处于就绪状态线程变为Zero状态，并移出就绪队列：PspUnreadyThread(Thread);

c、插入挂起线程队列的末尾ListInsertTail(&SuspendListHead, &Thread->StateListEntry);此时，TCB状态为zero。

d、开中断：KeEnableInterrupts(IntState);

**（8）KeEnableInterrupts函数**

该函数定义在hal1386.asm的164行，cmp dword [ebp + 8], 0;是因为[ebp + 8]是传入的参数EnableInt。

a、EnableInt=0表示不许中断，于是执行cli禁用硬件中断。

b、EnableInt=1表示允许中断，于是sti启动硬件中断。

**3、结合有关数据结构和算法，总结线程的状态及转换方法**

**（1）某状态->就绪：PspUnreadyThread**

将线程从就绪队列中移除，状态码变为 Zero。必须先调用这个函数使线程脱离就绪状态，再将就绪线程转入运行状态或结束处于就绪状态的线程。

**（2）就绪->zero：PspReadyThread**

将线程插入对应优先级的就绪队列队尾，状态码变为 Ready。一般都通过调用此函数使线程进入就绪状态，除了当前运行线程因被抢先而进入就绪状态的情况外。

**（3）PspWait**

当前运行线程插入指定等待队列的队尾，状态码改为 Waiting，线程调度。如果等待超时则会被系统自动唤醒进入就绪状态。运行线程都通过该函数变为阻塞状态。

**（4）等待->zero：PspUnwaitThread**

从等待队列移除线程，状态码修改为 Zero。必须先调用这个函数使线程脱离阻塞状态，再使阻塞线程进入就绪状态或结束处于阻塞状态的线程。

**（5）等待->就绪：PspWakeThread**

PspWakeThread函数先调用 PspUnwaitThread 函数，脱离阻塞状态，然后再调用PspReadyThread 函数进入就绪状态。线程被唤醒后将从函数 PspWait 返回并继续运行。PspWakeThread 的第二个参数将作为被唤醒线程从 PspWait 返回时的返回值。

**（6）线程调度程序函数：PspSelectNextThread**

使被抢先的线程从运行状态进入就绪状态，并决定哪个就绪线程应该进入运行状态。任何线程进入运行状态都是这个函数执行的结果。

该函数先扫描就绪位图，获得当前最高优先级。如果存在比当前运行线程优先级更高的就绪线程，当前运行线程变为就绪。然后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，换入线程绑定运行的地址空间。

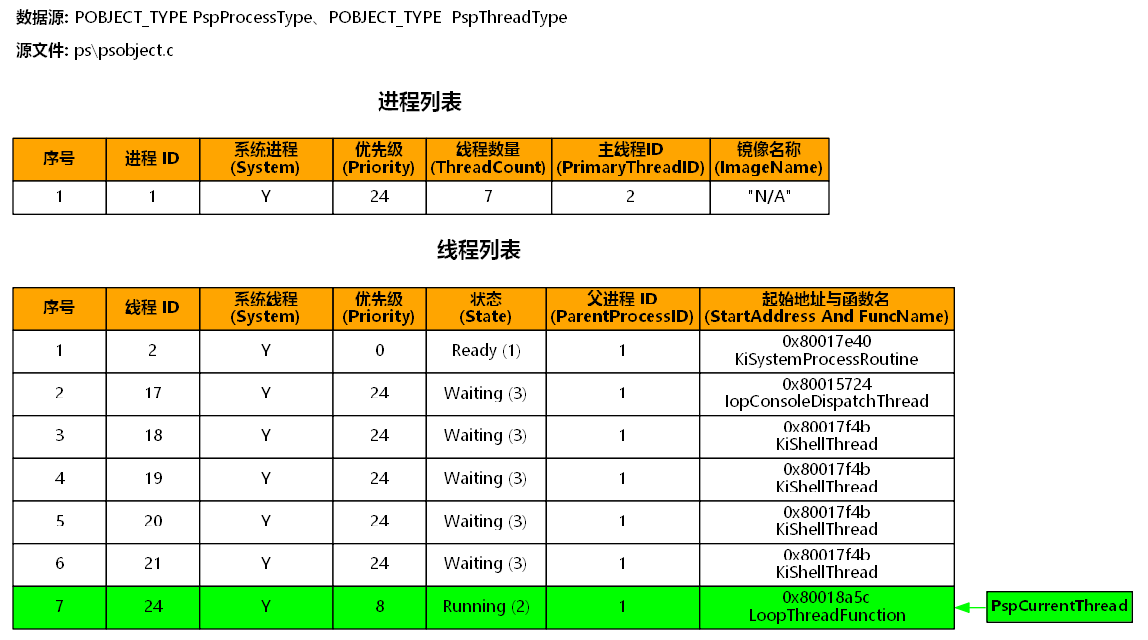
**4、简要说明在本部分实验过程中完成的主要工作**

**（1）调试线程状态的转换过程**

a、F7生成项目，F5调试，输入loop，看到计数在增加。



b、在ke/sysproc.c 文件的 LoopThreadFunction 函数中添加断点，F5调试，输入loop，在中断处中断。查看“进程线程”窗口。



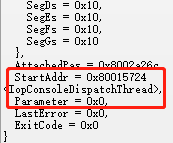
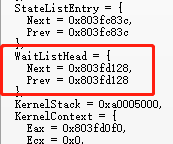
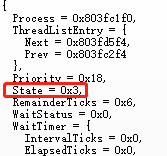
c、系统空闲线程处于就绪状态，其优先级为0；控制台派遣线程和所有控制台线程处于阻塞状态，其优先级为24；只有优先级为8的loop线程处于运行状态，能够在处理器上执行。

d、删除原有断点，在ps/sched.c文件中与线程状态转换相关的函数中添加断点，F5继续执行。loop在不停执行。



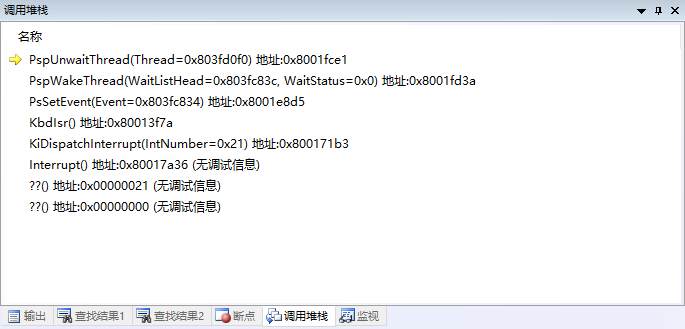
**（2）控制台派遣线程由阻塞状态进入就绪状态**

a、按下空格，在PspUnwaitThread函数中断，快速监视\*Thread，查看TCB。

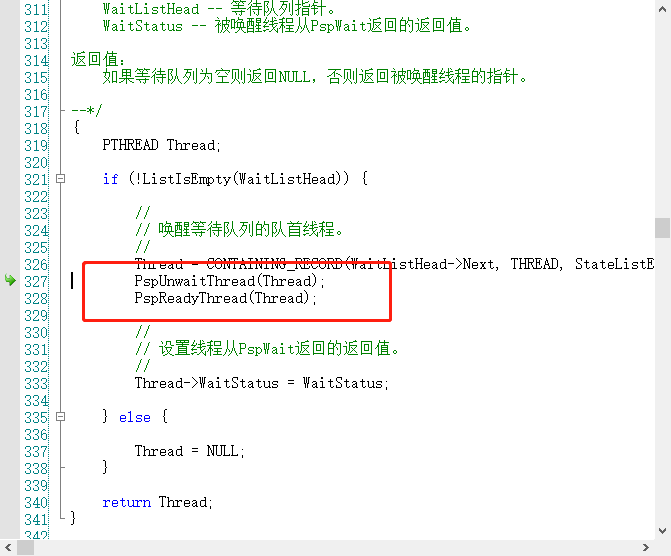


其中，State域的值为3（Waiting），双向链表项StateListEntry的Next和Prev指针的值都不为0，说明这个线程还处于阻塞状态，并在某个同步对象的等待队列中；StartAddr域的值为IopConsoleDispatchThread，说明在PspUnwaitThread函数中要处理的线程就是控制台派遣线程。

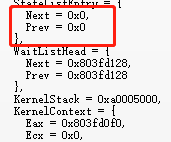
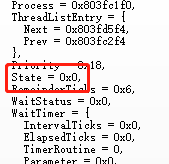
b、激活调用堆栈，查看当前的调用堆栈。



c、查看PspWakeThread函数的堆栈顶。

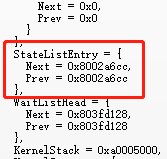
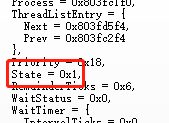


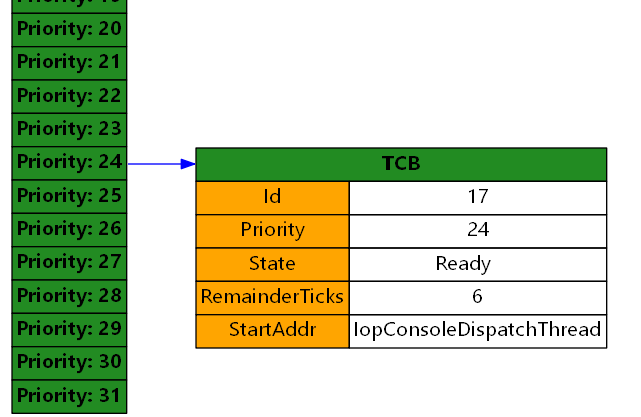
d、查看PspUnwaitThread函数的堆栈顶，F10单步调试到函数的最后，观察\*Thread。



其中，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。

e、F5继续执行，在PspReadyThread函数中断，F10单步调试至函数的最后，查看\*Thread。

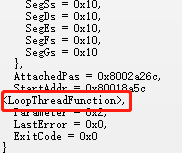
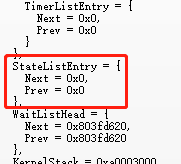
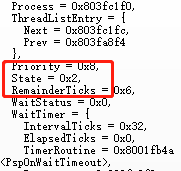




其中，State域的值为1（Ready），双向链表项StateListEntry的Next和Prev指针的值都不为0，说明这个线程已经处于就绪状态，并已经被放入优先级为24的就绪队列中。

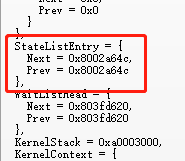
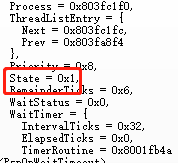
**（3）loop线程由运行进入就绪状态**

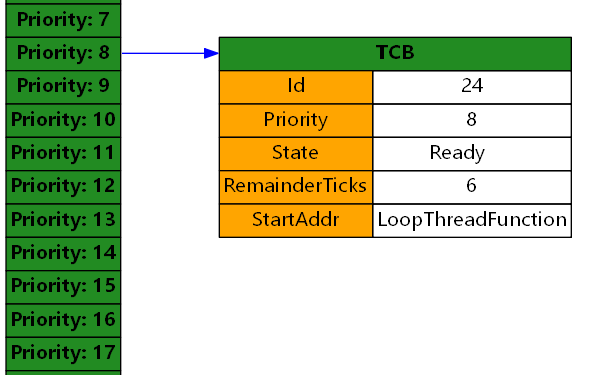
a、F5继续调试，在PspSelectNextThread函数中断，快速监视\*PspCurrentThread。



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，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域的值为LoopThreadFunction，说明这个线程就是loop线程。

b、F10单步调试，当前线程的操作完成，查看\*PspCurrentThread和就绪线程队列。

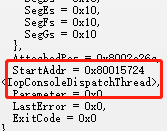
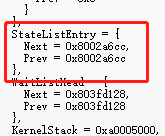
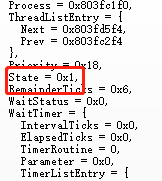




其中，State 域的值为 1（Ready），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都不为0，说明 loop 线程已经进入了就绪状态，并已经被放入优先级为 8 的就绪队列中。刷新“就绪线程队列”窗口中，可以看到在优先级为 8 的就绪队列中已插入线程ID 为 24 的线程控制块。

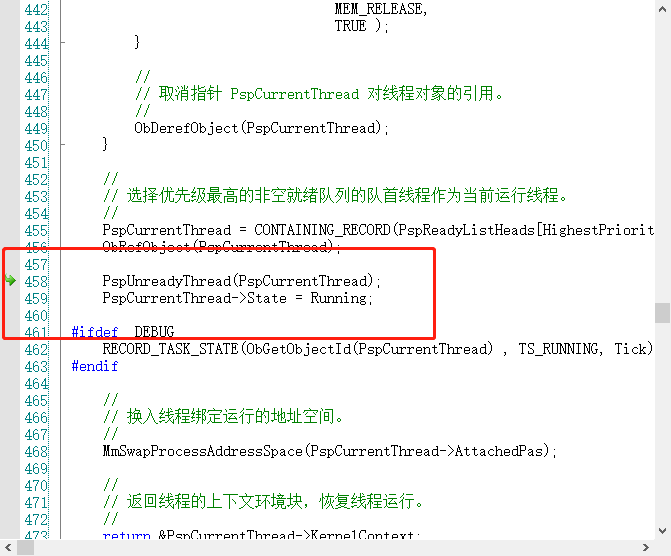
**（4）控制台派遣线程由就绪状态进入运行状态**

a、F5继续调试，在PspUnreadyThread函数中断，查看\*Thread。

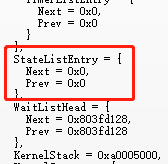
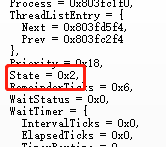


其中，State 域的值为 1（Ready），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都不为 0，说明这个线程处于就绪状态，并在优先级为24 的就绪队列中；StartAddr 域的值为 IopConsoleDispatchThread，说明这个线程就是控制台派遣线程。

b、激活PspSelectNextThread函数对应的堆栈顶。



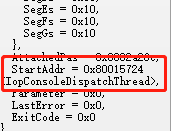
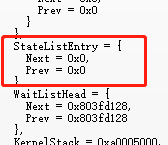
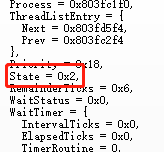
1. 激活 PspUnreadyThread 函数对应的堆栈项，F10单步调试，直到返回 PspSelectNextThread 函数并将线程状态修改为 Running，查看\*PspCurrentThread。



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。

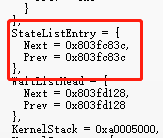
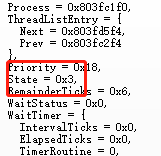
**（5）控制台派遣线程由运行状态进入阻塞状态**

a、F5继续执行，在PspWait函数中断，查看\*PspCurrentThread。



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。

b、F10单步调试，到达255行，查看\*PspCurrentThread。



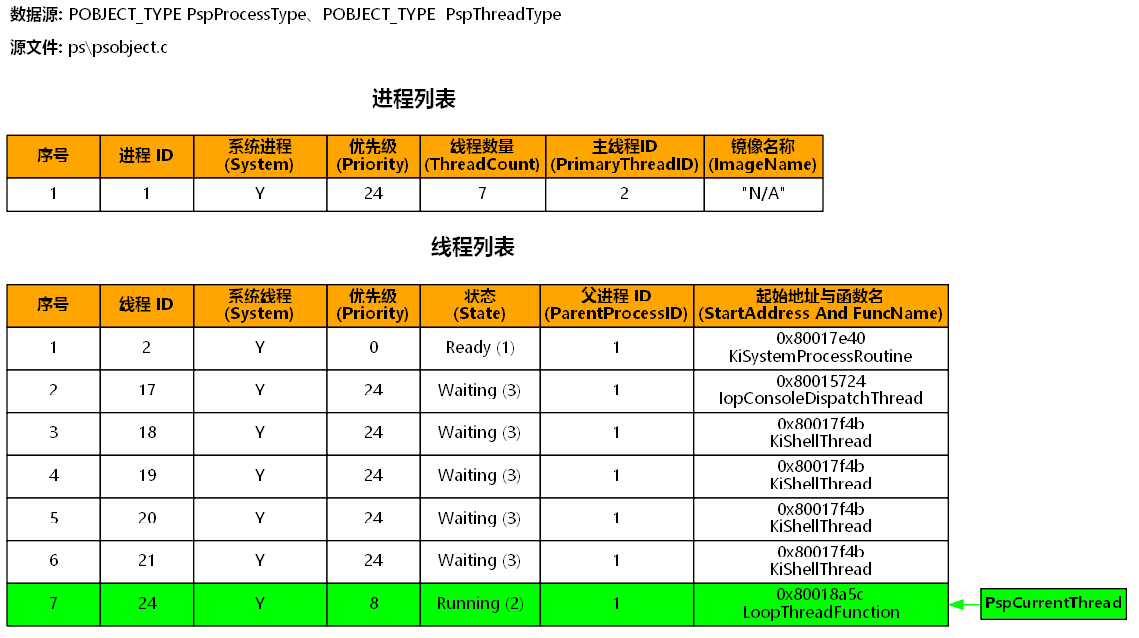
其中，State 域的值为 3（Waiting），双向链表项 StateListEntry 的 Next 和 Prev 指针的值都不为 0，说明控制台派遣线程已经处于阻塞状态了，并在某个同步对象的等待队列中。

**2）为线程增加挂起状态**

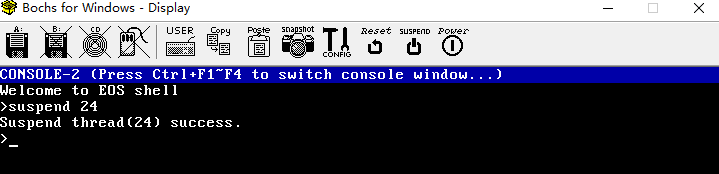
（给出实现方法的简要描述、源代码、测试及结果等）

**（1）测试已经实现的suspend命令**

a、开始调试，输入loop，查看loop线程的ID。



b、切换到控制台2，输入suspend 24。



c、切换到控制台，查看loop，不在增加。

**（2）完成Resume原语**

**A、简要描述**

根据描述，找到要修改的位置，即PspResumThread函数的119行。分析该函数的源代码：

1. 调用ObRefObjectByHandle，根据线程句柄获得线程对象的指针。
2. IntState = KeEnableInterrupts(FALSE)关中断，保证原子性。如果当前是ZERO状态，调用ListRemoveEntry把当前线程从挂起队列中移除，调用PspReadyThread当前线程变为就绪状态，执行调度线程PspThreadSchedule，让当前线程有可能获取处理器。
3. KeEnableInterrupts(IntState)开中断。

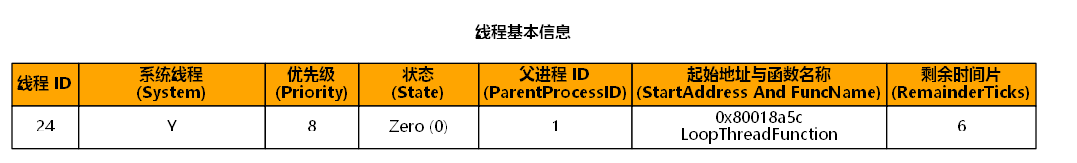
**B、源代码**

|  |
| --- |
| STATUS  PsResumThread(  IN HANDLE hThread  )  {  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) {  //  // 在此添加代码将线程恢复为就绪状态  /\*在 PsResumThread 函数第 119 行需要添加的代码的流程可以是：  1. 首先调用 ListRemoveEntry 函数将线程从挂起线程队列中移除。  2. 然后调用 PspReadyThread 函数将线程恢复为就绪状态。  3. 最后调用 PspThreadSchedule 宏函数执行线程调度，让刚刚恢复的线程有机会执行。  \*/  ListRemoveEntry(&Thread->StateListEntry);  PspReadyThread(Thread);  PspThreadSchedule();    Status = STATUS\_SUCCESS;  } else {  Status = STATUS\_NOT\_SUPPORTED;  }  KeEnableInterrupts(IntState); // 开中断  ObDerefObject(Thread);  }  return Status;  } |

**C、结合断点测试**

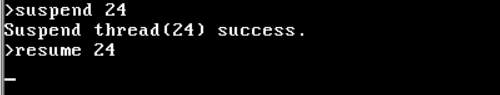
## a、当程序运行到PsSuspendThread函数的return语句：

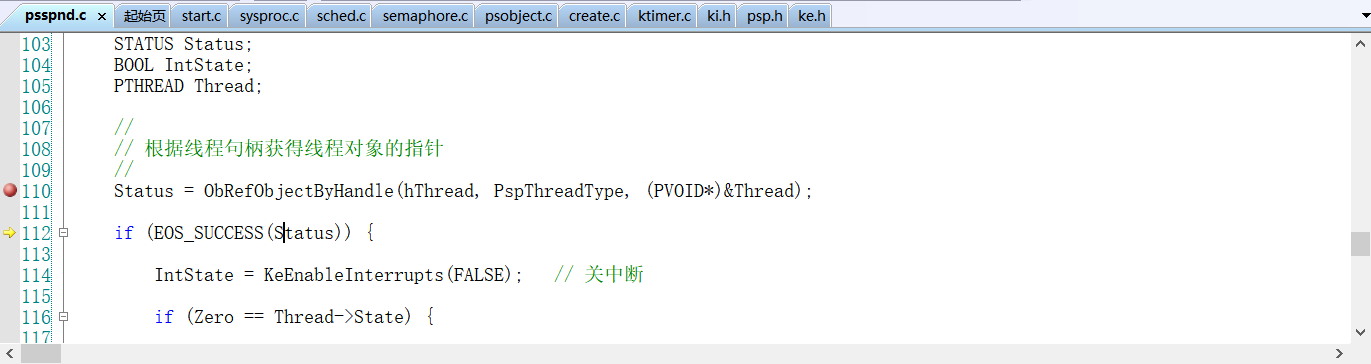
loop线程（startAddr=LoopThreadFuction）变成Zero状态（state=0x0），成功挂起，在挂起队列中（stateListEntry的next,prev不是0）。

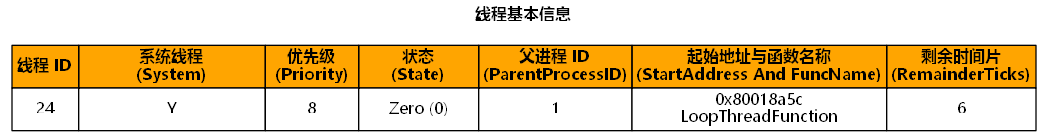


## b、当程序运行到PsResumThread函数的开头：

loop线程（startAddr=LoopThreadFuction）保持Zero状态（state=0x0），在挂起队列中（stateListEntry的next,prev不是0）。



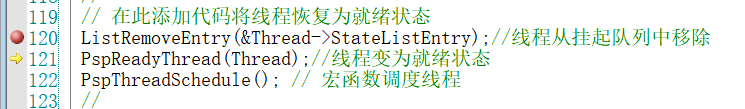


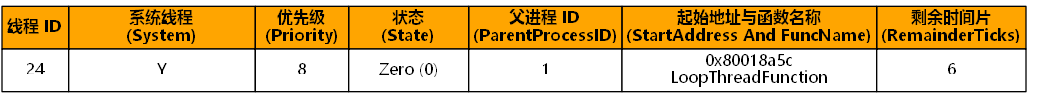




## c、程序执行完PsResumThread函数的ListRemoveEntry(&Thread->StateListEntry)：

loop线程（startAddr=LoopThreadFuction）保持Zero状态（state=0x0），从挂起队列中移除（stateListEntry的next,prev变成0）。

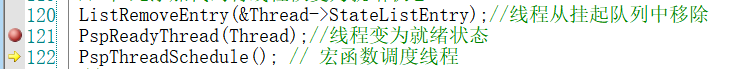


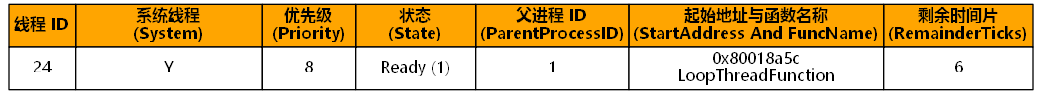


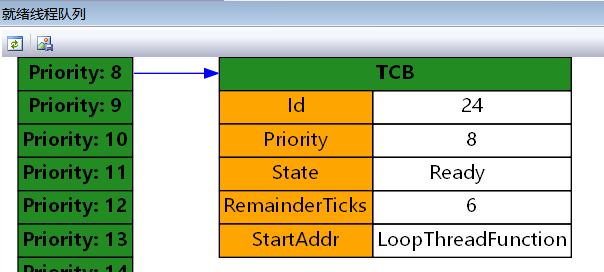


## d、程序执行完PsResumThread函数的PspReadyThread(Thread)：

loop线程（startAddr=LoopThreadFuction）保持Ready状态（state=0x1），进入就绪队列（stateListEntry的next,prev不是0）。







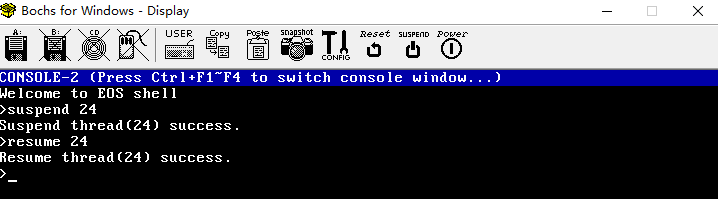
1. **控制台测试**
2. 开始调试，输入loop，计数增加。
3. 切换到控制台2，输入suspend 24。



c、切换到控制台1，loop计数不增加。



d、切换到控制台2，输入resume 。



e、切换到控制台1，loop计数继续增加。

**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS线程状态及其转换方法的特点、不足及改进意见；结合EOS对线程状态及其转换的相关问题提出自己的思考，分析线程状态转换与线程调度的关系；分析为线程增加挂起状态实现方法的有效性、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线程状态及其转换方法**

**（1）EOS线程状态**

在EOS定义的线程状态有5种状态：Zero（线程状态转换过程的中间状态）、Ready（就绪状态）、Running（运行状态）、Waiting（阻塞状态）、Terminted（结束状态）。

五种线程之间可以相互切换。任意状态都可以由本状态到结束状态。可以从就绪到运行，运行到阻塞，运行到就绪，阻塞到就绪。

此外，引入挂起状态，就绪和阻塞又分为活动就绪、活动阻塞、静止就绪、静止阻塞。挂起状态目前也算作zero的一种。目前只实现了静止就绪，没有静止阻塞。

**（2）EOS进程转换的特点**

下面对线程的转换过程简单介绍：

a、新建->就绪：当创建一个进程或线程时，新进程的主线程或者新线程都会被初始化为就绪状态，并被放入就绪队列中。

b、就绪->运行：当调度程序认为某个处于就绪状态的线程应当执行时，便使其成为当前运行的线程，该线程就会从就绪状态进入运行状态。

c、运行->就绪：当前运行线程因时间片用完或被更高优先级线程抢先时，当前运行线程就会由运行状态转入就绪状态。

d、运行->阻塞：当前运行线程可能因调用 API 函数 WaitForSingleObject 等待事件或者执行 I/O 请求而被阻塞，从而由运行状态转入阻塞状态。

e、阻塞->就绪：

f、任意状态->结束状态：线程可以由任意一个状态转入结束状态。例如，线程执行完毕会由运行状态转入结束状态，就绪线程或者阻塞线程如果被强制结束，也会转入结束状态。

**（3）改进意见**

zero状态是线程进行状态转换时的中间状态，可以通过关中断与开中断，将线程转换过程转变为原子操作。此外，各种线程状态之间的转换也可以封装成函数，这样一来可以忽略中间状态，让线程的状态转换过程更加清晰。由于取消zero状态，还需要单独添加挂起状态。

**2、线程状态转换与线程调度的关系**

线程调度依赖于线程状态来做出决策。调度器会优先选择处于就绪状态的线程，以便它们能够运行。如果一个线程进入阻塞状态，调度器将选择另一个就绪线程来执行，而被阻塞的线程将在特定条件满足时返回到就绪状态。线程的状态改变触发了调度器的决策，以确保系统的高效运行。

**（1）线程调度会完成一部分线程状态转换**

a、调度程序认为某个处于就绪状态的线程应当执行时，便使其成为当前运行的线程，该线程就会从就绪状态进入运行状态。

b、线程被更高优先级线程抢先，从运行变为就绪状态，是因为线程调度的存在。

**（2）有些线程状态转换结束后，会执行线程调度**

如PspWait函数，将当前运行线程插入指定等待队列的队尾，并修改状态码为 Waiting，然后执行线程调度，让出处理器。

**3、为线程增加挂起状态分析**

**（1）实现方法的有效性**

a、更灵活的线程管理：挂起状态允许线程在不终止的情况下进入一种休眠或等待状态，而不会占用 CPU 时间。这对于那些需要等待外部事件的线程非常有用，例如等待用户输入或I/O完成的线程。以前，这些线程可能会以不必要的方式占用 CPU 时间。

b、资源节省：线程在挂起状态时，不会被调度执行，这可以降低系统资源的使用率。这意味着更多的线程可以在系统上同时运行，而不会过度竞争 CPU 时间。

c、更快的响应时间：系统中的线程可以更快地响应外部事件，因为它们可以立即从挂起状态切换到就绪状态，而不必等待线程被唤醒后才开始执行。这可以提高系统的响应速度。

d、线程优先级管理：引入挂起状态可以让操作系统更有效地管理线程的优先级。挂起的线程可以根据需要被唤醒，并根据优先级决定其执行顺序。

e、避免忙等待：在以前的模型中，一些线程可能会在某些条件满足之前采用忙等待方式，不断轮询状态，占用大量的 CPU 时间。引入挂起状态后，这些线程可以等待条件满足而不浪费 CPU 资源。

f、更好的资源共享：操作系统可以更好地管理共享资源，因为线程可以在不需要资源时进入挂起状态，而不是一直占用资源。

**（2）实现方法的改进**

a、挂起时限：引入一个可配置的挂起时间限制，允许开发者指定线程挂起的最长时间。一旦时间限制达到，系统自动将线程唤醒。这可以防止线程无限期挂起，从而提高系统的可响应性。

b、条件挂起：允许线程在挂起时指定一个条件，只有当条件满足时才会被唤醒。这可以通过引入条件变量或信号量来实现，以便线程可以等待某些事件的发生。

c、挂起队列：为挂起的线程引入队列，使得线程可以按照特定顺序唤醒。这可以帮助实现更复杂的线程同步和协作。

d、优先级控制：允许挂起线程的优先级发生变化。这意味着线程在唤醒后可以重新进入就绪队列，并根据其新的优先级进行调度。这可以通过实时操作系统的功能来实现。

e、资源管理：改进挂起状态下的资源管理。当线程挂起时，系统可能需要释放其占用的资源，以便其他线程可以使用。当线程恢复时，需要确保它能够重新获取所需的资源。

f、性能优化：考虑如何优化线程挂起和唤醒操作，以降低系统开销。这可以包括采用更高效的数据结构、算法，以及减少上下文切换的开销。

g、错误处理：提供更详细的错误处理机制，以便在出现问题时能够更好地诊断挂起操作的失败原因。

h、用户界面：如果EOS系统是为开发者创建的，那么可以改进用户界面和API，以使线程挂起功能更易于使用和理解。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期： 2023年 10月20日

**实验名称：操作系统实验3 进程同步（4分）**

**实验目的：**以一个教学型操作系统EOS为例，深入理解进程（线程）同步的原理、意义及信号量的含义和实现方法；能对核心源代码进行分析和修改，能运用信号量实现同步问题；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：**

使用EOS的信号量，实现生产者-消费者问题；跟踪EOS信号量的工作过程，分析EOS信号量实现的源代码，理解并阐述其实现方法；修改EOS信号量的实现代码，使之支持等待超时唤醒和批量释放功能。

**1）使用EOS的信号量实现生产者-消费者问题**

（给出使用EOS的信号量解决生产者-消费者问题的实现方法，包括实现方法的简要描述、源代码、测试及结果等）

**1、对实现方法的简要描述**

生产者和消费者需要有同步和互斥的操作。

a、同步：消费者消耗的前提是剩余生产者生产的内容，分别设置empty信号和full信号用于同步，两者都是私有信号量。

b、互斥：使用mutex信号量来实现互斥操作，防止同时对缓冲区内容进行修改。

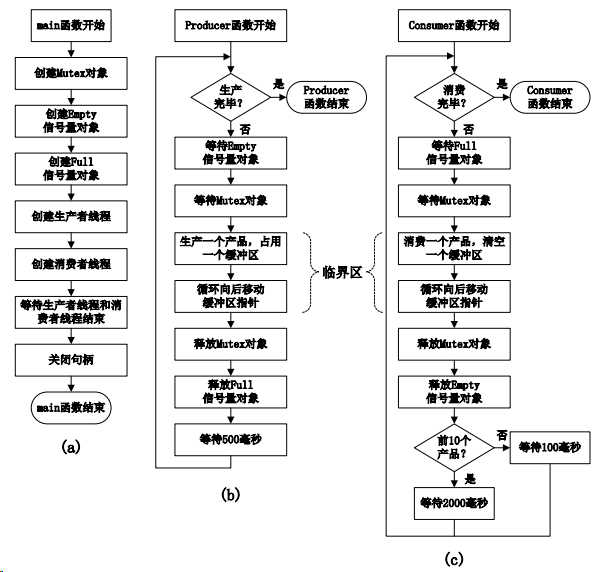
在执行的过程中，每个操作执行完毕后都会有等待的过程，consumer的等待时间为producer的4倍，所以一开始会出现4次生成后才有1次消费的现象。主要涉及到如下这三个函数：

a、main函数：main函数开始后，创建mutex、empty、full对象，创建生产者、消费者线程，等待生产者、消费者线程结束并关闭句柄。

b、producer函数：判断是否生产完毕，若生产完毕则producer函数结束；否则等待empty信号量对象、mutex信号量对象，生产商品、占用缓冲区、循环向后移动缓冲区指针，最后释放mutex对象、full对象并等待500ms，继续检测是否生产完毕。

c、consumer函数：判断是否消费完毕，若消费完毕则consumer函数结束；否则等待full信号量对象、mutex信号量对象，消费商品、清空缓冲区、循环向后移动缓冲区指针，最后释放mutex对象、empty对象并根据是否是前十个产品等待不同的时间，继续检测是否消费完毕。

如下图所示是这三个函数的流程图。



**2、源代码**

2.1 全局变量

|  |
| --- |
| // 缓冲池。  #define BUFFER\_SIZE 10  int Buffer[BUFFER\_SIZE];  // 产品数量。  #define PRODUCT\_COUNT 30  // 用于生产者和消费者同步的对象句柄。  HANDLE MutexHandle;  HANDLE EmptySemaphoreHandle;  HANDLE FullSemaphoreHandle;  // 生产者和消费者的线程函数  ULONG Producer(PVOID Param);  ULONG Consumer(PVOID Param); |

2.2 main函数

|  |
| --- |
| int main(int argc, char\* argv[])  {  // 启动调试 EOS 应用程序前要特别注意下面的问题：  // 1、如果要在调试应用程序时能够调试进入内核并显示对应的源码，  // 必须使用 EOS 核心项目编译生成完全版本的 SDK 文件夹，然  // 后使用此文件夹覆盖应用程序项目中的 SDK 文件夹，并且 EOS  // 核心项目在磁盘上的位置不能改变。  HANDLE ProducerHandle;  HANDLE ConsumerHandle;  // 创建用于互斥访问缓冲池的 Mutex 对象。  MutexHandle = CreateMutex(FALSE, NULL);  if (NULL == MutexHandle) {  return 1;  }  // 创建 Empty 信号量，表示缓冲池中空缓冲区数量。初始计数和最大计数都为 BUFFER\_SIZE。  EmptySemaphoreHandle = CreateSemaphore(BUFFER\_SIZE, BUFFER\_SIZE, NULL);  if (NULL == EmptySemaphoreHandle) {  return 2;  }  // 创建 Full 信号量，表示缓冲池中满缓冲区数量。初始计数为 0，最大计数为 BUFFER\_SIZE。  FullSemaphoreHandle = CreateSemaphore(0, BUFFER\_SIZE, NULL);  if (NULL == FullSemaphoreHandle) {  return 3;  }  // 创建生产者线程。  ProducerHandle = CreateThread( 0, // 默认堆栈大小  Producer, // 线程函数入口地址  NULL, // 线程函数参数  0, // 创建标志  NULL ); // 线程 ID  if (NULL == ProducerHandle) {  return 4;  }  // 创建消费者线程。  ConsumerHandle = CreateThread( 0,  Consumer,  NULL,  0,  NULL );  if (NULL == ConsumerHandle) {  return 5;  }  // 等待生产者线程和消费者线程结束。  WaitForSingleObject(ProducerHandle, INFINITE);  WaitForSingleObject(ConsumerHandle, INFINITE);  // 关闭句柄  CloseHandle(MutexHandle);  CloseHandle(EmptySemaphoreHandle);  CloseHandle(FullSemaphoreHandle);  CloseHandle(ProducerHandle);  CloseHandle(ConsumerHandle);  return 0;  } |

2.3 producer函数

|  |
| --- |
| // 生产者线程函数。  ULONG Producer(PVOID Param)  {  int i;  int InIndex = 0;  for (i = 0; i < PRODUCT\_COUNT; i++) {  WaitForSingleObject(EmptySemaphoreHandle, INFINITE);  WaitForSingleObject(MutexHandle, INFINITE);  printf("Produce a %d\n", i);  Buffer[InIndex] = i;  InIndex = (InIndex + 1) % BUFFER\_SIZE;  ReleaseMutex(MutexHandle);  ReleaseSemaphore(FullSemaphoreHandle, 1, NULL);  // 休息一会。每 500 毫秒生产一个数。  Sleep(500);  }  return 0;  } |

2.4 consumer函数

|  |
| --- |
| // 消费者线程函数。  ULONG Consumer(PVOID Param)  {  int i;  int OutIndex = 0;  for (i = 0; i < PRODUCT\_COUNT; i++) {  WaitForSingleObject(FullSemaphoreHandle, INFINITE);  WaitForSingleObject(MutexHandle, INFINITE);  printf("\t\t\tConsume a %d\n", Buffer[OutIndex]);  OutIndex = (OutIndex + 1) % BUFFER\_SIZE;  ReleaseMutex(MutexHandle);  ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL);  // 休息一会儿。让前 10 个数的消费速度比较慢，后面的较快。  if (i < 10) {  Sleep(2000);  } else {  Sleep(100);  }  }  return 0;  } |

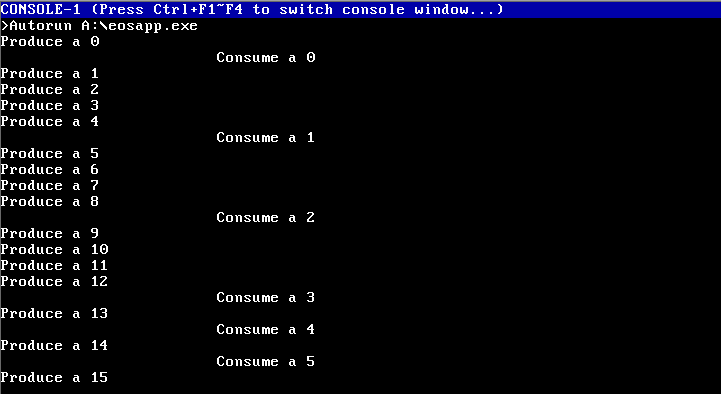
**3、测试**

创建EOS应用程序项目，将pc.c的内容替换EOSApp.c的内容。

测试内容在main函数中，main函数创建mutex、empty、full对象，创建生产者、消费者进程，并等待生产者、消费者进程结束。生产者等待500ms，消费者根据排名先后等待2000ms或者100ms。

因此对于消费者的前十个产品，生产者和消费者的比例是4：1；对于消费者其余的产品，生产者与消费者的比例是1：5。

运行项目，得到如下的结果，可以看到开始时执行4次produce会有1次consume。



**4、结果**

通过查看运行结果，符合测试预期，即对于消费者的前十个产品，生产者和消费者的比例是4：1；对于消费者其余的产品，生产者与消费者的比例是1：5。

**2）EOS信号量工作过程的跟踪与源代码分析**

（分析EOS信号量实现的核心源代码，阐述其实现方法，包括数据结构和算法等；简要说明在本部分实验中完成的主要工作）

**1、EOS信号量实现的核心源代码**

**（1）数据结构**

EOS系统提供了三种用于线程同步的对象：互斥对象mutex、信号量对象semaphore、事件对象event。这些对象有两种状态：signaled和nonsignaled，县城对于处于momsignalewd状态的对象执行wait操作会被阻塞，直到同步对象的状态变为signaled。

a、Mutex对象：用于同步多个线程对临界资源的互斥访问。Mutex对象包括一个持有线程指针、一个递归计数器和一个等待队列，其结构如下：

|  |
| --- |
| typedef struct \_MUTEX {  PVOID OwnerThread; // 当前拥有 Mutex 的线程指针  ULONG RecursionCount; // 递归拥有 Mutex 的计数器  LIST\_ENTRY WaitListHead; // 等待队列  }MUTEX, \*PMUTEX; |

当持有线程指针为NULL时，Mutex不被任何线程持有，此时Mutex处于signaled状态；当指针指向持有Mutex的线程时，Mutex处于nonsignaled状态。当一个线程对Mutex成功调用了WaitForSingleObject后，Mutex的线程指针就会指向此线程，此线程就持有了Mutex，Mutex变为nonsignaled状态。此时其它线程再对同一个Mutex调用WaitForSingleObject，将会被阻塞在Mutex的等待队列中，直到持有该Mutex的线程调用ReleaseMutex释放Mutex使之变为signaled状态。

b、Semaphore对象：记录型信号量，当Semaphore对象中的整形变量大于0时，其处于signaled 状态，当整型变量小于等于0时，其处于nonsignaled状态。其结构如下：

|  |
| --- |
| typedef struct \_SEMAPHORE {  LONG Count; // 信号量的整形值  LONG MaximumCount; // 允许最大值  LIST\_ENTRY WaitListHead; // 等待队列  }SEMAPHORE, \*PSEMAPHORE; |

c、Event对象：弹性同步对象，Event的两种状态signaled和nonsignaled可完全由程序控制，使用非常灵活。其结构如下：

|  |
| --- |
| typedef struct \_EVENT {  BOOL IsManual; // 是否手动类型事件  BOOL IsSignaled; // 是否处于 Signaled 状态  LIST\_ENTRY WaitListHead; // 等待队列  }EVENT, \*PEVENT; |

对Event调用SetEvent可使之变为signaled状态，调用ResetEvent可使之复位为nonsignaled状态。Event分为手动和自动两种类型。

**（2）算法**

**a、wait操作**：通过函数WaitForSingleObject函数实现。

|  |
| --- |
| EOSAPI  ULONG  WaitForSingleObject(  IN HANDLE Handle,  IN ULONG Milliseconds  ); |

WaitForSingleObject中主语句是ObWaitForObject(Handle, Milliseconds);ObWaitForObject中主语句是调用对象类型中注册的Wait操作 Type->Wait(Object, Milliseconds);

以semaphore对象为例，对create和wait进行初始化：

|  |
| --- |
| Initializer.Create = PspOnCreateSemaphoreObject;  Initializer.Wait = (OB\_WAIT\_METHOD)PsWaitForSemaphore; |

查看PsWaitForSemaphore函数，主要操作包括：1、IntState = KeEnableInterrupts(FALSE);关中断；2、Semaphore->Count--;3、判断Semaphore->Count < 0则进行等待；4、KeEnableInterrupts(IntState);恢复中断。

**b、create操作：**如果create函数成功创建了一个对象，便会返回对象的句柄，否则返回NULL。下面以CreateSemaphore为例：内部调用PsCreateSemaphoreObject函数。

|  |
| --- |
| EOSAPI  HANDLE  CreateSemaphore(  IN LONG InitialCount,  IN LONG MaximumCount,  IN PSTR Name  ); |

对于PsCreateSemaphoreObject函数：1、调用ObCreateObject创建PspSemaphoreType类型的信号量对象；2、调用ObCreateHandle函数在当前进程中为对象创建一个句柄。

**c、release操作**：release函数执行成功返回true，否则返回false

|  |
| --- |
| EOSAPI  BOOL  ReleaseSemaphore(  IN HANDLE Handle,  IN LONG ReleaseCount,  IN PLONG PreviousCount  ); |

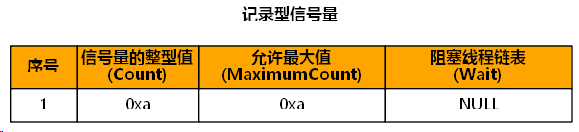
在ReleaseSemaphore函数内部调用PsReleaseSemaphoreObject函数，调用PsReleaseSemaphore函数：1、KeEnableInterrupts(FALSE)关中断；2、Semaphore->Count++;使信号量的值增加 1；3、如果Semaphore->Count <= 0，就PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);从等待队列里唤醒一个等待该信号量的的线程，让它进入就绪队列，使他能够消费这个信号量；4、PspThreadSchedule();线程调度；5、恢复中断KeEnableInterrupts(IntState)

**2、本部分实验过程中完成的主要工作**

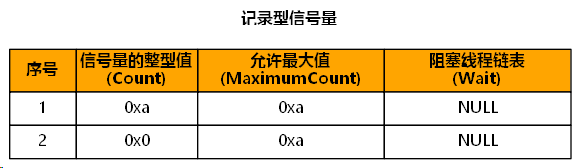
**（1）创建信号量**

a、main函数中创建信号量处添加断点，启动调试，进入CreateSemaphore函数，通过PsCreateSemaphoreObject来创建信号量对象。

b、进入PsCreateSemaphoreObject内部，初始化信号量对象中各个成员的操作是在PsInitializeSemaphore函数中完成的，PsInitializeSemaphore函数添加断点，在断点处观察PsInitializeSemaphore函数中用来初始化信号量结构体成 员的值。查看系统已经创建的信号量，如下图所示：

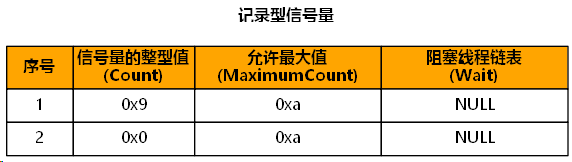


c、同理，继续调试，查看full信号量对象的初始状态如下图所示：



**（2）等待信号量（不阻塞）**

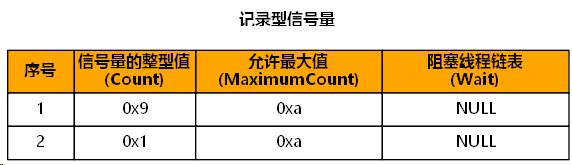
a、生产者函数中，等待empty信号量处添加断点，进入WaitForSingleObject函数，PsWaitForSemaphore处添加断点，当完成PsWaitForSemaphore时，查看记录型信号量如下图所示，此次执行没有等待，只是empty由10变为9。



**（3）释放信号量（不唤醒）**

a、在producer中的ReleaseSemaphore(FullSemaphoreHandle, 1, NULL)函数处加入断点，该函数的作用相当于V（Full）。在对V（Full）操作时由于当前的Full信号大于0。在当前阻塞线程链表中并没有内容，所以这次释放信号量并不能唤醒其他线程。

b、单步调试，执行到PsReleaseSemaphoreObject结束。刷新“记录型信号量”，此时可以看到此时的第二个信号量的内容从0变为1。说明完成V原语的加1。



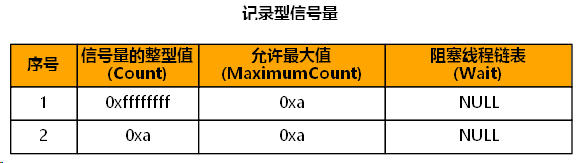
**（4）等待信号量（阻塞）**

由于开始时生产者线程生产产品的速度较快，而消费者线程消费产品的速度较慢，所以当缓冲池中所有的缓冲区都被产品占用时，生产者再生产新的产品时就会被阻塞。这个是一定可以到达的，这是因为在一个限定生产者和消费者等待的时间不同。

a、设断点在if (Semaphore->Count < 0) {PspWait(&Semaphore->WaitListHead, INFINITE);}如果可以在此击中断点就可以说明当前P原语需要将线程阻塞。

b、调试到断点处,查看“调用堆栈”，上层的堆栈是producer函数，证明是生产者时进入到该函数中。双击 Producer 函数所在的堆栈帧,绿色箭头指向等待 Empty 信号量的代码行,查看 Producer 函数中变量 i 的值为 14,表示生产者线程正在尝试生产 14 号产品。在bochs的输出结果中也可以看出生产者生产第13号产品后就停止了。

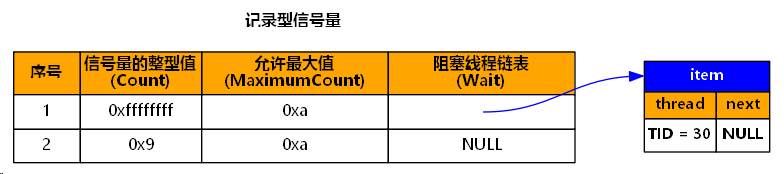
c、执行函数的内容，再次打开“记录型信号量”窗口，可以看出此时“记录型信号量”窗口中已经有wait的内容。此时的Empty信号量已经变成-1，表示要产生等待的过程。



**（5）释放信号量（唤醒）**

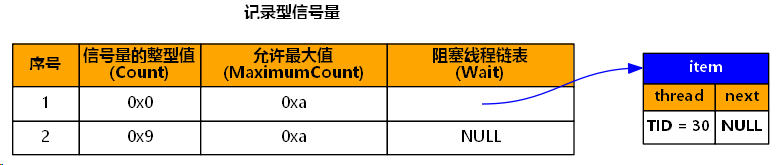
已经有一个线程被阻塞在等待队列中，产生一个空缓冲区后，生产者线程才会被唤醒生产 14 号产品。

a、在ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL);处加入断点。调试到断点，看“记录型信号量”窗口，内容和之前的是相同的。因为刚进入V原语，还没有将Empty信号进行加1操作。



b、在PsReleaseSemaphore函数的PspWakeThread函数处加入断点。根据前面的判断语句可以看出只有当Empty信号小于等于0时才会执行。该函数的作用就是将等待队列中的第一个线程释放。

c、执行到PspWakeThread中断处，并继续向下执行。再次打开“记录型信号量”窗口。可以看到此时Empty的等待队列中已经没有内容。表明一个完整的V原语操作已经完成。生产者被阻塞的线程已经被释放。



**3）支持等待超时唤醒和批量释放功能的信号量实现**

（给出实现方法的简要描述、源代码、测试及结果等）

**1、简要描述**

**a、PsWaitForSemaphore函数**

增加Milliseconds功能，等待不会让信号量-1，Empty信号量的内容只能大于等于0。信号量内容大于0，只需要对信号量减1,表示对信号量P操作，返回状态为成功。如果信号量等于0，调用函数PspWait等待超时上限。

**b、PsReleaseSemaphore函数**

循环调用PspWakeThread唤醒线程，直到达到ReleaseCount次，或者信号量的等待队列已经空了。如果循环了ReleaseCount次，那么信号量的计数量Semaphore->Count回归0，表示所有信号量被占用。如果循环不够ReleaseCount次，等待队列就已经空了，那么Semaphore->Count变为正数，表示有可用的空闲信号量。最后，PspThreadSchedule()线程调度。

c、将Producer、Consumer函数中等待Empty信号量的代码替换，设等待时间300ms，从而检验超时唤醒功能。

d、如果超时等待功能已经能够正常执行，可以考虑将消费者线程修改为一次消费两个产品，来测试ReleaseCount参数是否能够正常使用。使用实验文件夹中NewConsumer.c文件中Consumer函数替换原有的Consumer函数。

**2、源代码**

**a、PsWaitForSemaphore函数**

|  |
| --- |
| STATUS  PsWaitForSemaphore(  IN PSEMAPHORE Semaphore,  IN ULONG Milliseconds  )  {  BOOL IntState;  ASSERT(KeGetIntNesting() == 0); // 中断环境下不能调用此函数。  IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。  //  // 目前仅实现了标准记录型信号量，不支持超时唤醒功能，所以 PspWait 函数  // 的第二个参数的值只能是 INFINITE。  //  /\*  Semaphore->Count--;  if (Semaphore->Count < 0) {  PspWait(&Semaphore->WaitListHead, INFINITE);  }  \*/  STATUS Status;  if(Semaphore->Count>0)  {  Semaphore->Count--;  Status=STATUS\_SUCCESS;  }  else  {  Status=PspWait(&Semaphore->WaitListHead, Milliseconds);  }  KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。  /\*  return STATUS\_SUCCESS;  \*/  return Status;  } |

**b、PsReleaseSemaphore函数**

|  |
| --- |
| STATUS  PsReleaseSemaphore(  IN PSEMAPHORE Semaphore,  IN LONG ReleaseCount,  OUT PLONG PreviousCount  )  {  STATUS Status;  BOOL IntState;  IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。  if (Semaphore->Count + ReleaseCount > Semaphore->MaximumCount) {  Status = STATUS\_SEMAPHORE\_LIMIT\_EXCEEDED;  } else {  //  // 记录当前的信号量的值。  //  if (NULL != PreviousCount) {  \*PreviousCount = Semaphore->Count;  }  //  // 目前仅实现了标准记录型信号量，每执行一次信号量的释放操作  // 只能使信号量的值增加 1。  //  /\*  Semaphore->Count++;  if (Semaphore->Count <= 0) {  PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);  }  \*/  int i;  for(i=0;i<ReleaseCount;i++)  {  if(ListIsEmpty(&Semaphore->WaitListHead)){break;}  PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);  }  if(i==ReleaseCount){Semaphore->Count=0;}  else if(i<ReleaseCount){Semaphore->Count=ReleaseCount-i+(Semaphore->Count);}  //  // 可能有线程被唤醒，执行线程调度。  //  PspThreadSchedule();  Status = STATUS\_SUCCESS;  }  KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。  return Status;  } |

**c、将Producer函数中等待Empty信号量的代码行进行替换**

|  |
| --- |
| ULONG Producer(PVOID Param)  {  int i;  int InIndex = 0;  for (i = 0; i < PRODUCT\_COUNT; i++) {  /\*  WaitForSingleObject(EmptySemaphoreHandle, INFINITE);  \*/  while(WAIT\_TIMEOUT == WaitForSingleObject(EmptySemaphoreHandle, 300)){  printf("Producer wait for empty semaphore timeout\n");  }  WaitForSingleObject(MutexHandle, INFINITE);  printf("Produce a %d\n", i);  Buffer[InIndex] = i;  InIndex = (InIndex + 1) % BUFFER\_SIZE;  ReleaseMutex(MutexHandle);  ReleaseSemaphore(FullSemaphoreHandle, 1, NULL);  //  // 休息一会。每 500 毫秒生产一个数。  //  Sleep(500);  }  return 0;  } |

**d、修改Consumer函数中等待Full信号量的代码**

|  |
| --- |
| ULONG Consumer(PVOID Param)  {  int i;  int OutIndex = 0;  for (i = 0; i < PRODUCT\_COUNT; i++) {  while(WAIT\_TIMEOUT == WaitForSingleObject(FullSemaphoreHandle, 300)){  printf("Consumer wait for full semaphore timeout\n");  }  WaitForSingleObject(MutexHandle, INFINITE);  printf("\t\t\tConsume a %d\n", Buffer[OutIndex]);  OutIndex = (OutIndex + 1) % BUFFER\_SIZE;  ReleaseMutex(MutexHandle);  ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL);  // 休息一会儿。让前 10 个数的消费速度比较慢，后面的较快。  if (i < 10) {  Sleep(2000);  } else {  Sleep(100);  }  }  return 0;  } |

**3、测试**

**（1）测试超时等待功能**

在原生产者-消费者问题的基础上，修改producer函数和consumer函数，修改等待时间从而验证超时唤醒功能。当缓冲区满以后，生产者和消费者的速度还是4:1的时候，每四次生产中，只有1次能成功占用缓冲区，另外3次都会timeout超时。

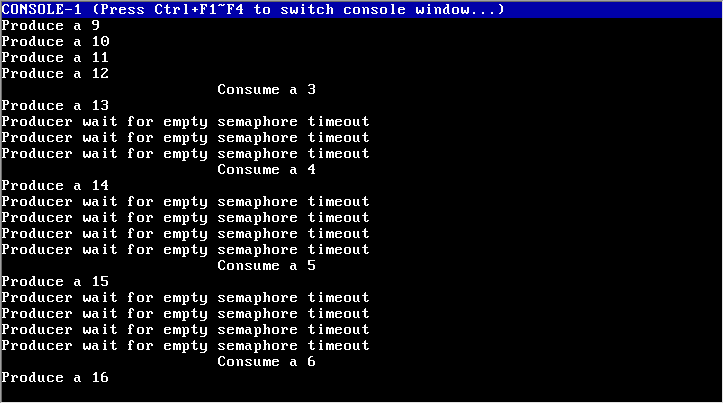
**（2）测试ReleaseCount能否正常使用**

将消费者线程修改为一次消费两个产品，来测试ReleaseCount 参数是否能够正常使用。使用NewConsumer.c文件中的Consumer 函数替换原有的 Consumer 函数。成功实现每次消费2个，释放2个缓冲区。

**4、测试结果**

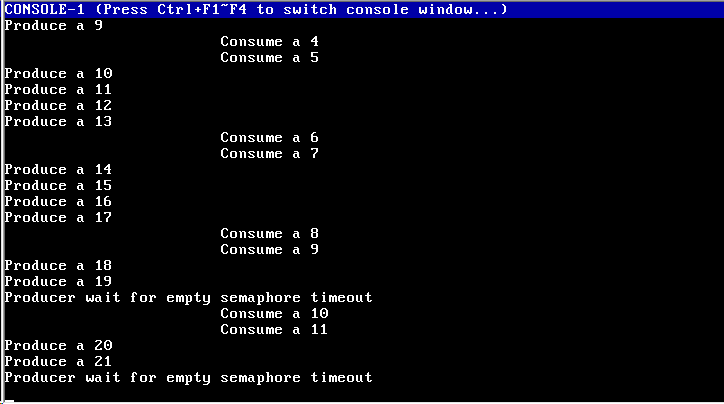
**（1）测试超时等待功能**

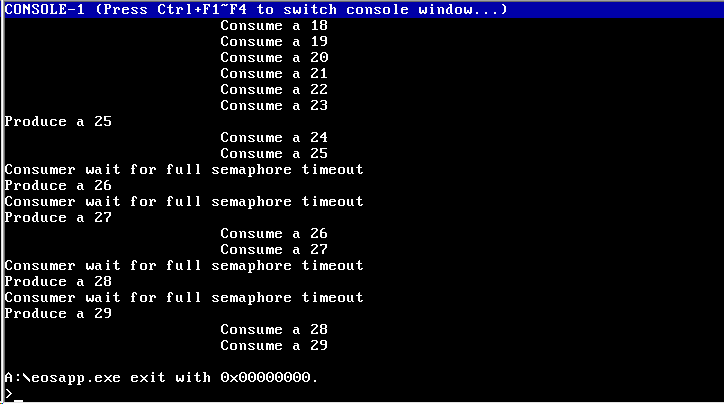
当缓冲区满以后，生产者和消费者的速度还是4:1的时候，每四次生产中，只有1次能成功占用缓冲区，另外3次都会timeout超时。



**（2）测试ReleaseCount能否正常使用**

成功实现每次消费2个，释放2个缓冲区。





**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS信号量实现方法的特点、不足及改进意见；结合EOS对信号量实现的相关问题提出自己的思考；分析支持等待超时唤醒和批量释放功能的信号量实现方法的有效性、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信号量实现方法**

**（1）特点**

**create，wait,release都采用关中断的方式。**

**a、EOS信号量Semaphore**

EOS 中同步对象都有两种状态：signaled和nonsignaled，线程对处于nonsignaled状态的同步对象执行Wait操作将会被阻塞，直到同步对象的状态变为signaled。当Semaphore对象中的整形变量大于0时，其处于signaled状态，当整型变量小于等于0时，其处于nonsignaled状态。

初始化信号量函数为CreateSemaphore，给信号量赋初值、最大值、对象名。

等待信号量函数为WaitForSingleObject，它最终调用的是PsWaitForSemaphore，实现P原语。

释放信号量函数PsReleaseSemaphore，实现V原语。

**b、EOS互斥量mutex**

当持有线程指针为NULL时，Mutex不被任何线程持有，此时Mutex处于signaled状态；当指针指向持有Mutex的线程时，Mutex处于nonsignaled状态。

当一个线程对 Mutex 成功调用了WaitForSingleObject后，Mutex的线程指针就会指向此线程，此线程就持有了Mutex，Mutex变为 nonsignaled状态。此时其它线程再对同一个Mutex调用WaitForSingleObject，将会被阻塞在 Mutex 的等待队列中，直到持有该 Mutex的线程调用ReleaseMutex释放Mutex使之变为signaled状态。

**c、测试**

通过测试生产者消费者程序，验证了有效性。

**（2）不足**

本实验采用关中断实现信号量。关中断实现互斥量只适用于单个处理器的，对多处理器（包括多核处理器）系统无效。在多处理器系统中，有可能存在一个以上的进程在不同处理器上同时执行，关中断是不能保证互斥的。

**（3）改进**

可以采用具有原子性的机器指令代替关中断，比如TestAndSet指令，保证信号量操作过程中的原子性。

**2、结合EOS对信号量实现的相关问题提出自己的思考**

通过本次实验，我意识到，信号量的等待和互斥量的等待的顺序非常重要。

实验中，先等待信号量，如果信号量大于0，说明可以使用信号量，然后才会申请访问临界区，等待互斥量。如果二者顺序反过来，先等待互斥量，成功拥有了互斥量以后，再等待信号量。那么另一个想要释放这个信号量的线程会因为没有互斥量，而一直无法释放信号量。于是陷入死锁。信号量的等待和互斥量的等待的顺序不能更改。

**3、支持等待超时唤醒和批量释放功能的信号量实现方法**

**（1）有效性**

a、PsWaitForSemaphore函数——等待超时唤醒

增加Milliseconds功能，等待不会让信号量-1，Empty信号量的内容只能大于等于0。信号量内容大于0，只需要对信号量减1,表示对信号量P操作，返回状态为成功。如果信号量等于0，调用函数PspWait等待，如果超时会结束等待。

b、 PsReleaseSemaphore函数——批量释放功能

循环调用PspWakeThread唤醒线程，直到达到ReleaseCount次，或者信号量的等待队列已经空了。如果循环了ReleaseCount次，那么信号量的计数量Semaphore->Count回归0，表示所有信号量被占用了。如果循环不够ReleaseCount次，等待队列就已经空了，那么Semaphore->Count变为正数，表示有可用的空闲信号量。最后，PspThreadSchedule()线程调度。

c、测试等待超时唤醒

在原生产者消费者问题基础上，将Producer、Consumer函数中等待Empty信号量的代码替换，设等待时间300，从而检验超时唤醒功能。

测试达到预期，当缓冲区满了以后，生产者和消费者的速度还是4:1的时候，每四次生产中，只有1次能成功占用缓冲区，另外3次都会timeout超时。

综上，证明了有效性。

d、测试超时等待功能

将消费者线程修改为一次消费两个产品，来测试ReleaseCount 参数是否能够正常使用。使用实验文件夹中 NewConsumer.c 文件中的Consumer 函数替换原有的 Consumer 函数。

测试达到预期，成功实现每次消费2个，释放2个缓冲区。

综上，证明了有效性。

**（2）不足**

a、信号量采用关中断实现信号量。关中断实现互斥量只适用于单个处理器的，对多处理器（包括多核处理器）系统无效。在多处理器系统中，有可能存在一个以上的进程在不同处理器上同时执行，关中断是不能保证互斥的。

b、测试设等待时间300，但没有验证等待时间为0的特殊情况。

**（3）改进**

a、可以采用具有原子性的机器指令代替关中断，比如TestAndSet指令，保证信号量操作过程中的原子性。

b、在原生产者消费者问题基础上，将Producer、Consumer函数中等待Empty信号量的代码替换，设等待时间0，从而检验超时唤醒功能的特殊情况。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期： 2023年 11月2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。

**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中完成的主要工作）

在EOS内核中，处理器调度的对象是线程，线程是调度的基本单元。EOS操作系统实现了基于优先级的抢占式调度，而此种正在执行的线程优先级高的线程处于“就绪”状态，利用此种调度会停止当前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到“就绪”状态，直到没有比它优先级更好的就绪线程时，才能重新获得处理器。

**1、相关数据结构**

为了实现基于优先级的抢占式调度，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。利用线程控制块结构体中的Priority域就是用来记录线程优先级的。

|  |
| --- |
| // 32个链表头组成的数组，分别对应了0~31的32个优先级的就绪队列。  // 下标为n的链表对应优先级为n的就绪队列。  LIST\_ENTRY PspReadyListHeads[32]; |

该数组保存了32个链表头，其中下标为n的链表对应的优先级为n的就绪队列，执行线程调度时，率先选择高优先级的就绪队列中的线程获得处理器，同一个优先级就绪队列中的多个线程，则按照先来先服务（FCFS）的顺序及进行调度。

为了更好的寻找优先级高的就绪线程，EOS维护了32位的就绪位图，如果位图的第n位为1，则表明优先级为n的就绪队列非空。通过从高优先级向低优先级扫描这个就绪位图，即可得到优先级的抢占式调度。

|  |
| --- |
| // 32位就绪位图。  // 如果位图的第n位为1，则表明优先级为n的就绪队列非空。  volatile ULONG PspReadyBitmap = 0; |

**2、基于优先级抢占式的线程调度**

线程调度执行的时机一定要用到中断处理，EOS操作系统定义了一个统一的处理函数Interrupt，各种中断都会到此函数进行处理，此函数定义在int.asm函数中，首先Interupt函数会调用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（实验二涉及过），源代码如下表所示。

ke/i386/int.asm IntEnter函数

|  |
| --- |
| ; 中断进入函数。  IntEnter:  ;{  T\_SIZE  mov eax, ebx  .SAVE\_CONTEXT:  ; 将 CPU 现场保存到 eax 指向的 CONTEXT 结构体中  mov [eax + OFF\_ECX], ecx  mov ecx, [\_EaxValue]  mov [eax + OFF\_EAX], ecx  mov ecx, [\_EbxValue]  mov [eax + OFF\_EBX], ecx  mov [eax + OFF\_EDX], edx  mov [eax + OFF\_ESP], esp  mov [eax + OFF\_EBP], ebp  mov [eax + OFF\_ESI], esi  mov [eax + OFF\_EDI], edi  xor ecx, ecx  mov cx, ds  mov [eax + OFF\_DS], ecx  mov cx, es  mov [eax + OFF\_ES], ecx  mov cx, fs  mov [eax + OFF\_FS], ecx  mov cx, gs  mov [eax + OFF\_GS], ecx  mov cx, ss  mov [eax + OFF\_SS], ecx  pop dword [eax + OFF\_EIP]  pop dword [eax + OFF\_CS]  pop dword [eax + OFF\_EFLAGS]  mov [eax + OFF\_ESP], esp  jmp dword [\_RetAddress]  ;} |

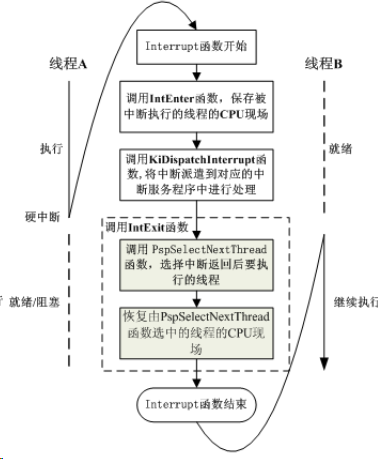
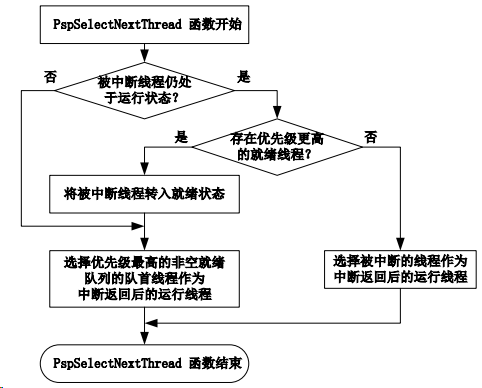
然后调用KiDispatchInterrupt函数根据向量号派遣到对应的中断服务程序中进行处理。

|  |
| --- |
| VOID  KiDispatchInterrupt(  ULONG IntNumber  )  // 根据向量号调用相应的中断处理程序。  switch (IntNumber)  {  case INT\_TIMER: //时钟中断  KiIsrTimer(); //时钟中断处理程序  break;  case INT\_KEYBOARD: //键盘中断  if (NULL != KeIsrKeyBoard)  KeIsrKeyBoard(); //键盘中断处理程序 |

然后调用PspSelectNextThread函数，选择下一个执行的线程，由于是基于优先级的抢占式调度，所以该函数会从就绪队列中找到优先级比当前执行的线程优先级更高的队列继续执行，如果没有比当前线程更高的优先级的就绪线程，则继续返回执行当前线程，源代码分析如下。

|  |
| --- |
| // 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) {  // 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。  if (0 != PspReadyBitmap && HighestPriority > PspCurrentThread->Priority) {  //将该线程插入到对应就绪队列队首  ListInsertHead( &PspReadyListHeads[PspCurrentThread->Priority],  &PspCurrentThread->StateListEntry );  //设置对应的位图  BIT\_SET(PspReadyBitmap, PspCurrentThread->Priority);  PspCurrentThread->State = Ready; //将状态转换为就绪状态  } else {  // 当前线程继续运行。  MmSwapProcessAddressSpace(PspCurrentThread->AttachedPas);  return &PspCurrentThread->KernelContext;  }  // 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)  PsPUnreadyThread(PspCurrentThread) //从就绪队列取下，设置为Zero状态  PspCurrentThread->State = Running; //设置为运行状态  return &PspCurrentThread->KernelContext; //返回线程的上下文，恢复线程运行 |

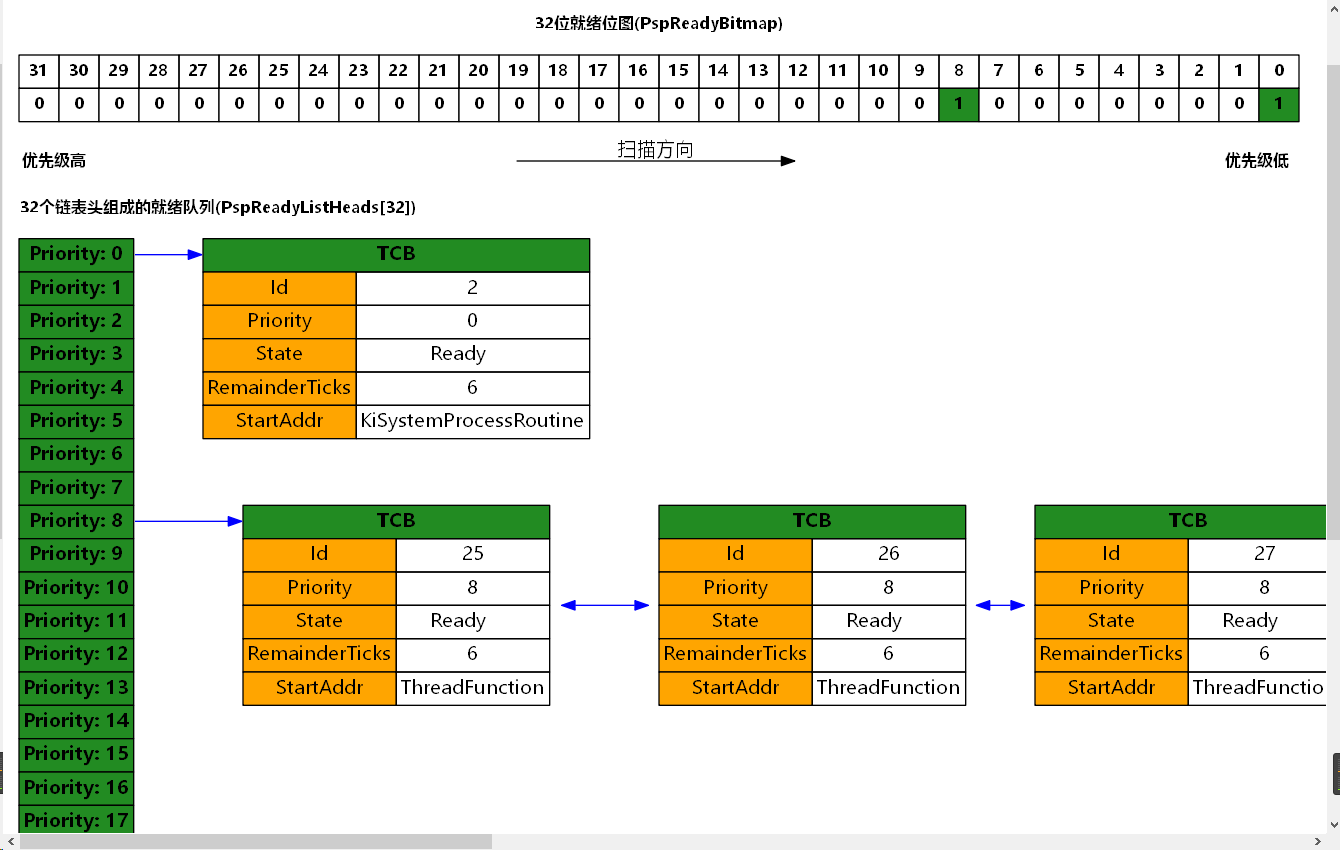
左图为PspSelectNextThread函数的整个执行过程，右图是EOS操作系统实现基于优先级的抢占式调度的方式，具体的流程如下：首先线程A执行，当发生中断后，Interrupt函数开始执行，调用IntEnter函数，保存被中断执行的CPU现场，然后调用KiDispatchInterrrupt函数，将中断派遣到对应的中断服务程序汇总进行处理。然后调用PspSelectNextThread函数，选择中断返回后要执行的线程，恢复由PspSelectNextThread函数选中的线程的CPU现场，继续执行线程B,如果没找到更高优先级的线程，则恢复执行线程A。



**3、主要工作**

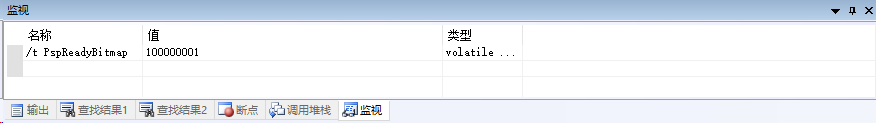
**A、调试当前线程不被抢先的情况**

首先执行rr命令开始创建新的线程，观察对应的就绪队列及优先级位图。



可以看到第0位和第8位为1，其他位都为0，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，其它优先级的就绪队列都为空。

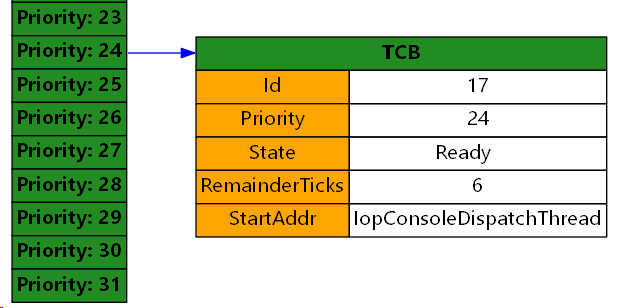
在PspSelectNextThread函数内设置断点后观察就绪位图的值为：



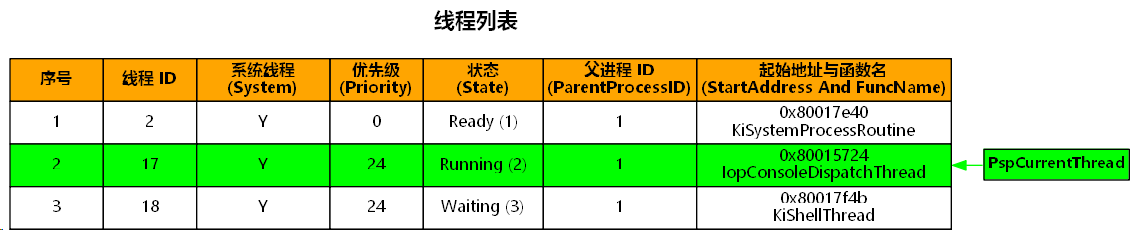
然后在扫描就绪位图后，会将当前最高的优先级放入变量HighestPriority中，通过快速查看变量HighestPriortiy的值为8，说明第0个线程仍然处于运行状态。

**B、调试当前线程被抢占的情况**

当第0个线程正在执行时在控制台按下空格，使之前阻塞状态的控制台派遣线程进入到就绪状态，并触发线程调度函数PspSelectNextThread，会在刚刚添加的断点处中断，观察就绪队列可以看到控制台派遣线程进入到就绪队列中。



由于控制台派遣线程的优先级为24高于优先级8的第0个线程，所以可以发现优先级为 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。



**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）

**1、实现方法**

时间片轮转调度，当同一个优先级就绪队列中的多个线程，为就绪队列中的每一个线程分配一个时间片（Time Slice），当线程调度执行时，把CPU分配给队首线程，待线程的时间片用完之后，会重新为它分配一个时间片，并将其移动到就绪队列的队尾，让新的队首开始执行。这样就可以保障就绪队列中的所有线程，均能获得一个时间片的CPU的执行时间。

**2、源代码**

|  |
| --- |
| VOID  PspRoundRobin(  VOID  )  {  if (NULL != PspCurrentThread && Running == PspCurrentThread->State) {  PspCurrentThread->RemainderTicks--; //时间片减少1  if(PspCurrentThread->RemainderTicks == 0)  {  //为被中断进程重新分配时间片  PspCurrentThread->RemainderTicks = TICKS\_OF\_TIME\_SLICE;  if(BIT\_TEST(PspReadyBitmap, PspCurrentThread->Priority))  {  //若有相同优先级线程，则将被中断线程放入队列最后  PspReadyThread(PspCurrentThread);  }  }  }  return;  } |

**3、测试**

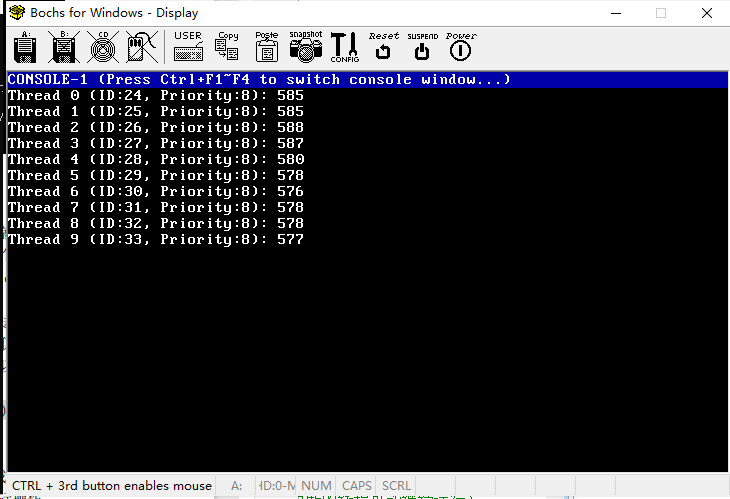
在 EOS 控制台中输入命令“rr”后按回车。

预期：应能看到 10 个线程轮转执行的效果。

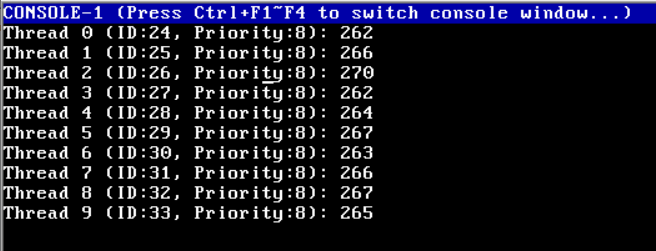
将 ps/psp.h 第 120 行定义的 TICKS\_OF\_TIME\_SLICE 的值修改为 1。在 EOS 控制台中输入命令“rr”后按回车。观察执行的效果。还可以按照上面的步骤为 TICKS\_OF\_TIME\_SLICE 取一些其它的极端值，例如 20 或 100 等，分别观察“rr”命令执行的效果。

**4、结果**

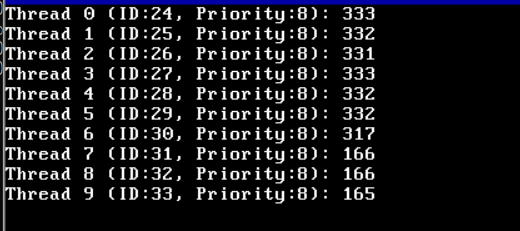
（1）时间片为6，各线程计数差距不大，速度适中



（2）时间片为1，看到 10 个线程轮转执行的效果，线程之间的计数值差距小。



（3）时间片为100，看到 10 个线程轮转执行的效果，线程之间的计数值差距大。



**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线程调度总结**

**（1）特点**

EOS处理器调度的对象是线程，线程是调度的基本单位。采用优先级抢占方式调度。

PspSelectNextThread函数负责优先级抢占的调度机制，该函数每隔几十毫秒就会执行一次，调度的线程是处于就绪队列的线程。

PspSelectNextThread函数先扫描就绪位图，获得当前最高优先级。如果存在比当前运行线程优先级更高的就绪线程，当前运行线程变为就绪。然后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，换入线程绑定运行的地址空间。

**（2）不足**

采用优先级抢占方式，但没有时间片概念，导致“死循环”或耗时多的进程会一直执行，同一优先级的后面的进程会饥饿。

**（3）改进意见**

采用时间片轮转调度，每个运行进程只能执行到时间片耗完为止。然后需要排到该优先级的队尾等待。这样可以避免同优先级的其他进程的饥饿。

**2、线程调度的执行时机和过程**

**A、优先级抢占的线程调度函数PspSelectNextThread执行时机有2个**

**（1）由外部设备触发的各种硬中断（定时计数器的定时中断、键盘中断）**

硬中断进入Interrupt 函数处理。

Interrupt 函数中：

第1步，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。

第2步，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。

第3步，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。

其中，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。

**（2）48 号软中断**

KeThreadSchedule函数（在头文件 inc/ke.h 中定义）就是48号软中断。

48号软中断定义在ke/i386/int.asm的Int\_48 标号所在行。处理 48 号软中断，没有进入 Interrupt 函数，而是调用了IntEnter 、 IntExit 函数，执行函数 PspSelectNextThread，从而选择一个“就绪”线程开始运行。

**B、线程调度过程**

PspSelectNextThread函数负责优先级抢占的调度机制，该函数每隔几十毫秒就会执行一次，调度的线程是处于就绪队列的线程。

PspSelectNextThread函数先扫描就绪位图，获得当前最高优先级。如果存在比当前运行线程优先级更高的就绪线程，当前运行线程变为就绪。然后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，换入线程绑定运行的地址空间。

**3、时间片轮转调度实现方法**

**A、有效性分析**

时间片轮转调度的执行时机：

8253 初始化为每秒钟向 CPU 发送 100 次定时计数器中断，KiDispatchInterrupt 函数会将定时计数器中断派遣到其对应的中断服务程序 KiIsrTimer 函数（在文件ke/ktimer.c 中定义）中处理。KiIsrTimer 函数会依次处理已经注册的各个计时器外，并最后调用KiIsrTimer 函数进行时间片轮转调度。

PspRoundRobin函数中实现时间片轮转调度算法：

（1）如果当前线程PspCurrentThread不是空，再去看当前线程状态。

有效性：这样避免了当前线程PspCurrentThread是空，导致查看PspCurrentThread->State的时候非法访问的可能。

（2）如果当前线程是空，或者当前线程状态如果不是运行状态，就无需进行处理。

有效性：避免了操作系统运行错误的可能。

（3）每次可编程定时计数器过了一个TICK，就会调用一次PspRoundRobin，所以需要对运行状态线程的剩余时间片减1。将当前线程可用时间片和0进行比较。如果时间片小于0表示时间片已经使用完成。如果没有用完继续执行即可。如果当前线程时间片用完了，首先重新赋值时间片为TICKS\_OF\_TIME\_SLICE。然后扫描就绪位图，判断该优先级的就绪队列中是否有线程。如果该优先级的就绪队列中有线程，就调用PspReadyThread让当前线程插入就绪队列队尾，状态码变为就绪。

有效性：不用扫描更高优先级的就绪队列，因为抢占的情况无需时间片调度，在更高优先级的进程被创建或唤醒（比如等待信号量或互斥锁结束了）的时候，就会调用PspSelectNextThread线程调度函数，进行抢占。

**B、不足**

时间片轮转也不能避免低优先级线程饥饿。

**C、改进**

可以使用多级反馈队列，优先级越高，时间片越小。每次运行完，优先级下降一格。从而让低优先级线程不再饥饿。

**北京科技大学实验报告**

学院：计通学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期： 2023 年 11 月 16 日

**实验名称：操作系统实验5 物理存储器与进程逻辑地址空间管理（2分）**

**实验目的：**以一个教学型操作系统EOS为例，深入理解物理存储器以及进程逻辑地址空间的管理方法；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：**

通过查看EOS物理存储器的使用情况，练习物理内存的分配与回收，分析相关源代码；通过查看进程逻辑地址空间的使用情况，练习虚拟内存的分配与回收，分析相关源代码，完成在应用进程中分配虚拟页和释放虚拟页的功能。

**实验步骤：**

**1）EOS物理内存分配和回收的练习以及源代码分析**

（分析相关源代码，阐述EOS中物理存储器的管理方法，包括数据结构和算法等；简要说明在本部分实验中完成的主要工作）

**1、数据结构**

**（1）描述符和描述符表**

描述符的长度是8个字节，描述符表中最多可以包含8192个描述符。描述符属性和描述符表均定义在cpu.asm文件中。

|  |
| --- |
| ; 描述符属性  DA\_32 equ 0x4000 ; 32 位段  DA\_LIMIT\_4K equ 0x8000 ; 段界限粒度为 4K 字节  DA\_DRW equ 0x92 ; 存在的可读写数据段属性值  DA\_CR equ 0x9A ; 存在的可执行可读代码段属性值 |
| ; 定义全局描述符表  ; 段基址, 段界限, 属性  \_Gdt: Descriptor 0, 0, 0 ; 空描述符  \_CsDesc: Descriptor 0, 0x0FFFFF, DA\_CR | DA\_32 | DA\_LIMIT\_4K ; 0 ~ 4G 的代码段  \_DsDesc: Descriptor 0, 0x0FFFFF, DA\_DRW | DA\_32 | DA\_LIMIT\_4K ; 0 ~ 4G 的数据段  GDT\_SIZE equ $ - \_Gdt ; 全局描述符表的长度  \_KeCodeSegmentSelector dw \_CsDesc - \_Gdt ; 代码段选择子  \_KeDataSegmentSelector dw \_DsDesc - \_Gdt ; 数据段选择子 |

**（2）二级页表的分页变换**

32位逻辑地址 ->分段变换 ->32位线性地址 ->分页变换 ->32位物理地址

分段变换：EOS 将代码段和数据段的段基址都设置为 0，大小为 4GB。分段变换会将逻辑地址和段基址 0 相加，得到的线性地址和逻辑地址是相同的，就好像不存在分段变换一样。

分页变换：在保护模式下启动了分页机制，就会对 32 位线性地址进行基于二级页表的分页变换，从而将线性地址转换为物理地址。线性地址高10位作为页目录索引，中10位是页表索引，低12位是物理页索引，得到物理地址。

页目录和页表的大小都是 4KB，并且都包含了1024 个页目录项PDE 或者 页表项PTE。PDE 和 PTE 的大小都会是 4 个字节。

**（3）页框号数据库**

通过页框号数据库来管理所有物理页。PFN Database链表的长度和物理存储器所包含的物理页数量是一致的。数组中的第 N 项描述了页框号为N 的物理页的状态，还指向了具有相同状态的另一个物理页的页框号，构成链表。元素的结构体在文件mm/i386/mi386.h 中定义如下。

|  |
| --- |
| typedef struct \_MMPFN  {  ULONG Unused : 9是未用  ULONG PageState : 3是物理页的状态  ULONG Next : 20是下一个物理页的页框号  }MMPFN, \*PMMPFN; |

**（4）物理页状态描述**

零页，此页空闲可用，已进行零初始化，每个字节的值都是 0；自由页，此页空闲可用，未进行零初始化，每个字节的值都不确定；占用页，此页正在被系统或某个进程使用。

只有零页和自由页对应的数据库项被组成链表。当申请分配物理页时,只需要从任意一个链表的首部移除数据库项,数据库项对应的物理页即可作为分配结果。大部分时候申请者都希望得到已经被零初始化的页。页的状态被定义为枚举类型，在文件 mm/mi.h 中定义如下。

|  |
| --- |
| typedef enum \_PAGE\_STATE {  ZEROED\_PAGE, // 零页  FREE\_PAGE, // 自由页  BUSY\_PAGE, // 占用页  } PAGE\_STATE; |

**2、算法**

**（1）MiAllocateZeroedPages函数**

该函数用于分配零页。首先从零页链表分配，循环分配NumberOfPages ，直到可用的零页链表用完；如果零页链表不足，则通过空闲页链表分配。

|  |
| --- |
| STATUS  MiAllocateZeroedPages(  IN ULONG\_PTR NumberOfPages,  OUT PULONG\_PTR PfnArray  )  参数：  NumberOfPages -- 期望分配的物理页的数量。  PfnArray -- 指针，指向用于输出物理页框号的缓冲区。  {  BOOL IntState;  ULONG\_PTR Pfn;  PVOID ZeroBuffer;  ULONG\_PTR i;  IntState = KeEnableInterrupts(FALSE);  if (NumberOfPages <= MiZeroedPageCount + MiFreePageCount) {  // 先从零页链表分配。  for (i = 0; i < NumberOfPages && MiZeroedPageCount > 0; i++) {  Pfn = MiZeroedPageListHead;  MiZeroedPageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;  MiZeroedPageCount--;  MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;  PfnArray[i] = Pfn;  }  // 如果零页链表不足，则继续从空闲页链表分配。  for (; i < NumberOfPages; i++) {  Pfn = MiFreePageListHead;  MiFreePageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;  MiFreePageCount--;  MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;  // 将物理页映射到系统PTE区域进行清零。  ZeroBuffer = MiMapPageToSystemPte(Pfn);  memset(ZeroBuffer, 0, PAGE\_SIZE);  MiFreeSystemPte(ZeroBuffer);  PfnArray[i] = Pfn;  }  }  KeEnableInterrupts(IntState);  return STATUS\_SUCCESS;  } |

**（2）MiAllocateAnyPages函数**

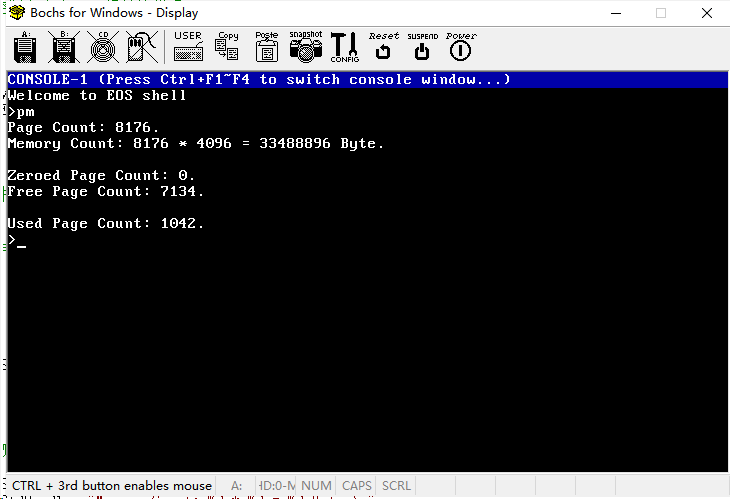
该函数与MiAllocateZerosPages相反，用于分配任意页。首先从空闲页链表分配，循环分配NumberOfPages ，直到可用的空闲页链表用完；如果空闲页链表不足，则通过零页链表分配。

|  |
| --- |
| STATUS  MiAllocateAnyPages(  IN ULONG\_PTR NumberOfPages,  OUT PULONG\_PTR PfnArray  )  {  BOOL IntState;  ULONG\_PTR Pfn;  ULONG\_PTR i;  IntState = KeEnableInterrupts(FALSE);  if (NumberOfPages <= MiZeroedPageCount + MiFreePageCount) {  // 先从空闲链表中分配。  for (i = 0; i < NumberOfPages && MiFreePageCount > 0; i++) {  Pfn = MiFreePageListHead;  MiFreePageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;  MiFreePageCount--;  MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;  PfnArray[i] = Pfn;  }  // 如果空闲链表不足则继续从零页链表分配。  for (; i < NumberOfPages; i++) {  Pfn = MiZeroedPageListHead;  MiZeroedPageListHead = MiGetPfnDatabaseEntry(Pfn)->Next;  MiZeroedPageCount--;  MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE;  PfnArray[i] = Pfn;  }  }  KeEnableInterrupts(IntState);  return STATUS\_SUCCESS;  } |

**3、完成的主要工作**

**（1）控制台命令“pm”**

在ConsoleCmdPhysicalMemoryy处添加断点，在断点处命中时，查看此时的物理内存的各项数据和内存使用情况。继续调试后可以看到控制台中打印的数据与此前查看的信息一致，查看情况如下图。



**（2）分配和释放物理页**

在控制台命令pm中添加分配物理页和释放物理页代码，在MiAllocateAnyPages添加断点，命中后进入MiAllocateAnyPages函数。进行单步调试，对比执行前后物理页面的变化。



可以看到，0x407页面已经被分配出去，从内存中从上到下选择空白的页面。

继续调试，命中下一个断点MiFreePages,进入MiFreePages函数，单步调试，对比执行前后物理页面的变化。



可以看到，“物理内存”窗口恢复原来的状态，表明回收页面完成。

**2）EOS进程逻辑地址空间分配和回收的练习以及源代码分析**

（分析相关源代码，阐述EOS中进程逻辑地址空间的管理方法，包括数据结构和算法等；给出在应用进程中分配虚拟页和释放虚拟页的实现方法简要描述、源代码、测试及结果等；简要说明在本部分实验中完成的主要工作）

**1、数据结构**

**（1）虚拟地址描述符VAD**

一个虚拟地址描述符VAD，记录一段被使用的地址范围。

将所有 VAD 按照地址增序组成链表，从而记录下了进程虚拟地址空间中的哪些地址区域已经被使用。VAD结构体和VAD链表结构体在文件 mm/mi.h 中定义如下。

|  |
| --- |
| typedef struct \_MMVAD{  ULONG\_PTR StartingVpn是被使用区域的开始虚页框号  ULONG\_PTR EndVpn是被使用区域的结束虚页框号  LIST\_ENTRY VadListEntry是链表项，用于将描述同一地址空间的所有VAD串成链表  }MMVAD, \*PMMVAD; |
| typedef struct \_MMVAD\_LIST{  ULONG\_PTR StartingVpn是记录的进程地址空间的开始虚页号  ULONG\_PTR EndVpn是记录的进程地址空间的结束虚页号  LIST\_ENTRY VadListHead是VAD链表头  }MMVAD\_LIST, \*PMMVAD\_LIST; |

**（2）PAS结构体**

PAS结构体，是 PCB 中的 Pas 域，每个 PAS 结构体对应于一个进程的虚拟地址空间，存储了进程地址空间的一个页目录和一个 VAD 链表。

**(3)二级页表的分页变换**

32位逻辑地址 ->分段变换 ->32位线性地址 ->分页变换 ->32位物理地址

分段变换：过程很简单。EOS 将代码段和数据段的段基址都设置为 0，大小为 4GB。分段变换会将逻辑地址和段基址 0 相加，得到的线性地址和逻辑地址是相同的，就好像不存在分段变换一样。

分页变换：在保护模式下启动了分页机制，就会对 32 位线性地址进行基于二级页表的分页变换，从而将线性地址转换为物理地址。线性地址高10位作为页目录索引，中10位是页表索引，低12位是物理页索引，得到物理地址。

页目录和页表的大小都是 4KB，并且都包含了1024 个页目录项PDE 或者 页表项PTE。PDE 和 PTE 的大小都会是 4 个字节。PTE和PDE定义在文件 mm/i386/mi386.h。

**2、算法**

**（1）VirtualAlloc分配虚拟地址的函数**

API 函数 VirtualAlloc 定义在eosapi.c中，VirtualAlloc 调用MmAllocateVirtualMemory，在进程地址空间内申请分配一段虚拟地址区域并映射物理内存。用户进程必须指定虚拟地址区域的大小，还可以选择是否指定虚拟地址区域的开始地址。

|  |
| --- |
| Status = MmAllocateVirtualMemory( &BaseAddress,  &RegionSize,  AllocationType,  FALSE ); |

**（2）VirtualFree函数调用MmFreeVirtualMemory函数**

MmFreeVirtualMemory函数在当前进程地址空间或系统地址空间中释放虚拟内存。

如果需要执行MEM\_RELEASE操作，也就是释放已保留地址区域，会调用MiFreeAddressRegion函数。

|  |
| --- |
| MiFreeAddressRegion(&Pas->VadList, Vad); |

如果需要执行MEM\_DECOMMIT操作，也就是对保留区域内部分已提交的区域进行反提交操作，也就是释放映射在连续虚拟页框上的物理页框，就调用MiDecommitPages函数

|  |
| --- |
| Status = MiDecommitPages(StartingVpn, EndVpn); |

**（3）MmAllocateVirtualMemory 分配虚拟内存的函数**

在virtual.c文件定义如下，如果传入参数是MEM\_RESERVE或（MEM\_RESERVE|MEM\_COMMIT），就调用MiReserveAddressRegion函数在地址空间中保留一段地址区域（即保留一个区域作为虚拟区域），保留的单位为页，然后记录实际保留的起始、结束虚页框号。判断如果传入参数只有MEM\_COMMIT，查询MEM\_COMMIT操作地址区域是否为已保留地址区域，如不是则返回失败。判断如果传入参数是MEM\_COMMIT或（MEM\_RESERVE|MEM\_COMMIT），就为保留的虚拟地址映射物理内存（即为连续的虚拟页框映射物理页框），执行MEM\_COMMIT操作。并且如果前面已执行MEM\_RESERVE则回滚MEM\_RESERVE。

|  |
| --- |
| STATUS  MmAllocateVirtualMemory(  IN OUT PVOID \*BaseAddress,  IN OUT PSIZE\_T RegionSize,  IN ULONG AllocationType,  IN BOOL SystemVirtual  )  {  STATUS Status;  BOOL IntState;  PMMPAS Pas;  PMMVAD Vad;  ULONG\_PTR StartingVpn;  ULONG\_PTR EndVpn;  ASSERT(BaseAddress != NULL && RegionSize != NULL);  // 确保参数有效（地址范围没有溢出）。  if (0 == \*RegionSize || \*BaseAddress + \*RegionSize - 1 < \*BaseAddress) {  return STATUS\_INVALID\_PARAMETER;  }  // 只能在MEM\_RESERVE、MEM\_COMMIT中选择，至少选择一个，可组合选择。  if ((AllocationType & (MEM\_RESERVE | MEM\_COMMIT)) == 0 ||  (AllocationType & ~(MEM\_RESERVE | MEM\_COMMIT)) != 0) {  return STATUS\_INVALID\_PARAMETER;  }  IntState = KeEnableInterrupts(FALSE);  if (SystemVirtual) {  Pas = &MiSystemPas;  } else {  Pas = MiCurrentPas;  }  do {  if ((AllocationType & MEM\_RESERVE) != 0) {  // 在地址空间中保留一段地址区域，保留的单位为页。  Status = MiReserveAddressRegion( &Pas->VadList,  \*BaseAddress,  \*RegionSize,  &Vad );  if (!EOS\_SUCCESS(Status)) {  break;  }  // 记录实际保留的起始、结束虚页框号。  StartingVpn = Vad->StartingVpn;  EndVpn = Vad->EndVpn;  } else {  // 查询MEM\_COMMIT操作地址区域是否为已保留地址区域，如不是则返回失败。  Status = MiFindReservedAddressRegion( &Pas->VadList,  \*BaseAddress,  \*RegionSize,  &Vad );  if (!EOS\_SUCCESS(Status)) {  break;  }  // MEM\_COMMIT操作的单位为页。  StartingVpn = MI\_VA\_TO\_VPN(\*BaseAddress);  EndVpn = MI\_VA\_TO\_VPN(\*BaseAddress + \*RegionSize - 1);  }  if ((AllocationType & MEM\_COMMIT) != 0) {  // 执行MEM\_COMMIT操作。  Status = MiCommitPages(StartingVpn, EndVpn);  if (!EOS\_SUCCESS(Status)) {  ASSERT(STATUS\_NO\_MEMORY == Status);  // 如果前面已执行MEM\_RESERVE则回滚MEM\_RESERVE。  if ((AllocationType & MEM\_RESERVE) != 0) {  MiFreeAddressRegion(&Pas->VadList, Vad);  }  break;  }  }  // 设置返回值。  \*BaseAddress = MI\_VPN\_TO\_VA(StartingVpn);  \*RegionSize = (EndVpn - StartingVpn + 1) << PAGE\_SHIFT;  Status = STATUS\_SUCCESS;  } while (0);  KeEnableInterrupts(IntState);  return Status;  } |

**（4）MiReserveAddressRegion函数**

如果指定了开始地址，首先遍历 VAD 链表，找到目标区域前的已保留区域P。如果P之后的已保留区域和目标区域重叠，说明查找指定的地址区域已经被使用，则返回失败。

如果没有指定了开始地址，就遍历 VAD 链表，从地址空间的起始端向后查找第一个满足申请大小的未保留区域。最终，从系统内存池中分配一个虚拟地址描述符结构体，将虚拟地址描述符插入已保留区域链表中。

**（5）MiCommitPages函数**

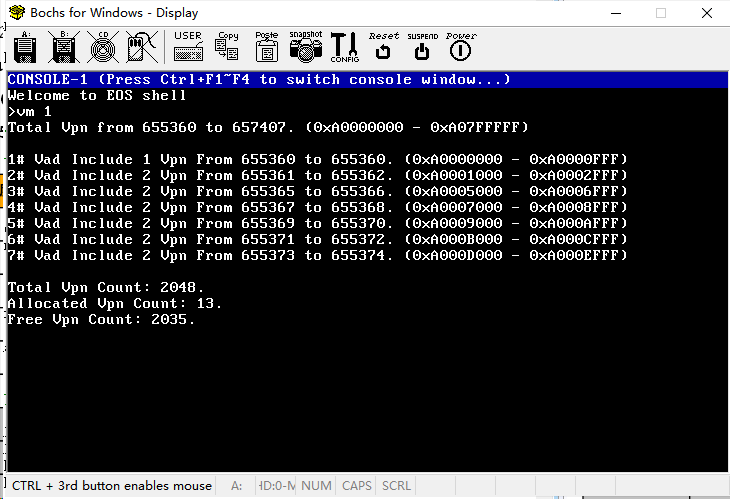
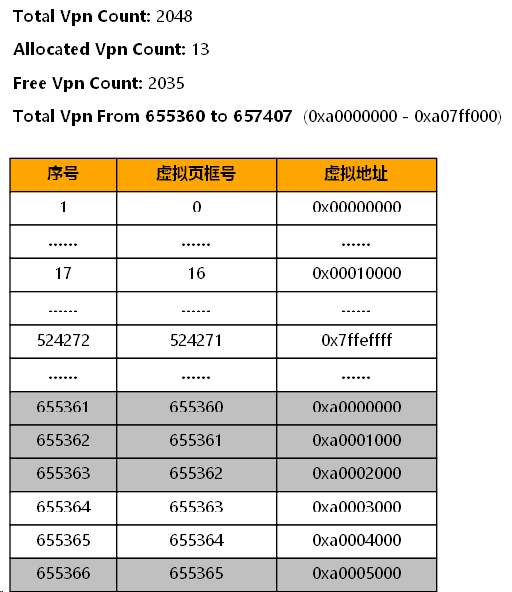
循环分配各个物理页，每次循环中，调用MiAllocateZeroedPages分配一个物理页并将之映射到虚拟页。其中，MiAllocateZeroedPages首先从零页链表中分配，如果零页链表不足则再从空闲页链表分配（会先清零）。

|  |
| --- |
| STATUS  MiCommitPages(  IN ULONG\_PTR StartingVpn,  IN ULONG\_PTR EndVpn  )  {  STATUS Status;  ULONG\_PTR Vpn;  ULONG\_PTR Pfn;  ULONG\_PTR DemandPages;  // 统计为虚拟地址区域提交内存所需的物理页面总数。  DemandPages = 0;  // 如果不是系统内存则统计这段虚拟地址空缺的页表数量，因为系统地址空间全部安  // 装了所需的页表而用户地址空间没有。  if (StartingVpn < MI\_VA\_TO\_VPN(MM\_SYSTEM\_RANGE\_START)) {  for (Vpn = StartingVpn & ~((1 << PTI\_BITS) -1); Vpn <= EndVpn; Vpn += (1 << PTI\_BITS)) {  if (0 == MiGetPdeAddress(Vpn)->u.Hard.Valid) {  DemandPages++;  }  }  }  // 统计这段虚拟地址的页面空洞数量。  for (Vpn = StartingVpn; Vpn <= EndVpn; Vpn++) {  if (0 == MiGetPdeAddress(Vpn)->u.Hard.Valid ||  0 == MiGetPteAddress(Vpn)->u.Hard.Valid) {  DemandPages++;  }  }  // 如果没有足够的空闲物理页则返回失败。  if (DemandPages > MiGetAnyPageCount()) {  return STATUS\_NO\_MEMORY;  }  for (Vpn = StartingVpn; Vpn <= EndVpn; Vpn++) {  // 如果页目录项或者页表项无效，说明当前地址没有映射物理内存。  if (0 == MiGetPdeAddress(Vpn)->u.Hard.Valid ||  0 == MiGetPteAddress(Vpn)->u.Hard.Valid) {  // 如果页目录项无效则分配一物理页作为页表并设置页目录项。  if (0 == MiGetPdeAddress(Vpn)->u.Hard.Valid) {  ASSERT(Vpn < MI\_VA\_TO\_VPN(MM\_SYSTEM\_RANGE\_START));  Status = MiAllocateZeroedPages(1, &Pfn);  ASSERT(EOS\_SUCCESS(Status));  MiMapPageTable(Vpn, Pfn);  }  // 分配一个物理页并将之映射到虚拟页。  Status = MiAllocateZeroedPages(1, &Pfn);  ASSERT(EOS\_SUCCESS(Status));  MiMapPage(Vpn, Pfn);  // 增加页表对应的有效PTE计数器。  // 注意：系统地址空间从不使用PTE计数器，页表从不被卸载或安装。  if (Vpn < MI\_VA\_TO\_VPN(MM\_SYSTEM\_RANGE\_START)) {  MiIncPteCounter(Vpn);  }  }  }  return STATUS\_SUCCESS;  } |

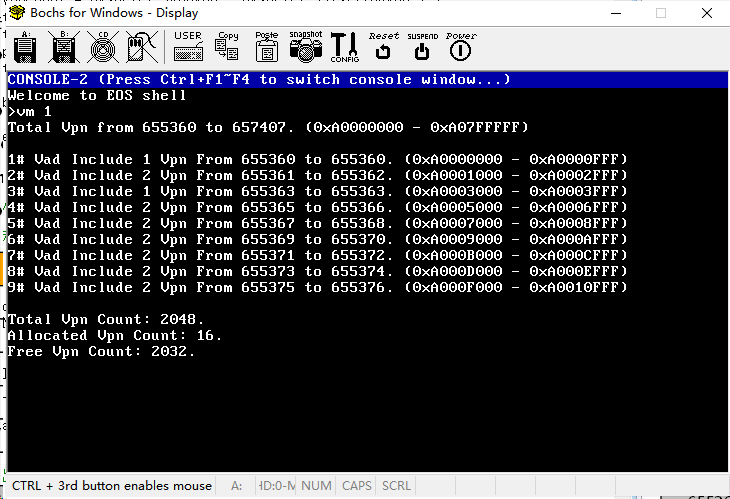
**3、完成的主要工作**

**（1）执行控制台命令“vm”**

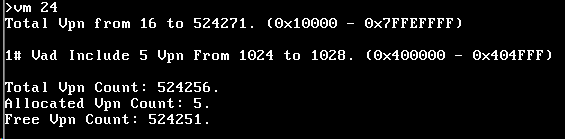
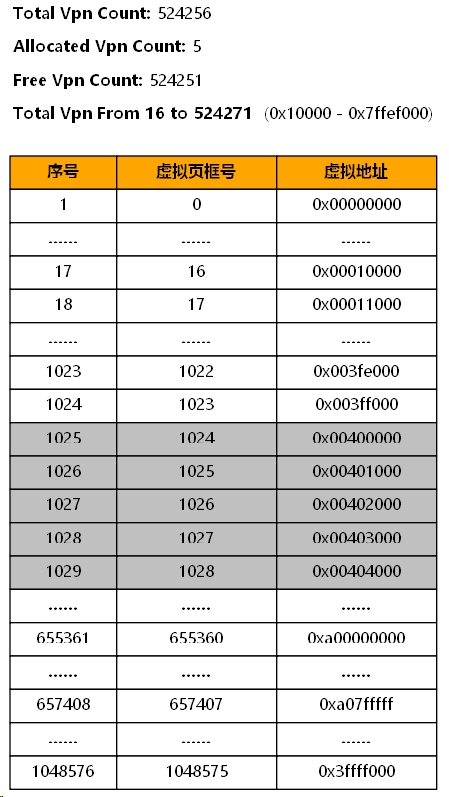
ConsoleCmdVM函数最后添加断点，输入“vm 1”后命中断点，查看系统进程的虚拟地址描述符信息。结果如下图所示。



LoopApp.exe添加进入Floppy.img文件中，ConsoleCmdVM函数最后添加断点，启动调试并输入LoopApp.exe。切换至控制台2，输入“vm 1”命中断点，查看虚拟描述符。

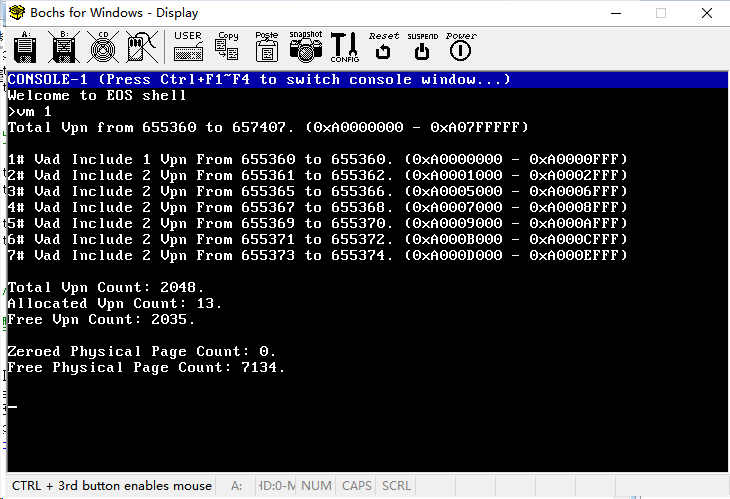


输入“vm 1”命中断点，查看虚拟描述符。

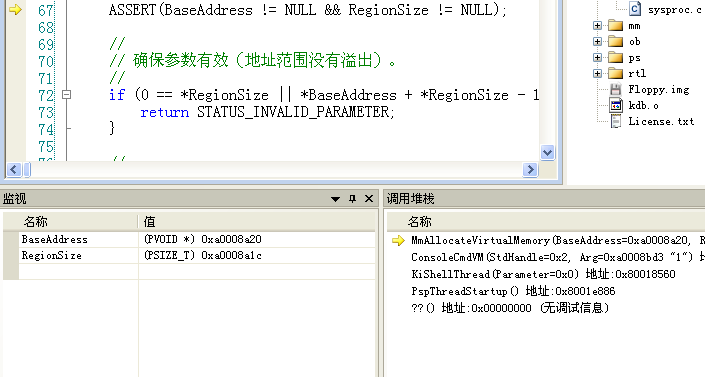
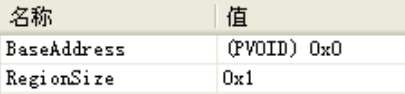


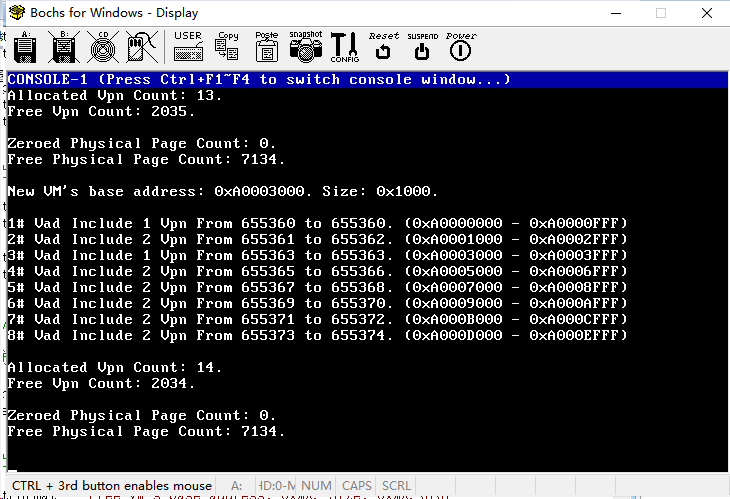
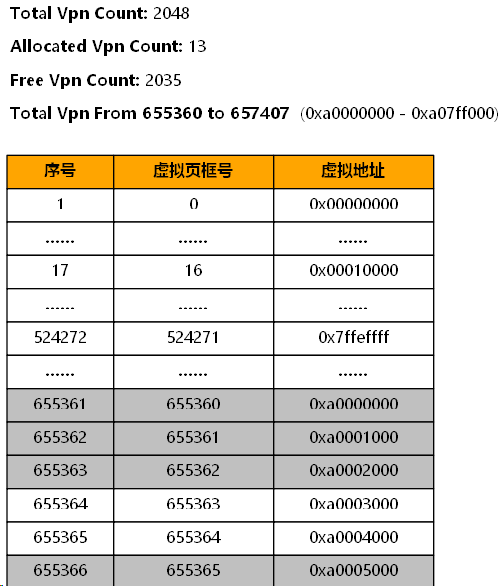
**（2）分配和释放虚拟页**

在vm命令对应的函数中添加分配和释放虚拟页的代码，在ke/sysproc.c文件的ConsoleCmdVM函数中，在调用MmAllocateVirtualMemory函数的代码行添加一个断点，在调用MmFreeVirtualMemory函数的代码行添加一个断点。开始调试后，在控制台中输入命令“vm 1”，命中断点，进入MmAllocateVirtualMemory执行完毕，查看虚拟地址描述符。

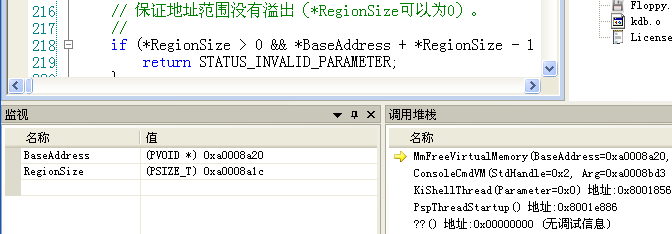


继续调试，命中MmFreeVirtualMemory处的断点，进入该函数，查看BaseAddress和RegionSize初始化的值。函数执行完毕时查看虚拟地址描述符。





分配在用户空间的第一部分，BaseAddress和RegionSize分别表示基地址的位置以及分配空间的大小。RegionSize和BaseAddress和的变化如下。

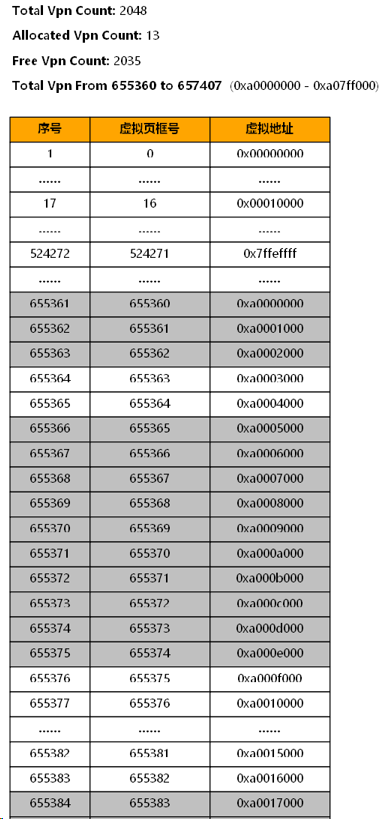


**（3）修改参数**

a、在调用MmAllocateVirtualMemory函数时将RegionSize参数的值设置为 PAGE\_SIZE+1，重复之前的调试，查看虚拟地址描述符。

b、在调用MmAllocateVirtualMemory函数时将BaseAddress参数的值设置为已经被占用的虚拟内存0xA0000000，查看虚拟地址描述符。

c、在调用MmAllocateVirtualMemory函数时将RegionSize参数的值设置为 PAGE\_SIZE\*2，将BaseAddress参数的值设置为 0xA0017004，查看虚拟地址描述符。



**4、在应用程序中分配和释放虚拟页**

**（1）简要描述EOSAPP.c**

先定义INT\*的指针类型声明指针，接收VirtualAlloc函数的返回的分配的虚拟空间地址。然后调用VirtualAlloc函数分配一个整型变量所需的空间。如果分配不成功再就直接终止程序。输出INT的值和地址。有4个参数：起始地址是NULL（不指定），空间大小是int类型大小，分配状态是MEM\_RESERVE|MEM\_COMMIT（先分配虚拟空间，再映射物理空间）。

修改整型变量的值为 0xFFFFFFFF。输出INT的新值。

调用Sleep函数等待 10 秒钟。

调用 API 函数 VirtualFree，释放之前分配的整型变量的空间。有3个参数：起始地址，空间大小，释放类型MEM\_RELEASE（释放虚拟内存，不释放映射在连续虚拟页框上的物理页框）

进入死循环，避免程序结束。

**（2）源代码**

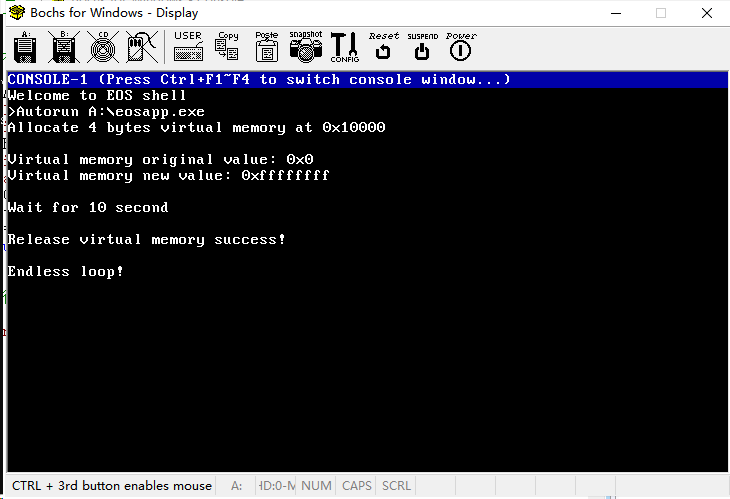
|  |
| --- |
| int main(int argc, char\* argv[])  {  int \*p;  p=VirtualAlloc(NULL,sizeof(int),MEM\_RESERVE|MEM\_COMMIT);  printf("Allocate %d bytes virtual memory at 0x%x\n\n",sizeof(int),p);  printf("Virtual memory original value: 0x%x\n",\*p);  \*p=0xFFFFFFFF;  printf("Virtual memory new value: 0x%x\n\n",\*p);  printf("Wait for 10 second\n\n");  Sleep(10000);//ms  int result=VirtualFree(p,0,MEM\_RELEASE);  if(result==1){printf("Release virtual memory success!\n\n");}  else {return 0;}  // 开始死循环，这样应用程序进程就不会结束。  printf("Endless loop!\n");  for(;;) {  }  return 0;  } |

**（3）测试**

代码修改完毕后，生成 EOS 应用程序项目，启动调试。应用程序自动执行后输出结果预计与指导书一致。

**（4）结果**

执行结果如下，与指导书一致，说明实现成功。



**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS物理存储器管理与进程逻辑地址空间管理方法的特点、不足及改进意见；结合EOS对物理存储器与进程逻辑地址空间管理相关问题提出自己的思考；其他需要说明的问题）

**1、EOS物理存储器管理与进程逻辑地址空间管理方法**

**（1）特点**

**A、EOS物理存储器管理**

采用二级页表的分页变换，对物理存储器也是以页为单位来进行管理，变换过程如下：32位逻辑地址 ->分段变换 ->32位线性地址 ->分页变换 ->32位物理地址

使用页框号数据库(PFN Database)来管理所有物理页。PFN 数组中的第 N 项描述了页框号为N 的物理页的状态，还指向了具有相同状态的另一个物理页的页框号，构成链表。

EOS定义了三种物理的页状态。零页，空闲可用，已进行零初始化。 自由页，空闲可用，未进行零初始化，每个字节的值都不确定。 占用页，正在被系统或某个进程使用。只有零页和自由页对应的数据库项被组成链表。

**B、EOS进程逻辑地址空间管理**

虚拟地址空间：EOS 中的每个进程都有独立虚拟地址空间4G。虚拟存储器的大小并不依赖于实际存储器的大小。低 2G 为进程私有的用户地址空间。用于存放用户进程的代码、数据等。高 2G 为所有进程共享的系统地址空间。用于存放内核的代码、内核运行时的各种数据结构以及所有线程的内核模式栈等。

虚拟地址空间的分配、映射：进程申请的时候，EOS才为进程分配一定数量的物理页，并在可用虚拟地址区域进行映射。可以是进程指定的虚拟地址区域或由系统选择的一个可用虚拟地址区域。虚拟地址描述符（VAD，Virtual Address Descriptor）用于记录一段被使用的地址范围。将所有 VAD 按照地址增序组成链表。用户进程可以调用 API 函数 VirtualAlloc 在进程地址空间内申请分配一段虚拟地址区域并映射物理内存，也可以可以调用 API 函数 VirtualFree 释放已分配地址区域，并释放映射在已分配地址区域的物理页。

**（2）不足**

EOS采用二级页表的分页变换，这样不利于存储器共享与保护，不利于用户视角，不利于动态申请内存，不利于管理统一化。

**（3）改进**

改用段页式，兼具段式管理和页式管理的优点。

段式管理。一方面利于共享，如果多个用户进程或作业需要共享某段程序或数据,可以使用各自的段名,在自己的段表中填入内存中的共享段的起始地址,并设置适当的读写控制权,就可以做到共享一个内存段的信息。

另一方面，利于存储保护。在多道程序的情况下,为了保证程序顺利执行,必须实行对段的保护,一般可利用段表及段长来实现段的保护,防止程序执行时地址越界;在段表中设有"存取权限”,对程序的访问权限进行各种必要的限制。

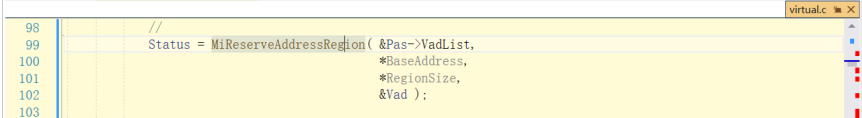
**2、相关问题提出自己的思考**

VirtualAlloc函数的Size参数如果是NULL，会怎么处理？

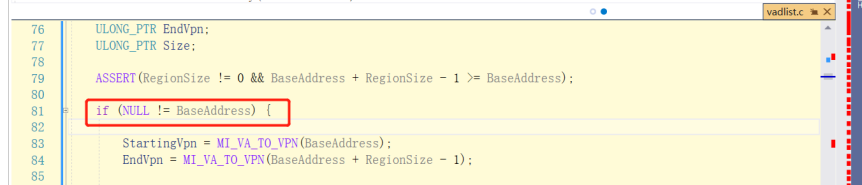
VirtualAlloc函数中，发现调用MmAllocateVirtualMemory函数。



MmAllocateVirtualMemory函数中调用MiReserveAddressRegion函数。



MiReserveAddressRegion函数中，对BaseAddress进行非空处理，如红框所示。



**北京科技大学实验报告**

学院：计通学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期： 2023 年 11 月 18 日

**实验名称：操作系统实验6 FAT12文件系统（3分）**

**实验目的：**以一个教学型操作系统EOS为例，理解磁盘存储器管理的基本原理与文件系统的实现方法；能对核心源代码进行分析和修改，具备实现一个简单文件系统的基本能力；训练分析问题、解决问题以及自主学习能力，通过6个实验的实践，达到能独立对小型操作系统的部分功能进行分析、设计和实现。

**实验环境：**EOS操作系统及其实验环境。

**实验内容：**

通过调用EOS API读取文件数据，跟踪FAT12文件系统的读文件功能，分析EOS中FAT12文件系统的相关源代码，理解并阐述EOS实现FAT12文件系统的方法；修改EOS的源代码，为FAT12文件系统添加写文件功能，。

**1）EOS中FAT12文件系统相关源代码分析**

（分析EOS中FAT12文件系统的相关源代码，简要说明EOS实现FAT12文件系统的方法，包括主要数据结构与文件基本操作的实现等）

**1、主要数据结构**

**（1）概述FAT12文件系统**

文件系统作为逻辑层，不关心用于存储文件的各个数据块的物理位置，而总是将这些数据块抽象为与其大小相同的逻辑扇区，从而使物理扇区可以不连续。EOS中1.44M 软盘上的 2880 个物理扇区分布在不同的盘面和磁道上，但是 FAT12 文件系统只是将这些物理扇区抽象为编号从 0 到 2879 的逻辑扇区。

开始的0-32扇区为系统区，随后的扇区为数据区。每个文件的存储位置和属性等信息保存在系统区中，而每个文件的实际数据都保存在数据区中。

系统区中，这 33 个扇区又分成了三部分，分别是引导扇区、文件分配表（FAT）和根目录。其中引导扇区占用 0 扇区，FAT 表占用 1 到 18 扇区，根目录占用 19到 32 扇区。余下的 33 到 2879 扇区是数据区。

数据区中，将簇做为数据区中存储文件数据的基本单位，簇包括一组连续的扇区。簇包含的扇区数量总是 2 的乘方。EOS中，一个簇只包含一个扇区。

相关结构体在fat12.h，包括：目录项结构体（DIRENT）、卷控制块（VCB）、文件控制块（FCB）等。

**（2）文件对象**

文件对象标记进程打开的设备或磁盘文件。进程在打开设备或磁盘文件时，IO 管理模块会创建一个与被打开设备或磁盘文件关联的文件对象，并返回此文件对象的句柄。此后，进程内的任何线程都可以通过此文件句柄来读写文件，从而完成对设备或磁盘文件的读写。

|  |
| --- |
| typedef struct \_FILE\_OBJECT {  PDEVICE\_OBJECT DeviceObject是关联的设备对象指针  PVOID FsContext是文件的上下文环境指针，  BOOLEAN ReadAccess是文件是否可读  BOOLEAN WriteAccess是文件是否可写  BOOLEAN SharedRead是文件是否允许共享读  BOOLEAN SharedWrite是文件是否允许共享写  ULONG FlagsAndAttributes是标志和属性位  ULONG CurrentByteOffset是文件当前读写偏移位置  MUTEX Mutex是用于多线程互斥访问文件对象的互斥体结构  } FILE\_OBJECT; |

**（3）目录项**

在根目录中，文件系统为每个文件或者文件夹都准备了一个 32 字节的目录项，目录项用来描述文件的名字、属性、文件大小以及文件起始簇的簇号。

由于根目录占用了从 19 到 32 的 14 个扇区，每个扇区的大小为 512 字节，所以，根目录中最多只能有 224 个目录项（14\*512/32=224），这就意味着在 FAT12 文件系统管理的 1.44M 软盘的根目录中最多只能有 224 个文件或者文件夹。

打开文件、关闭文件，都需要访问文件在根目录中对应的目录项。由于CPU无法直接访问磁盘，只能访问内存，所以在内存中准备一个目录项的副本DIRENT 供 CPU 访问。

在目录项结构体中比较重要的参数有：文件名和扩展名Name：文件名占用前8个字节，扩展名占用后3个字节。第一个字节也可以用来表示文件的状态：0x00 表示目录项没有对应文件；0x2E表示目录项对应的是一个子目录；0xE5 表示目录项对应的文件已经被删除。 文件的属性Attributes：比如只读 、隐藏 、系统 、子目录 、存档 。文件起始簇号FirstCluster。文件的大小FileSize以字节为单位。

**（4）文件控制块FCB**

EOS 内核为每个打开的文件在内存中准备了一个文件控制块，方便管理已打开的文件。

使用内存中目录项的副本DIRENT来初始化文件控制块（FCB）中对应的域。这个过程在函数 FatOpenFileInDirectory（在文件io/driver/fat12.c 中定义）中完成。接下来对文件进行读操作时，使用的就只是文件控制块中的信息，与文件的目录项就没有关系了。

**（5）文件分配表FAT**

根目录中的目录项记录了文件的属性和起始簇号，文件分配表记录了文件的簇链，数据区中的簇保存了文件数据。

文件分配表（FAT）用于将数据区中的磁盘空间分配给文件，是显式链接方式。文件分配表被划分为紧密排列的若干个表项，每个表项都与数据区中的一个簇相对应，而且表项的序号也是与簇号一一对应，例如序号为 0 的表项与簇 0 对应，序号为 1 的表项与簇 1 对应。

在每个表项中，只存放了文件的下一个簇号，从而将文件占用的簇连接成一个簇链，链头由目录项中的起始簇号确定。

**a、序号为0和1的表项：**

共占用了 FAT最开始的3个字节，在1.44M软盘上这三个字节的值必须是固定的，分别是0xF0、0xFF、0xFF，用于表示这是一个应用在 1.44M 软盘上的 FAT12 文件系统。数据区就起始于簇2。

**b、其他指针表项**

其余的表项可以存储文件占用的下一个簇号。

0x000：表项对应的簇未被文件占用

0x002-0xFEF：文件占用的下一个簇号（也就是下一个表项的序号）。

0xFF0-0xFF6：保留未用。

0xFF7：表项对应的簇不可用（坏簇）。

0xFF8-0xFFF：表项对应的簇是文件占用的最后一个簇。

**（6）卷控制块VCB**

当我们尝试打开某一文件，执行打开文件相关函数的调用流程，在FatOpenFile 函数开始，在函数的参数中就会传入一个卷控制块的指针。用于描述文件系统的相关信息。

在文件 io/driver/fat12.h 文件中定义了卷控制块结构体，其中重要的参数有：文件系统下层的软盘或者硬盘卷设备对象(目前仅仅是软盘)；文件系统的参数；文件分配表（File Allocation Table）缓冲区。FAT12 的 FAT 表最大不过 6KB，所以完全加载到内存中比较合适；根目录起始扇区、根目录大小以及根目录文件链表头；文件数据区的起始扇区以及簇的总数。

**（7）BPB（BIOS Parameter Block）**

在文件 io/device/fat12.h 文件中定义，其中比较重要的参数有：每扇区字节数SectorsPerCluster；每簇扇区数ReservedSectors；保留扇区数Fats；FAT表的数量RootEntries；根目录项数Sectors；扇区总数Media；扇区总数(Sectors 为 0 时使用)。

**2、文件基本操作的实现**

**（1）FAT12文件系统的初始化**

**A、IoInitializeSystem2函数**

a、创建FAT12驱动程序对象：IopCreateDriver("FAT12 Driver", &DriverObject);

b、初始化：FatInitializeDriver(DriverObject);

c、软驱设备对象上添加FAT12设备对象。DriverObject->AddDevice(DriverObject, DeviceObject, 0, &DeviceObject);

**B、FatInitializeDriver函数**

该函数主要初始化各个操作的函数入口。

|  |
| --- |
| DriverObject->AddDevice = FatAddDevice;  DriverObject->Create = FatCreate;  DriverObject->Close = FatClose;  DriverObject->Read = FatRead;  DriverObject->Write = FatWrite;  DriverObject->Query = FatQuery;  DriverObject->Set = FatSet; |

**（2）打开文件**

**A、CreateFile 函数**

打开一个设备还是在打开一个磁盘文件时，都是通过调用EOS API函数CreateFile来完成的。

参数：参数 FileName 是设备名或磁盘文件名的字符串指针。参数 DesiredAccess 指定使用什么样的读写权限来打开文件，可以为 GENERIC\_READ和 GENERIC\_WRITE 的一种或它们的组合。参数 ShareMode 指定使用什么样的共享方式来打开文件，可为 0，也可以是 FILE\_SHARE\_READ 和 FILE\_SHARE\_WRITE 的一种或它们的组合。参数 CreationDisposition 指定创建新文件的方式。CreateFile 函数在文件eosapi.c，主要调用IoCreateFile函数。

|  |
| --- |
| Status = IoCreateFile( (PSTR)FileName,  DesiredAccess,  ShareMode,  CreationDisposition,  FlagsAndAttributes,  &Handle ); |

**B、IoCreateFile函数**

用于创建或打开文件。目标可以是位于磁盘文件系统中的文件，也可以是IO设备。在io.c文件定义。主要调用函数IopCreateFileObject。

|  |
| --- |
| Status = IopCreateFileObject(FileName,  DesiredAccess,  ShareMode,  CreationDisposition,  FlagsAndAttributes,  &FileObject); |

**C、IopCreateFileObject函数**

a、创建文件对象并初始化之。

|  |
| --- |
| Status = ObCreateObject( IopFileObjectType,  NULL,  sizeof(FILE\_OBJECT),  0,  (PVOID\*)&File ); |

b、执行驱动程序的Create功能函数，即FatCreate函数。

|  |
| --- |
| Status = DeviceObject->DriverObject->Create(DeviceObject, RelativName, CreationDisposition, File); |

**（3）读文件**

调用CreateFile函数打开文件后，就可以通过调用EOS API函数ReadFile和WriteFile

来读写文件。

ReadFile 函数在执行时，仅是向文件对象的 DeviceObject 指针指向的设备对象发起读请求——调用设备驱动程序的 Read 接口函数（详情可参见在文件 io/file.c 文件中定义的 IopReadFileObject 函数）。

在使用 CreateFile 函数打开磁盘文件时，CreateFile 函数创建的文件对象是和磁盘文件的控制块相关联，磁盘文件控制块由文件系统创建。以后对文件对象进行读写时，文件系统根据文件对象关联的文件控制块（FCB, File Control Block）进行操作。

**A、ReadFile函数**

在文件eosapi.c，主要调用ObRead函数。

|  |
| --- |
| Status = ObRead( Handle,  Buffer,  NumberOfBytesToRead,  NumberOfBytesRead ); |

**B、ObRead函数**

用于读支持读操作的内核对象，例如文件、控制台、管道、串口设备等对象。在obmethod.c文件定义，主要调用对象类型中注册的 Read 操作，即FatRead函数。

|  |
| --- |
| Status = Type->Read(Object, Buffer, NumberOfBytesToRead, NumberOfBytesRead); |

**C、FatRead函数**

驱动程序的 Read 功能函数。定义在file.c文件，调用FatReadFile。

|  |
| --- |
| return FatReadFile((PVCB)DeviceObject->DeviceExtension,  (PFCB)FileObject->FsContext,  FileObject->CurrentByteOffset,  Request,  Buffer,  Result); |

**D、FatReadFile函数**

在文件的指定偏移处读取指定字节的数据，实际读取的字节数可能受到文件长度的限制而小于期望值。从偏移位置所在的簇开始向后读取文件的簇，直到读取完成。计算偏移位置在簇内的第几个扇区，然后从这个扇区开始读取簇内的连续扇区。

|  |
| --- |
| for (i = ((Offset + ReadCount) / Vcb->Bpb.BytesPerSector) % Vcb->Bpb.SectorsPerCluster;  i < Vcb->Bpb.SectorsPerCluster; i++) { |

计算读取位置在扇区内的字节偏移。

|  |
| --- |
| OffsetInSector = (Offset + ReadCount) % Vcb->Bpb.BytesPerSector; |

计算需要在这个扇区内读取的字节数。

|  |
| --- |
| if (BytesToRead - ReadCount > Vcb->Bpb.BytesPerSector - OffsetInSector) {  BytesToReadInSector = Vcb->Bpb.BytesPerSector - OffsetInSector;  }  else {  BytesToReadInSector = BytesToRead - ReadCount;  } |

读取扇区数据。

|  |
| --- |
| Status = IopReadWriteSector(Vcb->DiskDevice,  FirstSectorOfCluster + i,  OffsetInSector,  (PCHAR)Buffer + ReadCount,  BytesToReadInSector,  TRUE); |

如果读取完成则返回，继续读文件的下一个簇。

**（4）写文件**

WriteFile 和 ReadFile 一样，区别是调用了设备驱动程序的 Write 接口函数（详情可参见在文件 io/file.c文件中定义的 IopWriteFileObject 函数）。

**A、WriteFile 函数**

在文件eosapi.c，主要调用ObWrite函数。

|  |
| --- |
| Status = ObWrite( Handle,  Buffer,  NumberOfBytesToWrite,  NumberOfBytesWritten ); |

**B、ObWrite函数**

写支持写操作的内核对象，例如文件、控制台、管道、串口设备等对象。在obmethod.c文件定义。主要调用对象类型中注册的 Write 操作，即FatWrite函数。

|  |
| --- |
| Status = Type->Write(Object, Buffer, NumberOfBytesToWrite, NumberOfBytesWritten); |

**C、FatWrite函数**

驱动程序的 Write 功能函数。定义在file.c文件，调用FatWriteFile。

|  |
| --- |
| Status = FatWriteFile((PVCB)DeviceObject->DeviceExtension,  (PFCB)FileObject->FsContext,  FileObject->CurrentByteOffset,  Request,  Buffer,  Result); |

**D、FatWriteFile函数**

在文件指定的偏移位置开始写数据，如果偏移位置小于文件大小则覆盖原有内容，如果写范围超出文件大小则自动增加文件大小，如果文件大小增加后超过文件占用的磁盘空间大小则自动为文件分配新的簇，增加文件占用的磁盘空间。

**（5）句柄和文件对象关闭**

**A、IopCloseFileObject函数**

对象在被对象管理器删除时，会调用文件对象的析构函数 IopCloseFileObject（在文件 io/file.c 文件中定义）。IopCloseFileObject 函数在执行时，会对文件对象中 DeviceObject 指针指向的设备对象发出 Close 请求——调用驱动程序的 Close 接口函数。

IopCloseFileObject 函数内容如下，如果FsContext不为NULL则说明已经打开了设备，需要关闭设备。

|  |
| --- |
| DeviceObject->DriverObject->Close(DeviceObject, FileObject); |

关闭指针FileObject->DeviceObject。

|  |
| --- |
| FileObject->DeviceObject = NULL;  ObDerefObject(DeviceObject); |

**B、IopCloseFileObject调用FatClose 调用FatCloseFile**

关闭文件控制块。减小文件控制块的打开计数器，如果计数器变为0则关闭文件控制块并减小文件所在目录的文件控制块的打开计数器，以此递归。

|  |
| --- |
| for (Current = Fcb; Current != NULL; Current = Parent) { |

如果当前文件的打开计数器减1后仍大于0则返回。

|  |
| --- |
| if (--Current->OpenCount > 0) {  break;  } |

记录当前文件的所在目录，关闭当前文件后还要递归关闭当前文件所在目录。

|  |
| --- |
| Parent = Current->ParentDirectory; |

将当前文件从所在目录的文件链表中移除。

|  |
| --- |
| ListRemoveEntry(&Current->FileListEntry); |

释放FCB占用的内存。

|  |
| --- |
| MmFreeSystemPool(Current);  } |

**2）EOS中FAT12文件系统读文件过程的跟踪**

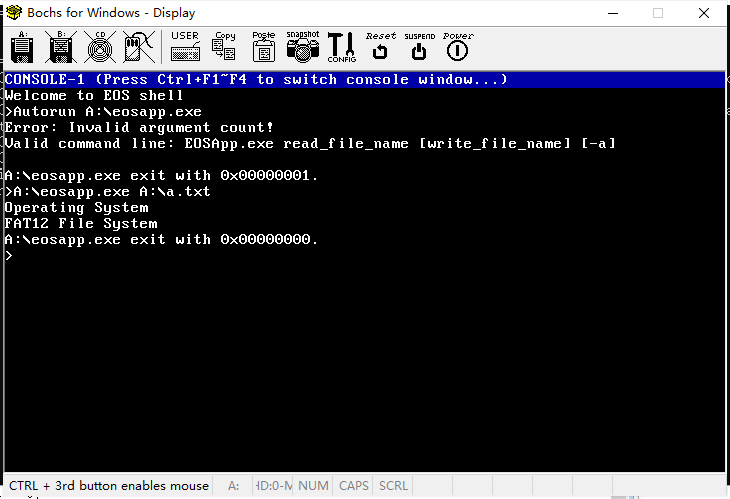
（简要说明在本部分实验中完成的主要工作；总结EOS中读文件的实现方法）

**1、对读文件的追踪**

**（1）调用EOS API函数读取文件中的数据**

a、创建EOS应用程序项目，在Floppy.img中加入a.txt，b.txt，c.txt，d.txt等文件。将其保存在Floppy.img文件中。

b、使用FileApp.c的代码将源代码进行替换，调试输入“A：\EOSApp.exe A:\a.txt”。输入该命令后文件的内容就会被打印在屏幕上，输出的内容如下。



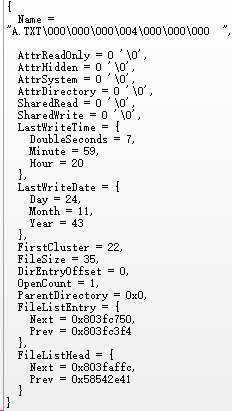
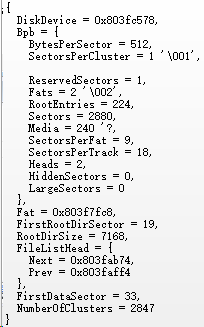
**（2）调试FAT12 文件系统的读文件功能**

a、在同一个IDE中打开fat12.c和eosapp项目，在EOXApp.c中的ReadFile 函数代码行添加断点。

b、开始调试，输入读取a.txt文件内容的命令后命中断点。在读文件时调用的 API 函数 ReadFile最终会调用FatReadFile函数，所以，在fat12.c文件中FatReadFile 函数的开始处添加一个断点。

c、继续执行，命中断点。利用十进制查看\*Vcb信息如下（左）。

d、查看\*File的信息如下（右）。



过程中，预期读取的字节数256，文件大小35，实际可读取的字节数为35。偏移位置0，从第一个簇读取。

**2、总结EOS中读文件的实现方法**

调用CreateFile函数打开文件后，就可以通过调用EOS API函数ReadFile和WriteFile

来读写文件。

ReadFile 函数在执行时，仅是向文件对象的 DeviceObject 指针指向的设备对象发起读请求——调用设备驱动程序的 Read 接口函数（详情可参见在文件 io/file.c 文件中定义的 IopReadFileObject 函数）。

在使用 CreateFile 函数打开磁盘文件时，CreateFile 函数创建的文件对象是和磁盘文件的控制块相关联，磁盘文件控制块由文件系统创建。以后对文件对象进行读写时，文件系统根据文件对象关联的文件控制块（FCB, File Control Block）进行操作。

**A、ReadFile函数**

在文件eosapi.c，主要调用ObRead函数。

|  |
| --- |
| Status = ObRead( Handle,  Buffer,  NumberOfBytesToRead,  NumberOfBytesRead ); |

**B、ObRead函数**

用于读支持读操作的内核对象，例如文件、控制台、管道、串口设备等对象。在obmethod.c文件定义，主要调用对象类型中注册的 Read 操作，即FatRead函数。

|  |
| --- |
| Status = Type->Read(Object, Buffer, NumberOfBytesToRead, NumberOfBytesRead); |

**C、FatRead函数**

驱动程序的 Read 功能函数。定义在file.c文件，调用FatReadFile。

|  |
| --- |
| return FatReadFile((PVCB)DeviceObject->DeviceExtension,  (PFCB)FileObject->FsContext,  FileObject->CurrentByteOffset,  Request,  Buffer,  Result); |

**D、FatReadFile函数**

在文件的指定偏移处读取指定字节的数据，实际读取的字节数可能受到文件长度的限制而小于期望值。从偏移位置所在的簇开始向后读取文件的簇，直到读取完成。计算偏移位置在簇内的第几个扇区，然后从这个扇区开始读取簇内的连续扇区。

|  |
| --- |
| for (i = ((Offset + ReadCount) / Vcb->Bpb.BytesPerSector) % Vcb->Bpb.SectorsPerCluster;  i < Vcb->Bpb.SectorsPerCluster; i++) { |

计算读取位置在扇区内的字节偏移。

|  |
| --- |
| OffsetInSector = (Offset + ReadCount) % Vcb->Bpb.BytesPerSector; |

计算需要在这个扇区内读取的字节数。

|  |
| --- |
| if (BytesToRead - ReadCount > Vcb->Bpb.BytesPerSector - OffsetInSector) {  BytesToReadInSector = Vcb->Bpb.BytesPerSector - OffsetInSector;  }  else {  BytesToReadInSector = BytesToRead - ReadCount;  } |

读取扇区数据。

|  |
| --- |
| Status = IopReadWriteSector(Vcb->DiskDevice,  FirstSectorOfCluster + i,  OffsetInSector,  (PCHAR)Buffer + ReadCount,  BytesToReadInSector,  TRUE); |

如果读取完成则返回，继续读文件的下一个簇。

**3）为EOS的FAT12文件系统添加写文件功能**

（给出实现方法的简要描述、源代码、测试及结果等）

**1、简要描述**

根据要求，修改FatWriteFile函数，需要完成三个功能，这三个功能难度逐渐增加。

a、完成一个最简单的情况，向一个空文件中写入数个字节的数据，写入多少字节便从文件起始位置覆盖多少字节。通过命令如“A:\eosapp.exe A:\a.txt A:\b.txt”实现。

b、使 FatWriteFile 函数写入文件的数据可以跨越一个扇区的边界，在这种情况下，最多跨越一个扇区，写入的字节从文件的结尾处写入。通过命令如“A:\eosapp.exe A:\a.txt A:\b.txt -a”实现。

c、在b的情况下进行改进，使 FatWriteFile 函数写入文件的数据可以跨越多个扇区的边界，写入的字节从文件的结尾处写入。通过命令如“A:\eosapp.exe A:\a.txt A:\b.txt -a”实现。

对于问题a，按照指导书，将相应的文件、代码进行替换，配置好环境后进行验证，属于验证性内容。

对于问题b，由于IopReadWriteSector函数只能对整个扇区进行读写操作，不能跨越扇区边界，所以只能通过修改 FatWriteFile 函数来解决该问题。

由于目前EOS应用程序中定义的缓冲区大小是256字节，所以调用函数 FatWriteFile 写入的数据最多也是 256 字节，这就意味着写入的数据只可能跨越一个扇区的边界。所以，可以尝试根据起始写入的位置和写入数据的大小将要写入的数据分割为不跨越扇区边界的两块数据，对分割后的两块数据分别处理。

对于要写入当前扇区内的数据，可以直接调用IopReadWriteSector函数并设置合适的参数来执行写扇区操作。如果整个数据不跨越扇区边界，当然就都写入当前扇区即可。

对于要写入下一个扇区内的数据，必须调用 FatGetFatEntryValue 函数根据当前簇号得到下一个簇号，如果得到的下一个簇号大于0xFF8，还需要调用 FatAllocateOneCluster函数分配一个新簇，并调用 FatSetFatEntryValue 函数将新簇链接到当前簇的后面。待下一个簇准备好后，可以根据下一个簇号计算出其对应的扇区号，然后就可以调用IopReadWriteSector 函数并设置合适的参数来执行写扇区操作。调用IopReadWriteSector 函数时使用的参数一定要设置正确，特别是扇区号、扇区内起始位置、写入数据缓冲区地址和写入的字节数目，在跨越扇区边界时这些参数都会有变化。

对于问题c，与问题b类似，必须使用循环来处理写入数据跨越多个扇区边界的问题。每次循环时只将合适的数据写入当前簇，在后面的循环中将余下的数据写入簇链中后面的簇，直到所有数据写入完毕。

每次向簇中写数据之前都需要判断是否需要分配新簇。

前一个簇号和当前簇号这两个变量对于管理簇链非常重要，在循环的过程中要注意维护好这两个 变量的值，保证在每次循环时这两个变量都保存了正确的簇号。 调用IopReadWriteSector 函数时使用的参数一定要设置正确，特别是扇区号、扇区内起始位置、写入数据缓冲区地址和写入的字节数目。

当前 FAT12 文件系统中一个簇只包含一个扇区，为了简化程序，可以不考虑一个簇中包含多个扇区的情况。

在函数结束前，注意要修改文件大小并返回实际写入的字节数量。

**2、源代码**

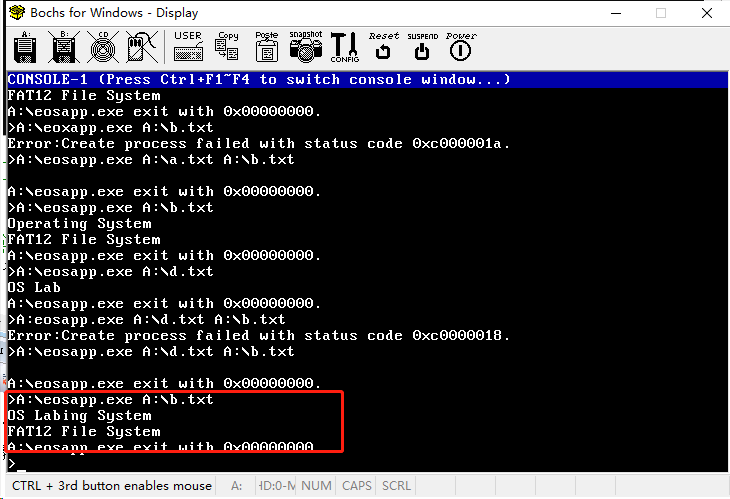
因为三个功能全部都是修改FatWriteFile函数，因此下面展示FatWriteFile函数的内容，并对修改内容进行标记。（受限于篇幅，只展示功能3，其余源代码见项目文件。因为功能3实现的基础是功能1、2）

|  |
| --- |
| STATUS  FatWriteFile(  IN PVCB Vcb,  IN PFCB File,  IN ULONG Offset,  IN ULONG BytesToWrite,  IN PVOID Buffer,  OUT PULONG BytesWriten  )  {  //-----------------------------3.4.3-------------------------------------  //-------------使 FatWriteFile 函数写入文件的数据可以跨越多个扇区的边界 --------------  STATUS Status;  // 由于在将新分配的簇插入簇链尾部时，必须知道前一个簇的簇号，  // 所以定义了“前一个簇号”和“当前簇号”两个变量。  USHORT PrevClusterNum, CurrentClusterNum;  USHORT NewClusterNum;  ULONG ClusterIndex;  ULONG FirstSectorOfCluster;  ULONG OffsetInSector;  //flag  ULONG ByteWrittenTemp = 0;//已写入字节数量  ULONG i;  // 写入的起始位置不能超出文件大小（并不影响增加文件大小或增加簇，想想原因？）  if (Offset > File->FileSize)  return STATUS\_SUCCESS;  // 根据簇的大小，计算写入的起始位置在簇链的第几个簇中（从 0 开始计数）  ClusterIndex = Offset / FatBytesPerCluster(&Vcb->Bpb);  // 顺着簇链向后查找写入的起始位置所在簇的簇号。  PrevClusterNum = 0;  CurrentClusterNum = File->FirstCluster;  for (i = ClusterIndex; i > 0; i--) {  PrevClusterNum = CurrentClusterNum;  CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);  }  // 如果写入的起始位置还没有对应的簇，就增加簇  if (0 == CurrentClusterNum || CurrentClusterNum >= 0xFF8) {  // 为文件分配一个空闲簇  FatAllocateOneCluster(Vcb, &NewClusterNum);  // 将新分配的簇安装到簇链中  if (0 == File->FirstCluster)  File->FirstCluster = NewClusterNum;  else  FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);  CurrentClusterNum = NewClusterNum;  }  // 计算写位置在扇区内的字节偏移。  OffsetInSector = Offset % Vcb->Bpb.BytesPerSector;  //分配了新簇，计算扇区内的字节偏移，  while (1) {  if (ByteWrittenTemp >= BytesToWrite)break;//写完了  //计算当前簇的第一个扇区的扇区号。簇从2开始计数。  FirstSectorOfCluster = Vcb->FirstDataSector + (CurrentClusterNum - 2) \* Vcb->Bpb.SectorsPerCluster;  //如果终点小于等于扇区字节数，就能一次全部写入扇区  //其中，“偏移+需要写入字节数-已经写入字节数”就是本次全部写入的终点地址  //Vcb->Bpb.BytesPerSector是扇区字节数  if (OffsetInSector + BytesToWrite - ByteWrittenTemp <= Vcb->Bpb.BytesPerSector)  {  Status = IopReadWriteSector(Vcb->DiskDevice,  FirstSectorOfCluster,  OffsetInSector,  (PCHAR)(Buffer + ByteWrittenTemp),  BytesToWrite - ByteWrittenTemp,  FALSE);  //修改已经写入的字节数  ByteWrittenTemp = BytesToWrite;  if (!EOS\_SUCCESS(Status))  return Status;  }  //如果终点超出扇区，就只先写入本扇区最大写入量Vcb->Bpb.BytesPerSector - OffsetInSector  else  {  Status = IopReadWriteSector(Vcb->DiskDevice,  FirstSectorOfCluster,  OffsetInSector,  (PCHAR)(Buffer + ByteWrittenTemp),  Vcb->Bpb.BytesPerSector - OffsetInSector,  FALSE);  //如果出错，就报错  if (!EOS\_SUCCESS(Status))  return Status;  //修改已经写入的字节数和扇区偏移地址  ByteWrittenTemp += Vcb->Bpb.BytesPerSector - OffsetInSector;  OffsetInSector = 0;  //新增一个簇，用于在下次循环中存入数据。与前文类似，簇链指针后移，插入新簇。  PrevClusterNum = CurrentClusterNum;  CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);//根据当前簇号得到下一个簇号，因为没有下一个簇，所以值大于等于0xFF8  //确保下一个簇是空  if (CurrentClusterNum >= 0xFF8) {  // 为文件分配一个空闲簇  FatAllocateOneCluster(Vcb, &NewClusterNum);  // 将新分配的簇安装到当前簇后面（也就是簇链结尾）  FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);//如果新簇不是第一个簇。把新簇链接到当前簇后面  // 让CurrentClusterNum指针指向新簇，用于下次循环  CurrentClusterNum = NewClusterNum;  }  }  }  // 如果文件长度增加了则必须修改文件的长度。  if (Offset + BytesToWrite > File->FileSize) {  File->FileSize = Offset + BytesToWrite;  // 如果是数据文件则需要同步修改文件在磁盘上对应的 DIRENT 结构  // 体。目录文件的 DIRENT 结构体中的 FileSize 永远为 0，无需修改。  if (!File->AttrDirectory)  FatWriteDirEntry(Vcb, File);  }  // 返回实际写入的字节数量  \*BytesWriten = BytesToWrite;  return STATUS\_SUCCESS;  } |

**3、测试及结果**

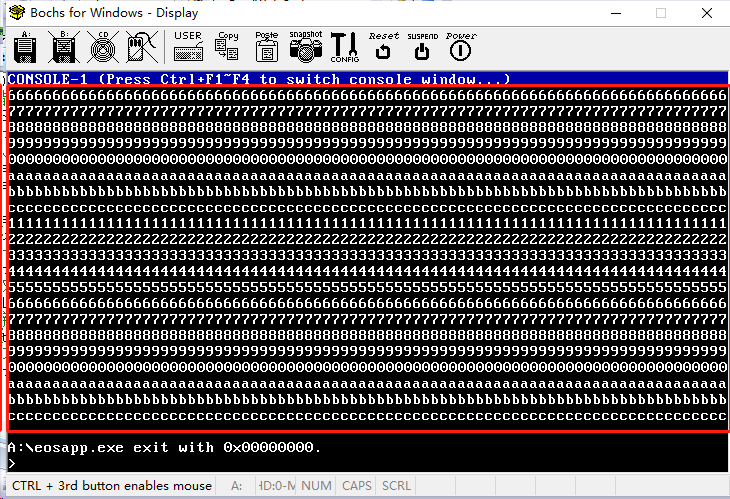
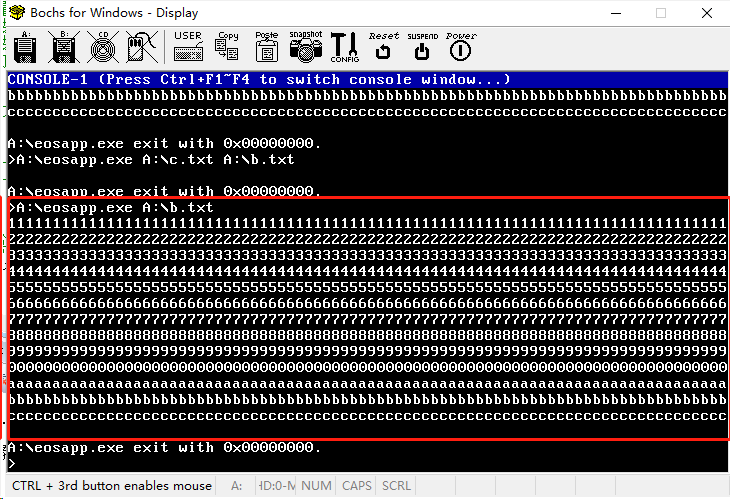
**（1）最简单的情况**

键入命令后，可以看到文件b的内容中，前几个字节被覆盖，与预期一致。



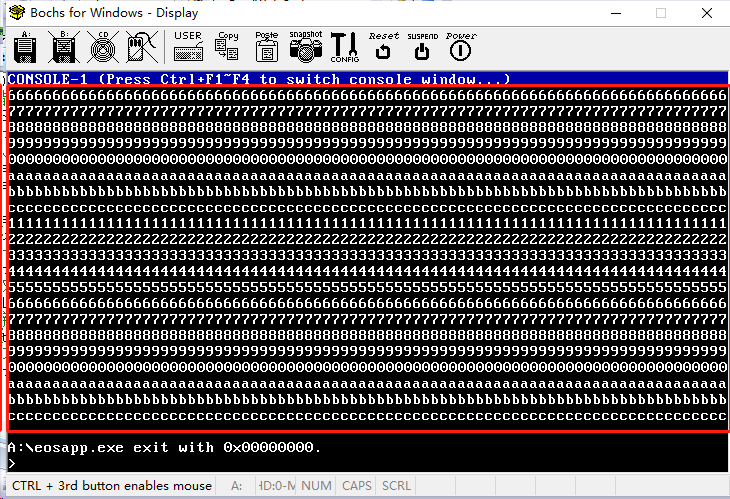
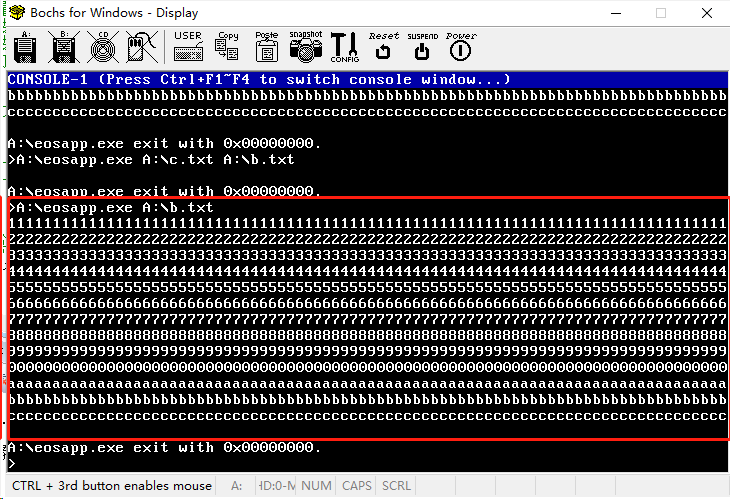
**（2）使 FatWriteFile 函数写入文件的数据可以跨越一个扇区的边界**

键入命令后，可以看到文件c的内容成功追加到了文件b的末尾。



**（3）使 FatWriteFile 函数写入文件的数据可以跨越多个扇区的边界**

键入命令后，可以看到文件c的内容成功追加到了文件b的末尾，成功跨越多个扇区，将内容写入。



**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS中FAT12文件系统实现方法的特点、不足及改进意见；结合EOS对文件系统实现相关问题提出自己的思考；分析写文件实现方法的有效性、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中FAT12文件系统的实现**

**（1）特点**

EOS中FAT12 文件系统将软盘的物理扇区抽象为编号从 0 到 2879 的逻辑扇区。

FAT12 文件系统的系统区从 0 扇区开始，到 32 扇区结束，分成了三部分：引导扇区占用 0 扇区，FAT 表占用 1 到 18 扇区，根目录占用 19到 32 扇区。余下的 33 到 2879 扇区是数据区。

**A、根目录：**文件系统为每个文件或者文件夹都准备了一个 32 字节的目录项，目录项用来描述文件的名字、属性、文件大小以及文件起始簇的簇号。

**B、文件分配表：**将数据区中的磁盘空间分配给文件，显式链接方式。

**C、数据区：**将簇做为数据区中存储文件数据的基本单位，一个簇只包含一个扇区。

对文件系统的重要操作包括：初始化文件系统、打开文件、读文件、写文件、关闭文件对象和句柄等。具体内容见前文“代码分析”。

**（2）不足**

EOS文件系统有很多对文件的操作在函数中留有相关代码的撰写位置，但没有具体代码，没实现对应功能。

另外，该文件系统所实现的磁盘调度算法是FCFS，效率、公平性等性能可能较低，需要后续进行修改。

**（3）改进建议**

在后续的拓展实验中，将该文件系统进行完善。对于留有相关代码的撰写位置，撰写相关代码，实现对应功能；基于文件系统的性能，修改磁盘调度算法，提高磁盘访问效率。

**2、写文件的实现方法**

**（1）有效性**

根据要求，修改FatWriteFile函数，需要完成三个功能，这三个功能难度逐渐增加。

a、完成一个最简单的情况，向一个空文件中写入数个字节的数据，写入多少字节便从文件起始位置覆盖多少字节。通过命令如“A:\eosapp.exe A:\a.txt A:\b.txt”实现。

b、使 FatWriteFile 函数写入文件的数据可以跨越一个扇区的边界，在这种情况下，最多跨越一个扇区，写入的字节从文件的结尾处写入。通过命令如“A:\eosapp.exe A:\a.txt A:\b.txt -a”实现。

c、在b的情况下进行改进，使 FatWriteFile 函数写入文件的数据可以跨越多个扇区的边界，写入的字节从文件的结尾处写入。通过命令如“A:\eosapp.exe A:\a.txt A:\b.txt -a”实现。

对于问题a，按照指导书，将相应的文件、代码进行替换，配置好环境后进行验证，属于验证性内容。

对于问题b，由于IopReadWriteSector函数只能对整个扇区进行读写操作，不能跨越扇区边界，所以只能通过修改 FatWriteFile 函数来解决该问题。

由于目前EOS应用程序中定义的缓冲区大小是256字节，所以调用函数 FatWriteFile 写入的数据最多也是 256 字节，这就意味着写入的数据只可能跨越一个扇区的边界。所以，可以尝试根据起始写入的位置和写入数据的大小将要写入的数据分割为不跨越扇区边界的两块数据，对分割后的两块数据分别处理。

对于要写入当前扇区内的数据，可以直接调用IopReadWriteSector函数并设置合适的参数来执行写扇区操作。如果整个数据不跨越扇区边界，当然就都写入当前扇区即可。

对于要写入下一个扇区内的数据，必须调用 FatGetFatEntryValue 函数根据当前簇号得到下一个簇号，如果得到的下一个簇号大于0xFF8，还需要调用 FatAllocateOneCluster函数分配一个新簇，并调用 FatSetFatEntryValue 函数将新簇链接到当前簇的后面。待下一个簇准备好后，可以根据下一个簇号计算出其对应的扇区号，然后就可以调用IopReadWriteSector 函数并设置合适的参数来执行写扇区操作。调用IopReadWriteSector 函数时使用的参数一定要设置正确，特别是扇区号、扇区内起始位置、写入数据缓冲区地址和写入的字节数目，在跨越扇区边界时这些参数都会有变化。

对于问题c，与问题b类似，必须使用循环来处理写入数据跨越多个扇区边界的问题。每次循环时只将合适的数据写入当前簇，在后面的循环中将余下的数据写入簇链中后面的簇，直到所有数据写入完毕。

每次向簇中写数据之前都需要判断是否需要分配新簇。

前一个簇号和当前簇号这两个变量对于管理簇链非常重要，在循环的过程中要注意维护好这两个 变量的值，保证在每次循环时这两个变量都保存了正确的簇号。 调用IopReadWriteSector 函数时使用的参数一定要设置正确，特别是扇区号、扇区内起始位置、写入数据缓冲区地址和写入的字节数目。

当前 FAT12 文件系统中一个簇只包含一个扇区，为了简化程序，可以不考虑一个簇中包含多个扇区的情况。

在函数结束前，注意要修改文件大小并返回实际写入的字节数量。

**（2）不足**

没有实现从指定起始字节数位置写入文件的操作；没有为簇增加多个扇区。

**（3）改进建议**

为了实现从指定起始字节数位置写入文件的操作，可以在EOSAPP.EXE中增加起始位置的参数。如果用户没有输入起始位置，就默认起始位置为文件开头。

在WriteFile、ObWrite、FatWrite函数的定义中没有初始OFFEST参数，所以需要在定义中加入该参数，并传给下一级。

FatWriteFile 函数有OFFEST参数，所以不需要再增加OFFEST，且该函数内，首先根据簇的大小，计算写入的起始位置在簇链的第几个簇中（从 0 开始计数）ClusterIndex = Offset / FatBytesPerCluster(&Vcb->Bpb);然后计算写位置在扇区内的字节偏移，OffsetInSector = Offset % Vcb->Bpb.BytesPerSector;

可以说FatWriteFile 函数已经基本实现了从指定起始字节数位置写入文件的操作。

**北京科技大学实验报告**

学院：计通学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期：2023 年 11 月 22 日

**实验名称：操作系统实验7 扩展实验1（3分）**

**实验目的：**以一个教学型操作系统EOS为例，再次从整体上理解EOS操作系统。

**实验环境：**Codecode实验平台和EOS操作系统

**实验内容：**

EOS 操作系统是一个包含一万多行源代码的小型操作系统，适用于教学和学生对操作系统的深入研究与学习，在开始之前需要对 EOS 操作系统的框架有一个深入的理解，这样可以方便读者从总体上把握 EOS 操作系统的结构，引导读者阅读 EOS 操作系统的源码，进而有针对性地、有重点地学习和研究 EOS 操作系统的相关内容。

**实验步骤：**

任务（一）：操作系统启动后，应用程序执行前，操作系统中有哪些进程和县城，如何创建？

**（1）补全start.c文件**

该文件中已经给出了部分提示，因此根据提示内容和内核文件中的函数，对待填充位置进行补全。其中，KiInitializeProcessor函数位于cpu.asm中，KiInitializeInterrupt在int.asm中，其余则在c语言源代码中，最终初始化程序代码如下（展示部分）。

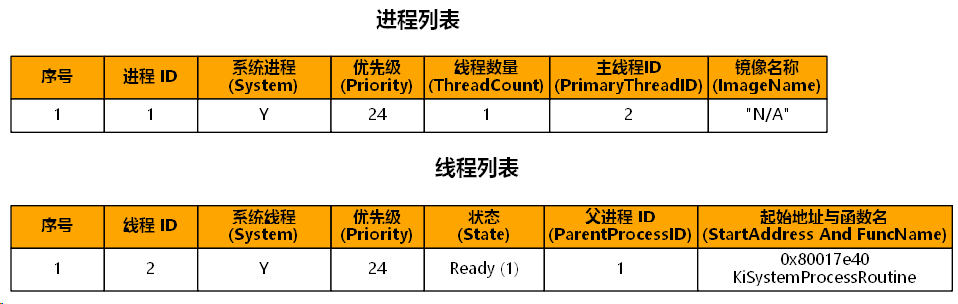
|  |
| --- |
| // 初始化处理器和中断。  // TODO:  KiInitializeProcessor();  KiInitializeInterrupt();  // 初始化可编程中断控制器和可编程定时计数器。  // TODO:  KiInitializePic();  KiInitializePit();  // 对各个管理模块执行第一步初始化，顺序不能乱。  // TODO:  MmInitializeSystem1(LoaderBlock);  ObInitializeSystem1();  PsInitializeSystem1();  IoInitializeSystem1();  // 创建系统启动进程。  // TODO:  PsCreateSystemProcess(KiSystemProcessRoutine);  // 执行到这里时，所有函数仍然在使用由 Loader 初始化的堆栈，所有系统线程  // 都已处于就绪状态。执行线程调度后，系统线程开始使用各自的线程堆栈运行。  // TODO:  KeThreadSchedule(); |

**（2）调试程序**

a、start.c文件中的PsCreateSystemProcess函数添加断点，命中后查看进程线程为空。

b、进入PsCreateSystemProcess内部，单步至PspCreateProcessEnvironment创建进程环境，可以看到已经创建一个系统进程。

c、单步至PspCreateThread创建线程，该线程是主线程。



d、单步至KeThreadSchedule触发48号中断（ke.h），创建控制台派遣进程、控制台线程，中断返回时执行线程调度。

任务（二）：操作系统启动之后，应用程序启动之前，查看软盘的使用情况和软盘包含的文件列表

**（1）补全代码，实现sd命令**

修改位置位于sysproc.c文件中的ConsoleCmdScanDisk函数处，用于扫描磁盘信息。主要使用VCB控制块来查看磁盘信息，获取文件系统信息后打印。

|  |
| --- |
| // 得到 FAT12 文件系统设备对象，然后得到卷控制块 VCB  // TODO:  FatDevice = (PDEVICE\_OBJECT)ObpLookupObjectByName(IopDeviceObjectType, "A:");  pVcb = (PVCB)FatDevice->DeviceExtension;  // 将卷控制块中缓存的 BIOS Parameter Block (BPB) ，以及卷控制块中的其它重要信息输出  // TODO:  fprintf(StdHandle, "\*\*\*\*\*\*\*\* BIOS Parameter Block (BPB) \*\*\*\*\*\*\*\*\n");  fprintf(StdHandle, "Bytes Per Sector : %d\n", pVcb->Bpb.BytesPerSector);  fprintf(StdHandle, "Sectors Per Cluster: %d\n", pVcb->Bpb.SectorsPerCluster);  fprintf(StdHandle, "Reserved Sectors : %d\n", pVcb->Bpb.ReservedSectors);  fprintf(StdHandle, "Fats : %d\n", pVcb->Bpb.Fats);  fprintf(StdHandle, "Root Entries : %d\n", pVcb->Bpb.RootEntries);  fprintf(StdHandle, "Sectors : %d\n", pVcb->Bpb.Sectors);  fprintf(StdHandle, "Media : 0x%X\n", pVcb->Bpb.Media);  fprintf(StdHandle, "Sectors Per Fat : %d\n", pVcb->Bpb.SectorsPerFat);  fprintf(StdHandle, "Sectors Per Track : %d\n", pVcb->Bpb.SectorsPerTrack);  fprintf(StdHandle, "Heads : %d\n", pVcb->Bpb.Heads);  fprintf(StdHandle, "Hidden Sectors : %d\n", pVcb->Bpb.HiddenSectors);  fprintf(StdHandle, "Large Sectors : %d\n", pVcb->Bpb.LargeSectors);  fprintf(StdHandle, "\*\*\*\*\*\*\*\* BIOS Parameter Block (BPB) \*\*\*\*\*\*\*\*\n\n");  fprintf(StdHandle, "First Sector of Root Directroy: %d\n", pVcb->FirstRootDirSector);  fprintf(StdHandle, "Size of Root Directroy : %d\n", pVcb->RootDirSize);  fprintf(StdHandle, "First Sector of Data Area : %d\n", pVcb->FirstDataSector);  fprintf(StdHandle, "Number Of Clusters : %d\n\n", pVcb->NumberOfClusters);  // 扫描 FAT 表，统计空闲簇的数量，并计算软盘空间的使用情况  // TODO:  FreeClusterCount = 0;  for (i = 2; i < pVcb->NumberOfClusters + 2; i++) {  if (0 == FatGetFatEntryValue(pVcb, i))  FreeClusterCount++;  }  UsedClusterCount = pVcb->NumberOfClusters - FreeClusterCount;  fprintf(StdHandle, "Free Cluster Count: %d (%d Byte)\n", FreeClusterCount, FreeClusterCount\*pVcb->Bpb.SectorsPerCluster\*pVcb->Bpb.BytesPerSector);  fprintf(StdHandle, "Used Cluster Count: %d (%d Byte)\n", UsedClusterCount, UsedClusterCount\*pVcb->Bpb.SectorsPerCluster\*pVcb->Bpb.BytesPerSector); |

**（2）补全代码，实现dir命令**

修改sysproc.c文件中的ConsoleCmdDir函数，通过控制台命令 dir 输出软盘根 目录中的文件信息。

|  |
| --- |
| // 得到 FAT12 文件系统设备对象，然后得到卷控制块 VCB  // TODO:  FatDevice = (PDEVICE\_OBJECT)ObpLookupObjectByName(IopDeviceObjectType, "A:");  pVcb = (PVCB)FatDevice->DeviceExtension;  // 分配一块虚拟内存做为缓冲区，然后将整个根目录区从软盘读入缓冲区。  // TODO:  pBuffer = NULL; // 不指定缓冲区的地址。由系统决定缓冲区的地址。  BufferSize = pVcb->RootDirSize; // 申请的缓冲区大小与根目录区大小相同。  MmAllocateVirtualMemory(&pBuffer, &BufferSize, MEM\_RESERVE | MEM\_COMMIT, TRUE);  RootDirSectors = pVcb->RootDirSize / pVcb->Bpb.BytesPerSector; // 计算根目录区占用的扇区数量  for(i=0; i<RootDirSectors; i++) {  // 将根目录区占用的扇区读入缓冲区  IopReadWriteSector( pVcb->DiskDevice,  pVcb->FirstRootDirSector + i,  0,  (PCHAR)pBuffer + pVcb->Bpb.BytesPerSector \* i,  pVcb->Bpb.BytesPerSector,  TRUE);  }  // 扫描缓冲区中的根目录项，输出根目录中的文件和文件夹信息  // TODO:  fprintf(StdHandle, "Name | Size(Byte) | Last Write Time\n");  for(i=0; i<pVcb->Bpb.RootEntries; i++) {    pDirEntry = (PDIRENT)(pBuffer + 32 \* i);  // 跳过未使用的目录项和被删除的目录项  if(0x0 == pDirEntry->Name[0]  || (CHAR)0xE5 == pDirEntry->Name[0])  continue;  FatConvertDirNameToFileName(pDirEntry->Name, FileName);    fprintf(StdHandle, "%s %d %d-%d-%d %d:%d:%d\n",  FileName, pDirEntry->FileSize, 1980 + pDirEntry->LastWriteDate.Year,  pDirEntry->LastWriteDate.Month, pDirEntry->LastWriteDate.Day,  pDirEntry->LastWriteTime.Hour, pDirEntry->LastWriteTime.Minute,  pDirEntry->LastWriteTime.DoubleSeconds);  }  // 释放缓冲区  // TODO:  BufferSize = 0; // 缓冲区大小设置为 0，表示释放全部缓冲区  MmFreeVirtualMemory(&pBuffer, &BufferSize, MEM\_RELEASE, TRUE); |

任务（三）：操作系统如何从控制台获取用户输入的命令？

**（1）补全源代码，实现键盘Read功能**

修改位置位于keyboard.c文件中的KbdRead函数处，实现键盘驱动程序的Read功能，对键盘事件响应，读取缓冲区。

|  |
| --- |
| // 读写大小须是键盘事件结构体大小的整数倍。  // TODO:  Request -= Request % sizeof(KEY\_EVENT\_RECORD);  while (Count < Request) {  // 阻塞等待直到缓冲区非空。  // TODO:  PsWaitForEvent(&Ext->BufferEvent, INFINITE);  // 读取缓冲区，如果缓冲区被读空了则复位非空事件。  // 注意，要和键盘中断服务程序互斥访问键盘事件缓冲区，要禁止中断。  // TODO:  IntState = KeEnableInterrupts(FALSE);  Count += IopReadRingBuffer(Ext->Buffer, Buffer + Count, Request - Count);  if (IopIsRingBufferEmpty(Ext->Buffer)) {  PsResetEvent(&Ext->BufferEvent);  }  KeEnableInterrupts(IntState);  } |

**（2）补全源代码，实现键盘中断服务程序**

修改位置位于/keyboard.c文件中的KbdIsr函数，该函数是键盘中断服务程序，用于服务键盘中断，在收到键盘按键信号后发出中断请求，会调用该程序，该函数将对应的中断信号封装入结构体再写入缓冲区。

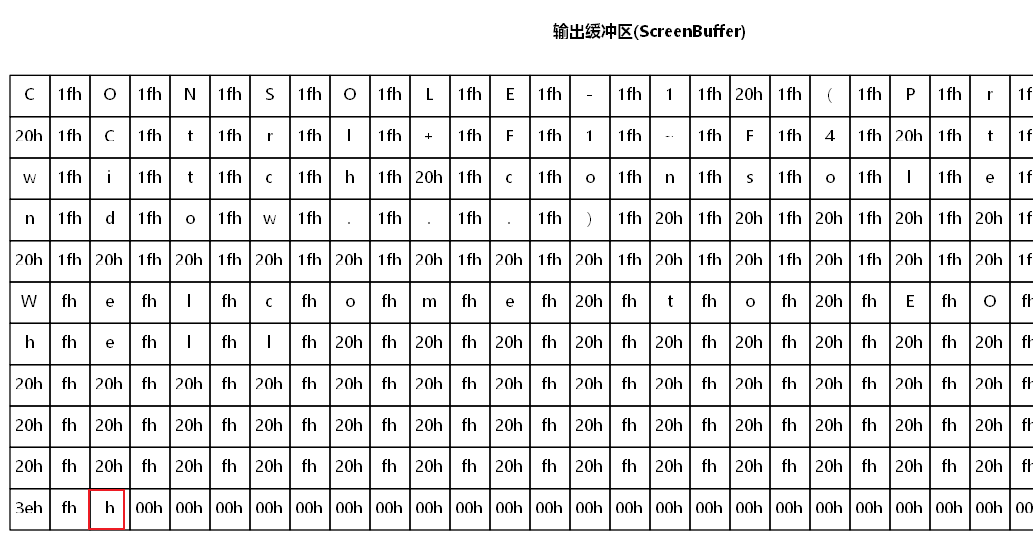
|  |
| --- |
| // 从8042数据端口读取键盘扫描码。  // TODO:  ScanCode[i] = READ\_PORT\_UCHAR((PUCHAR)KEYBOARD\_PORT\_DATA);  i++; |
| // 将键盘事件写入缓冲区，并设置键盘非空事件。  //TODO:  IopWriteRingBuffer(Ext->Buffer, &KeyEventRecord, sizeof(KEY\_EVENT\_RECORD));  PsSetEvent(&Ext->BufferEvent); |

**（3）程序调试**

在console.c文件中添加断点，命中后在PKEYBOARD\_DEVICE\_EXTENSION Ext = (PKEYBOARD\_DEVICE\_EXTENSION)KbdDevice[0]->DeviceExtension;处添加断点。进入Obread函数，将键盘设备作为文件对象访问。

触发断点处会依据键盘扫描码获取对应的虚码，会判断虚码的类型，如果是扩展码则不处理，如果需要显示则写入显示缓冲区，无论何种键盘输入都会被当作读入事件写入缓冲区。

根据键盘缓冲区的虚码，显示的模块会读取显示缓冲区的内容，显示对应的字符在显示器上。



任务（四）：应用程序进程如何在操作系统内核中创建？

**（1）补全代码，实现PsCreateProcess函数**

修改位置位于补全ke/sysproc.c文件中的PsCreateProcess函数，创建一个应用程序进程。

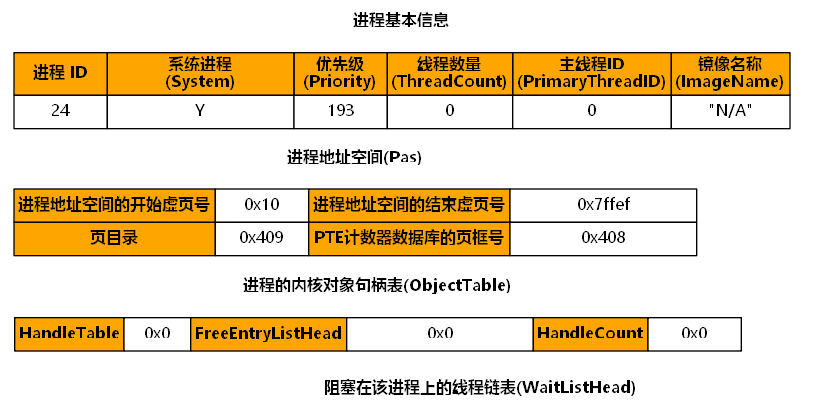
|  |
| --- |
| // 创建一个进程环境（进程的控制块以及进程的地址空间和句柄表）。  // TODO:  Status = PspCreateProcessEnvironment(8, ImageName, CmdLine, &ProcessObject); |
| // 加载可执行映像（程序的指令和数据）到新建进程的用户地址空间中。  // TODO:  Status = PspLoadProcessImage( ProcessObject,  ProcessObject->ImageName,  &ProcessObject->ImageBase,  (PVOID\*)&ProcessObject->ImageEntry ); |
| // 创建新进程的主线程，所有进程的主线程都从函数PspProcessStartup开始执行。  // TODO:  Status = PspCreateThread( ProcessObject,  0,  PspProcessStartup,  NULL,  CreateFlags,  &ThreadObject ); |

**（2）调试程序**

a、PsCreateProcess处添加断点，键入hello.exe后命中断点，第一个参数Image是路径，进入PsCreateProcess内部，命中PspCreateProcessEnvironment，进入内部后ObCreateObject处添加断点命中。单步至MmCreateProcessAddressSpace并进入。

b、单步至MmAllocateSystemPool函数，分配内存块，按照先零页再空闲页的顺序，成功分配虚拟页和物理块。

c、在mm/pas.c中，将物理页框号映射到页目录和PTE计数器数据库，并对页目录进行初始化。



任务（五）：应用程序进程的主线程如何创建和状态转换？

**（1）补全代码，实现加载EOS应用程序的可执行文件到内存**

修改位置位于ps/peldr.c文件中的PspLoadProcessImage函数处，加载 EOS 应用程序的 可执行文件到内存。

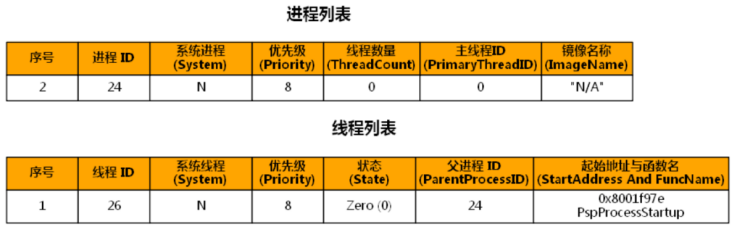
|  |
| --- |
| // 打开可执行文件。  // TODO:  Status = IoCreateFile( ImageName,  GENERIC\_READ,  FILE\_SHARE\_READ,  OPEN\_EXISTING,  0,  &FileHandle ); |
| // 当前线程附着到新建进程的地址空间中执行。  // TODO:  PspThreadAttachProcess(Process); |
| // 读取PE文件的全部头。  // TODO:  Status = ObRead( FileHandle,  FileHeaders,  0x400,  &NumberOfBytesRead ); |
| // 验证是否EOS应用程序。  // TODO:  if (NtHeaders->OptionalHeader.MajorSubsystemVersion != IMAGE\_SUBSYSTEM\_EOS\_CUI) {  Status = STATUS\_INVALID\_APP\_IMAGE;  break;  } |
| // 在当前进程地址空间中分配虚拟内存。  // TODO:  Status = MmAllocateVirtualMemory( &AppImageBase,  &VirtualSize,  MEM\_RESERVE,  FALSE ); |
| // 将文件指针移到节在文件中的位置，如果超出文件长度则说明文件无效。  // TODO:  Status = IoSetFilePointer( FileHandle,  SectionHeader->PointerToRawData,  FILE\_BEGIN,  &FilePointer); |
| // 读取节的内容到内存中。  // TODO:  Status = ObRead( FileHandle,  SectionAddress,  VirtualSize,  &NumberOfBytesRead ); |
| // 下面将应用程序和内核进行连接。  // TODO:  n = NtHeaders->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].VirtualAddress;  if (0 != n) {  if (!PspLinkAppWithKernel(AppImageBase, (PIMAGE\_IMPORT\_DESCRIPTOR)(AppImageBase + n))) {  Status = STATUS\_SYMBOL\_NOT\_FOUND;  break;  }  } |
| // 释放PE文件头，不再使用。  // TODO:  if (NULL != FileHeaders) {  MmFreeSystemPool(FileHeaders);  } |
| // 当前线程返回所属进程地址空间继续执行。  // TODO:  PspThreadAttachProcess(PspCurrentProcess); |

**（2）程序调试**

a、与任务（四）类似，通过PsCreateProcess创建进程，并在分配内存块后对进程控制块进行初始化。

b、通过PspLoadProcessImage函数加载可执行映像，调用 PspThreadAttachProcess 函数将当前线程附着到新建进程的地址空间中执行。这个过程依赖文件系统，需要按照镜像名查找文件并加载。

c、创建应用程序的主线程，通过PspCreateThread函数进行，添加断点命中后进入内部，使用PspProcessStartup作为线程函数，其中调用了应用程序可执行文件映像的入口。ObCreateObject函数创建线程对象，在之后分配虚拟内存并初始化线程控制块。



d、PspReadyThread函数使应用程序线程进入就绪。此时，线程创建完毕，需要PspThreadSchedule进行线程调度。



**任务（六）：应用程序的执行结果如何输出到控制窗口？**

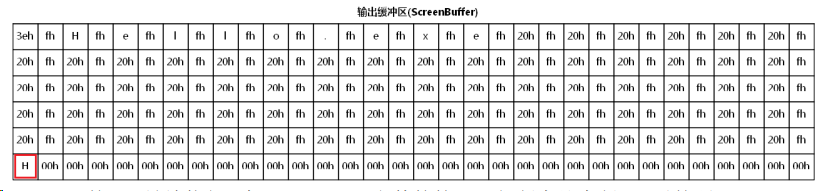
**（1）补全源代码，实现应用程序结果输出到窗口**

修改位置位于console.c文件中的IopWriteConsoleOutput函数，这是写控制台的输出缓冲区。由于控制台写入是临界区操作，因此需要先使用PsWaitForMutex函数上锁，缓冲区后再解锁，具体的写入操作就是将当前要输出的内容输入到输出缓冲区即可，使用的是IopWriteScreenBuffer函数。

|  |
| --- |
| ULONG i;  PsWaitForMutex(&Console->AccessMutex, INFINITE);  for (i = 0; i < NumberOfBytesToWrite; i++) {  IopWriteScreenBuffer( Console->ScreenBuffer,  &Console->CursorPosition,  ((PCHAR)Buffer)[i],  Console->TextAttributes);  }  // 如果控制台是激活的，那么同时还要更新显示器上的光标位置。  // 注意：要互斥访问变量IopActiveConsole。  PsWaitForMutex(&IopActiveMutex, INFINITE);  if (IopActiveConsole == Console) {  IopSetScreenCursor(Console->ScreenBuffer, Console->CursorPosition);  }  PsReleaseMutex(&IopActiveMutex);  PsReleaseMutex(&Console->AccessMutex);  \*NumberOfBytesWritten = NumberOfBytesToWrite;  return STATUS\_SUCCESS; |

**（2）调试程序**

当应用程序调用printf函数时，会通过eos提供的api接口函数writefile调用Obwrite函数。在Obwrite内部调用IopWriteConsoleOutput函数，其中会进行IopWriteScreenBuffer操作，即将字符写入屏幕缓冲区。



**任务（七）：应用程序进程如何退出？**

**（1）补全源代码，实现结束制定进程和线程的功能**

退出程序时需要完成两个操作，分别是结束当前进程和结束当前线程，也就是将要补全的两段代码。

**a、结束进程**

释放所有资源，将进程结束标志置为结束码，主线程设置为NULL，然后唤醒所有线程，将所有线程销毁。最后删除线程环境，再执行线程调度，唤醒其他线程执行。

|  |
| --- |
| BOOL IntState;  PTHREAD Thread;  IntState = KeEnableInterrupts(FALSE);  if (NULL != Process->PrimaryThread) {  // 设置进程结束标志（主线程指针为NULL）和结束码。  Process->PrimaryThread = NULL;  Process->ExitCode = ExitCode;  // 唤醒等待进程结束的所有线程。  while (!ListIsEmpty(&Process->WaitListHead)) {  PspWakeThread(&Process->WaitListHead, STATUS\_SUCCESS);  }  // 结束进程内的所有线程。  // 注意：并不急于在结束每个线程后都立刻执行线程调度，所有线程都被结束后再执  // 行一次调度即可。  while (!ListIsEmpty(&Process->ThreadListHead)) {  Thread = CONTAINING\_RECORD(Process->ThreadListHead.Next, THREAD, ThreadListEntry);  PspTerminateThread(Thread, ExitCode, TRUE);  }  // 删除进程环境。  PspDeleteProcessEnvironment(Process);  // 执行线程调度。  PspThreadSchedule();  }  KeEnableInterrupts(IntState); |

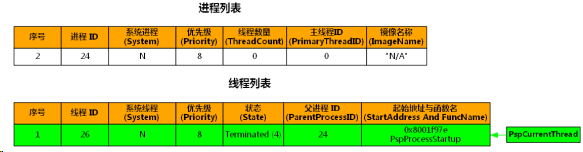
**b、结束线程**

具体到结束线程则需要区分是主线程还是子线程，如果是子线程则直接结束，如果是主线程则需要等待子线程全部结束再结束，为避免有线程还在阻塞，因此唤醒所有线程。然后将线程转入结束状态，并且释放内核栈，回收资源。

|  |
| --- |
| STATUS Status;  BOOL IntState;  SIZE\_T StackSize;  IntState = KeEnableInterrupts(FALSE);  ASSERT(Thread->State != Zero);  if (Thread->State != Terminated) {  if (Thread == Thread->Process->PrimaryThread) {  // 被结束线程是所在进程的主线程，结束线程所在的整个进程。  PspTerminateProcess(Thread->Process, ExitCode);  } else {  // 唤醒等待线程结束的所有线程。  while (!ListIsEmpty(&Thread->WaitListHead)) {  PspWakeThread(&Thread->WaitListHead, STATUS\_SUCCESS);  }  // 线程脱离目前所处状态并转入结束状态。  if(Ready == Thread->State) {  PspUnreadyThread(Thread);  } else if (Waiting == Thread->State) {  PspUnwaitThread(Thread);  }  Thread->State = Terminated;  #ifdef \_DEBUG  RECORD\_TASK\_STATE(ObGetObjectId(Thread) , TS\_STOPPED, Tick);  #endif  ListInsertTail(&PspTerminatedListHead, &Thread->StateListEntry);  // 设置线程结束码并将线程从进程的线程链表中移除。  Thread->ExitCode = ExitCode;  ListRemoveEntry(&Thread->ThreadListEntry);  // 释放线程的内核模式栈。  // 注意：如果当前线程正在结束自己，则不能释放线程正在使用的内核栈。  if (Thread != PspCurrentThread) {  StackSize = 0;  Status = MmFreeVirtualMemory( &Thread->KernelStack,  &StackSize,  MEM\_RELEASE,  TRUE );  ASSERT(EOS\_SUCCESS(Status));  }  // 如果被删除线程不是系统线程，还需释放线程的用户模式栈。目前所有线  // 程都执行在内核模式栈中，没有用户模式栈，忽略此操作。  // 线程结束，线程不再保留对自己的引用。  ObDerefObject(Thread);  // 注意：正在结束进程时不需要执行线程调度，因为结束进程函数最后会执行线程调  if (!IsTerminatingProcess) {  PspThreadSchedule();  }  }  }  KeEnableInterrupts(IntState); |

**（2）调试程序**

通过操作系统提供的接口函数PsExitProcess退出当前进程，其中会调用我们补全的函数PspTerminateProcess，需要进行的工作有：设置进程结束的标志和结束码、唤醒进程结束的线程、删除进程环境、线程调度。通过“进程线程”查看线程状态。



**结果分析：**

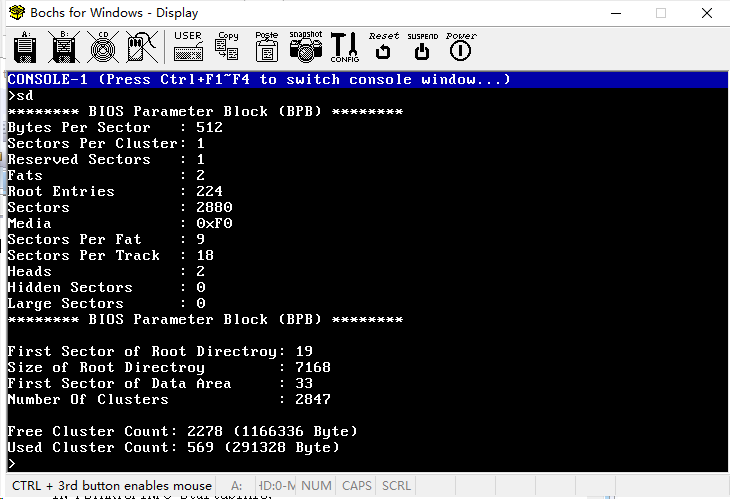
下面依此对实现的七个任务进行测试，不同任务实现成功的现象不同。

**任务（一）：**

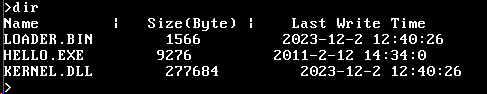
操作系统启动成功，说明该功能完成。

**任务（二）：**

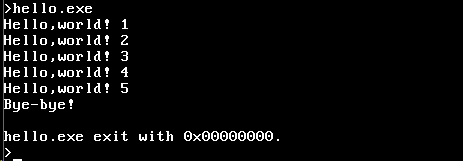
（1）键入sd命令，查看软盘情况



（2）键入dir命令，查看软盘根目录的文件信息



（3）键入hello.exe，查看应用程序的结果。



**任务（三）--（七）：**

键入hello.exe后，能够正常打印，并结合前述的调试过程，说明实现成功。

**北京科技大学实验报告**

学院：计通学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期：2023 年 11 月 26 日

**实验名称：操作系统实验7 扩展实验2（3分）**

**实验目的：**掌握并修改EOS内核的引导过程，使Bochs可以从平坦的软盘镜像启动操作系统内核。掌握并修改EOS内核的引导过程，使物理机可以通过U盘进行引导并启动操作系统内核。

**实验环境：**Codecode实验平台和EOS操作系统

**实验内容：**

通过对EOS操作系统引导程序进行修改，使其能够在Bochs虚拟机和物理机上平坦的启动。

**实验步骤：**

**任务（一）：改进EOS内核引导过程，实现裸机从无文件系统的平坦软盘镜像引导**

裸机裸机无文件系统的平坦软盘镜像引导OS启动需要将镜像加载到内存中，因此需要先获得内存的信息，主要是物理内存的大小，然后将内核加载到指定地址。

这个部分需要修改的代码就是将原本的含有FAT12的代码删除，然后对int 0x13相关参数的查询。

    push word szLoading

    call TextOut

    mov cx , 0x258;600

    mov si , 0x5

    mov di , 0x1000

ReadKernel:

    push cx

    push 0

    push di

    push 0x0001

    push si

    call ReadSector

    add di , 0x20

    add si , 0x1

    pop cx

    loop ReadKernel

这里用到了ReadSector的子程序，这个子程序是负责读取簇的，但是原本的是读取FAT的簇相关的，不能直接用作无文件系统的平坦加载。

    mov ax, [bp + 4]            ; ax = wSector

    mov bl, 18                  ;

    div bl                      ; y 在 al 中, z 在 ah 中

    inc ah                      ; z ++

    mov cl, ah                  ; cl <- 起始扇区号

    mov dh, al                  ; dh <- y

    shr al, 1                   ; y >> 1 (其实是 y / Heads, 这里 Heads = 2)

    mov ch, al                  ; ch <- 柱面号

    and dh, 1                   ; dh & 1 = 磁头号

    mov dl, 0

**任务（二）：改进EOS内核引导过程，实现物理机从U盘引导加载EOS内核**

本实验要求从物理机上启动EOS内核，因此首先需要在MBR中加入硬盘分区表，boot代码原本采用chs模式启动，但是物理机使用的是lba方式，所以需要对loader进行一些修改。

大部分的内容已经在工程中给出，这里我们需要补充从镜像文件读600个扇区到物理内存0x10000处的代码。

    push word szLoading

    call TextOut

ReadKernel:

    mov ax,cs   ;初始化寄存器

    mov ds,ax

    mov ds,ax

    mov es,ax

    mov ah,0x42

    mov dl,0x80

    mov si,packet

    int 0x13

    jc  error

    mov ax,[bufferseg]

    mov bx,bufferseg

    add ax,0x0020

    mov [bx],ax

    ;将起始LBA加1，以读取下一个扇区

    mov ax,[blockNum]

    mov bx,blockNum

    add ax,1

    mov [bx],ax

    cmp ax,600

    je  finish

    jmp ReadKernel

packet:

packet\_size:     db 10h             ;packet大小，16个字节

reserved:        db 0

count:           dw 1               ;读1个扇区

bufferoff:       dw 0x0000          ;读到内存0x1000处，偏移地址

bufferseg:       dw 0x1000              ;段地址

blockNum:        dd 5               ;起始LBA块

                 dd 0

finish:

error:

完成代码后就可以根据实验指导书的内容进行镜像装载，在此之前需要注释掉可能会导致内核初始化异常的相关代码，包括ioinit.c中的软驱初始化。mempool.c中的内存校验断言，keyboard.c中用于更新键盘LED灯的相关代码。

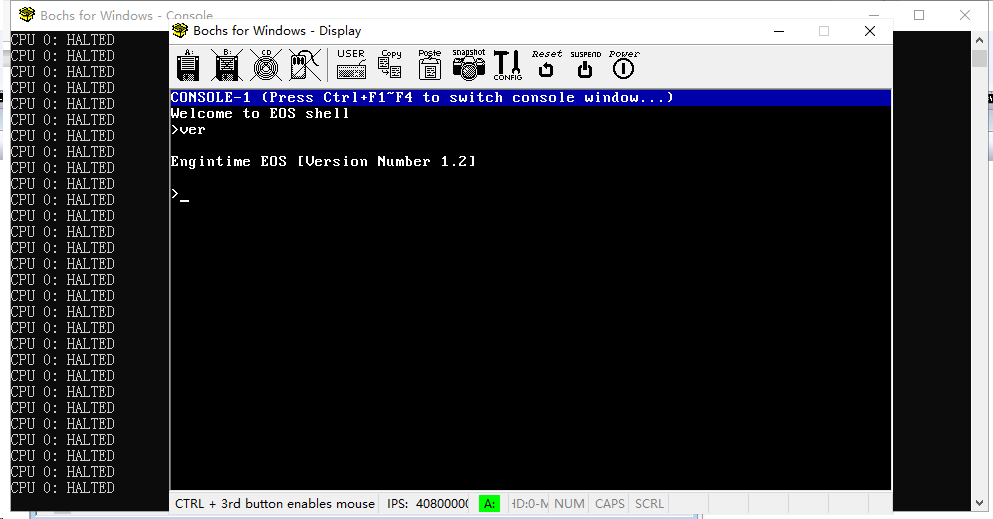
装载成功后的U盘应该为如下布局。



**结果分析：**

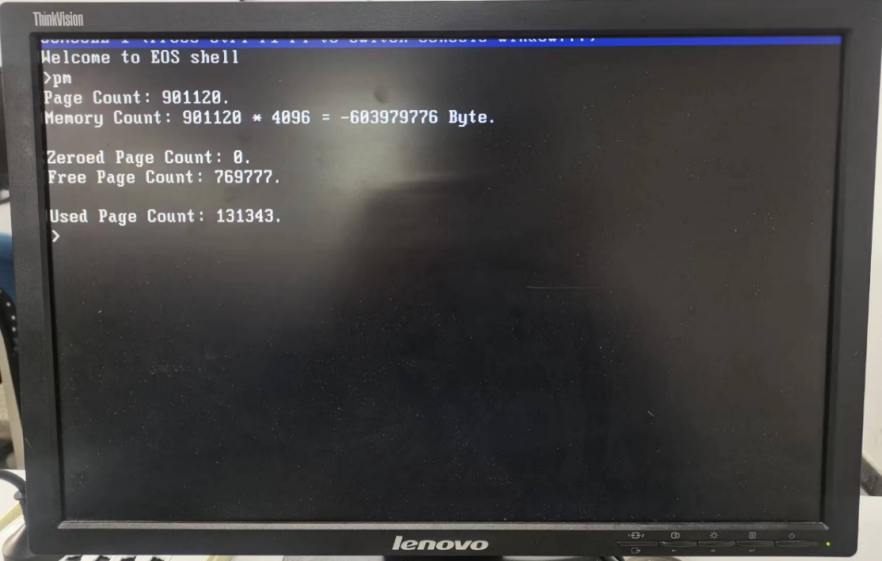
**任务（一）：**

按照要求修改配置后，生成项目并开始调试，键入“c”后控制台光标闪烁；键入“ver”命令，正常打印信息，说明修改后的代码可以直接由硬盘启动到Bochs虚拟机，启动调试可以看到测试成功。



**任务（二）：**

将U盘插入PC的USB接口，调整BIOS，将U盘作为启动盘，随后机器重新启动，最终可以看到EOS成功在物理机上启动。



**北京科技大学实验报告**

学院：计通学院 专业：信息安全 班级：信安211

姓名：李晓坤 学号：U202141863 实验日期：2023 年 12 月 1 日

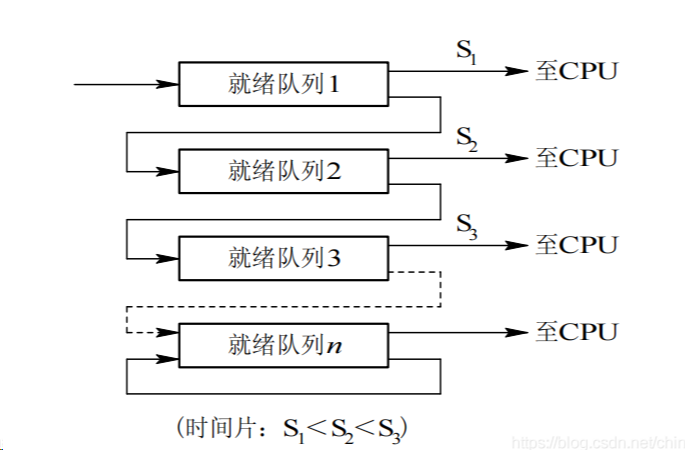
**实验名称：操作系统实验7 扩展实验3（3分）**

**实验目的：**理解并实现多级反馈队列

**实验环境：**Codecode实验平台与EOS操作系统

**实验内容：**

通过线程调度，在时间片轮转调度算法的基础上修改完善，最终实现多级反馈队列调度算法，解决低优先级进程长时间得不到调度的情况。如下图是多级反馈队列的简单示意图。针对实验指导书中的要求，在接下来的实验步骤中，会详细介绍。



**实验步骤：**

**（一）实现时间片轮转调度算法**

该实验要求在基础性实验中已经完成，受限于实验报告的篇幅，这里不再赘述，详细内容见之前的报告内容。

**（二）修改时间片大小**

修改时间片的大小 TICKS\_OF\_TIME\_SLICE 为 100，方便观察执行后的效果。主要修改位于psp.h文件中的宏定义。

|  |
| --- |
| // 用于时间片轮转调度的时间片大小（即时间片包含的时钟滴答数）  #define TICKS\_OF\_TIME\_SLICE 100 |

**（三）修改Sleep时间**

在控制台命令“rr”的处理函数中，将 Sleep 时间更改为 200\*1000，这样可以有 充足的时间查看优先级降低后的效果。主要修改位于sysproc.c文件中的ConsoleCmdRoundRobin函数。

|  |
| --- |
| // 当前线程等待一段时间。由于当前线程优先级 24 高于新建线程的优先级 8，  // 所以只有在当前线程进入“阻塞”状态后，新建的线程才能执行。  //Sleep(40 \* 1000);  Sleep(200 \* 1000); |

**（四）修改TCB**

修改线程控制块（TCB）结构体，在其中新增两个成员，一个是线程整个生命周期中合计使用的时间片数量，另一个是线程的初始时间片数量。在psp.h文件中的\_THREAD结构体内增加成员；在sysproc.c的ConsoleCmdRoundRobin中进行初始化。

|  |
| --- |
| ULONG InitTicks; //线程初始时间片数量，用于多级反馈队列  ULONG UsedTicks; //线程整个生命周期合计使用的时间片数量，多级反馈队列 |
| //初始化增加的两个变量  Status = ObRefObjectByHandle(ThreadHandleArray[i], PspThreadType, (PVOID\*)&Thread);  Thread->InitTicks = 100;  Thread->UsedTicks = 0; |

**（五）修改rr命令的输出内容和格式**

修改“rr”命令在控制台输出的内容和格式，不再显示线程计数，而是显示线程初始化时间片的大小，已使用时间片的合计数量，剩余时间片的数量。“rr”命令最终会调用\_THREAD\_PARAMETER函数进行打印，因此主要修改该函数，在sysproc.c文件中，其中修改的地方已标出。

|  |
| --- |
| PRIVATE  ULONG  ThreadFunction(  PVOID Param  )  {  ULONG i;  UCHAR Priority;  COORD CursorPosition;  PTHREAD\_PARAMETER pThreadParameter = (PTHREAD\_PARAMETER)Param;  PTHREAD Thread;//线程指针  STATUS Status;  //获取线程  Status = ObRefObjectByHandle(CURRENT\_THREAD\_HANDLE, PspThreadType, (PVOID\*)&Thread);  if (!EOS\_SUCCESS(Status)) {  return Status;  }  // 根据线程参数设置输出内容显示的位置。  CursorPosition.X = 0;  CursorPosition.Y = pThreadParameter->Y;  // 在线程参数指定的行循环显示线程执行的状态。死循环。通过开关中断互斥访问控制台。  // 格式：Thread 序号 (优先级): 执行计数  for (i = 0; ; i++) {  \_\_asm("cli");  PsGetThreadPriority(CURRENT\_THREAD\_HANDLE, &Priority);  SetConsoleCursorPosition(pThreadParameter->StdHandle, CursorPosition);  //fprintf(pThreadParameter->StdHandle, "Thread %d (ID:%d, Priority:%d, ): %u ",  // pThreadParameter->Y, ObGetObjectId(PspCurrentThread),Priority, i);  //打印内容的修改  fprintf(pThreadParameter->StdHandle, "Thread %d , ID:%d, Priority:%d, InitTicks: %u , UsedTicks:%u, RemainderTicks:%d ",  pThreadParameter->Y, ObGetObjectId(PspCurrentThread), Priority,  Thread->InitTicks,  Thread->UsedTicks,  Thread->RemainderTicks);  \_\_asm("sti");  }  return 0;  } |

**（六）实现多级反馈队列**

修改位置主要位于sched.c文件中的PspRoundRobin函数，该函数原本是时间片轮转调度，在此基础上修改为多级反馈队列调度。主要修改思路如下：

（1）正常情况下，每次触发中断时剩余时间片减少，已消耗时间片增加。

（2）对时间片是否耗尽进行判断。若时间片已经耗尽：【1】线程优先级不是最低，则修改优先级、时间片；【2】线程优先级最低，则分配时间片。

（3）线程调度

|  |
| --- |
| VOID  PspRoundRobin(  VOID  )  {  //多级反馈队列的实现  if (NULL != PspCurrentThread && Running == PspCurrentThread->State) {//正在运行的线程才调用时间片轮转  //时间片减少1  PspCurrentThread->RemainderTicks--;  //  PspCurrentThread->UsedTicks++;  if(PspCurrentThread->RemainderTicks==0)//时间片耗尽  {  //重新分配时间片  //优先级不是最低  if (PspCurrentThread->Priority > 0) {  PspCurrentThread->Priority--;  PspCurrentThread->InitTicks = (9- PspCurrentThread->Priority) \* TICKS\_OF\_TIME\_SLICE;  PspCurrentThread->RemainderTicks = PspCurrentThread->InitTicks;  }  //优先级最低  else {  PspCurrentThread->RemainderTicks = PspCurrentThread->InitTicks;  }  //PspCurrentThread->RemainderTicks=TICKS\_OF\_TIME\_SLICE;  //若存在相同优先级的就绪线程，则将该线程放至就绪队列末尾  if(BIT\_TEST(PspReadyBitmap, PspCurrentThread->Priority))  {  PspReadyThread(PspCurrentThread);  }  }  }  return;  } |

**（七）增加空格事件提升线程优先级**

由于EOS 没有提供鼠标，可以使用键盘事件或者控制台命令使线程优先级提升。由于键盘事件与线程之间没有建立一个明确的会话关系，所以还需要解决使用键盘事件提升哪个线程优先级的问题。一个简单的方式是，在键盘的中断处理程序中（在 io/driver/keyboard.c 文件的 396 行的 KbdIsr 函数），如果当前线程（注意不能 是 2 号线程）处于运行状态并且优先级大于 0 小于 8 的话（由于空闲线程的优先级为 0，不能更改该线程的优先级，如果当前线程的优先级为 8，没有必要再做提升 线程优先级的操作），按下空格键，响应键盘事件后，就将其优先级提升为默认的优先级即可。添加如下内容：

|  |
| --- |
| //增加空格事件，提高当前线程（除了2号）的优先级，最高为8（为了测试）  case VK\_SPACE:  if (PspCurrentThread->Priority > 0 && PspCurrentThread->Priority < 8 && ObGetObjectId(PspCurrentThread) != 2) {  PsSetThreadPriority(CURRENT\_THREAD\_HANDLE, 8);  }  break; |

**（八）增加控制台“up”命令，提升指定线程的优先级**

使用控制台命令提升线程优先级，在 EOS 操作系统中实现一个“up ThreadID”命令,通过输入的线程 ID 来提升对应线程的优先级。在实现命令的过程中需要做如下判断：需要提升线程的优先级应该大于0 并且小于 8，如果是处于就绪状态的线程，需要先将该线程移出队列，然后设置该线程的优先级为默认值 8，并设置线程的初始时间片大小和剩余时间片大小，如果是处于运行状态或阻塞状态的线程，直接设置线程的优先级即可。主要修改sysproc.c中的三处地方：（1）函数声明；（2）增加命令；（3）函数实现

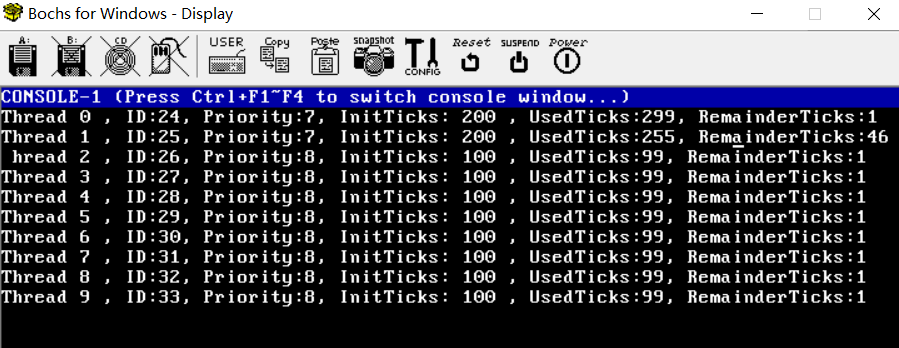
|  |
| --- |
| //up命令函数声明  PRIVATE  VOID  ConsoleCmdUp(  IN HANDLE StdHandle,  IN PCSTR Arg  ); |
| else if (0 == stricmp(Line, "up")) {//增加命令up  ConsoleCmdUp(StdHandle,Arg);  continue;  } |
| //下面增加一个up命令，用于提升线程优先级：up ID  PRIVATE  VOID  ConsoleCmdUp(//实现up命令函数  IN HANDLE StdHandle,  IN PCSTR Arg  )  {  ULONG ThreadID;  HANDLE hThread;  //新增  STATUS Status;  BOOL IntState;  PTHREAD Thread;  //获取线程ID  ThreadID = atoi(Arg);  if (0 == ThreadID) {  fprintf(StdHandle, "Please input a valid thread ID.\n");  return;  }  //获取线程句柄  hThread = (HANDLE)OpenThread(ThreadID);  if (NULL == hThread) {  fprintf(StdHandle, "%d is an invalid thread ID.\n", ThreadID);  return;  }  //获取线程对象  Status = ObRefObjectByHandle(hThread, PspThreadType, (PVOID\*)&Thread);  if (EOS\_SUCCESS(Status)) {  IntState = KeEnableInterrupts(FALSE); // 关中断  //就绪状态的线程处理  if (Ready == Thread->State) {  PspUnreadyThread(Thread);  Thread->Priority = 8;  Thread->InitTicks = 100;  Thread->RemainderTicks = 100;  PspReadyThread(Thread);  }  //正在运行，只设置优先级  else if (Running == Thread->State ) {  Thread->Priority = 8;  }  //阻塞状态，修改优先级和时间片  else if (Thread->State == Waiting) {  Thread->Priority = 8;  Thread->InitTicks = 100;  Thread->RemainderTicks = 100;  }  KeEnableInterrupts(IntState); // 开中断  ObDerefObject(Thread);//释放指针  }  // 关闭线程句柄  CloseHandle(hThread);  } |

**结果分析：**

针对各个要求进行测试并对结果进行分析。

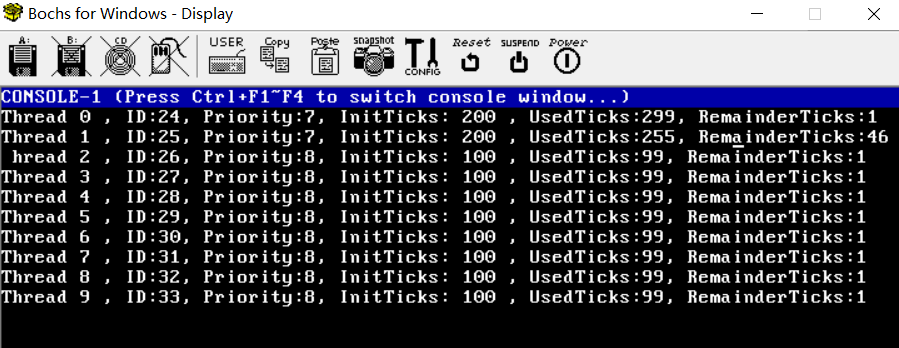
**（一）到（五）**

均可以通过一个（五）来进行验证。进行调试，键入“rr”命令后，可以看到新创建的个线程在进行调度，循环打印线程的时间片使用情况。

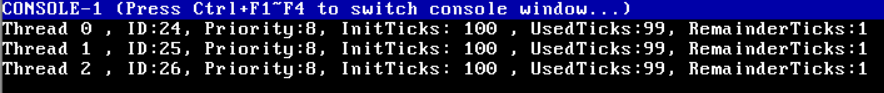


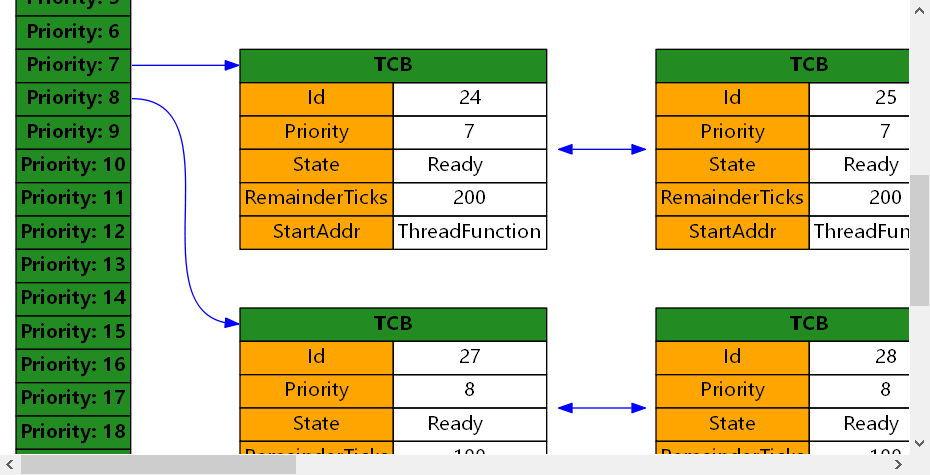
**（六）多级反馈队列**

进行调试，键入“rr”命令后，可以看到个线程得到调度，且时间片用尽后优先级会降低，分配的时间片会增加，成功验证。



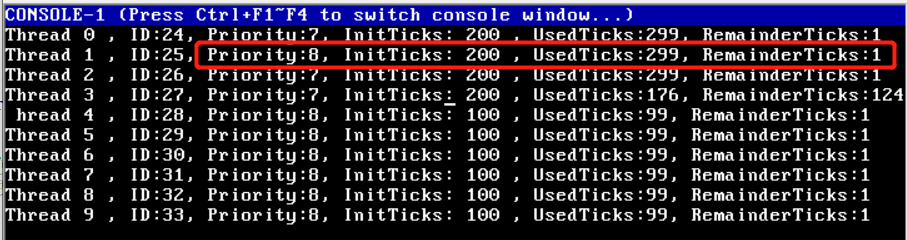
从就绪队列窗口看到，未执行的线程优先级8，时间片100，执行过一次的线程优先级7，时间片100。





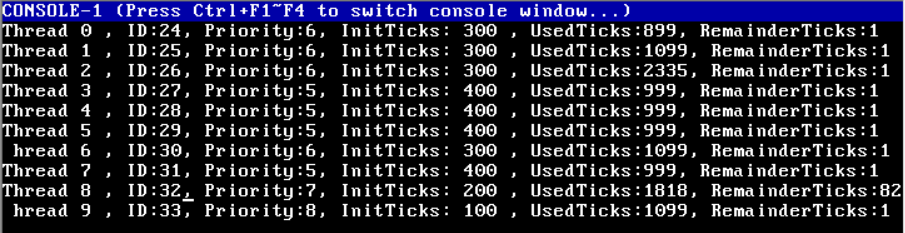
**（七）键盘事件提升线程优先级**

开始调试，键入“rr”命令后，按下空格，可以看到对应线程的优先级提升为8，成功实现。



**（八）“up”命令提升指定线程优先级**

开始调试，键入“rr”命令后，切换到控制台2，键入“up 33”提升id为33的线程的优先级。回到控制台1可以看到33号线程优先级得到提高。



24号线程运行时“按空格”或用up指令，可以看到线程当时优先级提升为8。继续执行该线程，执行结束后，该线程优先级减一，变为7，如下图：

