**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

**实验名称：操作系统实验1 操作系统启动与线程状态转换（3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；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成功启动的全过程，查看EOS启动后的状态和行为；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操作系统的启动**

（简要说明在本部分实验中完成的主要工作，总结EOS操作系统的启动过程以及启动后的工作方式）

EOS操作系统的启动过程包括以下基本步骤：a、CPU执行BIOS的自检和初始化程序，然后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程序执行。b、Boot程序将Loader.bin文件加载到物理内存的0x1000处，并执行Loader程序。c、Loader程序将操作系统内核Kernel.dll文件加载到物理内存中，启动CPU的保护模式和分页机制，最后执行Kernel.dll的入口点函数。d、EOS内核完成初始化后，用户可以与之进行交互，此时EOS启动完成。

**2）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）

以下是对EOS操作系统启动过程的详细描述：

1.BIOS程序执行：在计算机加电时，CPU的寄存器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程序执行自检和初始化后，将软盘引导扇区（512字节）加载到物理地址0x7c00-0x7dff，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。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。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。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。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。在FindFile部分查找Loader.bin文件，并检查文件是否符合要求。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。结构体LOADER\_PARAMETER\_BLOCK用于描述加载器的参数。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。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。在ProtectionMode代码块中，会对页表进行初始化，然后启动分页机制。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：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。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。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。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。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。

5.系统进程的执行过程：系统进程（ID为1）的主线程（ID为2）首先执行KiSystemProcessRoutine函数，在该函数中，系统进程创建了初始化线程KiInitializationThread，然后将主线程的优先级降至0，并进入死循环状态。同时，系统进程还创建了控制台派遣函数IopConsoleDispatchThread，用于将键盘事件派遣到活动的控制台线程中。系统进程还创建了控制台线程KiShellThread，这些线程使得操作系统能够响应用户的输入和请求，并管理系统资源的分配和调度。

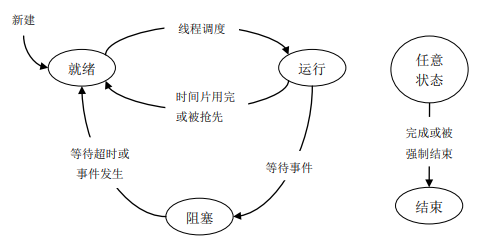
6.初始化线程的执行过程：初始化线程（KiInitializationThread）是系统进程中的一个线程，负责执行系统的初始化工作。它首先初始化各个管理器，包括中断管理器、内存管理器等，然后创建系统的第一个用户进程，即Shell进程，用于与用户进行交互。初始化线程的工作完成后，它将进入休眠状态，等待系统的其他部分需要进行初始化时唤醒它。

7.Shell进程的执行过程：Shell进程是用户与操作系统交互的主要界面，通常是一个命令解释器。在EOS操作系统中，Shell进程是由系统进程创建的第一个用户进程。Shell进程首先会初始化自己的环境，包括设置命令提示符、加载配置文件等。然后，它会不断地等待用户输入命令，并解释和执行这些命令。Shell进程通过调用操作系统提供的系统调用接口来访问操作系统的功能和资源，如文件系统、进程管理等。Shell进程的主要作用是提供一个友好的界面，使用户能够方便地与操作系统进行交互，并完成各种操作和任务。

**3）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）

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。EOS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。

源代码中，1）TCB结构体定义位于ps/psp.h文件中，具体包括线程所属进程指针、线程链表项、优先级、状态、剩余时间片等字段。此外，线程运行在内核状态，故无用户空间栈。

在这个实验中，常用的字段有Process（进程指针）、KernelStack（栈位置）、KernelContext（处理器上下文）、Priority（优先级）、State（状态）、RemainderTicks（剩余时间片）。

2）线程状态定义在ps/psp.h文件中，包括了中间状态、就绪状态、运行状态、等待状态和结束状态。

3）记录不同状态线程的数据结构定义在sched.c文件中，包括就绪队列头、就绪位图、睡眠等待队列和已结束线程队列。

4）等待计数器定义在ke.h文件中，用于等待超时后调用回调函数，包括初始化、注册和注销操作。

5）静止就绪状态的线程队列定义在psspnd.c文件中，使用了循环链表实现。

**源代码如下：**

1）线程对象结构体TCB

|  |
| --- |
| 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; |

2）线程的状态

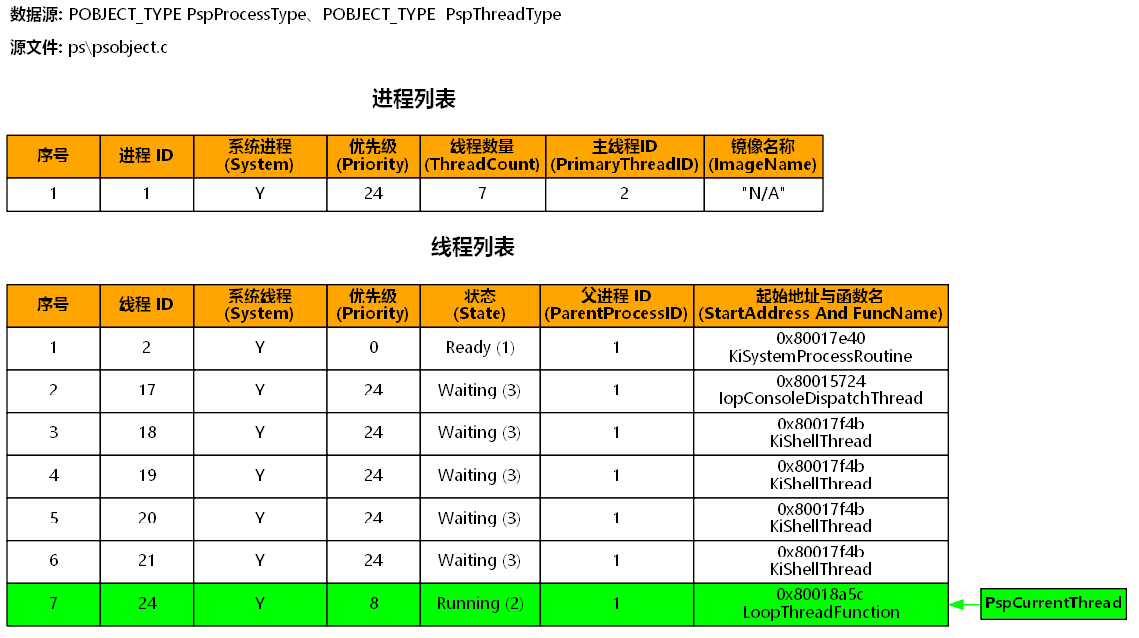
|  |
| --- |
| typedef enum \_THREAD\_STATE {  Zero, // 0  Ready, // 1  Running, // 2  Waiting, // 3  Terminated // 4  } THREAD\_STATE; |

3）等待计数器

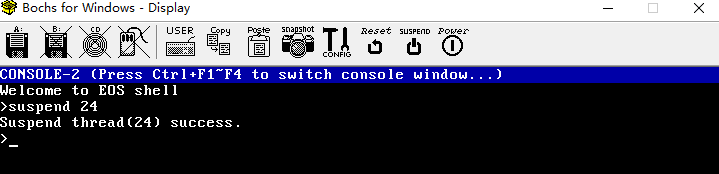
|  |
| --- |
| typedef struct \_KTIMER  {  ULONG IntervalTicks;  ULONG ElapsedTicks;  PKTIMER\_ROUTINE TimerRoutine;  ULONG\_PTR Parameter;  LIST\_ENTRY TimerListEntry;  }KTIMER, \* PKTIMER; |

测试及结果如下：

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，不再增加。

**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、不足及改进意见等；对EOS启动过程与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）

线程状态转换对线程调度至关重要。调度器根据线程状态做出决策，优先选择就绪状态的线程来执行，确保系统高效运行。

（1）线程调度导致部分线程状态转换：

a、当调度器选定一个就绪状态的线程执行时，该线程从就绪状态转换为运行状态。

b、如果一个线程被更高优先级的线程抢占，它会从运行状态转换为就绪状态，这归功于线程调度的存在。

（2）某些线程状态转换完成后，会触发线程调度：

例如，PspWait函数将当前线程插入指定等待队列的队尾，将状态码修改为Waiting，然后进行线程调度，让出处理器。

关于增加挂起状态的分析：

（1）有效方法的实现：

a、更灵活的线程管理：挂起状态允许线程在等待外部事件时休眠或等待，而不占用CPU时间，提高了系统的利用率。

b、资源节省：挂起状态降低了系统资源的使用率，允许更多线程同时运行，减少了竞争。

c、更快的响应时间：挂起状态下的线程能更快地响应外部事件，提高了系统的响应速度。

d、优先级管理：挂起状态使得操作系统能更有效地管理线程的优先级，根据需要唤醒线程。

e、避免忙等待：引入挂起状态后，线程不再需要忙等待，节省了CPU资源。

f、更好的资源共享：挂起状态使得操作系统能更好地管理共享资源。

（2）改进实现方法：

a、挂起时限：引入可配置的挂起时间限制，防止线程无限期挂起，提高系统的可响应性。

b、条件挂起：允许线程在挂起时指定条件，只有满足条件时才唤醒。

c、挂起队列：为挂起线程引入队列，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变化，并根据新的优先级重新调度。

e、资源管理：改进挂起状态下的资源管理，释放占用的资源以便其他线程使用。

f、性能优化：优化线程挂起和唤醒操作，减少系统开销。

g、错误处理：提供详细的错误处理机制，诊断挂起操作的失败原因。

h、用户界面：改进用户界面和API，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： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

**实验名称：操作系统实验2 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）

首先在EOS应用程序项目中导入pc.c文件，首先对pc.c中实现的生产者消费者同步问题进行分析。

定义生产者消费者句柄。

|  |
| --- |
| HANDLE ProducerHandle;  HANDLE ConsumerHandle; |

创建互斥信号量mutex，初始值为1；empty信号量初始值为buffer\_size，即缓冲池中空缓冲区数量；full信号量为满缓冲区数量，初始值为0。

|  |
| --- |
| MutexHandle = CreateMutex(FALSE, NULL);  if (NULL == MutexHandle) {  return 1;  }  EmptySemaphoreHandle = CreateSemaphore(BUFFER\_SIZE, BUFFER\_SIZE, NULL);  if (NULL == EmptySemaphoreHandle) {  return 2;  }  FullSemaphoreHandle = CreateSemaphore(0, BUFFER\_SIZE, NULL);  if (NULL == FullSemaphoreHandle) {  return 3;  } |

创建信号量的代码如下。

|  |
| --- |
| EOSAPI  HANDLE  CreateMutex(  IN BOOL InitialOwner,  IN PCSTR Name  )  {  STATUS Status;  HANDLE Handle;  Status = PsCreateMutexObject(InitialOwner, Name, &Handle);  PsSetLastError(TranslateStatusToError(Status));  return EOS\_SUCCESS(Status) ? Handle : NULL;  } |

调用PsCreateMutexObject函数创建互斥信号量对象，创建初始值InitialOwner，名称和句柄。

设置生产者和消费者线程，对应代码如下：

|  |
| --- |
| 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;  } |

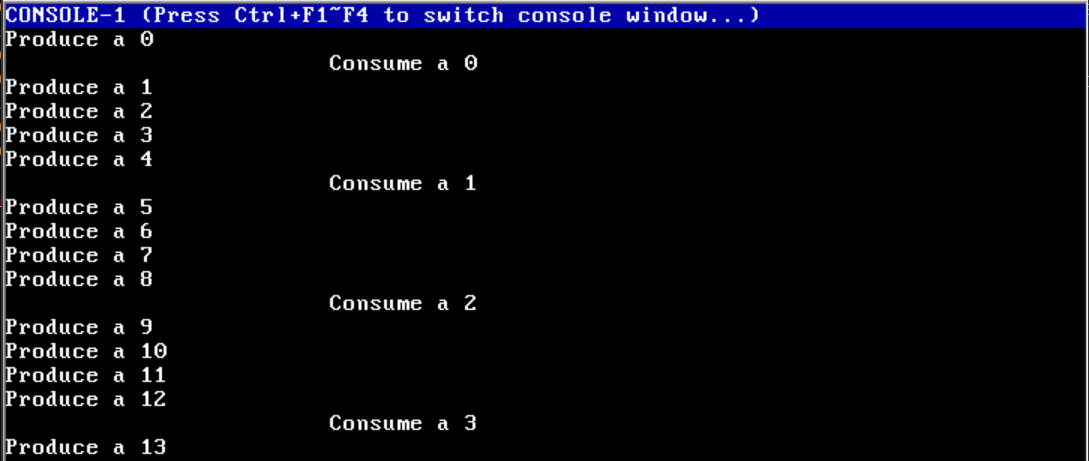
由加粗部分的函数可以看出，生产者线程先进行P操作，再使用mutex对缓冲区互斥访问，最后V操作释放缓冲区。

对这三个同步对象的操作不能改变顺序，因为假设先执行P(mutex），再执行P(empty)，如果生产者线程已经将缓冲区放满，但是消费者线程没有取产品，下次生产者线程运行时，先对缓冲区进行封锁，再执行P(empty)将被阻塞，需要消费者消费后将其唤醒；但是消费者线程执行P(mutex)时，由于已被生产者线程封锁，消费者线程也会被阻塞，都需要对方来唤醒自己，陷入等待中。

同理可以分析消费者线程的代码。

|  |
| --- |
| 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;  } |

消费者线程对信号量的操作和生产者线程类似，但是延时操作略有不同。在最初的消费中，消费者的消费比生产者慢，使生产者线程很快填满缓冲池，进入等待机制，只有消费者消费一个产品后，生产者线程才能继续生产，以此验证信号量实现的正确性。

 例如生产者在生产13号产品后本来要继续生产14号产品，但是因为此时缓冲池已满，所以需要等待消费者消费4号产品才可以继续生产14号产品。

由上图可知测试正确。

**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. 创建信号量：

|  |
| --- |
| EOSAPI  HANDLE  CreateSemaphore(  IN LONG InitialCount,  IN LONG MaximumCount,  IN PSTR Name  )  {  STATUS Status;  HANDLE Handle;  Status = PsCreateSemaphoreObject( InitialCount,  MaximumCount,  Name,  &Handle );  PsSetLastError(TranslateStatusToError(Status));  return EOS\_SUCCESS(Status) ? Handle : NULL;  } |

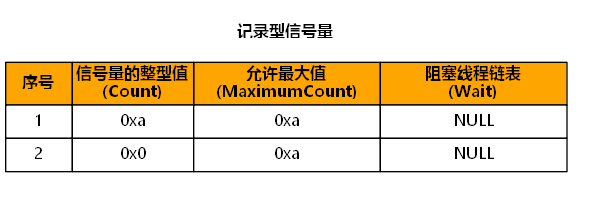
上文已提到过，通过此函数创建信号量，此API函数调用EOS内核中的PsCreateSemaphoreObject函数创建信号量对象。

|  |
| --- |
| STATUS  PsCreateSemaphoreObject(  IN LONG InitialCount,  IN LONG MaximumCount,  IN PSTR Name,  OUT PHANDLE SemaphoreHandle  )  {  STATUS Status;  PVOID SemaphoreObject;  SEM\_CREATE\_PARAM CreateParam;  if(InitialCount < 0 || MaximumCount <= 0 || InitialCount > MaximumCount){  return STATUS\_INVALID\_PARAMETER;  }  //  // 创建信号量对象。  //  **CreateParam.InitialCount = InitialCount;**  **CreateParam.MaximumCount = MaximumCount;**  **Status = ObCreateObject( PspSemaphoreType,**  **Name,**  **sizeof(SEMAPHORE),**  **(ULONG\_PTR)&CreateParam,**  **&SemaphoreObject);**  if (!EOS\_SUCCESS(Status)) {  return Status;  }  Status = ObCreateHandle(SemaphoreObject, SemaphoreHandle);  if (!EOS\_SUCCESS(Status)) {  ObDerefObject(SemaphoreObject);  }  return Status;  } |

通过此函数在EOS内核管理的内存中船舰一个信号量对象，在PsInitializeSemaphore函数中完成信号量对象中各个成员的初始化。

|  |
| --- |
| VOID  PsInitializeSemaphore(  IN PSEMAPHORE Semaphore,  IN LONG InitialCount,  IN LONG MaximumCount  )  {  ASSERT(InitialCount >= 0 && InitialCount <= MaximumCount && MaximumCount > 0);  Semaphore->Count = InitialCount;  Semaphore->MaximumCount = MaximumCount;  ListInitializeHead(&Semaphore->WaitListHead);  } |

PsInitializeSemaphore中传入的参数和CreateSemaphore函数中传入的参数对应一致。

创建完毕后查看“记录型信号量”如下图。

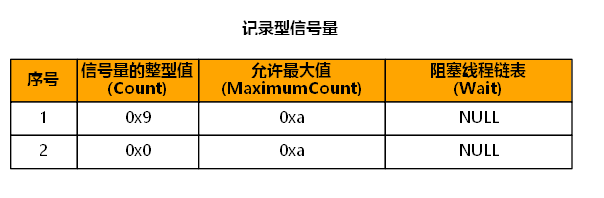
1. **等待信号量（P操作）和释放信号量（V操作）：**

**等待信号量（不阻塞）**

程序刚开始运行时，缓冲区都为空，此时生产者线程等待empty信号量时，不需要阻塞就可以返回，通过调用PsWaitForSemaphore函数使empty信号量的计数减一。

|  |
| --- |
| STATUS  PsWaitForSemaphore(  IN PSEMAPHORE Semaphore,  IN ULONG Milliseconds  )  {  BOOL IntState;  STATUS Status;//定义STATUS类型变量  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。  //  **if(Semaphore->Count>0){**  **Semaphore->Count--;**  **Status=STATUS\_SUCCESS;**  **}**  else if (Semaphore->Count == 0) {  Status=PspWait(&Semaphore->WaitListHead, Milliseconds);  }  KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。  return Status;  } |

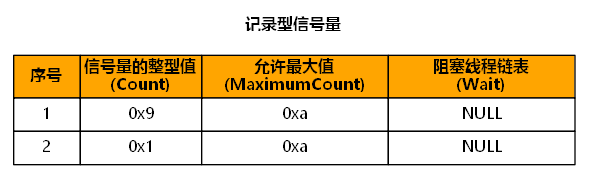
完成PsWaitForSemaphore函数中的全部操作后，查看“记录型信号量”窗口如下图，验证empty信号量的确减一。



**释放信号量（不唤醒）**

在生产者线程中，ReleaseSemaphore中释放FULL信号量，具体修改FULL信号量的值是在PsReleaseSemaphore函数中完成的。

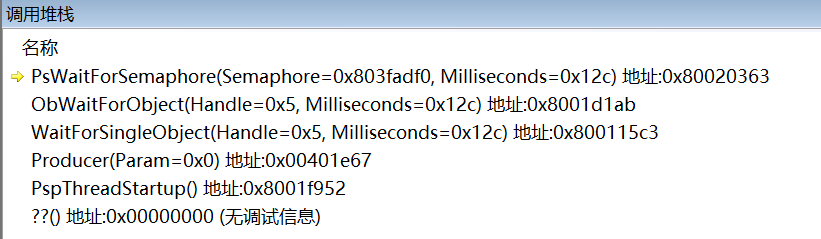
|  |
| --- |
| STATUS  PsReleaseSemaphore(  IN PSEMAPHORE Semaphore,  IN LONG ReleaseCount,  OUT PLONG PreviousCount  )  {  STATUS Status;  BOOL IntState;  IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。  if (Semaphore->Count + ReleaseCount > Semaphore->MaximumCount) {  Status = STATUS\_SEMAPHORE\_LIMIT\_EXCEEDED;  } 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。  //  while((ListIsEmpty(&Semaphore->WaitListHead)==0)&&(ReleaseCount!=0)){  **PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);**  **ReleaseCount--;**  }  **Semaphore->Count+=ReleaseCount;**  //  // 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;  } |

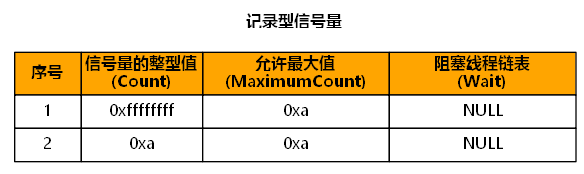
****通过以上代码完成对Full信号量的增加，查看“记录型信号量”窗口如下图，验证Full信号量确实增加了1。

**等待信号量（阻塞）**

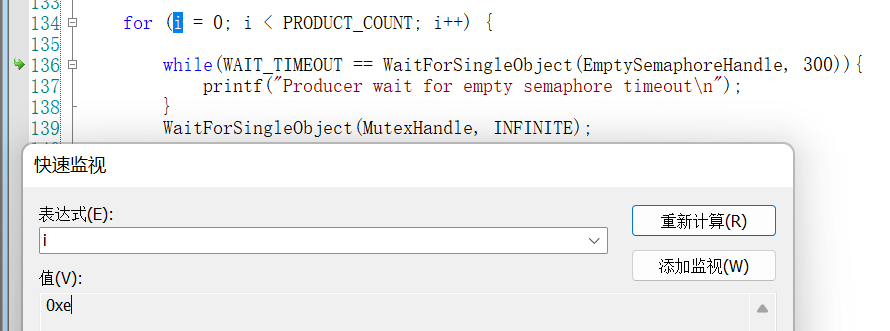
如前文所说，生产者线程生产产品的速度较快，消费者消费产品的速度慢，因此当生产者线程将缓冲池全部放满后，再生产新的产品时就会被阻塞。

在Producer线程中的PsWaitForSemaphore函数中添加断点，在程序执行一段时间后在断点处中断，可以发现是从生产者线程进入了此函数，如下图。



此时查看“记录型信号量”窗口，可以发现empty信号量值为-1，调用PspWait函数将生产者线程放入empty信号量的等待队列中等待，让出处理器。

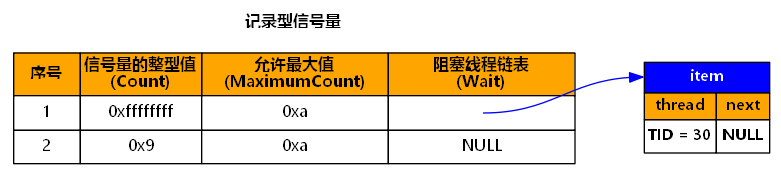
生产者线程尝试生产14号产品，因此此时producer函数中的变量i的值为14。



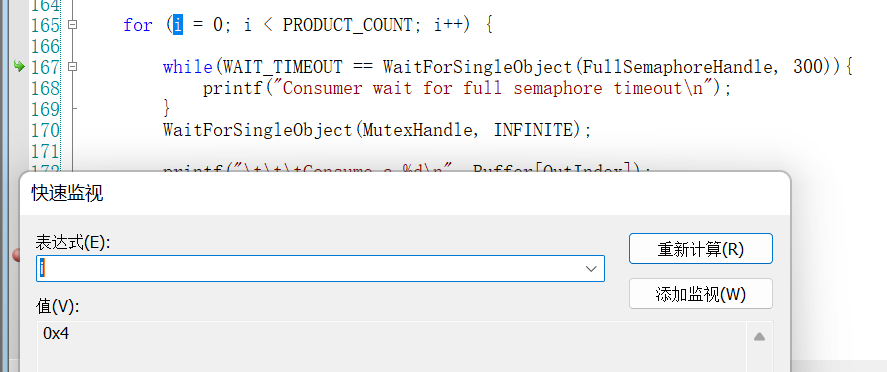
**释放信号量（唤醒）**

缓冲池满的情况下，只有当消费者线程从缓冲池中消费一个产品，即产生一个空缓冲区之后，生产者线程才会被唤醒并继续生产14号产品。

此时在Consumer线程中的ReleaseSemaphore中添加断点，在断点处中断后，查看“记录型信号量”如下图。此时生产者线程依然阻塞在empty信号量上。



此时consumer函数中变量i的值为4，说明已经消费了4号产品。

****

在PsReleaseSEmaphore函数中empty信号量的计数为-1，和生产者线程被阻塞的值是一样的；在此函数中调用PspWakeThread函数唤醒阻塞在empty信号量等待队列中的生产者线程，再调用PspSchedule函数执行调度，使生产者线程继续执行。

**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）

为了使信号量对象支持等待超时功能和批量释放功能，需要修改以下两个函数：PsWaitForSemaphore函数，PsReleaseSemaphore函数。

1. 实现等待超时功能

|  |
| --- |
| if(Semaphore->Count>0){  Semaphore->Count--;  Status=STATUS\_SUCCESS;  }  else if (Semaphore->Count == 0) {  Status=PspWait(&Semaphore->WaitListHead, Milliseconds);  }  KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。 |

支持等待超时唤醒功能的信号量，计数值只能大于等于0，当计数值大于0时，计数值减一后返回成功；当计数值等于0时，调用PspWait函数阻塞线程的执行。

1. 实现唤醒和批量释放功能

|  |
| --- |
| while((ListIsEmpty(&Semaphore->WaitListHead)==0)&&(ReleaseCount!=0)){  PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);  ReleaseCount--;  }  **Semaphore->Count+=ReleaseCount;** |

ReleaseCount为信号量计数增加的数量，使用宏定义函数ListEmpty判断信号量的等待队列是否为空，如果为空并且ReleaseCount值不为0，则此时使用PspWakeThread函数唤醒线程。

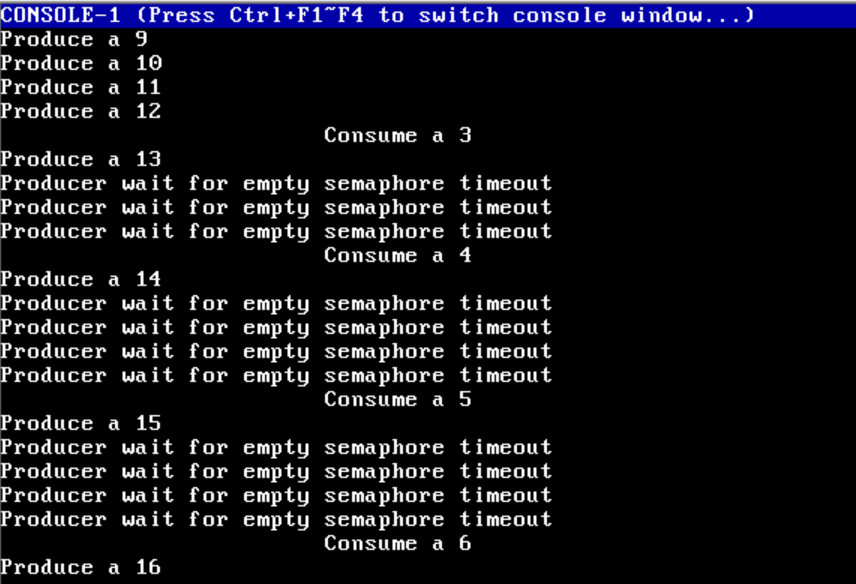
C．测试结果

修改Producer函数和Consumer函数中的对应行。

|  |
| --- |
| while(WAIT\_TIMEOUT == WaitForSingleObject(EmptySemaphoreHandle, 300)){  printf("Producer wait for empty semaphore timeout\n");  } |

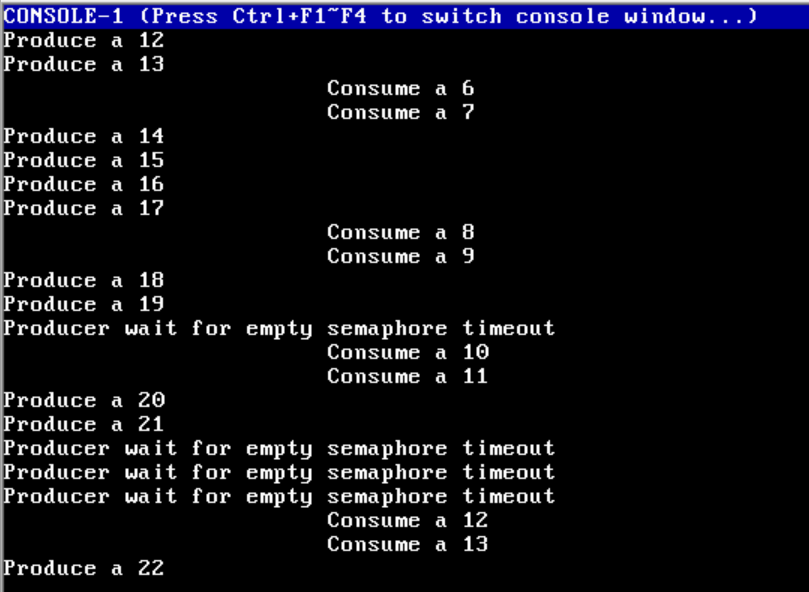
|  |
| --- |
| while(WAIT\_TIMEOUT == WaitForSingleObject(FullSemaphoreHandle, 300)){  printf("Consumer wait for full semaphore timeout\n");  } |

对应修改之后再次启动调试，验证上述功能是否实现。



以上运行结果说明超时等待功能实现。

将消费者线程修改为以此修改两个产品，验证批量释放功能。（使用学生包中的NewConsumer.c替换原有的Consume函数）



由上图可以验证消费者线程一次消费两个产品并批量释放empty信号量。

**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）

通过本实验，我领悟到了信号量在实现进程和线程间同步和互斥时的重要作用。在实现互斥访问时，将互斥信号量的初始值设为1，将访问互斥资源的代码夹在P(互斥信号量)V(互斥信号量)之间；而在实现同步访问时，则需要根据具体问题具体分析，确定同步信号量的初始值。本实验以生产者消费者问题为例，充分展现了信号量的作用。通过P、V操作，信号量在实现过程中屏蔽了中断信号，确保了操作的原子性，防止了各进程因争用临界区而陷入死锁状态。另外，在实现超时等待功能和批量唤醒时，我们通过调整等待时间和记录释放数量的方式，实现了更灵活的控制，确保了程序的稳定性和可靠性。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

**实验名称：操作系统实验3 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，线程被视为调度的基本单元。EOS操作系统采用了基于优先级的抢占式调度策略。在这种调度策略下，优先级较高的线程处于“就绪”状态时，会中断当前正在执行的低优先级线程，并将处理器分配给高优先级的线程执行。低优先级线程则进入“就绪”状态，等待重新获得处理器，直到系统中没有比它优先级更高的线程为止。

**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。

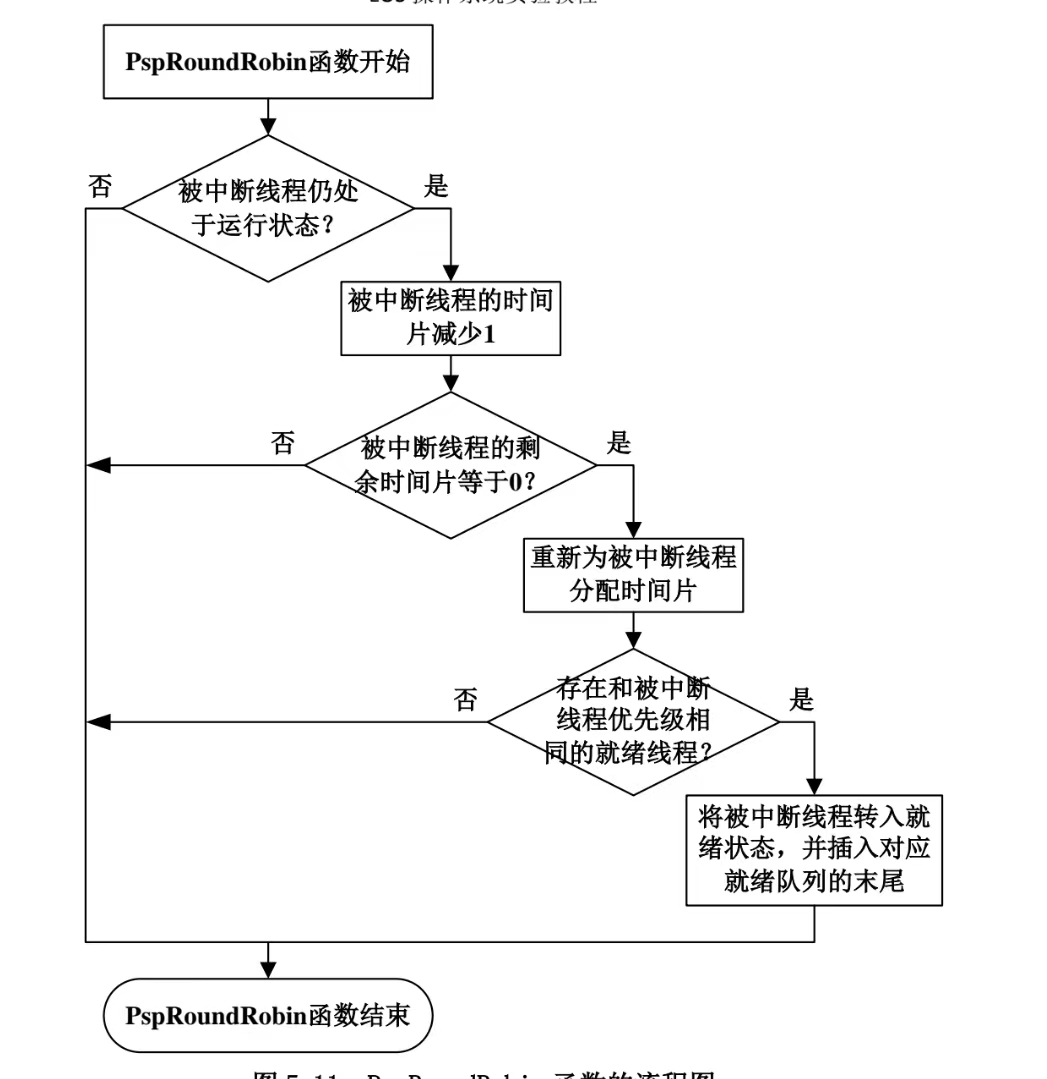
**2）为EOS添加时间片轮转调度**

（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。

时间片轮转调度算法的作用对象是正在“运行”的线程，因此要对被中断线程的状态进行判断；同时，线程所拥有的时间片是在ReminderTicks中给定，因此在进行调度时，需要对该域进行对应的判断、重新分配或者减少的操作。

**2、实现的算法流程：**

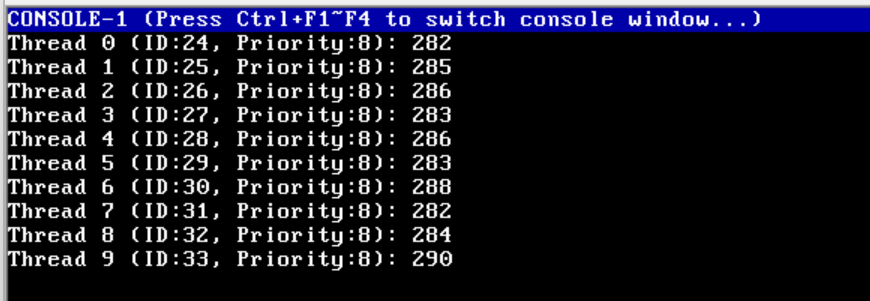
**3、具体代码如下：**

|  |
| --- |
| if(NULL!=PspCurrentThread && Running == PspCurrentThread->State){  PspCurrentThread->RemainderTicks--;  if(PspCurrentThread->RemainderTicks == 0){  PspCurrentThread->RemainderTicks = TICKS\_OF\_TIME\_SLICE;  if(BIT\_TEST(PspReadyBitmap,PspCurrentThread->Priority) != 0){  PspReadyThread(PspCurrentThread);  }  }  } |

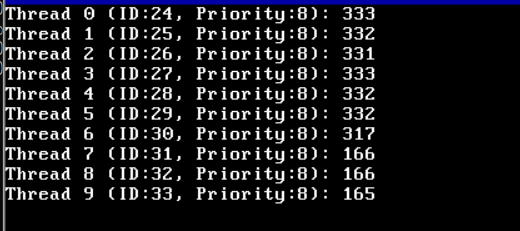
在这段代码中，时间片大小是由TICKS\_OF\_TIME\_SLICE定义的。

**4、测试：**

（1）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。



（2）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）

在调试未实现时间片流转调度算法的EOS内核时，验证了基于优先级的抢占式调度。发现当新建的10个优先级相同的线程时，只有第0个线程一直在运行，而更高优先级的控制台派遣线程能够抢先执行。但是，一旦实现了时间片轮转调度算法，这10个线程可以在时间片选择合理的情况下，较为公平地获得执行的机会。在ThreadFunction函数中，只能使用开关中断的方法来互斥访问控制台这种临界资源，而不能使用互斥信号量。使用开关中断不会改变线程状态，并且线程会被放在对应优先级的就绪队列中，而不是互斥信号量的等待队列。同时，测试时使用了多种时间片，可以发现只要时间片选取合理，线程是可以达到几乎同时执行，“独占”CPU的效果的，可以实现虚拟性。如果要实现多级反馈队列，则需要在时间片轮转调度算法的基础上做对应的修改；因为时间片用完而被中断的线程不是放在对应优先级的就绪队列中，而是放在下一级优先级的就绪队列中，并且不同优先级的时间片大小设置不同，优先级越高，时间片越小。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

**实验名称：操作系统实验4 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）

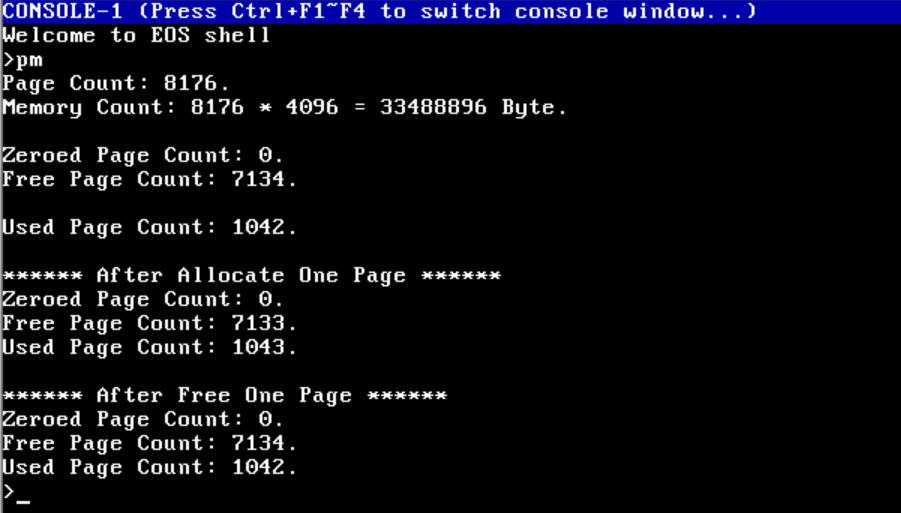
本实验为了测试时间片轮转调度，使用了控制台命令pm，由ConsoleCmdPhysicalMemory函数定义。代码中全局变量MiTotalPagezFrameCount保存了物理页的总数，MiZeroedPageCount和MiFreePageCount分别保存了零页和空闲页的数量，已用物理页数量应等于物理页总数减去零页数量和空闲页数量，同时在统计输出物理存储器信息前要关中断，防止在命令执行过程中有其他线程分配或者释放物理页。

下面对pm命令进行调试。

在ConsoleCmdPhysicalMemory对应位置添加断点，启动调试后命中断点，查看“物理内存”窗口如下图。



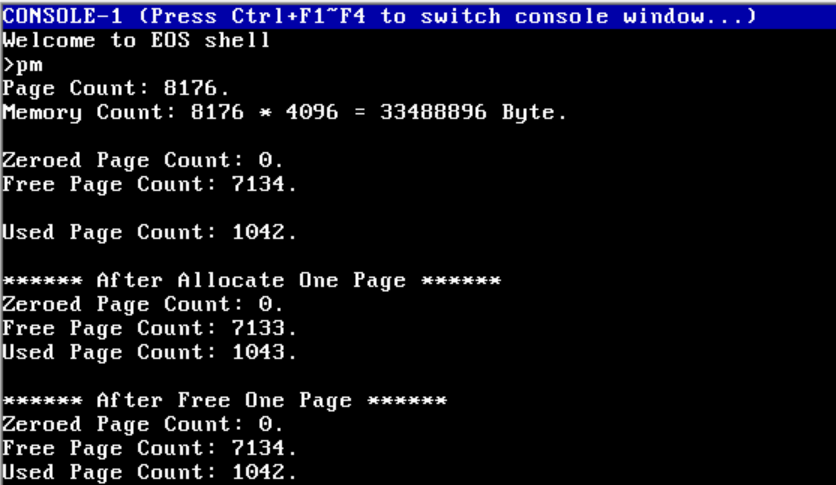
从图中可以查看到物理内存的使用情况，物理内存是从低地址向高地址连续分配的。但是在图中可以发现有0x408和0x409时空闲的，这是因为初始化时线程ID为3的线程完成初始化后，所使用的两个物理页被操作系统回收了。



继续执行后，可以查看pm命令的输出情况，与“物理内存”窗口的信息一致：物理页有8176页，零页为0，空闲页为7134页，已使用页1042页。

1. 分配物理页和释放物理页

在pm命令函数中添加分配和释放物理页的代码，使用学生包中的pm.c文件替换ConsoleCmdPhysicalMemory函数，则执行pm命令后的现象如下图。



可以看出此时展示出了分配一个物理页和释放一个物理页后零页、空闲页、已使用页的情况。下面对新添加的代码进行调试。

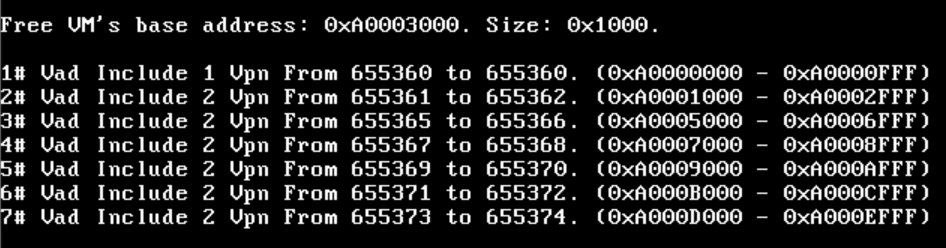
首先是分配物理页的过程，在分配物理页MiAllocateAnyPages函数和MiFreePages函数的代码处加断点，执行项目输入pm命令后，首先命中MiAllocateAnypages的断点。此时查看“物理内存”窗口如下图。

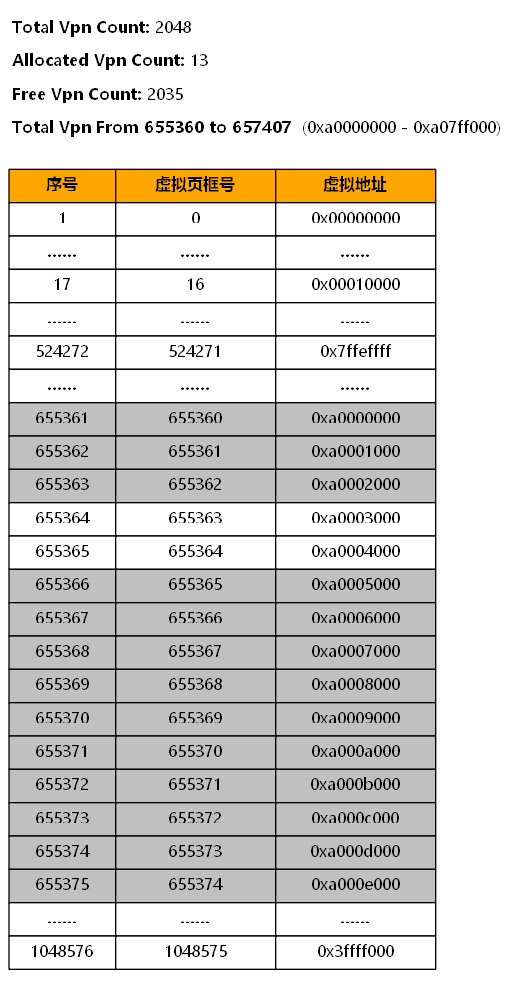
1. vm命令分析

本实验为了测试时间片轮转调度，使用了控制台命令vm，由ConsoleCmdVM函数定义。vm和pm不同，vm命令输出的是某个进程的虚拟地址描述符信息，因此此命令使用了进程ID这个参数，这个进程可以是系统进程也可以是用户进程。

下面对此命令进行调试。

在ConsoleCmdVM函数中添加断点，启动项目输入vm 1查看系统进程的虚拟地址描述符信息，命中该断点。此时查看虚拟机窗口输出的结果如下图。



查看“虚拟地址描述符”窗口如下图。系统进程中由虚拟地址描述符管理的虚拟页只会分配给进程的句柄表（使用一个虚拟页）和子线程的栈（每个线程的栈使用两个虚拟页），因此可以看出上图1号描述符包含的一个虚拟页是系统进程句柄表，2号到7号描述符分别包含两个虚拟页，但是很明显看出2和3号之间存在两个空闲页，与前文的原因相同，是由于线程ID为3的线程结束后，它的栈使用的虚拟页被操作系统释放。

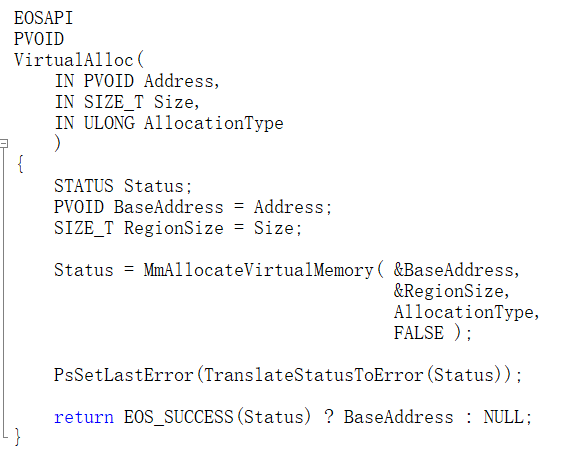
**2）EOS进程逻辑地址空间分配和回收的练习以及源代码分析**

（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）

在创建的EOS应用程序中编写代码实现指导书上写的对应功能。源代码如下。

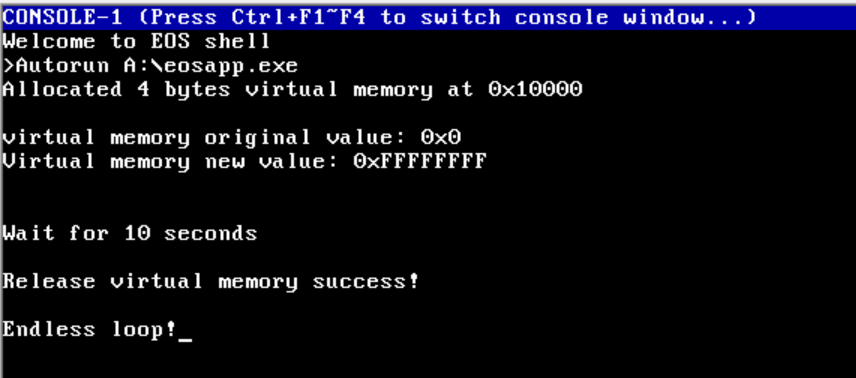
|  |
| --- |
| int main(int argc, char\* argv[])  {  INT \*pointer;  **if(pointer = VirtualAlloc(0,sizeof(int),MEM\_RESERVE|MEM\_COMMIT)){**  printf("Allocated %d bytes virtual memory at 0x%X\n\n",sizeof(int),pointer);    printf("virtual memory original value: 0x%X \n",\*pointer);  **\*pointer=0xFFFFFFFF;**  printf("Virtual memory new value: 0x%X \n\n",\*pointer);    printf("\nWait for 10 seconds\n");  **Sleep(1000);**    **if(VirtualFree(pointer,0,MEM\_RELEASE))**  printf("\nRelease virtual memory success!\n\n");  else printf("\nRelease virtual memory failed!\n\n");    **printf("Endless loop!");**  **while(1);**  **}**  else {  printf("Allocated Failed!\n");  return 0;  }  printf("Hello world!\n");  return 1;  } |

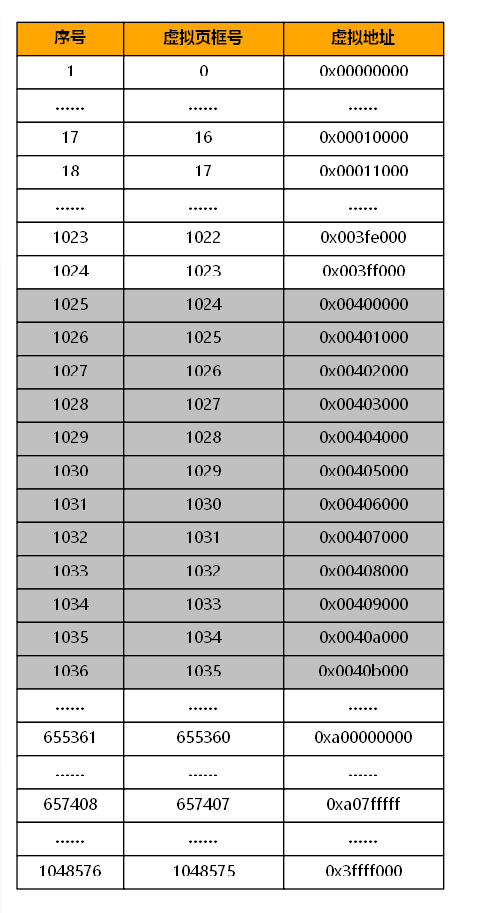
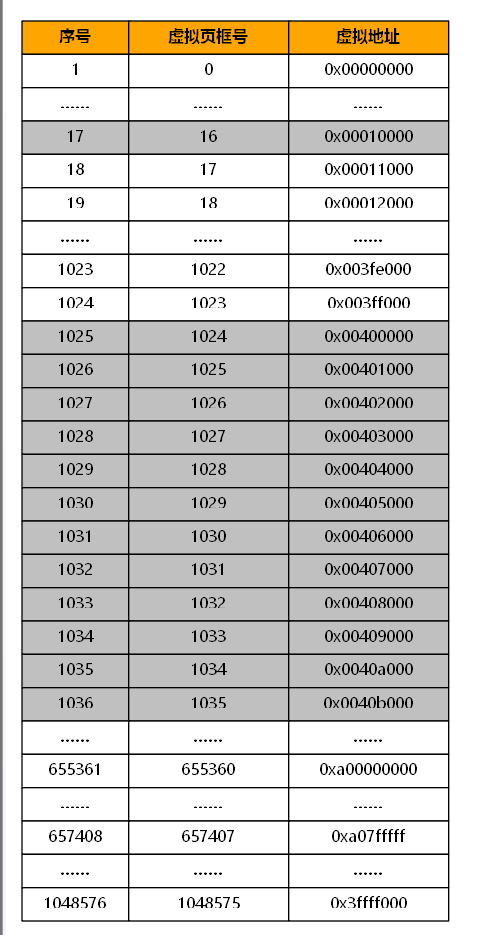
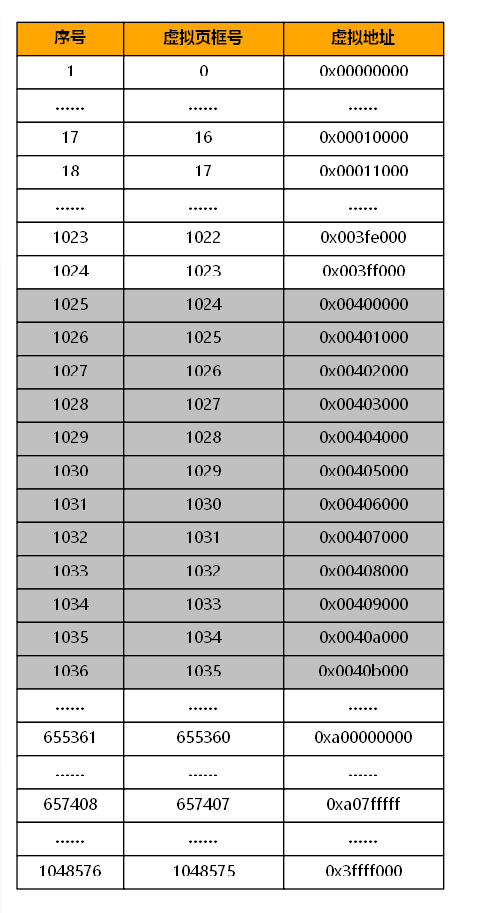
本程序调用API函数VirtualAlloc分配一个整型变量所需的空间，使用一个整型变量的指针指向该空间；修改该整型变量的指针为0xFFFF\_FFFF；调用Sleep函数等待10秒钟；调用API函数VirtualFree函数释放之前分配的整型变量的空间之后进入死循环。（加粗标识）

 着重注意这一行代码。

|  |
| --- |
| if(pointer = VirtualAlloc(0,sizeof(int),MEM\_RESERVE|MEM\_COMMIT)) |

VirtualAlloc函数中传入的参数0对应Address，进入VirtualAlloc代码处调试，可以发现令BaseAddress为Address，调用MmAllocateVirtualMemory对参数进行对应处理。

测试结果如下图。

在调用VirtualAlloc函数代码行中添加断点，单步调试查看分配虚拟页之前的、分配虚拟页之后的、释放虚拟页之后的虚拟地址描述符。如下图，从左到右分别对应分配前、分配后、释放后的“虚拟地址描述符”窗口。

**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）

在EOS中，采用了分页内存管理方式，对物理存储器以页为单位进行管理，通过数组管理页框号。物理页的分配首先从空闲页链表中分配，然后从零页链表中分配，为了防止安全问题，设置了零页。虚拟页的分配过程与物理页相似，但在EOS中可以选择是否将虚拟页映射到物理页上。释放物理页和虚拟页是分配的逆过程。通过这个实验，我对EOS的物理存储器和进程逻辑地址管理有了更深入的了解，了解了操作系统在分配和释放物理页或虚拟页时的操作过程。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

**实验名称：操作系统实验5 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）

调用EOS API函数读取文件数据，对源代码进行分析如下。

|  |
| --- |
| //  // 以只读的方式，打开要读取的文件  //  hFileRead = **CreateFile**(argv[1], GENERIC\_READ, 0, OPEN\_EXISTING, 0);  if (INVALID\_HANDLE\_VALUE == hFileRead)  {  printf("Open file \"%s\" error: %d\n", argv[1], GetLastError());  goto RETURN;  } |

调用CreateFile函数打开要读取的文件。如果不需要将读取的数据写入文件，则使用标准输出句柄作为输出句柄；如果需要将读取的数据写入文件，则调用CreateFile函数打开要写入的文件，因为需要写入，则将文件指针移动到文件的末尾并使用写入文件句柄作为输出句柄；

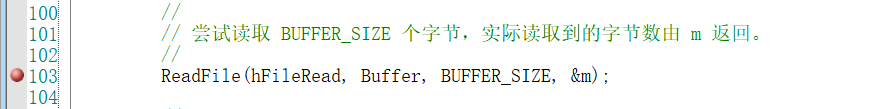
|  |
| --- |
| hFileWrite = CreateFile(argv[2], GENERIC\_WRITE, 0, OPEN\_EXISTING, 0);  if (INVALID\_HANDLE\_VALUE == hFileWrite)  {  printf("Open file \"%s\" error: %d\n", argv[2], GetLastError());  goto RETURN;  }  //  // 如果是增加写，则将文件指针移动到文件的末尾。  //  if (4 == argc && 0 == stricmp(argv[3], "-a"))  {  **SetFilePointer(hFileWrite, GetFileSize(hFileWrite), FILE\_BEGIN);**  }  **hOutput = hFileWrite;** |

调用ReadFile函数读取文件，调用WriteFile函数将读取的数据写入输出句柄，读取完毕后关闭文件。

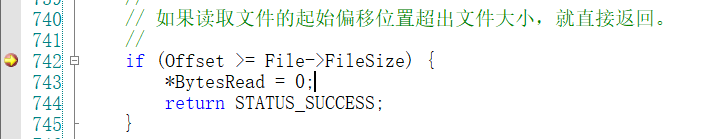
**2）EOS中FAT12文件系统读文件过程的跟踪**

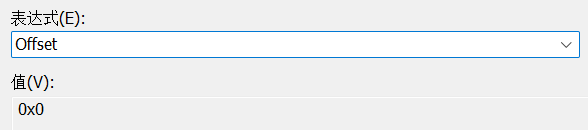
（简要说明在本部分实验中完成的主要工作；总结EOS中读文件的实现方法）

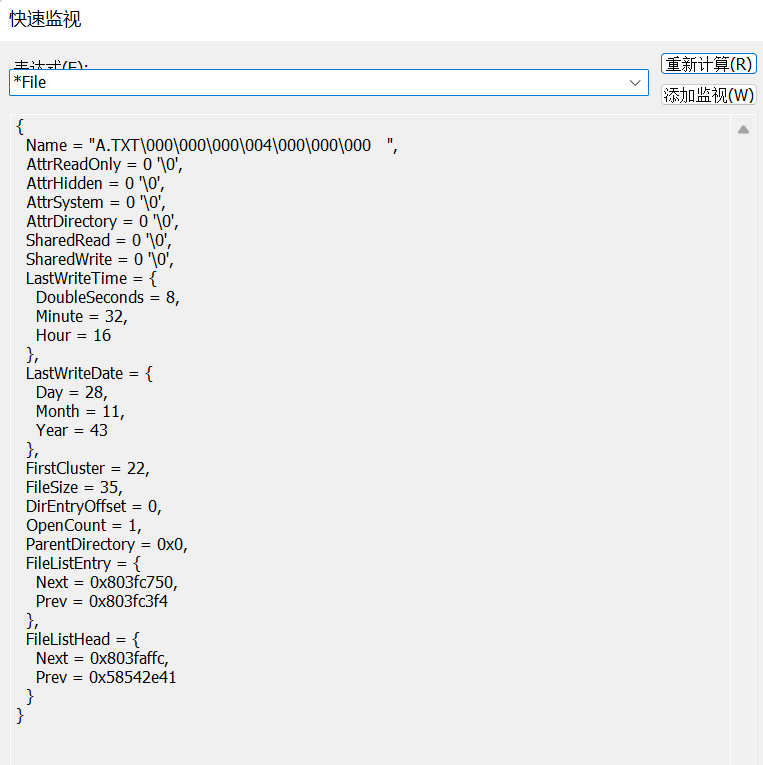
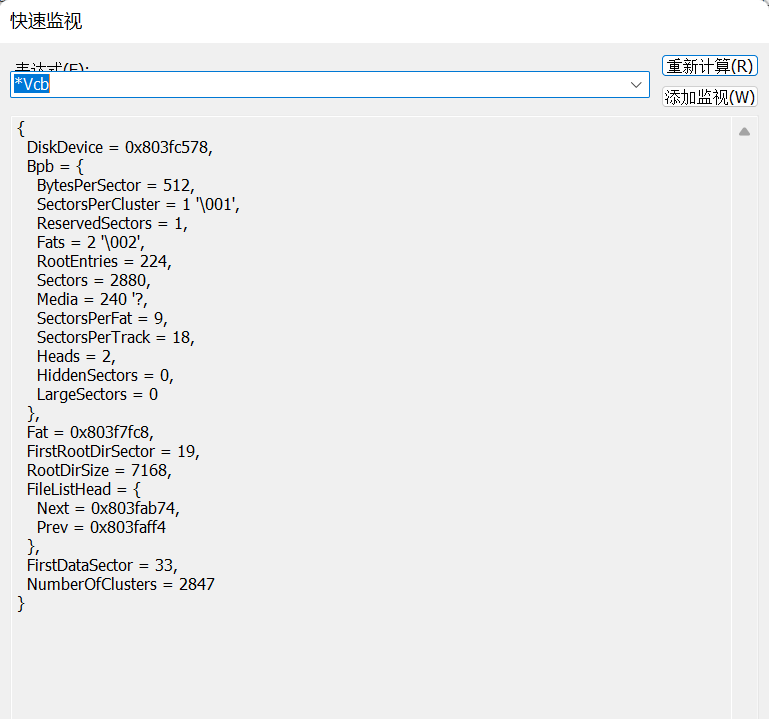
在EOSApp.c文件指定位置添加断点，执行命令命中断点。



在fat12.c文件中FatReadFile函数的开始处添加断点，继续执行后命中断点。



此时查看\*Vcb和\*File的“快速监视”窗口如下图。

此时查看参数Offset的值，其值是0，说明从文件头开始读取；查看参数ByteToRead的值，是0x400；查看参数Buffer的值，显示缓冲区所在地址在内核地址的空间。

此时因为读取文件的起始偏移位置没有超过文件大小，所以单步调试继续读取文件；预期读取的字节数大于文件大小，因此实际读取大小以文件大小为准。

继续执行，从文件的第一个簇开始读取，计算簇的起始扇区号，计算方法说明如下。

Vcb->FirstDataSector是文件数据区的起始扇区号。在FAT文件系统中，文件数据通常存储在簇中，而簇的数据则从数据区的某个扇区开始存储。(Cluster - 2) \* Vcb->Bpb.SectorsPerCluster这部分是为了找到簇的偏移量。FAT文件系统中前两个簇（簇号为0和1）被保留，因此实际的文件数据簇号从2开始。这部分计算得到了从第2个簇开始到目标簇的偏移扇区数，乘以簇的扇区数（Vcb->Bpb.SectorsPerCluster）得到总的偏移扇区数。最终，将上述两部分相加，得到目标簇的起始扇区号。这个值存储在FirstSectorOfCluster中。

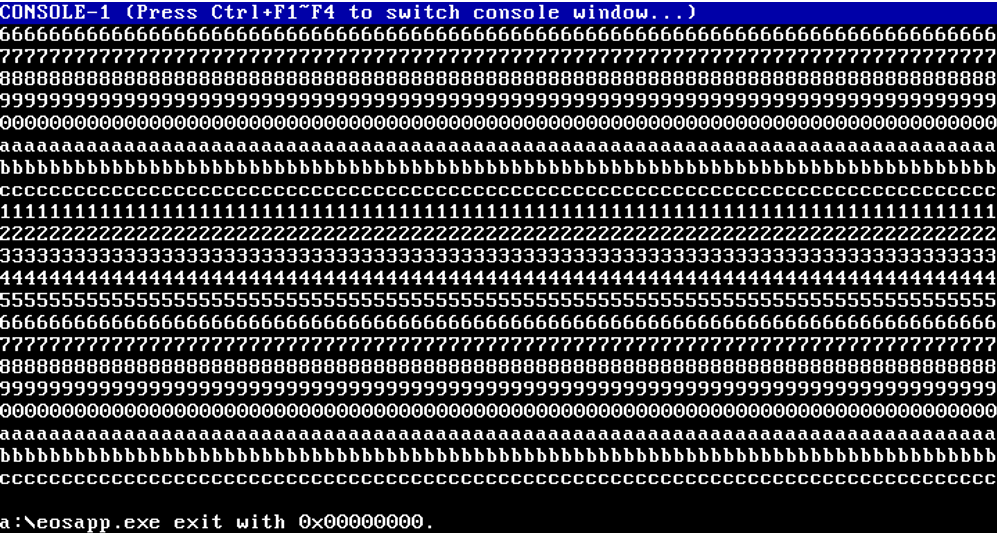
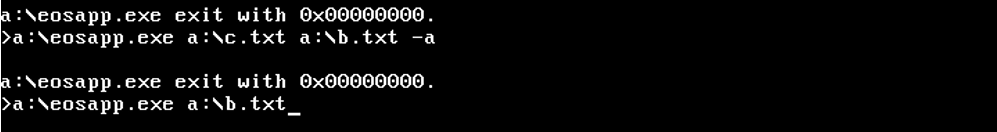
**3）为EOS的FAT12文件系统添加写文件功能**

（给出实现方法的简要描述、源代码、测试及结果等）

最终代码如下：

使FatWriteFile函数写入文件的数据跨越多个扇区的边界

|  |
| --- |
| **ULONG WriteCount=0;//记录实际已经写入的字节数**    for (j = 0; j<sharpTimes; ++j) {  PrevClusterNum = CurrentClusterNum;  CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);  if (0 == CurrentClusterNum || CurrentClusterNum >= 0xFF8) {//当前簇号为0或者大于等于0xff8  FatAllocateOneCluster(Vcb, &NewClusterNum);  //更新首簇号  if (0 == File->FirstCluster)  File->FirstCluster = NewClusterNum;  else  //写fat项中指定的值  FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);  CurrentClusterNum = NewClusterNum;  }  **//计算当前簇的第一个扇区号**  FirstSectorOfCluster = Vcb->FirstDataSector + (CurrentClusterNum - 2) \* Vcb->Bpb.SectorsPerCluster;  //写入  Status = IopReadWriteSector(Vcb->DiskDevice,  FirstSectorOfCluster,  0,  (PCHAR)(Buffer + WriteCount),  Vcb->Bpb.BytesPerSector,  FALSE);  if (!EOS\_SUCCESS(Status))  return Status;  WriteCount += Vcb->Bpb.BytesPerSector;  }  **//如果还有剩余不够一个扇区的数据，写入**  if (BytesToWriteLastTime != 0) {  PrevClusterNum = CurrentClusterNum;  CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);  if (0 == CurrentClusterNum || CurrentClusterNum >= 0xFF8) {  FatAllocateOneCluster(Vcb, &NewClusterNum);  if (0 == File->FirstCluster)  File->FirstCluster = NewClusterNum;  else  FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);  CurrentClusterNum = NewClusterNum;  }  FirstSectorOfCluster = Vcb->FirstDataSector + (CurrentClusterNum - 2) \* Vcb->Bpb.SectorsPerCluster;  Status = IopReadWriteSector(Vcb->DiskDevice,  FirstSectorOfCluster,  0,  (PCHAR)(Buffer + WriteCount),  BytesToWriteLastTime,  FALSE);  if (!EOS\_SUCCESS(Status))  return Status;  }  } |

测试结果如下：

**4）为EOS的FAT12文件系统添加文件复制功能**

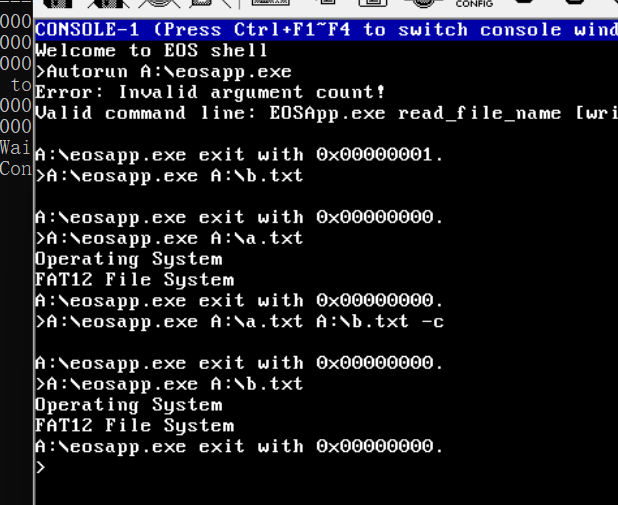
（给出实现方法的简要描述、源代码、测试及结果等）

文件复制与文件写入的功能区别在于文件复制会将之后多出来的数据置0，由此可简单实现文件复制功能：基本上等于创建文件、读取文件、写入新文件。

代码如下：

|  |
| --- |
| **ULONG WriteCount=0;//记录实际已经写入的字节数**    for (j = 0; j<sharpTimes; ++j) {  PrevClusterNum = CurrentClusterNum;  CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);  if (0 == CurrentClusterNum || CurrentClusterNum >= 0xFF8) {//当前簇号为0或者大于等于0xff8  FatAllocateOneCluster(Vcb, &NewClusterNum);  //更新首簇号  if (0 == File->FirstCluster)  File->FirstCluster = NewClusterNum;  else  //写fat项中指定的值  FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);  CurrentClusterNum = NewClusterNum;  }  **//计算当前簇的第一个扇区号**  FirstSectorOfCluster = Vcb->FirstDataSector + (CurrentClusterNum - 2) \* Vcb->Bpb.SectorsPerCluster;  //写入  Status = IopReadWriteSector(Vcb->DiskDevice,  FirstSectorOfCluster,  0,  (PCHAR)(Buffer + WriteCount),  Vcb->Bpb.BytesPerSector,  FALSE);  if (!EOS\_SUCCESS(Status))  return Status;  WriteCount += Vcb->Bpb.BytesPerSector;  }  **//如果还有剩余不够一个扇区的数据，写入**  if (BytesToWriteLastTime != 0) {  PrevClusterNum = CurrentClusterNum;  CurrentClusterNum = FatGetFatEntryValue(Vcb, PrevClusterNum);  if (0 == CurrentClusterNum || CurrentClusterNum >= 0xFF8) {  FatAllocateOneCluster(Vcb, &NewClusterNum);  if (0 == File->FirstCluster)  File->FirstCluster = NewClusterNum;  else  FatSetFatEntryValue(Vcb, PrevClusterNum, NewClusterNum);  CurrentClusterNum = NewClusterNum;  }  FirstSectorOfCluster = Vcb->FirstDataSector + (CurrentClusterNum - 2) \* Vcb->Bpb.SectorsPerCluster;  Status = IopReadWriteSector(Vcb->DiskDevice,  FirstSectorOfCluster,  0,  (PCHAR)(Buffer + WriteCount),  BytesToWriteLastTime,  FALSE);  if (!EOS\_SUCCESS(Status))  return Status;  }  } |

测试结果：



**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）

在这个实验中，“文件大小”和“文件占用磁盘空间大小”代表了两个不同的概念。文件大小以字节为单位，而文件占用磁盘空间以簇为单位计算，每个簇包含512个字节，因此文件大小的增加并不意味着文件占用磁盘空间增大。

同时，EOS操作系统采用链式分配方式管理磁盘空间，这种方式可以有效解决连续存储的碎片问题，提高磁盘利用率。但是，这种方式的随机访问效率较低，可能需要改进。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

**实验名称：操作系统实验6 扩展实验1（1分）**

**实验目的：**从整体上理解操作系统

**实验环境：**EOS操作系统及其实验环境。

**实验内容：**了解以下问题：1、操作系统启动之后，应用程序执行之前，操作系统中有哪些进程和线程，它们是如何创建的？2、操作系统启动之后，应用程序执行之前，查看软盘的使用情况和软盘包含的文件列表？3、操作系统如何从控制台窗口获取用户输入的命令？4、应用程序进程是如何在操作系统内核中创建的？5、操作系统如何为应用程序分配物理内存，如何在虚拟地址空间中为应用程序分配虚拟页？6、已为应用程序分配的虚拟页是如何与物理内存映射的？7、应用程序进程的主线程是如何创建的，创建后是如何进行线程状态转换的？8、系统进程创建的线程和应用程序进程创建的线程之间是如何调度的？9、应用程序的运行结果是如何输出到控制台的？10、应用程序进程是如何退出的？

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线程，如何创建？

操作系统启动后，第一个执行的是入口点函数KiSystemStartup。

下面根据所给注释和相关函数声明补充源代码。

|  |
| --- |
| #include "ki.h"  #include "mm.h"  #include "ob.h"  #include "ps.h"  #include "io.h" |

根据以上引用的头文件可以查找对应的初始化函数，初始化处理器和中断、初始化可编程中断控制器课可编程定时计数器，对应“ki.h”。

|  |
| --- |
| // 初始化处理器和中断。  KiInitializeProcessor(); //初始化全局描述符表  KiInitializeInterrupt(); //初始化全局中断符表 |

对各个管理模块执行第一步初始化，对应“mm.h”，“ob.h”，“ps.h”，“io.h”，模块初始化有顺序要求。

|  |
| --- |
| // 对各个管理模块执行第一步初始化，顺序不能乱。  MmInitializeSystem1(LoaderBlock);  ObInitializeSystem1();  PsInitializeSystem1();  IoInitializeSystem1(); |

创建系统启动进程，对应“ps.h”

|  |
| --- |
| // 创建系统启动进程。  PsCreateSystemProcess(KiSystemProcessRoutine); |

系统线程调度，使系统线程使用各自的线程堆栈运行，对应“ke.h”。

|  |
| --- |
| // 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。 KeThreadSchedule(); |

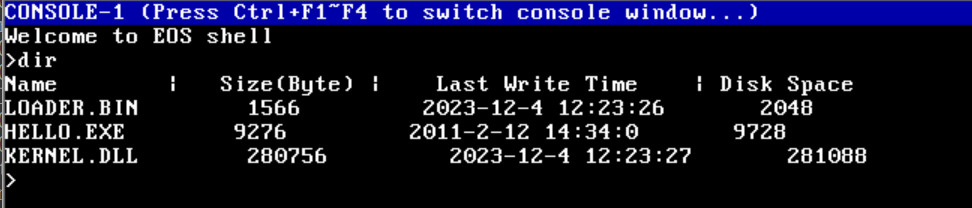
任务（二）：操作系统启动之后，应用程序启动之前，查看软盘的使用情况和软盘包含的文件列表

修改dir命令函数的源代码，要求输出每个文件的名称、大小、最后改写时间后，再输出每个文件所占用的磁盘空间（以字节为单位）。

修改如下。

|  |
| --- |
| //修改输出信息  // 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  fprintf(StdHandle, "Name | Size(Byte) | Last Write Time **| Disk Space\**n");  fprintf(StdHandle, "%s %d %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,  **(pDirEntry->FileSize/512)\*512 + (pDirEntry->FileSize%512==0?0:512)** |

计算文件占用磁盘空间就是读取文件占用多少扇区，(pDirEntry->FileSize / 512)部分表达式计算了文件大小中完整的 512 字节扇区数。在找到完整的 512 字节扇区数之后，这部分表达式将其乘以 512，以获取这些完整扇区的总大小。后一部分处理任何不能组成完整 512 字节扇区的剩余字节。三元运算符 (? :) 检查文件大小是否不是 512 的倍数，以确定是否有剩余字节。如果存在剩余字节，则添加 512 以考虑部分扇区。

启动调试后，查看控制台信息如下。

任务（三）：操作系统如何从控制台获取用户输入的命令？

补全io/driver/keyboard.c文件中的KbdRead函数。

|  |
| --- |
| STATUS  KbdRead(  IN PDEVICE\_OBJECT DeviceObject,  IN PFILE\_OBJECT FileObject,  OUT PVOID Buffer,  IN ULONG Request,  OUT PULONG Result OPTIONAL  )  {  BOOL IntState;  ULONG Count = 0;  PKEYBOARD\_DEVICE\_EXTENSION Ext = (PKEYBOARD\_DEVICE\_EXTENSION)DeviceObject->DeviceExtension;  //  // 读写大小须是键盘事件结构体大小的整数倍。  //  **Request -= Request % sizeof(KEY\_EVENT\_RECORD);**  while (Count < Request) {  //  // 阻塞等待直到缓冲区非空。  //  **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。  //  **IntState = KeEnableInterrupts(FALSE); //关中断**  **Count += IopReadRingBuffer(Ext->Buffer, Buffer + Count, Request - Count);**  **if (IopIsRingBufferEmpty(Ext->Buffer)) {**  **PsResetEvent(&Ext->BufferEvent);**  **}**  **KeEnableInterrupts(IntState); //开中断**  }  \*Result = Count;  return STATUS\_SUCCESS;  } |

修改io/driver/keyboard.c中的KbdIsr函数，如下。

|  |
| --- |
| VOID  KbdIsr(  VOID  )  {  static UCHAR ScanCode[3];  static UCHAR i = 0;  KEY\_EVENT\_RECORD KeyEventRecord;  ULONG ControlKeyStateMask;  PKEYBOARD\_DEVICE\_EXTENSION Ext = (PKEYBOARD\_DEVICE\_EXTENSION)KbdDevice[0]->DeviceExtension;  //  // 从8042数据端口读取键盘扫描码。  //    ScanCode[i]=READ\_PORT\_UCHAR((PUCHAR)KEYBOARD\_PORT\_DATA);  i++;  //处理具体按键  //  // 将键盘事件写入缓冲区，并设置键盘非空事件。  //  IopWriteRingBuffer(Ext->Buffer, &KeyEventRecord, sizeof(KEY\_EVENT\_RECORD));  PsSetEvent(&Ext->BufferEvent); |

任务（四）：应用程序进程如何在操作系统内核中创建？

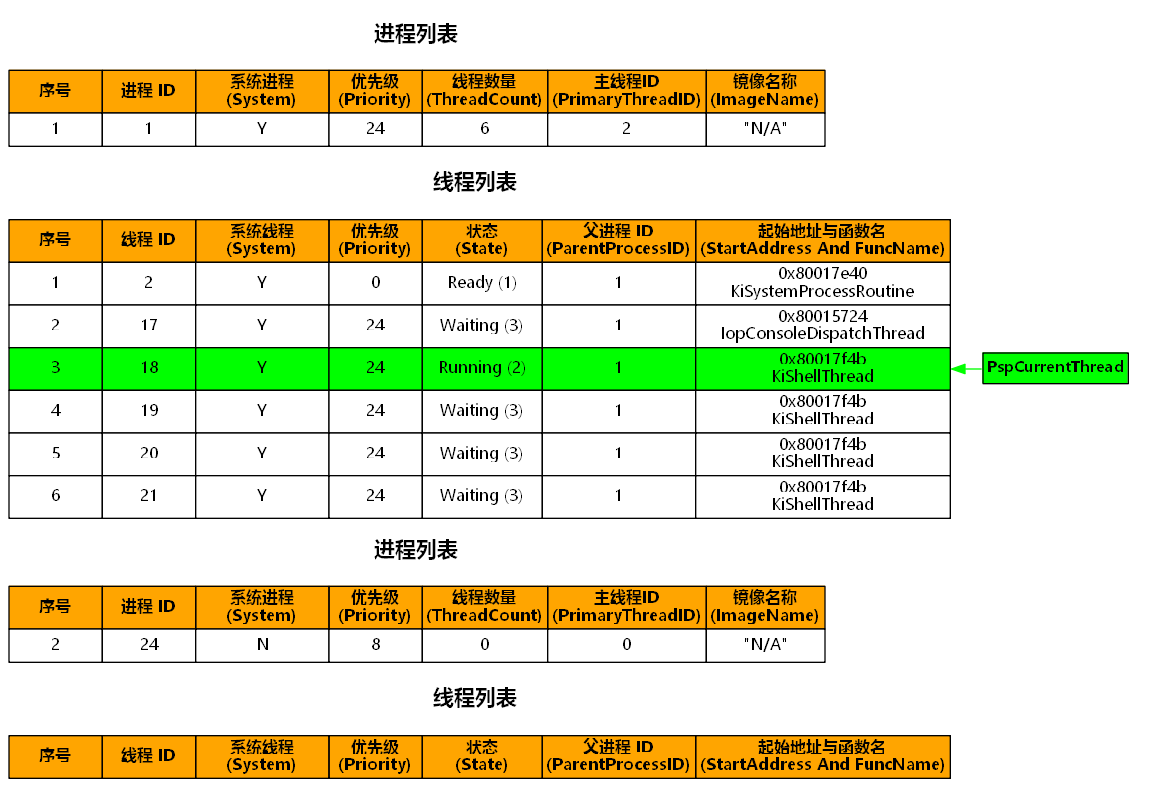
补全ps/create.c文件中的PsCreateProcess函数，源代码如下。

|  |
| --- |
| //  // 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）。  //  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。  //  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。  //  Status = PspCreateThread( ProcessObject,  0,  PspProcessStartup,  NULL,  CreateFlags,  &ThreadObject ); |

任务（五）：应用程序进程的主线程如何创建和状态转换？

**创建应用程序进程的主线程**

在ps/create.c文件的指定代码处添加断点，继续执行进入PspCreateThread函数内部。此函数在指定进程内创建一个线程。此时查看“进程线程”窗口如下，还未创建应用程序线程。

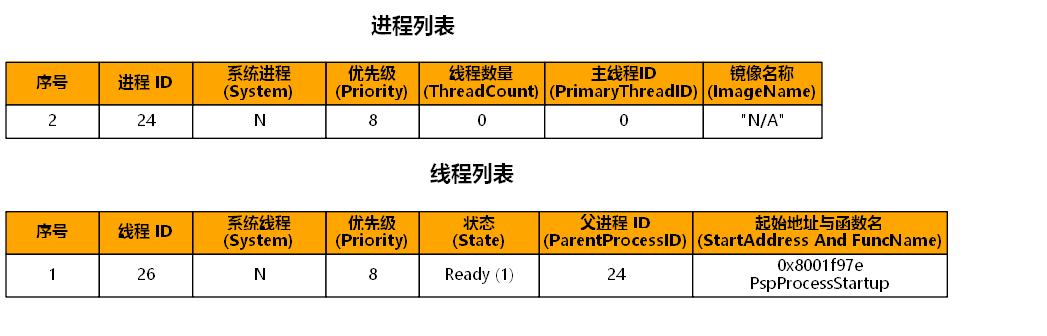


继续单步执行，ObCreateObject函数创建线程对象，MmAllocateVirtualMemory分配虚拟内存，运行到PspReadyThread(NewThread)对线程控制块进行初始化，此时查看“进程线程”窗口如下图。此时应用程序线程已经插入所在进程的线程链表中并完成了初始化。



**线程的状态转换**

单步调试PspReadyThread(NewThread)，使应用程序线程进入就绪状态。



**任务（六）：应用程序的执行结果如何输出到控制窗口？**

STATUS

IopWriteConsoleOutput(

IN PCONSOLE Console,

IN PVOID Buffer,

IN ULONG NumberOfBytesToWrite,

OUT PULONG NumberOfBytesWritten

)

{

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。

// 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;

}

**任务（七）：应用程序进程如何退出？**

补全ps/delete.c中的PspTerminateProcess函数和PspTerminateThread函数，源代码如下。

|  |
| --- |
| VOID  PspTerminateProcess(  IN PPROCESS Process,  IN ULONG ExitCode  )  {  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);  } |

|  |
| --- |
| VOID  PspTerminateThread(  IN PTHREAD Thread,  IN ULONG ExitCode,  IN BOOL IsTerminatingProcess  )  {  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));**  **}**  //  // 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度。  //  if (!IsTerminatingProcess) {  PspThreadSchedule();  }  }  }  KeEnableInterrupts(IntState);  } |

应用程序退出后，执行线程调度，下一个即将处于运行状态的线程是控制台线程，需要输出等待下一条指令的提示信息，并继续从控制台等待键盘输入。

**结果分析：**

通过在EOS内核中调试应用程序的执行过程并补充模块代码，对进程线程管理、存储器相关、IO输入输出等都有了系统的认识，穿插在指导书中的思考题，也使得对操作系统运行过程有了更加细节的认知。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

**实验名称：操作系统实验6 扩展实验2（1分）**

**实验目的：**（1）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。（2）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：**EOS操作系统及其实验环境。

**实验内容：**

1. 改进EOS内核引导过程，实现逻辑从无文件系统的平坦软盘镜像引导

改写loader.asm如下。

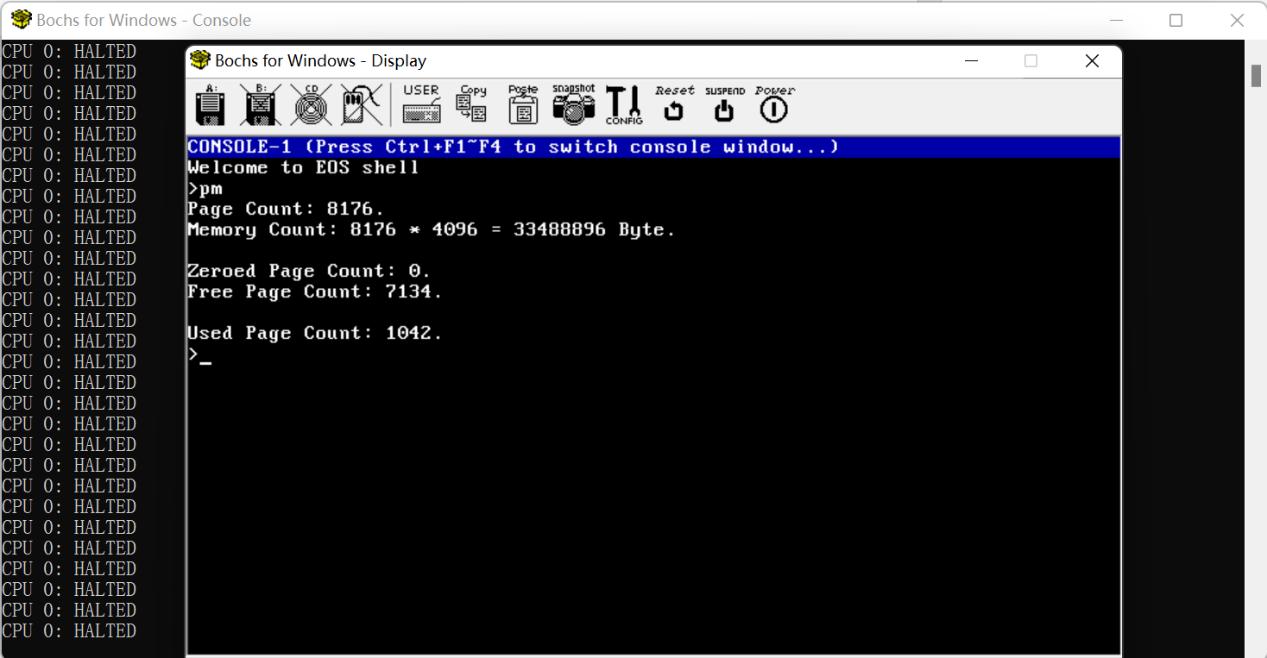
|  |
| --- |
| push word szLoading  call TextOut  mov cx , 0x258;600  mov si , 0x5  mov di , 0x1000  ReadKernel:  push cx  push 0  push di  push 0x0001  push si  call ReadSector  add di , 0x20  add si , 0x1  pop cx  loop ReadKernel |

修改ReadSector子程序如下

|  |
| --- |
| mov ax, [bp + 4]  mov bl, 18  div bl  inc ah  mov cl, ah  mov dh, al  shr al, 1  mov ch, al  and dh, 1  mov dl, 0 |

依照指导书对内核代码做相应调整，注释掉ioinit.c的174-226行，同时对项目配置做对应修改。

此时生成项目进行调试，在console窗口输入c，可以直接执行。发现由硬盘启动到虚拟机。



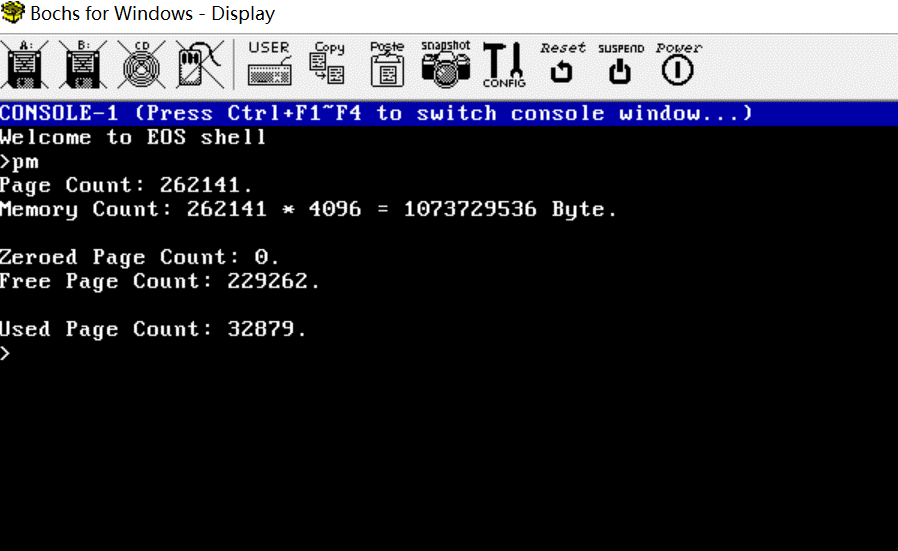
1. 改进EOS内核引导过程，实现物理机从U盘引导加载EOS内核

此时由于Bochs版本较低，则调整配置，替换项目中的Bochs配置。

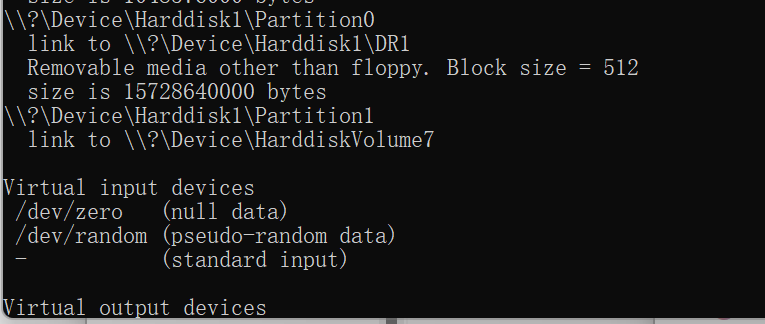
对loader.asm进行调整，源代码如下。

|  |
| --- |
| 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: |

对内核代码进行对应调整，将keyboard.c，mempool.c，ioinit.c的对应代码行注释，启动调试，进入内核后输入pm统计并输出物理存储器信息。

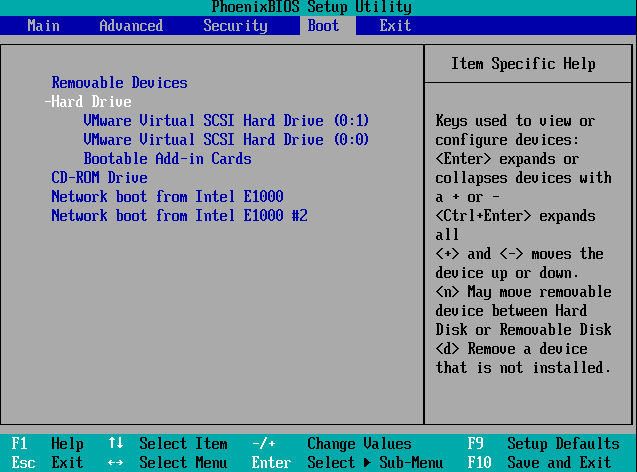


直接使用上文的U盘镜像，在物理机上使用U盘完成EOS内核的引导和加载。

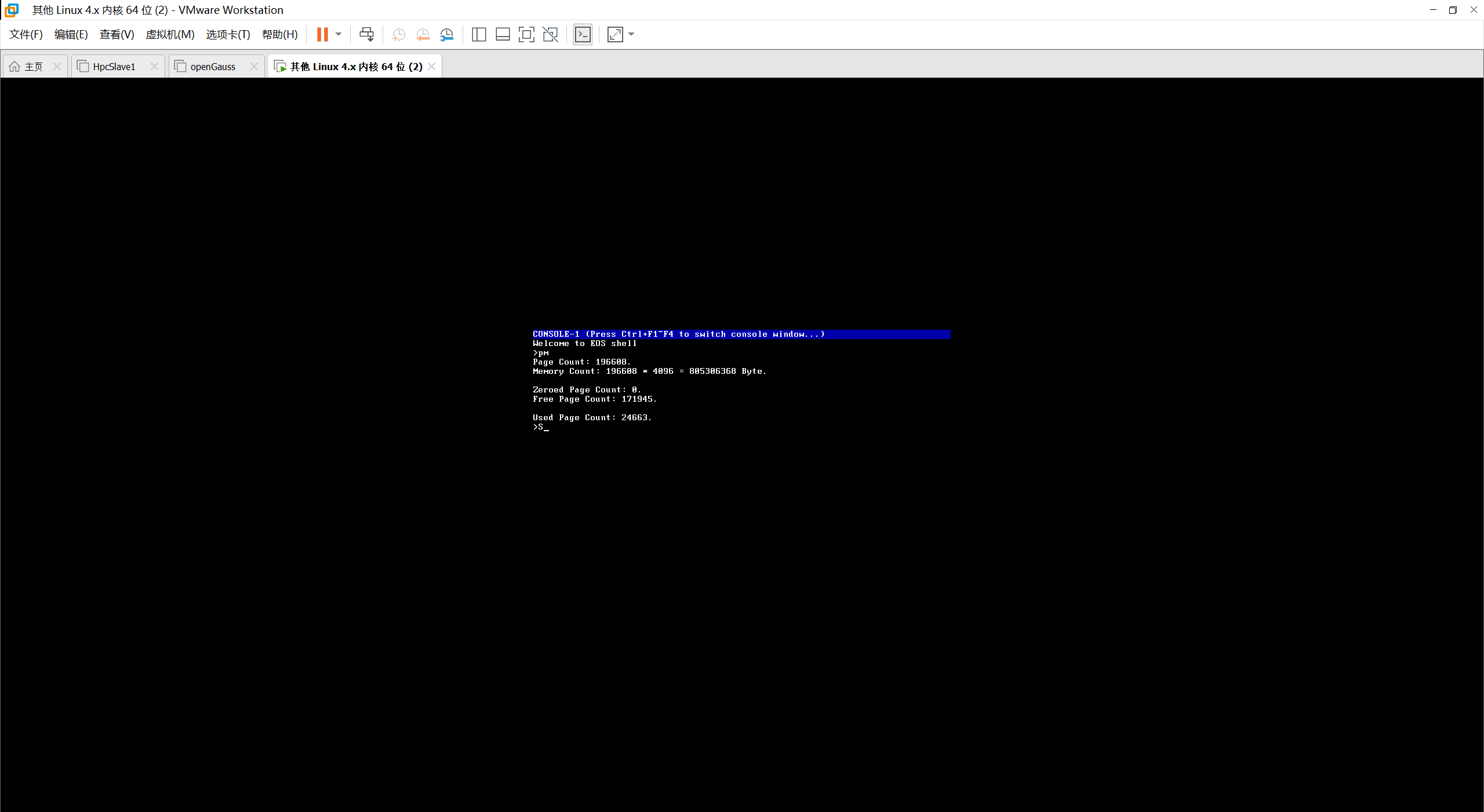
首先将U盘镜像文件写入U盘，下图展示的是输入.\ddrelease64.exe –list展示的可读写设备，其中有16GB的可移除U盘设备信息。

查看U盘是否正确写入镜像，如下图。



对虚拟机进行BIOS设置，如下。将U盘添加为虚拟机硬盘，并在boot中调整硬盘启动顺序。

启动虚拟机如下图。输入pm命令后可以查看对应信息。



**结果分析：**

通过本实验的操作，我对EOS的内核引导过程有了更深刻的认识，通过U盘引导的方式较硬盘引导的方式更加简单便携。不仅如此，因为涉及到汇编语言的编写，我的汇编代码阅读理解能力、编写能力也都有了提升。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

**实验名称：操作系统实验6 扩展实验3（1分）**

**实验目的：**实现多级反馈队列调度算法

**实验环境：**EOS操作系统及其实验环境。

**实验内容：**

源代码如下：

实现时间片轮转调度算法：

实验4已实现，在ps/schd.c中实现PspRoundRobin函数。

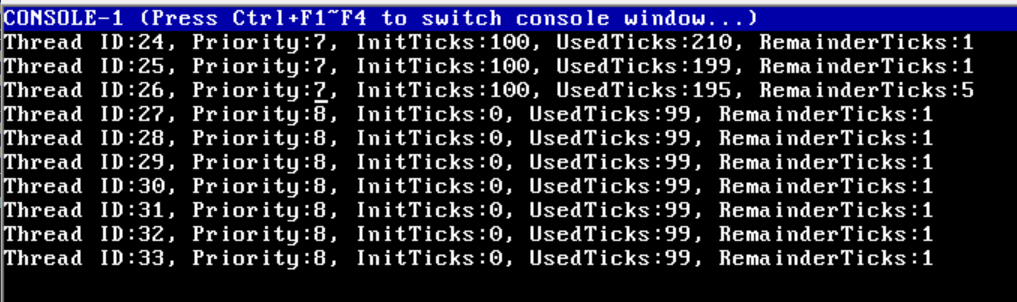
|  |
| --- |
| if( NULL != PspCurrentThread && Running == PspCurrentThread->State) {  // 在此实现时间片轮转调度算法  PspCurrentThread->RemainderTicks--;  // 被中断线程的剩余时间片等于0  if(PspCurrentThread->RemainderTicks==0){  PspCurrentThread->RemainderTicks = TICKS\_OF\_TIME\_SLICE;  if(BIT\_TEST(PspReadyBitmap, PspCurrentThread->Priority)!=0){  PspReadyThread(PspCurrentThread);  }  }  } |

修改时间片TICKS\_OF\_TIME\_SLICE为100，方便观察执行效果。

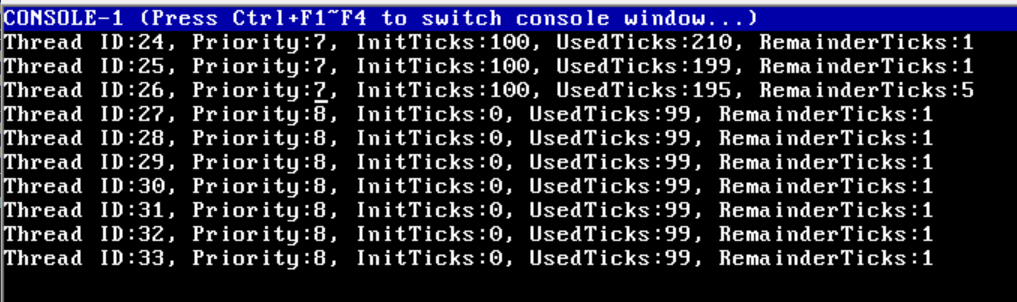
修改ps/psp.h对应代码行如下。



实现多级反馈队列

修改时间片调度算法代码如下。

|  |
| --- |
| // 线程生命周期结束  if(PspCurrentThread->UsedTicks == PspCurrentThread->LifeTimeTicks) {  PspCurrentThread->State = Terminated;  PspThreadSchedule();  return;  }  if (PspCurrentThread->RemainderTicks == 0) {  PspCurrentThread->RemainderTicks = TICKS\_OF\_TIME\_SLICE;  //多级反馈队列核心  if(PspCurrentThread->Priority > 0) {  PspCurrentThread->Priority--;  PspCurrentThread->InitTicks += TICKS\_OF\_TIME\_SLICE;  PspCurrentThread->RemainderTicks = PspCurrentThread->InitTicks;  PspThreadSchedule();  }  if (BIT\_TEST(PspReadyBitmap, PspCurrentThread->Priority) != 0) {  PspReadyThread(PspCurrentThread);  }  }  }  if (NULL != PspCurrentThread && Running == PspCurrentThread->State) {  PspCurrentThread->RemainderTicks--; // 当前运行线程的剩余时间减1  PspCurrentThread->UsedTicks++; // 当前运行线程已使用的时间片加1 |

**结果分析：**

通过实现多级反馈队列调度算法，我了解到了算法实现的具体细节，比如如何添加多个队列，队列的时间片设置，如何提升线程优先级等等。实现多级反馈队列主要是为了避免由于一个运行时间长的优先级较高作业导致较低优先级长时间无法运行，可以说多级反馈队列是已实现的时间片轮转调度算法的优化。

**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 物联网工程 班级： 物联222

姓名： 陈晓天 学号：U202241926 实验日期： 2024 年 5 月 17 日

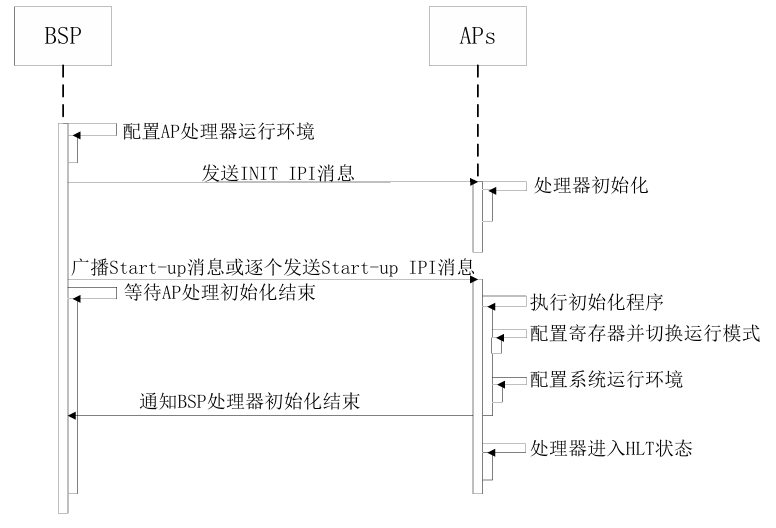
**实验名称：操作系统实验6 扩展实验6（1分）**

**实验目的：**实现支持用多处理机的信号量。

**实验环境：**EOS操作系统及其实验环境。

**实验内容：**

多核处理器概述如下：

****

自旋锁的概述如下：

在多核处理器中，自旋锁是一种用于实现临界区保护的同步机制。当一个线程试图获取自旋锁时，如果锁已经被其他线程持有，则该线程会一直循环（自旋）等待直到锁被释放。自旋锁的主要特点是在等待期间，线程会一直占用CPU资源，不会进行阻塞或者切换到其他任务。因此，自旋锁适用于临界区较小、锁被持有时间较短的情况，避免了线程上下文切换的开销。然而，如果临界区较大或者锁被持有时间过长，使用自旋锁可能会导致CPU资源浪费，因此在设计中需要权衡使用场景。

源代码如下：

1）APIC.C

|  |
| --- |
| #include "eos.h"  #include "stdint.h"  // 假定的硬件地址  #define LAPIC\_ADDR 0xFEE00000  #define IOAPIC\_ADDR 0xFEC00000  // 常用 LAPIC 寄存器偏移  #define LAPIC\_ID 0x20  #define LAPIC\_VER 0x30  #define LAPIC\_TPR 0x80  #define LAPIC\_EOI 0xB0  #define LAPIC\_SVR 0xF0  #define LAPIC\_ESR 0x280  #define LAPIC\_ICR\_LOW 0x300  #define LAPIC\_ICR\_HIGH 0x310  #define LAPIC\_LVT\_TIMER 0x320  #define LAPIC\_LVT\_PERF 0x340  #define LAPIC\_LVT\_LINT0 0x350  #define LAPIC\_LVT\_LINT1 0x360  #define LAPIC\_LVT\_ERROR 0x370  #define LAPIC\_TIMER\_INITCNT 0x380  #define LAPIC\_TIMER\_CURRCNT 0x390  #define LAPIC\_TIMER\_DIV 0x3E0  // 硬编码的处理器数量  #define NUM\_CPUS 4  // LAPIC 和 IOAPIC 的基地址映射  volatile uint32\_t \*lapic = (uint32\_t \*)(LAPIC\_ADDR);  volatile uint32\_t \*ioapic = (uint32\_t \*)(IOAPIC\_ADDR);  // 简化的写操作函数  void lapic\_write(int index, int value) {  lapic[index >> 2] = value;  }  int lapic\_read(int index) {  return lapic[index >> 2];  }  // LAPIC 初始化  void lapicinit(void) {  lapic\_write(LAPIC\_SVR, 0x100 | 0xFF); // 启用 LAPIC，并设置 Spurious Interrupt Vector  lapic\_write(LAPIC\_EOI, 0); // 发送 EOI  lapic\_write(LAPIC\_TPR, 0); // 允许所有优先级的中断  }  // IOAPIC 初始化  void ioapicinit(void) {  // IOAPIC REDTBL 的基本配置  int i=0;  while(i<24) {  ioapic[0] = 0x10 + i \* 2; // 设置中断向量  ioapic[4] = 0x20 + i; // 设置中断处理向量  ioapic[0] = 0x10 + i \* 2 + 1; // 高32位  ioapic[4] = 0; // 目标 APIC ID  i=i+1;  }  }  // CPU 启动函数  void start\_cpu(int cpu\_id) {  // 简化的启动逻辑，实际需要更多的控制，如发送 INIT 和 STARTUP IPIs  lapic\_write(LAPIC\_ICR\_HIGH, cpu\_id << 24);  lapic\_write(LAPIC\_ICR\_LOW, 0x4500); // 发送 INIT  lapic\_write(LAPIC\_ICR\_LOW, 0x4601); // 发送 STARTUP  }  // 主初始化函数  void mpinit(void) {  lapicinit();  ioapicinit();  // 启动所有其他 CPU  int i=1;  while (i < NUM\_CPUS) {  start\_cpu(i);  i=i+1;  }  } |

Spin Lock.c

|  |
| --- |
| #include <stdio.h>  // 自旋锁结构体  struct spinlock {  unsigned int locked; // 锁是否被持有  char \*name; // 锁的名称  // 为了调试：  struct cpu \*cpu; // 持有锁的CPU  unsigned int pcs[10]; // 锁住锁的调用栈（程序计数器的数组）  };  // 初始化自旋锁函数  void initlock(struct spinlock \*lk, char \*name) {  lk->locked = 0; // 锁未被持有  lk->name = name;  lk->cpu = NULL;  }  // 申请自旋锁函数  void acquire(struct spinlock \*lk) {  while (lk->locked) {} // 自旋直到获取到锁  lk->locked = 1;  // 在这里可以添加调试信息，比如记录持有锁的CPU和调用栈  }  // 释放自旋锁函数  void release(struct spinlock \*lk) {  lk->locked = 0; // 释放锁  // 在这里可以清除调试信息，比如清除持有锁的CPU和调用栈  } |

CPU.h

|  |
| --- |
| #ifndef EOS\_H  #define EOS\_H  #include <stdint.h>  // 自旋锁结构体  struct spinlock {  unsigned int locked; // 锁是否被持有  char \*name; // 锁的名称  // 为了调试：  struct cpu \*cpu; // 持有锁的CPU  unsigned int pcs[10]; // 锁住锁的调用栈（程序计数器的数组）  };  // 初始化自旋锁函数  void initlock(struct spinlock \*lk, char \*name);  // 申请自旋锁函数  void acquire(struct spinlock \*lk);  // 释放自旋锁函数  void release(struct spinlock \*lk);  // 常用 LAPIC 寄存器偏移  #define LAPIC\_ID 0x20  #define LAPIC\_VER 0x30  #define LAPIC\_TPR 0x80  #define LAPIC\_EOI 0xB0  #define LAPIC\_SVR 0xF0  #define LAPIC\_ESR 0x280  #define LAPIC\_ICR\_LOW 0x300  #define LAPIC\_ICR\_HIGH 0x310  #define LAPIC\_LVT\_TIMER 0x320  #define LAPIC\_LVT\_PERF 0x340  #define LAPIC\_LVT\_LINT0 0x350  #define LAPIC\_LVT\_LINT1 0x360  #define LAPIC\_LVT\_ERROR 0x370  #define LAPIC\_TIMER\_INITCNT 0x380  #define LAPIC\_TIMER\_CURRCNT 0x390  #define LAPIC\_TIMER\_DIV 0x3E0  // 硬编码的处理器数量  #define NUM\_CPUS 4  // LAPIC 初始化  void lapicinit(void);  // IOAPIC 初始化  void ioapicinit(void);  // CPU 启动函数  void start\_cpu(int cpu\_id);  // 主初始化函数  void mpinit(void);  #endif /\* EOS\_H \*/ |

**结果分析：**

通过该实验，我了解到了多核处理器的实现。已经掌握了多核处理器的一部分工作原理。在多核处理器中，为了充分利用多核心的计算资源，需要有效地管理线程间的并发访问。自旋锁是其中一种常见的同步机制，通过不断尝试获取锁来实现线程之间的同步。此外，多核处理器还涉及到任务调度、内存管理、缓存一致性等方面的内容，需要深入学习。