EOS 操作系统扩展实验教程

V3.0

万老师

18725138505

目录

实验一 从整体上理解EOS操作系统 3
实验二 在物理机上运行EOS操作系统 32
实验三 线程调度算法改进 45
实验四 内存管理算法改进 48
实验五 文件系统改进 54
实验六 多核处理器改进 64
实验七 搭建自己的操作系统 68

实验一 从整体上理解 EOS 操作系统

实验性质：验证
建议学时：2 学时
任务数：7 个

一、 实验目的

- 从整体上理解 EOS 操作系统。

二、 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。

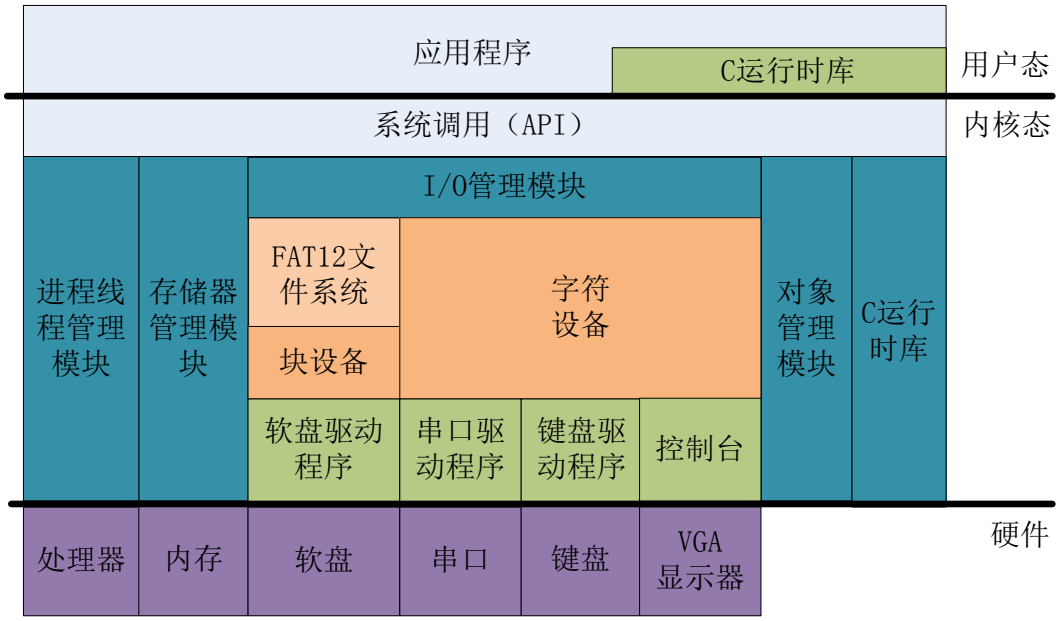


图 1-1：EOS 操作系统架构图

本实验主要通过在 EOS 操作系统中运行应用程序，引导读者理解操作系统各个功能模块与应用程序在执行前、执行中、执行后如何工作的，在不同的阶段操作系统会有哪些模块参与到工作中，从总体上把握进程创建、线程状态转换、线程调度、内存分配、文件系统等重要概念，这样，读者就可以对 EOS 操作系统的总体构成有一个清晰的认识。

在进行实验之前了解一下 EOS 操作系统的文件目录结构、主要的源代码文件和调用的重要函数是十分必要的，这样读者可以更全面的认识 EOS 操作系统。其中，某几个文件（包括：start.c、ioinit.c、sysproc.c、sysproc.c、mempool.c）的重要函数已经在下表中列出，对于未列出重要函数的文件，请读者根据自己对源代码的理解对下面的表格进行补充。

目录	文件	说明	重要函数
\api	eosapi	EOS 中 API 函数的实现	
\bochs	BIOS-bachs-latest	BIOS 镜像	
	bochs.exe	Bochs Release 版的可	

		执行程序	
	bochsdbg.exe	BochsDebug 版的可执行程序	
	bochsrc.bxrc	Bochs Release 版的配置文件	
	bochsrcdbg.bxrc	BochsDebug 版的配置文件	
	HardDisk.vhd	Bochs 的虚拟磁盘文件	
	VGABIOS-lgpl-latest	VGABIOS 镜像	
\boot	boot.asm	引导程序	
	loader.asm	加载程序	
\inc	eos.h	EOS 内核导出头文件，EOS 应用程序需包含此头文件	
	eosdef.h	EOS 数据类型、结构体、常量以及宏定义	
	error.h	EOS 错误码常量的定义	
	io.h	IO 模块公共头文件	
	ke.h	系统支撑模块对外接口的头文件，供内核其它模块使用	
	mm.h	内存管理模块公开接口头文件	
	ob.h	对象管理器公开接口头文件	
	ps.h	进程管理模块的对外接口头文件	
	rtl.h	内核运行时库头文件	
	status.h	内部错误状态码常量的定义	
\io	driver/fat12.c	FAT12 文件系统驱动的实现	
	driver/fat12.h	FAT12 文件系统驱动程序的内部头文件	
	driver/floppy.c	EOS 软盘驱动程序	
	driver/keyboard.c	键盘驱动的实现	
	driver/serial.c	串口驱动程序的实现	
	block.c	实现了块设备层	
	console.c	控制台模块的实现，包括控制台的初始化、打开、读写	
	file.c	文件对象的实现，包括文件对象的创建、读、	

		写、查询、设置	
	io.c	IO 模块接口函数的实现	
	ioinit.c	I/O 模块的初始化，包括数据结构体的初始化和设备驱动的加载	IoInitializeSystem1：I/O 管理器第一步初始化，初始化不可阻塞，主要是创建 IO 对象类型。
			IoInitializeSystem2：I/O 管理器第二步初始化函数，可阻塞，安装驱动并初始化设备。
	iomgr.c	驱动程序对象和设备对象的管理	
	iop.h	IO 模块私有头文件	
	rbuf.c	环形缓冲区的实现	
\ke	i386/8253.c	PC 机 8253 可编程定时计数器的初始化	
	i386/8259.c	PC 机 8259 可编程中断控制器的控制	
	i386/bugcheck.c	PC 机 KeBugCheck 的实现，KeBugCheck 依赖硬件平台	
	i386/cpu.asm	定义并加载全局描述符表	
	i386/dispatch.c	中断派遣模块的实现	
	i386/int.asm	中断的实现	
	ki.h	内核支撑模块的内部函数及变量的声明	
	ktimer.c	内核定时器的实现	
	start.c	EOS 内核的入口函数	KiSystemStartup 的功能是系统的入口点，Kernel.dll 被 Loader 加载到内存后从这里开始执行
	sysproc.c	系统进程函数，控制台线程函数，及各个控制台命令对应函数	KiInitializationThread：初始化线程的线程处理函数
			KiShellThread：Shell 线程的线程处理函数
			ConsoleCmdVersionNumber：打印输出 EOS 操作系统的版本号。
			ConsoleCmdMemoryMap：转储系统进程的二级页表映射信息。
			ConsoleCmd……:控制台命令的相关函数
\mm	i386/mi386.h	i386 内存相关的实现	
	mempool.c	内存池模块的实现	PoolInitialize：初始化内存池结构体，初始化后内存池是空的

			PoolCommitMemory：提交内存给内存池托管分配
			PoolAllocateMemory：从内存池中分配一块内存
	mi.h	内存管理器内部头文件	
	mminit.c	初始化内存管理模块	
	pas.c	进程地址空间的创建、删除和切换	
	pfnlist.c	物理内存管理，包括物理页的分配、回收以及零页线程	
	ptelist.c	系统 PTE 管理，实现了单页物理内存的快速映射	
	syspool.c	系统内存池的管理，包括初始化、分配和回收	
	vadlist.c	虚拟地址描述符链表管理模块	
	virtual.c	虚拟内存管理，包括虚拟内存的分配、回收等	
\ob	obhandle.c	对象句柄表的实现	
	obinit.c	对象管理器之初始化	
	object.c	对象管理模块的实现	
	obmethod.c	对象虚函数的调用，包括 Wait、Read 和 Write	
	obp.h	对象管理模块的内部头文件	
	obtype.c	对象类型管理模块的实现	
\ps	i386/pscxt.c	线程上下文环境的初始化，依赖具体的硬件	
	i386/psexp.c	进程（线程）异常处理模块	
	create.c	进程、线程的创建	
	delete.c	线程、进程结束以及相关函数的实现	
	event.c	进程同步对象之事件的实现	
	mutext.c	进程同步对象之互斥信号量的实现	
	peldr.c	PE 文件的加载	

	psinit.c	进程管理模块的初始化	
	psobject.c	和进程、线程对象相关的一些函数	
	psp.h	进程管理模块的内部头文件	
	psspnd.c	挂起线程的简单实现	
	psstart.c	线程、用户进程的启动函数	
	sched.c	线程调度的实现。包括线程状态的转换。	
	semaphore.c	进程同步对象之记录型信号量的实现	PsInitializeSemaphore：初始化信号量结构体
			PsWaitForSemaphore：信号量的 Wait 操作
			PsReleaseSemaphore：信号量的 Signal 操作
			PspOnCreateSemaphoreObject：semaphore 对象的构造函数，在创建新 semaphore 对象时被调用
			PspCreateSemaphoreObjectType：semaphore 对象类型的初始化函数
			PsCreateSemaphoreObject：semaphore 对象的构造函数
			PsReleaseSemaphoreObject：semaphore 对象的 signal 操作函数
\rtl	i386/hal386.asm	i386 中断的实现	
	i386/setjmp.asm	保存和恢复当前栈的信息	
	crt.c	内核 C 运行库函数	
	generr.c	内部状态码到错误码的转换	
	keymap.c	按键字符映射表	
	list.c	链表数据结构模块	
\	Floppy.img	软盘镜像文件	
\	License.txt	源代码协议	

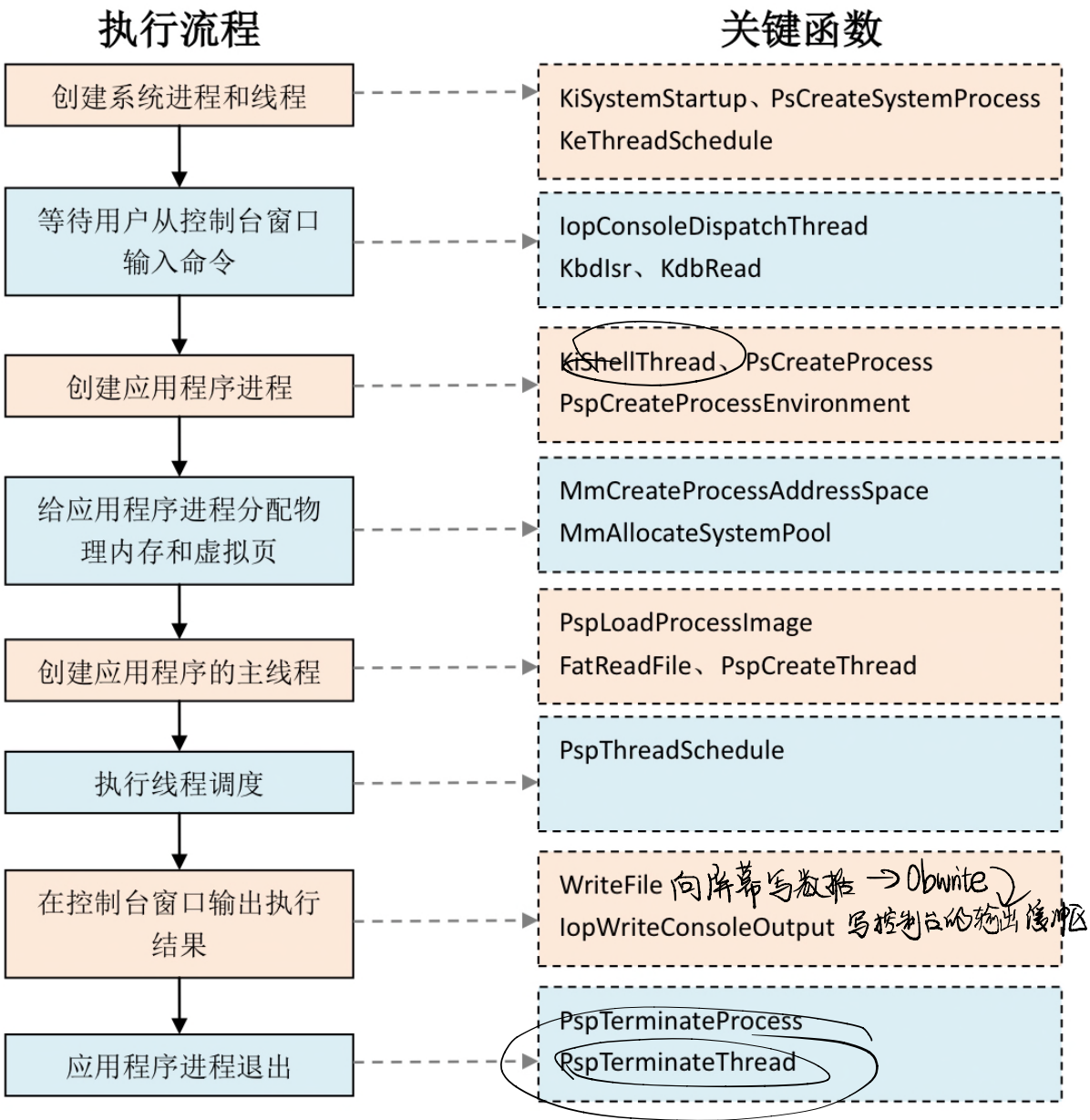
一个应用程序在操作系统中执行看似很简单，但用户在看到应用程序的输出结果之前，操作系统做了大量的幕后工作，因此，对操作系统更为深入的学习，除了学习每个模块的功能之外，还需要从根本上来研究操作系统是如何让应用程序执行的整个流程。

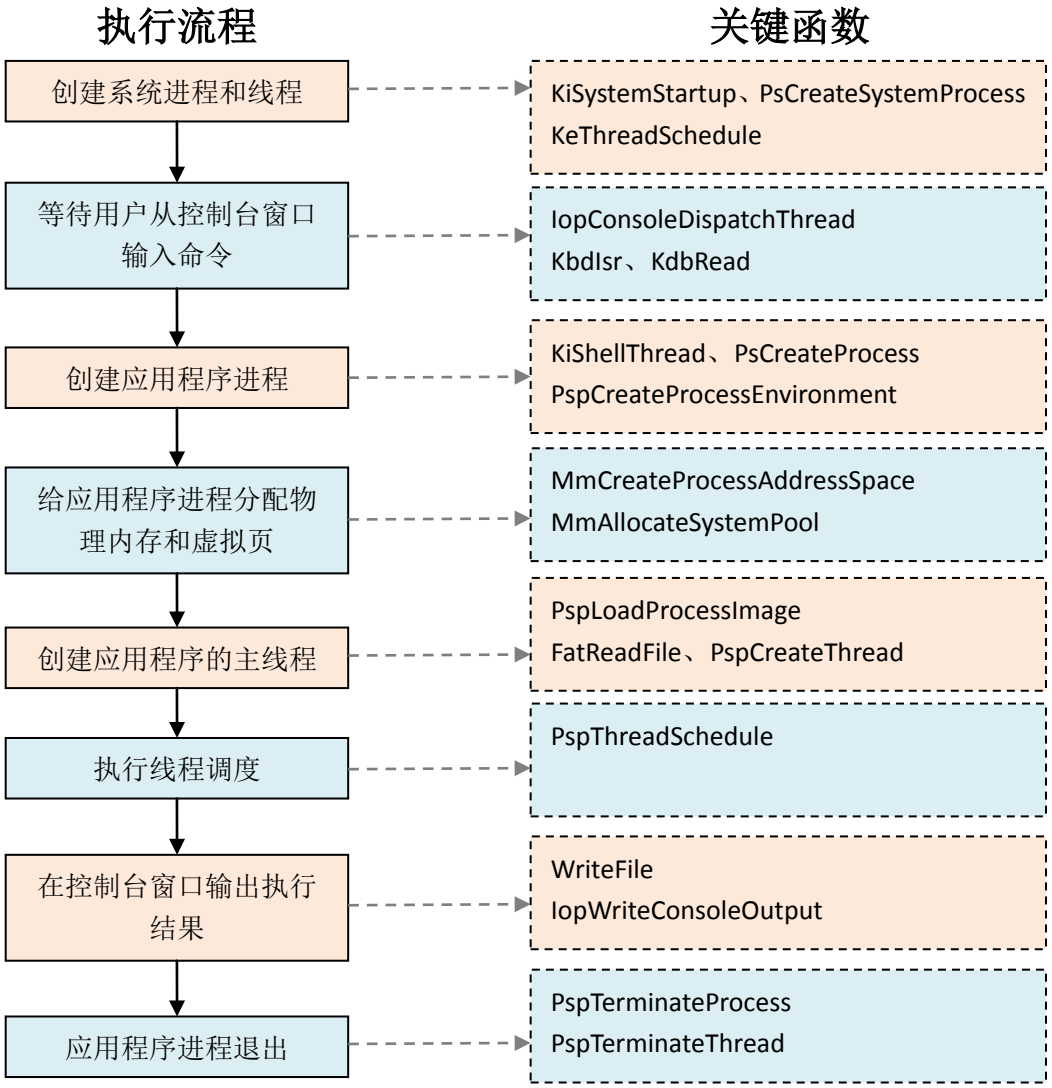
应用程序在执行过程中可以解决以下问题，这些问题的不断解决可以将应用程序的执行

过程串联起来。

序号	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？

在解决以上问题之前，有必要通过下面的流程图理解应用程序的执行流程和调用到的关键函数，这样可以帮助读者在较短的时间内把握整个执行流程。





以下的内容将通过调试 EOS 操作系统的源代码，并对每个流程中的关键内容进行讲解，使读者将已经学习到的理论知识和实际的代码关联起来，对操作系统的研究就可以提升到一个新的高度。

2.1 任务（一）：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？

操作系统启动后（也就是 Kernel.dll 被 Loader 加载到内存后），读者最想了解的是最先进入的是哪个函数，最关注的是操作系统的重要模块的功能是如何实现的，其中进程线程管理模块不仅在 EOS 内核发挥着非常重要的作用，而且在今后的高级语言编程中也会经常用到，那么深入理解进程线程管理模块的原理变得尤为重要，这样不仅可以编写出高质量的程序，而且对以后从事操作系统相关方面的工作也有很大帮助。

第一部分的重点是引导读者理解进程线程管理模块，以解决读者的疑问。第一个执行的是入口点函数 KiSystemStartup，该函数主要功能是初始化处理器、中断、可编程中断控制器、可编程定时计数器、并对各个管理模块执行第一步初始化，接着创建系统进程、线程，系统线程处于就绪状态后，执行线程调度，系统线程开始使用各自的线程栈运行，然后创建初始化线程，系统线程退化为空闲线程，再创建控制台派遣线程，最后创建了 4 个控制台线程，这些线程之间的调度是基于优先级的抢先式调度。

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/eos-kernel-01.git

补全源代码

打开 Ke/start.c 文件，KiSystemStartup 函数的函数体是空的，是留给读者完成的。要求读者先阅读注释，然后根据注释和对操作系统原理的理解在源代码中查找相关函数的声明和实现（这些需要调用的函数在源代码中已实现），最后编写代码实现该函数的功能。

提示：选择“调试”菜单中的“启动调试”菜单项，如果 EOS 操作系统显示蓝屏，需要修改程序，直到可以正常启动。

调试程序

请读者按照下面的步骤调试系统进程线程的创建过程。

1. 打开 ke/start.c 文件，在 KiSystemStartup 函数中的代码 PsCreateSystemProcess 处添加一个断点，按 F5 启动调试，在断点的位置中断，在“进程线程”窗口中，点击“刷新”按钮，可以看到系统中还未创建任何进程和线程。
2. 按 F11 进入 PsCreateSystemProcess 函数的内部调试，按 F10 单步执行完毕第 48 行代码 PspCreateProcessEnvironment。PspCreateProcessEnvironment 函数的功能是创建一个进程环境，此时刷新“进程线程”窗口，可以看到已经创建了一个系统进程。

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	0	0	"N/A"

3. 继续按 F10 单步执行完毕第 55 行代码 PspCreateThread，PspCreateThread 函数的功能是创建一个线程（线程函数是 KiSystemProcessRoutine），刷新“进程线程”窗口，可以看到已经创建了一个系统线程，还可以查看该线程的状态，优先级、父进程 ID、起始地址与函数名等信息。此时，在进程列表中还没有为系统进程设置主线程 ID。
4. 继续按 F10 调试到第 65 行代码，在“进程线程”窗口中，点击“刷新”按钮，可以看到已经为系统进程设置了主线程 ID，该值为在上一步创建的系统线程 ID。

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	1	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	24	Ready (1)	1	0x80017e40 KiSystemProcessRoutine

5. 按 F10 单步调试，返回到上一层函数的代码 KeThreadSchedule 处，但是先不要执行此行代码。KeThreadSchedule 的功能是触发线程调度软中断(对应的中断号是 48 号中断，在 inc/ke.h 文件中的第 131 代码处定义)，在中断返回时会执行线程调度，这样操作系统就会转去执行刚刚创建的那个系统线程了。
6. 在系统线程函数中添加一个断点：在右侧的“项目管理器”窗口中打开 ke/sysproc.c 文件，在第 123 行代码处添加一个断点。
7. 按 F5 继续调试，在断点位置中断。刷新“进程线程”窗口，可以看到系统中唯一的线程处于运行状态，说明该线程正在执行其线程函数并命中了断点。

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)	
1	2	Y	24	Running (2)	1	0x80017e40 KiSystemProcessRoutine	← PspCurrentThread

8. 当前中断位置处的函数 PsCreateThread 用于创建一个新的系统线程（线程函数是 KiInitializationThread)，然后使用这个新建的线程继续进行操作系统的初始化工作。按 F10 单步调试一次后，刷新“进程线程”窗口，可以看到已经创建了初始化线程。

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)	
1	2	Y	24	Running (2)	1	0x80017e40 KiSystemProcessRoutine	← PspCurrentThread
2	3	Y	24	Ready (1)	1	0x80017ed0 KiInitializationThread	

9. 继续按 F10 单步执行，当黄色箭头指向第 140 行的代码 PsSetThreadPriority 时，就不要再执行了。此行代码可以将当前线程(即系统给创建的第一个线程)的优先级降为 0，使之退化为空闲线程。这就会让刚刚新建的还处于就绪状态的系统初始化线程（优先级为 24）抢占处理器并开始运行。
10. 接下来，在初始化线程的线程函数中添加一个断点：打开 ke/sysproc.c 文件，在 KiInitializationThread 函数中（第 160 行代码处）添加一个断点。
11. 按 F5 继续调试，在断点处中断。刷新“进程线程”窗口，可以看到系统初始化线程处于运行状态，说明该线程正在执行其线程函数并命中了断点。

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)	
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine	
2	3	Y	24	Running (2)	1	0x80017ed0 KiInitializationThread	← PspCurrentThread

12. 按 F10 单步执行 IoInitializeSystem2 函数，完成基本输入输出的初始化。刷新“进程线程”窗口，可以看到当前系统中已经创建了控制台派遣线程，并处于就绪状态。请读

者自行查找创建控制台派遣线程的代码位置，并尝试说明其函数调用层次。

线程列表

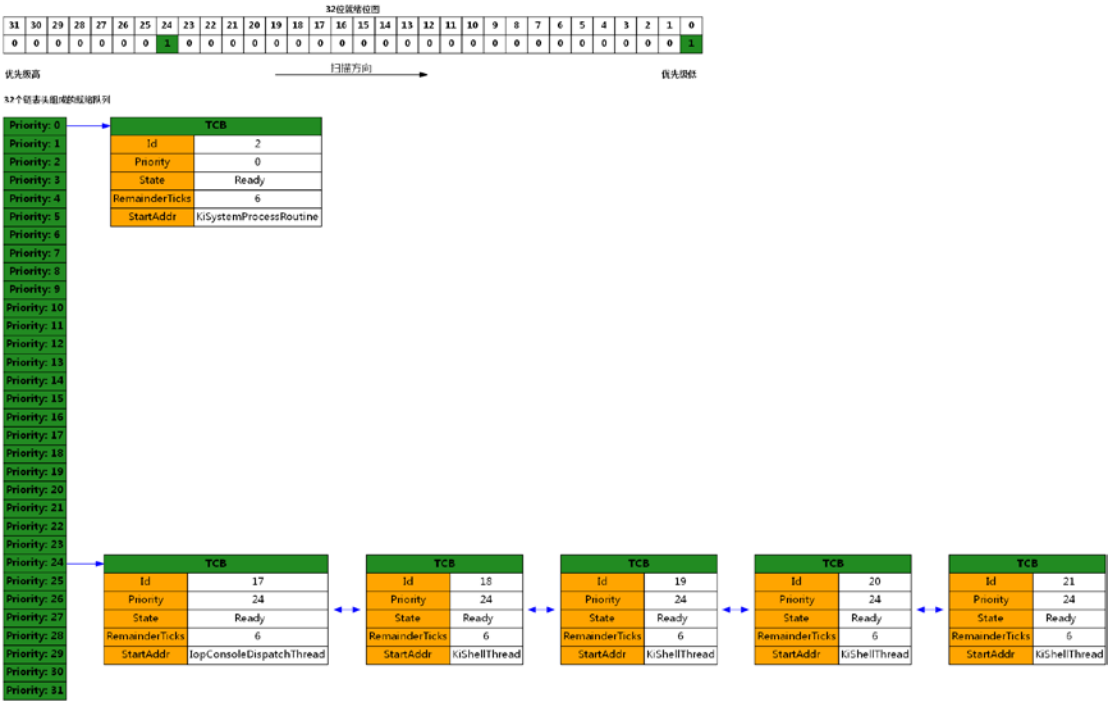
序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	3	Y	24	Running (2)	1	0x80017ed0 KiInitializationThread ← PspCurrentThread
3	17	Y	24	Ready (1)	1	0x80015724 IopConsoleDispatchThread

13. 由于第 168 行的 for 语句会循环创建四个控制台线程，所以，读者可以在 184 行代码处添加一个断点，按 F5 继续调试。待命中断点后，刷新“进程线程”窗口，可以看到当前系统已经完成了四个控制台线程的创建，并且所有的控制台线程都处于就绪状态。

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	3	Y	24	Running (2)	1	0x80017ed0 KiInitializationThread ← PspCurrentThread
3	17	Y	24	Ready (1)	1	0x80015724 IopConsoleDispatchThread
4	18	Y	24	Ready (1)	1	0x80017f4b KiShellThread
5	19	Y	24	Ready (1)	1	0x80017f4b KiShellThread
6	20	Y	24	Ready (1)	1	0x80017f4b KiShellThread
7	21	Y	24	Ready (1)	1	0x80017f4b KiShellThread

14. 刷新“就绪线程队列”窗口，可以看到优先级为 0 的空闲线程，以及优先级为 24 的控制台派遣线程和四个控制台线程分别在其优先级对应的就绪队列中，而当前处于运行态的系统初始化线程（线程 ID 为 3）就不在就绪队列中。



15. 刷新“线程运行轨迹”窗口可以查看这些系统线程的状态转换过程和运行轨迹。注意，系统初始化线程（TID=3）在初始化的过程中会由于等待一些硬件设备的响应，从而频繁进入阻塞状态，当其进入阻塞状态时，空闲线程（TID=2）就会获得处理器并开始运行。

63	运行							PspSelectNextThread	462
64		就绪						PspReadyThread	134
64	就绪							PspSelectNextThread	408
64		运行						PspSelectNextThread	462
64		阻塞						PspWait	230
64	运行							PspSelectNextThread	462
66		就绪						PspReadyThread	134
66	就绪							PspSelectNextThread	408
66		运行						PspSelectNextThread	462
66		阻塞						PspWait	230
66	运行							PspSelectNextThread	462
67		就绪						PspReadyThread	134
67	就绪							PspSelectNextThread	408
67		运行						PspSelectNextThread	462
71			创建					PspCreateThread	572
71			就绪					PspReadyThread	134
72				创建				PspCreateThread	572
72				就绪				PspReadyThread	134
73					创建			PspCreateThread	572
73					就绪			PspReadyThread	134
74						创建		PspCreateThread	572
74						就绪		PspReadyThread	134
75							创建	PspCreateThread	572
75							就绪	PspReadyThread	134
Tick	TID=2	TID=3	TID=17	TID=18	TID=19	TID=20	TID=21	函数	行号

16. 当操作系统启动完毕后，初始化线程（TID=3）就会结束，空闲线程（TID=2）会进入死循环一直运行，控制台派遣线程和四个控制台线程就会进入阻塞状态，等待用户通过键盘输入命令。读者可以在 ke/sysproc.c 文件的第 143 行添加一个断点，按 F5 继续运行，命中断点后，刷新“进程线程”窗口，可以看到空闲线程处于运行态，其它线程都处于阻塞态；刷新“线程运行轨迹”窗口，可以看到之前描述的线程运行过程；刷新“就绪线程队列”窗口，可以看到没有处于就绪状态的线程；刷新“键盘”窗口，可以看到控制台派遣线程（TID=17）阻塞在键盘的缓冲区事件上，也就是正在等待键盘上的输入；四个控制台线程阻塞在各自的缓冲区中，当缓冲区非空时，控制台就会进入就绪状态，准备开始接收命令了。
17. 按 Shift+F5 停止调试。
18. 练习。
    (1) 说明 PsCreateThread 函数和 PspCreateThread 函数的区别。
    (2) 在下面的表格中填写系统线程相关的信息。

系统线程名称	默认优先级	线程处理函数	功能

通过对第一部分内容的调试，读者对**进程线程管理模块**已经有初步的了解，在下一部分将带领读者了解 **FAT12 文件系统**，查看软盘的使用情况和软盘包含的文件列表。

2.2 任务（二）：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？

读者在上一部分已经对系统进程线程的创建和执行过程有一定的理解，在接下来的内容中将围绕一个应用程序在操作系统中开始运行到结束的过程展开探索和讨论，通过一个应用程序的完整生命周期来理解操作系统的各个模块是如何协同工作的。本部分将带领读者了解**FAT12 文件系统**，查看软盘的使用情况和软盘包含的文件列表，应用程序的可执行文件就在软盘的 FAT12 文件系统中。

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/eos-kernel-02.git

补全源代码

1. 在项目管理器窗口，双击“Floppy.img”文件，可以使用 Floppy Image Editor 工具打开软盘镜像，打开后可以查看软盘中的内容。
2. 打开 io/driver/fat12.h 文件，在第 115 行的代码定义了 FAT12 文件系统设备对象的扩展结构体——卷控制块（VCB）。

```
typedef struct _VCB {
    PDEVICE_OBJECT DiskDevice;  // 文件系统下层的软盘卷设备对象
    BIOS_PARAMETER_BLOCK Bpb;   // 文件系统的参数。
    PVOID Fat;                  // 文件分配表缓冲区。
    ULONG FirstRootDirSector;   // 根目录起始扇区
    ULONG RootDirSize;          // 根目录大小
    LIST_ENTRY FileListHead;    // 根目录文件链表头。
    ULONG FirstDataSector;      // 文件数据区的起始扇区
    USHORT NumberOfClusters;    // 簇的总数
}VCB, *PVCB;
```

3. 打开 ke/sysproc.c 文件，ConsoleCmdScanDisk 函数的功能是通过控制台命令 sd 输出软盘的 FAT12 文件系统相关信息，该函数的函数体是空的，要求读者编写代码实现函数的功能。编写完成后，按 F5 启动调试，EOS 操作系统启动后，输入 sd 命令，查看软盘的相关信息，输出的内容可以参考下图。

```
>sd
******** BIOS Parameter Block (BPB) ********
Bytes Per Sector    : 512
Sectors Per Cluster: 1
Reserved Sectors    : 1
Fats                : 2
Root Entries        : 224
Sectors             : 2880
Media               : 0xF0
Sectors Per Fat     : 9
Sectors Per Track   : 18
Heads               : 2
Hidden Sectors      : 0
Large Sectors       : 0
******** BIOS Parameter Block (BPB) ********

First Sector of Root Directroy: 19
Size of Root Directroy        : 7168
First Sector of Data Area     : 33
Number Of Clusters            : 2847

Free Cluster Count: 2272 (1163264 Byte)
Used Cluster Count: 575 (294400 Byte)
>
```

4. 打开 ke/sysproc.c 文件,ConsoleCmdDir 函数的功能是通过控制台命令 dir 输出软盘根目录中的文件信息，该函数的函数体是空的，要求读者编写代码实现函数的功能。编写完成后，按 F5 启动调试，EOS 操作系统启动后，输入 dir 命令查看软盘根目录文件中的信息，输出的内容可以参考下图。

```
>dir
Name         |   Size(Byte) |    Last Write Time
LOADER.BIN        1566          2020-6-3 15:10:13
HELLO.EXE         9276         2011-2-12 14:34:0
KERNEL.DLL        280756          2020-6-3 15:10:13
>
```

5. 输入 Hello.exe 命令，可以查看应用程序的输出结果。在接下来的内容中将探索应用程序的整个执行流程。

```
>Hello.exe
Hello,world! 1
Hello,world! 2
Hello,world! 3
Hello,world! 4
Hello,world! 5
Bye-bye!
```

6. 练习

（1）修改“dir”命令函数 ConsoleCmdDir 的源代码，要求在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）。

2.3 任务（三）：操作系统如何从控制台窗口获取用户输入的命令？

在上一部分读者已经对 FAT12 文件系统有了一定的了解，在本部分通过在控制台中输入一个命令的过程引导读者理解**输入输出模块**的相关内容，特别是加深对**字符设备**的理解。

操作系统接受用户输入命令的主要过程是，操作系统通过控制台派遣线程函数 IopConsoleDispatchThread，在 for 循环中调用 ObRead 函数读取键盘事件，然后调用 IopWriteConsoleInput(IopActiveConsole, &KeyEventRecord)函数，将用户键盘事件发送给当前活动的控制台执行命令并在屏幕上输出。

```
IopConsoleDispatchThread() {
```

```
    for(;;) {
            //
            // 读取键盘事件。
            //
            Status = ObRead( KeyboardHandle,
                             &KeyEventRecord,
                             sizeof(KEY_EVENT_RECORD),
                         &NumberOfBytesRead );
            ......
            IopWriteConsoleInput(IopActiveConsole, &KeyEventRecord);
    }
}
```

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/eos-kernel-03.git

补全源代码

1. 打开 io/driver/keyboard.c 文件，KbdRead 函数的功能是键盘驱动程序提供的 Read 功能，该函数的功能还没有完全实现，请读者根据注释的内容编写代码。
2. 打开 io/driver/keyboard.c 文件，KbdIsr 函数的功能是键盘中断服务程序，该函数的功能还没有完全实现，请读者编写代码实现该函数的功能，包括：

（1） 从 8042 数据端口读取键盘扫描码。

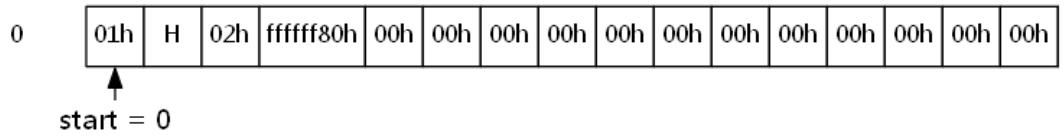
（2） 将键盘事件写入缓冲区，并设置键盘非空事件。

调试程序

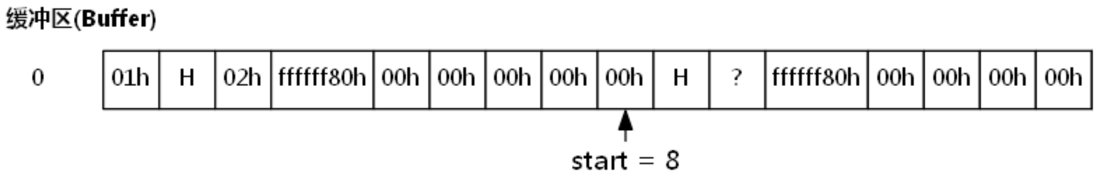
1. 使用快捷键 Ctrl + Shift + F9 删除所有的断点。
2. 打开 io/console.c 文件，在第 615 行和 666 行代码处添加断点，当控制台派遣线程开始运行时，会首先在第 615 行处中断。按 F5 启动调试,在第 615 行处中断后，在 io/driver/keyboard.c 文件中的 KbdIsr 函数中的代码 PKEYBOARD_DEVICE_EXTENSION Ext = (PKEYBOARD_DEVICE_EXTENSION)KbdDevice[0]->DeviceExtension;所在的行再添加一个断点，在键盘中断处理程序中添加断点后，当有键盘按下或抬起时都会命中这个断点。
3. 按 F11 进入 ObRead 函数的内部，ObRead 函数的功能是调用内核对象的读操作。这里访问的对象是一个文件对象，也就是说操作系统将键盘设备作为一个文件对象来访问。按 F10 单步调试到第 111 行，在这里调用对象类型中注册的 Read 操作。
4. 按 F11 进入 IopReadFileObject 函数的内部（io/file.c 298 行），该函数的功能是读取文件对象。

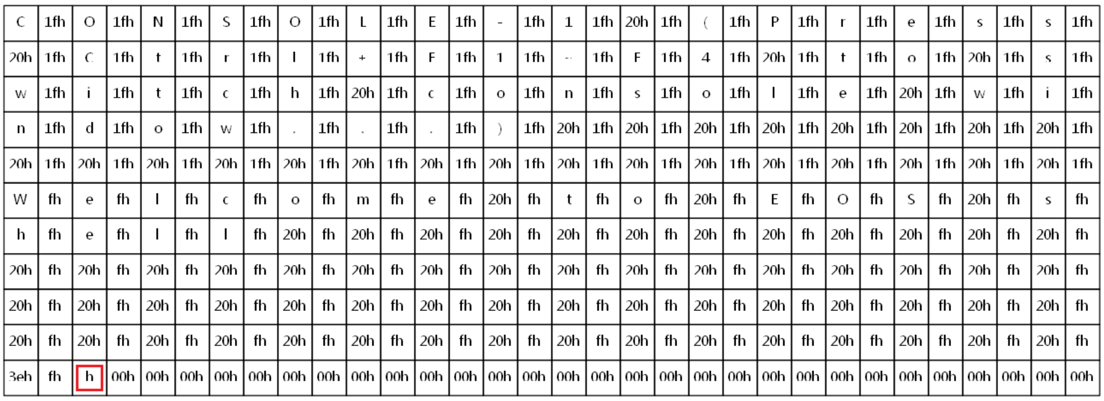
5. 按 F10 单步执行到第 324 行的代码处，此时已经从键盘文件对象得到了键盘设备对象，在这里将要调用键盘设备对象绑定的键盘驱动程序所提供的 Read 函数，按 F11 就会进入 KbdRead 函数内部。
6. 按 F10 单步调试到“阻塞等待直到缓冲区非空”所在的代码行，刷新“键盘”窗口，可以看到键盘缓冲区为空，还没有任何内容，因为此时还没有通过键盘输入任何内容；刷新“进程线程”窗口，可以看到当前正在运行的仍然是控制台派遣线程。
7. 按 F10 单步调试，由于键盘缓冲区为空，所以控制台派遣线程就会阻塞在键盘缓冲区事件中。此时用户就可以从键盘输入命令了。
8. 激活 Bochs 虚拟机的 Display 窗口,可以看到 EOS 内核已经完全启动,当输入 hello.exe 命令，在输入第一个小写 h 字符时，会在文件 io/driver/keyboard.c 的
   PKEYBOARD_DEVICE_EXTENSION Ext = (PKEYBOARD_DEVICE_EXTENSION)KbdDevice[0]->DeviceExtension;
   所在的代码行的断点处中断。
9. 按 F10 单步调试，将要执行的代码行为“从 8042 数据端口读取键盘扫描码”对应的代码。此行代码是从键盘控制器 8042 数据端口读取键盘扫描码，按 F10 单步执行该行代码后,打开“快速监视”窗口,输入表达式查看键盘扫描码的值(表达式为 ScanCode[i])，点击“重新计算”按钮，可以查看该变量的值为 0x23，这个就是键盘上的 h 按下后得到的键盘扫描码。
10. 按 F10 调试到 KeyEventRecord.VirtualKeyValue = KbdVirtualKeyMap[ScanCode[0]]; 所在的代码行，按 F10 单步执行该行代码，打开“快速监视”窗口，输入表达式“KeyEventRecord.VirtualKeyValue”，点击“重新计算”按钮，可以查看该变量的值为 0x48。由于输入的第一个字符是 h，通过“单字节键盘扫描码对应的虚键查询表”(在 io/driver/keyboard.c 文件的第 38 行)可以 根据键盘扫描码得到虚拟键码的值,例如：h 的键盘扫描码为 0x23,通过查表可以得到对应的虚键值为字符 ‘H’,其 ASCII 码的值就是 0x48。
11. 按 F10 调试到“将键盘事件写入缓冲区”所在的代码行，按 F10 单步执行该行代码将键盘事件写入缓冲区。刷新“键盘”窗口，可以看到 H 键对应的虚拟键码已经写入到了缓冲区中。

缓冲区(Buffer)



12. 按 F5 继续调试，直到在 io/console.c 文件的第 666 行代码处中断，此行代码就是将从键盘缓冲区中读取的字符发送给控制台。所以，刷新“键盘”窗口，可以看到 start 游标的位置已经更新，表明键盘缓冲区中的内容已经被读取。刷新“控制台 1”窗口，可以看到字符“h”还没有写入到控制台输出缓冲区。此时按 F10 单步调试一次后，刷新“控制台 1”窗口，可以看到字符“h”已经写入到控制台输出缓冲区（下图中的红框位置处)。激活 Bochs 虚拟机的 Display 窗口，可以看到刚才输入的字符“h”已经显示出来。




输出缓冲区(ScreenBuffer)

C	1fh	O	1fh	N	1fh	S	1fh	O	1fh	L	1fh	E	1fh	-	1fh	1	1fh	20h	1fh	(	1fh	P	1fh	r	1fh	e	1fh	s	1fh	s	1fh
20h	1fh	C	1fh	t	1fh	r	1fh	l	1fh	+	1fh	F	1fh	1	1fh	--	1fh	F	1fh	4	1fh	20h	1fh	t	1fh	o	1fh	20h	1fh	s	1fh
w	1fh	i	1fh	t	1fh	c	1fh	h	1fh	20h	1fh	c	1fh	o	1fh	n	1fh	s	1fh	o	1fh	l	1fh	e	1fh	20h	1fh	w	1fh	i	1fh
n	1fh	d	1fh	o	1fh	w	1fh	.	1fh	.	1fh	.	1fh	)	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh
20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh
W	fh	e	fh	l	fh	c	fh	o	fh	m	fh	e	fh	20h	fh	t	fh	o	fh	20h	fh	E	fh	O	fh	S	fh	20h	fh	s	fh
h	fh	e	fh	l	fh	l	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
3eh	fh	h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h

13. 停止调试，删除所有断点。
14. 练习。

(1) 记录输入命令 hello 的 5 次按键对应的键盘扫描码。

通过以上的调试，读者对 **I/O 设备管理**中的**字符设备**的工作流程已经有了深入的理解，通过按键可以触发键盘中断，由键盘中断处理程序通过键盘扫描码得到虚拟键码，然后将字符写入键盘缓冲区，最后由控制台派遣线程将键盘缓冲区中的字符写入到控制台缓冲区并显示到屏幕上。在下一部分将调试应用程序进程的创建过程。

2.4 任务（四）：应用程序进程是如何在操作系统内核中创建的？

读者已经在上一部分熟悉了**字符设备**的工作流程，EOS 内核可以通过键盘接收用户输入的命令，那么应用程序又该如何执行呢？本部分再次聚焦到**进程线程管理模块**，与第一部分不同的是本部分重点是探索应用程序进程的创建过程。应用程序进程是在控制台线程处理函数（KiShellThread）中创建的，创建进程调用的函数是 PsCreateProcess，以下将对关键函数的执行流程和功能进行说明，然后调试应用程序进程从创建、执行到结束的整个过程。

关键函数 KiShellThread

```
ULONG KiShellThread(IN PVOID Parameter)
{
    ......
    for (;;){

        fprintf(StdHandle, ">");


        ......


        // 将可执行文件名从 Line 中拷贝至 Image 中。如果仅给出了文件名则自动在
        // 文件名前面加上 a 盘根目录的路径，如果没有给出“.exe”扩展名则自动
```

```
        //添加。
        if (strnicmp(Line, "a:\\", 3) != 0 &&strnicmp(Line, "a:/", 3) != 0)
        {
            sprintf(Image, "a:\\%s", Line);
        } else {
            strcpy(Image, Line);
        }

        if (strnicmp(Line + strlen(Line) - 4, ".exe", 4) != 0) {
            strcpy(Image + strlen(Image), ".exe");
        }

        StartInfo.StdInput = StdHandle;
        StartInfo.StdOutput = StdHandle;
        StartInfo.StdError = StdHandle;

        Status = PsCreateProcess( Image, Arg, 0, &StartInfo, &ProcInfo );

        if (!EOS_SUCCESS(Status)) {
            fprintf(StdHandle, "Error:Create process failed with status code
0x%x.\n", Status);
            continue;
        }

        // 等待进程结束，然后关闭进程和主线程的句柄。
        ObWaitForObject(ProcInfo.ProcessHandle, INFINITE);
        PsGetExitCodeProcess(ProcInfo.ProcessHandle, &ExitCode);
        ObCloseHandle(ProcInfo.ProcessHandle);
        ObCloseHandle(ProcInfo.ThreadHandle);

        fprintf(StdHandle, "\n%s exit with 0x%.8X.\n", Line, ExitCode);
    }
}
```

解析：KiShellThread 函数的主要功能是通过 for 循环来循环处理控制台命令或者执行应用程序，如果没有要执行的控制台命令或应用程序，就一直处于等待状态，直到操作系统退出。在执行应用程序时，需要先将从控制台输入的应用程序名称拷贝到 Image 中，接着调用 PsCreateProcess 函数创建进程，通过该进程执行应用程序，调用 ObWaitForObject 函数等待应用程序进程结束，然后调用 PsGetExitCodeProcess 函数获取进程的退出码，最后调用 ObCloseHandle 函数关闭进程和线程的句柄。

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在

CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/eos-kernel-04.git

补全源代码

打开 ps/create.c 文件，PsCreateProcess 函数的功能是创建一个应用进程，该函数的功能还没有完全实现，请读者根据注释的内容编写代码。

按照下面的步骤调试应用程序进程的创建过程

1. 使用快捷键 Ctrl + Shift + F9 删除所有的断点。
2. 打开 ke/sysproc.c 文件，在第 326 行代码 PsCreateProcess 处添加一个断点。
3. 按快捷键 F5 启动调试，操作系统启动后，在控制台输入 Hello.exe 命令后按回车，在断点的位置中断，可以在以下可视化窗口中查看当前操作系统的状态信息。
   (1) 刷新“进程线程”窗口，可以看到当前正在运行的线程是控制台线程，当前也是在控制台线程处理函数中中断的，在进程列表中只包含一个系统进程，由此可知，还未创建应用程序进程。

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	6	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	17	Y	24	Waiting (3)	1	0x800157 24 IopConsoleDispatchThread
3	18	Y	24	Running (2)	1	0x80017f4b KiShellThread ← PspCurrentThread
4	19	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
5	20	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
6	21	Y	24	Waiting (3)	1	0x80017f4b KiShellThread

(2) 刷新“物理内存”窗口，可以查看物理内存的分配情况。

物理页的数量: 8176
物理内存的大小: 8176 * 4096 = 33488896 Byte
零页的数量: 0
空闲页的数量: 7134
已使用页的数量: 1042

物理页框号	页框号数据库项	状态
0x0	0x80100000	BUSY
0x1	0x80100001	BUSY
……	……	……
0x406	0x80100406	BUSY
0x407	0x80100407	BUSY
0x408	0x80100408	FREE
0x409	0x80100409	FREE
0x40a	0x8010040a	BUSY
0x40b	0x8010040b	BUSY
……	……	……
0x412	0x80100412	BUSY
0x413	0x80100413	BUSY
0x414	0x80100414	FREE
0x415	0x80100415	FREE
……	……	……
0x1fee	0x80101fee	FREE
0x1fef	0x80101fef	FREE

(3) 刷新“控制台”窗口，在输出缓冲区中还未输出程序的执行结果。

输入缓冲区(InputBuffer)

H	e	l	l	o	.	e	x	e	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h

输出缓冲区(ScreenBuffer)

C	1fh	O	1fh	N	1fh	S	1fh	O	1fh	L	1fh	E	1fh	-	1fh	1	1fh	20h	1fh	(	1fh	P	1fh	r	1fh	e	1fh	s	1fh	s	1fh
20h	1fh	C	1fh	t	1fh	r	1fh	l	1fh	+	1fh	F	1fh	1	1fh	-	1fh	F	1fh	4	1fh	20h	1fh	t	1fh	o	1fh	20h	1fh	s	1fh
w	1fh	i	1fh	t	1fh	c	1fh	h	1fh	20h	1fh	c	1fh	o	1fh	n	1fh	s	1fh	o	1fh	l	1fh	e	1fh	20h	1fh	w	1fh	i	1fh
n	1fh	d	1fh	o	1fh	w	1fh	.	1fh	.	1fh	.	1fh	)	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh
20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh	20h	1fh
W	fh	e	fh	l	fh	c	fh	o	fh	m	fh	e	fh	20h	fh	t	fh	o	fh	20h	fh	E	fh	O	fh	S	fh	20h	fh	s	fh
h	fh	e	fh	l	fh	l	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
3eh	fh	H	fh	e	fh	l	fh	l	fh	o	fh	.	fh	e	fh	x	fh	e	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h

4. 将鼠标移动到 PsCreateProcess 函数的第一个参数 Image 上，可以看到第一个参数的值是可执行程序的路径（a:\\Hello.exe）。
5. 按 F11 进入 PsCreateProcess 函数内部调试，在代码 PspCreateProcessEnvironment 处添加一个断点，按 F5 执行到该行代码。
6. 按 F11 进入 PspCreateProcessEnvironment 函数内部调试，在代码 ObCreateObject 所在行添加一个断点，按 F5 执行到该行代码，刷新“进程线程”窗口，可以看到此时还未创建应用程序进程。
7. 按 F10 单步执行 ObCreateObject 所在行的代码，打开 ps/psp.h 文件，在第 44 行的代码可以查看进程对象结构体包含（PCB）的成员，进程的各个成员的值可以通过“进程控制块”窗口查看，也可以通过“监视”窗口查看。刷新“进程线程”窗口，可以看到已经创建了一个进程，还未创建该进程对应的线程；打开“进程控制块”窗口，选择 PID=24 的进程控制块，可以看到该进程的控制块还未初始化。

进程基本信息

进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
24	Y	193	0	0	"N/A"

进程地址空间(Pas)

进程地址空间的开始虚页号	0x0	进程地址空间的结束虚页号	0x0
页目录	0x0	PTE计数器数据库的页框号	0x0

进程的内核对象句柄表(ObjectTable)

HandleTable	0x0	FreeEntryListHead	0x0	HandleCount	0x0

8. 按 F10 执行到 MmCreateProcessAddressSpace 代码所在行，在下一部分继续将进一步调试该函数的执行过程。

在本部分通过调试，查看创建应用程序进程前的系统状态，然后创建了应用程序进程控制块，还未进行初始化，在下一部分的内容中将继续调试为应用程序进程分配物理内存和虚拟页的过程。

2.5 如何为应用程序分配物理内存和在虚拟地址空间如何为应用程序分配虚拟页？

在上一部分创建了应用程序进程控制块，但是还未对其进行初始化，本部分主要通过调试为应用程序分配物理内存和在虚拟地址空间为应用程序分配虚拟页的过程，讲解**存储器管理模块**的相关内容。

在 **2.4** 节的基础上继续完成实验

9. 按 F11 进入 MmCreateProcessAddressSpace 函数内部进行调试，这个函数的功能是分配一个进程的地址空间，第 135 行代码 MmAllocateSystemPool 函数的功能是从系统内存池中分配进程空间（MMPAS）结构体，其中 MMPAS 结构体是在 mm/mi.h 文件中的第 72 行定义的。

```
typedef struct _MMPAS {
    MMVAD_LIST VadList;                 // 分配给进程使用的地址空间。
    ULONG_PTR PfnOfPageDirectory; // 页目录
    ULONG_PTR PfnOfPteCounter;          // PTE 计数器数据库的页框号。
}MMPAS;
```

10. 按 F11 进入 MmAllocateSystemPool 函数内部进行调试，按 F10 执行到第 76 行代码 PoolAllocateMemory 处。
11. 按 F11 进入 PoolAllocateMemory 函数内部进行调试，该函数的主要功能是通过伙伴算法分配内存块，按 F10 单步执行该函数，查看伙伴算法的具体实现过程。
12. 打开 mm/pas.c 文件，在第 137 行代码 if(NULL == Pas)处添加一个断点，按 F5 继续执行到此断点处中断。
13. 按 F10 执行到第 144 行代码 if (!EOS_SUCCESS(MiAllocateZeroedPages(2, PfnArray))) 处，按 F11 进入 MiAllocateZeroedPages 函数的内部，该函数的功能是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。
14. 在 MiAllocateZeroedPages 函数中，按 F10 单步执行到第 223 行代码 for (; i<NumberOfPages; i++)处，打开“物理内存”窗口，点击“刷新”按钮，可以看到还未分配新的物理页。

物理页框号	页框号数据库项	状态
0x0	0x80100000	BUSY
0x1	0x80100001	BUSY
.....	.....	.....
0x406	0x80100406	BUSY
0x407	0x80100407	BUSY
0x408	0x80100408	FREE
0x409	0x80100409	FRFF
0x40a	0x8010040a	BUSY
0x40b	0x8010040b	BUSY
.....	.....	.....
0x412	0x80100412	BUSY
0x413	0x80100413	BUSY
0x414	0x80100414	FRFF
0x415	0x80100415	FREE
.....	.....	.....
0x1fee	0x80101fee	FRFF
0x1fef	0x80101fef	FREE

15. 按 F10 单步执行到第 234 行代码 ZeroBuffer = MiMapPageToSystemPte(Pfn);处，刷新“物理内存”窗口，可以看到已经分配了一个新的物理页，将鼠标移动到 Pfn 变量上，可以查看该变量的值与已分配物理页框号的值是一样的，接下来的代码可以将物理页映射到系统 PTE 区域进行清零。继续调试程序，可以查看另外一个物理页的分配过程。

物理页框号	页框号数据库项	状态
0x0	0x80100000	BUSY
0x1	0x80100001	BUSY
.....	.....	.....
0x406	0x80100406	BUSY
0x407	0x80100407	BUSY
0x408	0x80100408	FREE
0x409	0x80100409	BUSY
0x40a	0x8010040a	BUSY
.....	.....	.....
0x412	0x80100412	BUSY
0x413	0x80100413	BUSY
0x414	0x80100414	FREE
0x415	0x80100415	FREE
.....	.....	.....
0x1fee	0x80101fee	FREE
0x1fef	0x80101fef	FREE

16. 打开 mm/pas.c 文件，在第 154 行代码 MiInitializeVadList 处添加一个断点，按 F5 执行到该代码行，该代码行对应的函数用于初始化可分配虚拟内存的虚拟地址空间描述符链表，按 F10 单步执行该行代码，接下来将在下一部分调试虚拟页与物理内存映射的过程。

读者对**存储器管理模块**有一定的理解之后，下一部分将调试已为应用程序分配的虚拟页与物理内存的映射过程，进一步研究**存储器管理模块**的相关内容。

2.6 已为应用程序分配的虚拟页是如何映射到物理内存的？

读者在上一部分对存储器管理模块已经有一定了解，本部分将通过调试已为应用程序分配的虚拟页与物理内存的映射过程，进一步研究**存储器管理模块**的相关内容。

在 **2.5** 节的基础上继续完成实验

17. 打开 mm/pas.c 文件，第 158 行代码是将物理页框号映射到页目录，第 159 行代码是将另一个物理页框号映射到 PTE 计数器数据库，按 F10 单步执行第 158 行和第 159 行代码，接下来的代码对页目录进行初始化。
18. 打开 ps/create.c 文件，在 PspCreateProcessEnvironment 函数中的 if (NULL == NewProcess->Pas)所在代码行添加一个断点，按 F5 执行到该行代码，在“进程控制块”窗口中，选择 PID=24 的进程控制块，可以看到进程地址空间已经初始化。

进程基本信息

进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
24	Y	193	0	0	"N/A"

进程地址空间(Pas)

进程地址空间的开始虚页号	0x10	进程地址空间的结束虚页号	0x7ffef
页目录	0x409	PTE计数器数据库的页框号	0x408

进程的内核对象句柄表(ObjectTable)

HandleTable	0x0	FreeEntryListHead	0x0	HandleCount	0x0

19. 停止调试，并删除所有的断点。

至此，读者对**存储器管理模块**已经有了一个全面的认识，在下一部分中，将调试初始化进程控制块、加载可执行映像，创建进程的主线程，线程的状态转换的过程。

2.7 任务（五）：应用程序进程的主线程如何创建的，创建后是如何进行状态转换的？

通过上面两部分的内容，读者对存储器管理模块已经有了一个全面的理解，应用程序的进程已经有了独立的空间，可以使用系统分配的内存了，接下来将调试加载应用程序可执行文件、创建应用程序主线程的过程和线程状态转换的相关内容。

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/eos-kernel-05.git

补全源代码

打开 ps/peldr.c 文件，PspLoadProcessImage 函数的功能是加载 EOS 应用程序的可执行文件到内存，该函数的功能还没有完全实现，请读者根据注释的内容编写代码。

调试程序

1. 初始化进程控制块

(1) 打开 ke/sysproc.c 文件，在第 326 行添加一个断点，按 F5 启动调试。

(2) 按快捷键 F5 启动调试，操作系统启动后，在控制台输入 Hello.exe 命令后按回车，在断点处中断。

(3) 在 ps/create.c 文件的第 452 行代码 NewProcess->ObjectTable = ObAllocateHandleTable()处添加一个断点，按 F5 执行到该代码行，接下来的代码为进程分配一个句柄表，初始化进程控制块，按 F10 单步执行到第 510 行代码 Status = STATUS_SUCCESS 处，打开“进程控制块”窗口，选择 PID = 24 的进程控制块，可以看到应用程序进程控制块已经被初始化。

进程基本信息

进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
24	N	8	0	0	"N/A"

进程地址空间(Pas)

进程地址空间的开始虚页号	0x10	进程地址空间的结束虚页号	0x7ffef
页目录	0x409	PTE计数器数据库的页框号	0x408

进程的内核对象句柄表(ObjectTable)

HandleTable	0xa0003000	FreeEntryListHead	0x1	HandleCount	0x0

2. 加载可执行映像

(1) 在 ps/create.c 文件的 195 行代码 Status = PspLoadProcessImage 处添加一个断点，按 F5 执行到该行代码处，按 F11 进入 PspLoadProcessImage 函数的内部。

(2) 在 PspLoadProcessImage 函数中，调用 IoCreateFile 函数打开可执行文件，接着调用 PspThreadAttachProcess 函数将当前线程附着到新建进程的地址空间中执行，按 F10 单步执行到代码 ObRead（读取 PE 文件的全部头）行，调用 ObRead 函数读取 PE 文件的全部头，按 F11 进入该函数的内部调试。

(3) 在 ObRead 函数内部，按 F10 单步调试到 111 行，该行代码的功能是调用对象类型中注册的 Read 操作，将鼠标移动到 Type->Read 上可以查看即将要调用的是 IopReadFileObject 函数以及该函数的地址，刷新“对象类型”窗口，根据函数的名称和地址，可以确定调用调用的类型是“FILE”类型，按 F11 进入函数函数内部进行调试。

(4) 在 IopReadFileObject 函数内部，按 F10 单步调试到 324 行，该行代码的功能是执行驱动程序的 Read 功能函数，在快速监视窗口查看 DeviceObject->DriverObject->Read 的值可以知道即将要调用的是 FatRead 函数，按 F11 进入函数的内部进行调试。

(5) 在 FatRead 函数内部调用 FatReadFile 函数，按 F11 进入 FatReadFile 函数内部进行调试，该函数的功能是在文件的指定偏移处读取指定字节的数据，在“快速监视”对话框中输入表达式“*Vcb”后按回车。参数 Vcb 提供的信息如下图。结合 VCB 结构体（文件 io/driver/fat12.h 的第 115 行）和 BPB 结构体（文件 io/driver/fat12.h 的第 27 行）的定义，尝试说明这些信息的含义。

```
{
DiskDevice = 0x803fc578,
Bpb = {
BytesPerSector = 0x200,
SectorsPerCluster = 0x1,
ReservedSectors = 0x1,
    Fats = 0x2,
RootEntries = 0xe0,
    Sectors = 0xb40,
    Media = 0xf0,
SectorsPerFat = 0x9,
```

```
    SectorsPerTrack = 0x12,
        Heads = 0x2,
    HiddenSectors = 0x0,
    LargeSectors = 0x0
      },
      Fat = 0x803f7fc8,
    FirstRootDirSector = 0x13,
    RootDirSize = 0x1c00,
    FileListHead = {
        Next = 0x803faaf4,
    Prev = 0x803faaf4
      },
    FirstDataSector = 0x21,
    NumberOfClusters = 0xb1f
}
```

(6) 在“快速监视”对话框中输入表达式“*File”后按回车。参数 File 提供的信息如下图。结合 FCB 结构体（文件 io/driver/fat12.h 的第 150 行）定义，尝试说明这些信息的含义。

```
{
  Name = "HELLO.EXE\000\000\000\001",
  ......
FirstCluster = 0x3,
FileSize = 0x243c,
DirEntryOffset = 0x20,
OpenCount = 0x1,
ParentDirectory = 0x0,
......
    }
```

(7) 关闭“快速监视”对话框后，将鼠标移动到参数 Offset 上，显示其值为 0，说明从文件头开始读取。将鼠标移动到参数 BytesToRead 上，显示其值为 0x400。将鼠标移动到参数 Buffer 上，显示缓冲区所在地址在内核地址空间（大于 0x80000000），也就是在内核中定义的缓冲区。

(8) 由于读取文件的起始偏移位置（0）没有超出文件的大小，所以按 F10 单步调试后，可以继续读取文件。

(9) 由于预期读取的字节数（0x400）小于文件的大小，所以实际可读取的字节数（BytesToRead）应为 0x400。按 F10 调试，直到在第 758 行中断。

(10) 由于要读取的偏移位置 Offset 是 0，所以开始读取的簇 Cluster 就是文件的第一个簇。按 F10 直到在第 770 行中断。注意，C 语言运算符“/”只取商。

(11) 第 770 行计算簇的起始扇区号。按 F10 单步调试。如果文件第一个簇是 3，查看计算的结果 FirstSectorOfCluster 就会为 34，尝试说明计算的方法。

(12) 接下来使用双重循环读取扇区中的数据，外层循环是遍历文件簇链中的所有簇，内层循环是遍历一个簇中的所有扇区。这里使用的 FAT12 文件系统每个簇只有一个扇区，所以并没有循环执行。按 F10 单步调试，直到该函数执行完毕，注意观察各个变量的值和计算方法。

(13) 打开 ps/peldr.c 文件，在“验证是否是 EOS 的应用程序”的代码行添加一个断点，按 F5 继续执行，按 F10 单步执行该行代码。

(14) 接下来得到可执行文件的基址和大小，然后分配虚拟内存，将可执行文件的内容读取到内存中，并将应用程序和内核进行连接，最后当前线程返回所属进程地址空间继续执行，可以按 F10 单步调试这部分内容。

3. 创建应用程序进程的主线程

(1) 在 ps/create.c 文件的第 207 行代码 Status = PspCreateThread 处添加一个断点，按 F5 执行到该行代码处，按 F11 进入 PspCreateThread 函数的内部，该函数的功能是在指定的进程内创建一个线程。刷新“进程线程”窗口，可以看到还未创建应用程序的线程。在 ps/psp.h 文件的第 68 行可以查看线程对象结构体（TCB）的相关成员。注意，PspCreateThread 函数中使用 PspProcessStartup 函数作为线程函数，而在 PspProcessStartup 函数中就是调用了应用程序可执行文件的入口点，也就是说当应用程序的主线程开始执行时，就会从其入口点开始执行。而应用程序的入口点就是在应用程序的 C 运行时库中定义的_start 函数，读者可以在 EOS 应用程序项目的 crt/src/crt0.c 文件中找到该函数的定义。

(2) 按 F10 单步执行到第 560 行，ObCreateObject 函数用于创建线程对象，按 F10 单步执行到第 582 行，调用 MmAllocateVirtualMemory 函数分配虚拟内存，按 F10 单步执行到第 602 行，接下来的代码用于初始化线程控制块，按 F10 执行到 623 行代码 PspReadyThread(NewThread)处，此时已经初始化了**线程控制块**，并将线程插入所在进程的线程链表中，刷新“进程线程”窗口，可以看到已经创建了应用程序的线程，并进行了初始化。

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
2	24	N	8	0	0	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	26	N	8	Zero (0)	24	0x8001f97e PspProcessStartup

4. 线程的状态转换

(1) 按 F10 单步执行 PspReadyThread(NewThread)，可以使应用程序的线程进入就绪态，关于线程状态转换的内容可以参考前面的实验四。

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	26	N	8	Ready (1)	24	0x8001f97e PspProcessStartup

2.8 在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？

读者在上一部分通过调试线程的创建和状态转换的过程，加深了对线程的认识，接下来调试线程的调度过程，线程调度本质上就是在系统中的就绪线程中选取下一个即将要运行的线程。

在 **2.7** 节的基础上继续完成实验

5. 线程调度

(1) 按 F10 单步执行到第 634 行代码 PspThreadSchedule()处，PspThreadSchedule 函数的功能是有线程进入就绪状态，执行线程调度，可以按 F11 进入该函数内部调试线程调度的过程，当前 EOS 操作系统是基于优先级调度的，关于线程调度的内容可以参考前面的实验六。

(2) 在 ps/create.c 文件的第 218 行代码处 ProcessObject->PrimaryThread = ThreadObject 添加一个断点，按 F5 执行到该行代码处中断，接下来需要设置应用程序进程的主线程，设置父进程中的句柄值，使之指向新建子进程对象及其主线程对象，设置返回结果，包括子进程及其主线程的句柄和 ID。

(3) 停止调试。

(4) 删除所有的断点。

6. 练习

(1) 应用程序线程处于运行状态时，哪些线程处于就绪状态，哪些线程处于阻塞状态，并说明原因。

(2) EOS 内核是如何解决让正在运行的线程主动中断后触发线程调度让出处理器的情况。

在下一部分的内容中，将调试应用程序的执行结果输出到控制台窗口的过程。

2.9 任务（六）：应用程序的执行结果如何输出到控制窗口中？

在前面的部分中，读者已经理解了应用程序线程的创建、状态转换、调度的过程，在接下来的内容中将调试应用程序将执行结果输出到控制台窗口的过程。

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/eos-kernel-06.git

补全源代码

打开 io/console.c 文件，IopWriteConsoleOutput 函数的功能是写控制台的输出缓冲区，该函数的函数体是空的，要求读者编写代码完成该函数。

调试程序

1. 查看应用程序的输出结果。

(1) 打开 ke/sysproc.c 文件，在第 326 行添加一个断点，按 F5 启动调试，然后在 EOS 操作系统的控制台窗口，输入 hello.exe 然后按回车，在断点位置中断。

(2) 打开 api/eosapi.c 文件，在第 498 行添加一个断点。这样当应用程序调用 C 标准库中的 printf 函数时，在此函数中就会就会调用 WriteFile 函数向标准输出设备（屏幕）写数据，就会命中这个断点。

(3) 按 F5 继续执行，命中 WriteFile 函数中的断点后，按 F11 可以进入 ObWrite 函数内部。

(4) 按 F10 单步执行到第 165 行代码 Type->Write 处，按 F11 进入 IopWriteConsoleOutput 函数的内部。

(5) 按 F10 单步执行到代码 IopWriteScreenBuffer，按 F11 进入函数内部，该函数的功能是将字符写入到屏幕缓冲区，按 F10 单步执行到 342 行，刷新“控制台 1”窗口，可以看到字符 H 已经写入到输出缓冲区中。激活控制台窗口，可以看到已经输出字符 H。

输出缓冲区(ScreenBuffer)

3eh	fh	H	fh	e	fh	l	fh	l	fh	o	fh	.	fh	e	fh	x	fh	e	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh	20h	fh
H	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h

(6) 按 F5 继续执行，在 api/eosapi.c 文件的第 498 行断点处中断，可以按照（2）～（4）步调试执行结果中其它字符的输出。

(7) 应用程序的执行结果都输出之后，删除所有断点，停止调试。

2. 练习。

(1) 请读者编写一个 EOS 应用程序，在其中调用 C 标准库中的 gets 函数，从标准输入获取用户输入的数据，并将此应用程序放入内核项目的软盘镜像文件中。然后读者在内核 api/eosapi.c 文件第 477 行添加一个断点调试应用程序调用 ReadFile 函数的过程，模仿上面的内容对执行的过程进行详细的描述，并将整个过程所调用的各个函数填入到下表中。

函数名称	功能	所在文件	所在行

在下一部分将调试应用程序退出的过程。

2.10 任务（七）：应用程序进程是如何退出的？

应用程序进程执行完之后，需要释放虚拟地址空间和物理内存，并结束该进程的所有线程，然后释放线程控制块、进程控制块，最后设置返回信息。可以按照下面的步骤调试应用程序进程退出的过程。

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/eos-kernel-07.git

补全源代码

1. 打开 ps/delete.c 文件，PspTerminateProcess 函数的功能是结束指定进程，该函数的功能还没有完全实现，请读者根据注释的内容编写代码。
2. 打开 ps/delete.c 文件，PspTerminateThread 函数的功能是结束指定线程，该函数的功能还没有完全实现，请读者根据注释的内容编写代码。

调试程序

1. 删除所有断点。
2. 打开 ke/sysproc.c 文件，在第 326 行添加一个断点，按 F5 启动调试，然后在 EOS 操作系统的控制台窗口，输入应用程序名称然后按回车，在断点位置中断。
3. 打开 api/eosapi.c 文件，在第 208 行代码 PsExitProcess 处添加一个断点，按 F5 继续执行，在断点处中断，激活控制台窗口，可以看到应用程序已经执行完毕，按 F11 进入 PsExitProcess 函数的内部，该函数的功能是退出当前进程。
4. 按 F10 执行到第 120 行代码 PspTerminateProcess 处，按 F11 进入函数的内部。
5. 按 F10 执行到 PspTerminateThread 代码处，该行代码用于结束进程中的指定线程，按 F11 进入 PspTerminateThread 函数，然后按 F10 单步调试该函数中的代码。
6. 打开 ps/delete.c 文件，在 PspTerminateProcess 函数中的 PspDeleteProcessEnvironment 代码处一个断点，按 F5 继续执行，在断点位置中断。
7. 按 F11 进入函数 PspDeleteProcessEnvironment 函数内部，该函数的功能是删除进程的地址空间。
8. 按 F10 执行到 166 行，该行代码用于释放句柄表。
9. 按 F10 执行到 172 行，该行代码用于当前线程附着到被结束进程的地址空间中执行，以执行清理操作。
10. 按 F10 执行到 177 行，MmCleanVirtualMemory 函数用于清理进程用户地址空间中的虚拟内存。
11. 按 F10 执行到 200 行，刷新“进程线程”窗口，可以看到应用程序进程的线程数量为 0，主线程 ID 为 0，在应用程序的线程列表中，线程状态为终止状态。

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
2	24	N	8	0	0	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	26	N	8	Terminated (4)	24	0x8001f97e PspProcessStartup ← PspCurrentThread

12. 打开 ps/delete.c 文件，在 PspTerminateProcess 函数中的代码 PspThreadSchedule 处添加一个断点，按 F5 继续执行，调用 PspThreadSchedule 函数执行线程的调度。
13. 打开 ke/sysproc.c 文件，在第 345 行添加一个断点，按 F5 继续执行，刷新“进程线程”窗口，可以看到已经不包含应用程序进程了，说明应用程序进程已经退出了。
14. 练习。
    (1) 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哪一个，并说明原因。

通过以上的调试，读者对应用程序的退出过程已经有了深入的理解，退出码会返回给操作系统，操作系统就可以获取应用程序的退出码并输出。

通过在 EOS 内核中调试一个应用程序的执行过程，带领读者详细了解了图 1-1 中 EOS 操作系统的所有重要模块，包括进程线程管理模块、存储器管理模块、输入输出管理模块、对象管理模块等，至此读者对 EOS 内核的主要模块已经进行了深入的学习和研究。

实验二 在物理机上运行 EOS 操作系统

实验性质：验证＋设计
建议学时：2 学时
任务数：2 个

一、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。

二、预备知识

在前面的操作系统的启动实验中通过模拟裸机从软盘进行引导加载 EOS 内核，完成了操作系统的启动。但是随着存储技术的发展，存储较小的软盘已经被淘汰，目前物理机上都是通过其他硬件进行引导的。因为采用硬盘引导的方式比较繁琐，需要进行硬盘的安装和配置，所以本实验采用比较简单便捷的 U 盘引导方式，通过修改 BIOS 引导方式，即插即用。而且可以直接将平坦的镜像文件写入 U 盘，实现 EOS 内核引导和加载的过程。

2.1 主引导记录 MBR

系统启动过程中，BIOS 按照“启动顺序”，把控制权转交给排在第一位的储存设备。即根据用户指定的引导顺序从硬盘或是可移动设备中读取启动设备的 MBR，并放入指定的位置（0x7c000）内存中。

这时，计算机读取该设备的第一个扇区，也就是读取最前面的 512 个字节。这最前面的 512 个字节，就叫做“主引导记录”（Master boot record，缩写为 MBR）。MBR 的结构如下图 2-1 所示：

MBR-Stucture

Address		Description		Size in bytes
Hex	Dec			
+000h	+0	Bootstrap code area		446
+1BEh	+446	Partition entry #1	*Partition table* (for primary partitions)	16
+1CEh	+462	Partition entry #2		16
+1DEh	+478	Partition entry #3		16
+1EEh	+494	Partition entry #4		16
+1FEh	+510	55h	*Boot signature*[nb 1]	2
+1FFh	+511	AAh		
		Total size: 446 + 4*16 + 2		**512**

图 2-1：主引导记录 MBR 的结构

“主引导记录”只有 512 个字节，放不了太多东西。它的主要作用是，告诉计算机到硬

盘的哪一个位置去找操作系统。主引导记录由三个部分组成：

（1）第 1-446 字节：调用操作系统的机器码。

（2）第 447-510 字节：分区表（Partition table）。

（3）第 511-512 字节：主引导记录签名（0x55 和 0xAA）。

其中第二部分“分区表”的作用，是将硬盘分成若干个区。硬盘分区有很多好处。考虑到每个区可以安装不同的操作系统，“主引导记录”因此必须知道将控制权转交给哪个区。分区表的长度只有 64 个字节，里面又分成四项，每项 16 个字节。所以，一个硬盘最多只能分四个一级分区，又叫做“主分区”。

每个主分区的 16 个字节，由 6 个部分组成：

（1）第 1 个字节：如果为 0x80，就表示该主分区是激活分区，控制权要转交给这个分区。四个主分区里面只能有一个是激活的。

（2）第 2-4 个字节：主分区第一个扇区的物理位置（柱面、磁头、扇区号等等）。

（3）第 5 个字节：主分区类型。

（4）第 6-8 个字节：主分区最后一个扇区的物理位置。

（5）第 9-12 字节：该主分区第一个扇区的逻辑地址。

（6）第 13-16 字节：主分区的扇区总数。

最后的四个字节（“主分区的扇区总数”），决定了这个主分区的长度。也就是说，一个主分区的扇区总数最多不超过 2 的 32 次方。

2.2 PE 文件概述

EOS 内核 Kernel.dll 文件遵从 PE 文件标准格式，当加载内核程序后，需要取出内核程序的入口点地址，并结合相关地址信息计算出跳转的位置，执行内核程序。想要了解更多的 PE 文件相关内容的读者，可自行阅读 OS Lab 提供帮助文档。打开 OS Lab，选择菜单栏中的“帮助”—》“其他帮助文档”—》“PE 文件格式说明文档”查看更多内容。

PE 文件格式是微软公司为其开发的 Windows 操作系统定义的一种文件组织形式，是 Portable Executable（可移植、可执行）的缩写，它是 32 位 Windows 操作系统中的可执行文件（EXE）和动态连接库文件（DLL）的组织形式。PE 文件格式如图 2-2 所示。

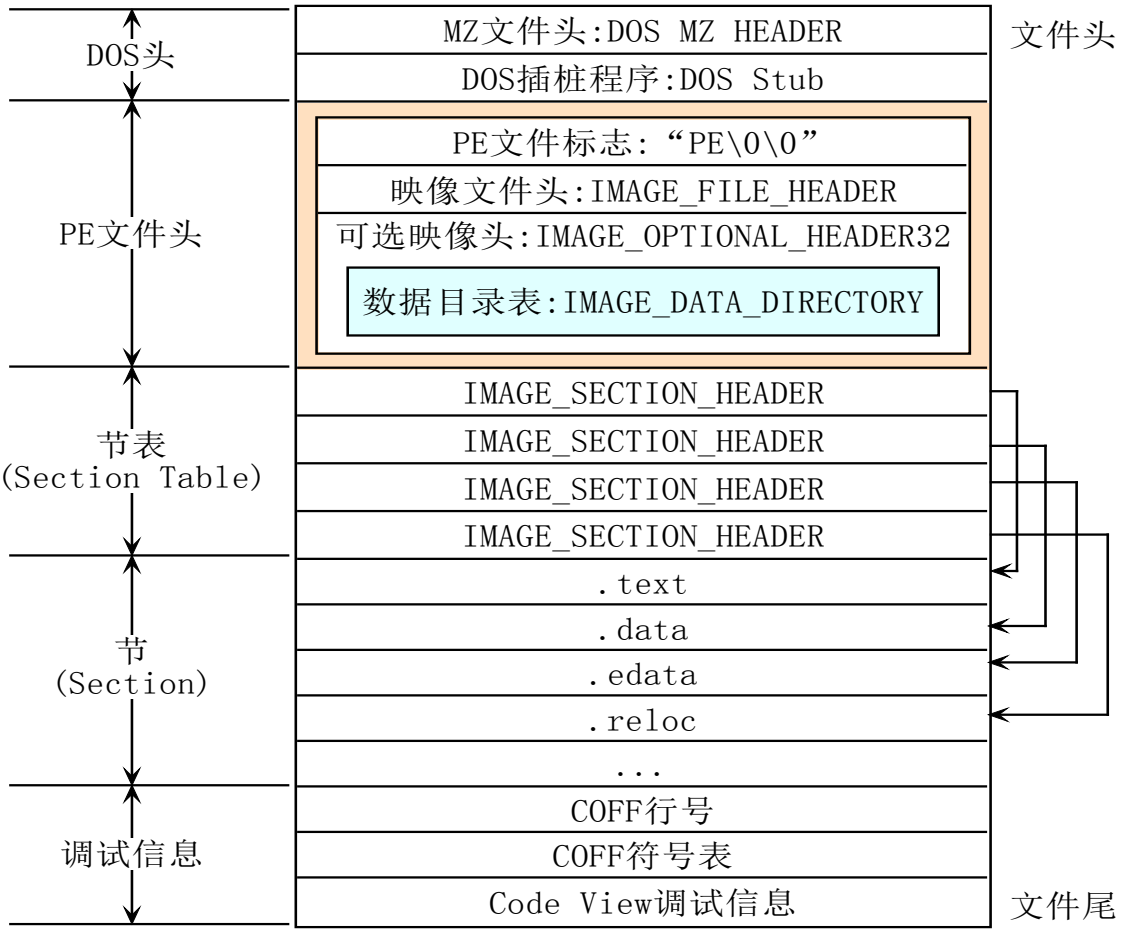


图 2-2：PE 文件格式

在 IMAGE_NT_HEADERS 中包含一个使用结构体 IMAGE_OPTIONAL_HEADER 定义的域 OptionalHeader，其表示的是 PE 文件逻辑分布的基本信息。结构如下所示。

```
typedef struct _IMAGE_OPTIONAL_HEADER {
WORD    Magic;
BYTE    MajorLinkerVersion;
BYTE    MinorLinkerVersion;
DWORD   SizeOfCode;
DWORD   SizeOfInitializedData;
DWORD   SizeOfUninitializedData;
DWORD   AddressOfEntryPoint;
DWORD   BaseOfCode;
DWORD   BaseOfData;
DWORD   ImageBase;
DWORD   SectionAlignment;
DWORD   FileAlignment;
WORD    MajorOperatingSystemVersion;
WORD    MinorOperatingSystemVersion;
WORD    MajorImageVersion;
WORD    MinorImageVersion;
WORD    MajorSubsystemVersion;
```

```
WORD    MinorSubsystemVersion;
DWORD   Win32VersionValue;
DWORD   SizeOfImage;
DWORD   SizeOfHeaders;
DWORD   CheckSum;
WORD    Subsystem;
WORD    DllCharacteristics;
DWORD   SizeOfStackReserve;
DWORD   SizeOfStackCommit;
DWORD   SizeOfHeapReserve;
DWORD   SizeOfHeapCommit;
DWORD   LoaderFlags;
DWORD   NumberOfRvaAndSizes;
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

该结构共 31 个域，一些很关键，另一些不太常用。这里只介绍那些真正有用的域。

字段名称	含义
AddressOfEntryPoint	PE 文件的入口点，通常指向某个函数的起始地址，在 PE 文件加载器将 PE 文件装入内存后，操作系统会从此入口点指定的函数开始执行。
ImageBase	PE 文件的默认加载地址。比如，如果该值是 0x400000，PE 文件加载器将尝试把文件装到虚拟地址空间的 0x400000。若该地址区域已被其他模块占用，那 PE 文件加载器会选用其他空闲地址，此时就必须对 PE 文件进行重定向操作了。
SectionAlignment	内存中节对齐的粒度。例如，如果该值是 4096 (4KB)，那么每个节在内存中的起始地址必须是 4096 的倍数。若第一个节从 0x401000 开始且大小是 10 个字节，则下一个节必须从 0x402000 开始，即使 0x401000 和 0x402000 之间还有很多空间没有被使用。
FileAlignment	文件中节对齐的粒度。例如，如果该值是 512，那么每个节在文件中的偏移必须是 512 的倍数。若第一个节在文件中的偏移量是 0x200，且大小是 10 个字节，则下一个节必须位于偏移 0x400 处，即使文件中在偏移量 0x200 和 0x400 之间还有很多空间没有被使用。
SizeOfImage	内存中整个 PE 映像体的尺寸。它是所有头和节经过节对齐处理后的大小。

SizeOfHeaders	所有的头加上节表的大小，也就等于文件尺寸减去文件中所有节的尺寸。可以以此值作为 PE 文件第一个节的文件偏移量。
SubSystem	用来识别 PE 文件属于哪个子系统。
DataDirectory	IMAGE_DATA_DIRECTORY 结构的数组。每个结构给出一个重要数据结构的 RVA，比如导入表等。

表 2-1： optional header 中的部分域

三、实验内容

3.1 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

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

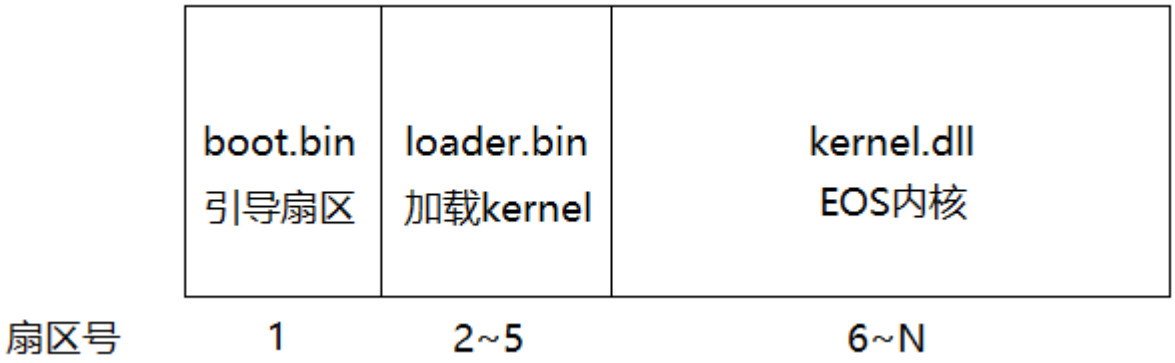
方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/lab02-mission1.git

阅读 boot.asm

不同于之前 EOS 内核中的 boot 程序需要通过 FAT12 文件系统的相关参数才能确定读取内容，本实验是从平坦的磁盘镜像文件读取，扇区位置已经固定，因此直接采用 CHS 模式，通过 int13h 读扇区进行操作。新的镜像文件结构如下图 2-3 所示。



2-3：软盘镜像文件结构

在上图中可以看到，loader.bin 一共占用 4 个扇区，扇区号是 2~5。因此 boot.asm 的功能是：读 loader.bin 共 4 个扇区到 0x1000 处，然后跳转到 0x1000 处执行。

boot.asm 程序中采用了 CHS 寻址方式，因为读扇区的个数较少而不涉及磁道和柱面的变化，因此磁道与柱面均设置为 0。注意应保证 boot.bin 的大小为 512 字节、末尾字节是 0x55aa。

改写 loader.asm

在图 2-3 中可以看到 kernel.dll 在镜像的第 6 扇区开始存储，kernel.dll 文件大小将

近 300KB，为了简化计算，可直接从镜像第 6 扇区开始，连续读 600 个扇区内容到物理内存 0x10000 处即可。

在 loader 程序中，保留获取物理内存大小、检查内核映像的虚拟基址和映像大小以及后面进入到保护模式等代码不变，在获取物理内存大小后面增加读 kernel.dll 到物理内存 0x10000 处的代码。

提示：

（1） 忽略之前与 FAT12 文件系统相关的计算。

（2） 可以使用 loop 循环来读扇区，每次读一个扇区。

（3） 读扇区时，将位置信息中的基址设置为 0x1000，这样每写一个扇区，基址增加 0x20。

（4） 因为本次需要读镜像中的 600 个扇区，需要用到磁道、柱面等参数的变化，因此可以在读扇区时调用 ReadSector。

内核代码调整

将 EOS 内核引导过程调整为 U 盘引导后，就不再需要对内核进行 FAT 文件系统相关内容的初始化，将其注释掉，否则会因为初始化失败而无法进入到内核。需要将文件 ioinit.c 的第 174 至第 226 行代码注释掉。

调整配置

因为本次任务是通过平坦镜像文件进行引导，因此需要调整项目配置，具体的操作步骤如下：

（1）右键点击“项目管理器”中的项目根节点，选择“属性”，将“配置属性”中的“远程目标机”设置为“Bochs Debug”，这样就可以在启动时进行调试。

（2）将“配置属性”—》“常规”—》“调试前命令”中的内容清空。

（3）在“生成事件”—》“生成后事件”—》“命令行”中添加如下命令：

```
echo 正在制作引导软盘映像文件...
mkfloppy.exe "$(OutDir)\boot.bin" "$(OutDir)\loader.bin" "$(TargetPath)" "Floppy.img"
```

命令的作用是调用写镜像工具 mkfloppy.exe 将二进制文件 boot.bin、loader.bin 以及内核程序 kernel.dll 依次写入到镜像 Floppy.img 中。

（4）以上配置完成后点击“应用”按钮并关闭属性页。

启动 Bochs 调试

首先生成项目，OS Lab 会自动将 boot.bin，loader.bin，kernel.dll 这三个文件依次写到平坦镜像文件 Floppy.img。这样启动调试后，Bochs 会加载这个镜像。

生成项目成功后，按 F5 启动调试，这时可直接在 console 窗口中的光标闪烁处输入“c”后按下回车，直接继续执行程序，查看是否正常进入内核。

若继续执行后没有进入到 EOS 内核程序，则可在几个关键位置上添加断点进行调试，查看寄存器状态以及运行的指令，如断点 0x0000:0x7c00、0x0000:0x1000 处查看 boot 和 loader 的运行过程。

3.2 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

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个

人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/lab02-mission2.git

调整配置

在 EOS 内核项目中，Bochs 版本较低，无法设置 U 盘引导，因此需要读者按以下步骤替换掉项目中的 Bochs 配置，具体操作步骤如下：

（1） 清空项目文件夹下的 Bochs 文件夹中的全部内容。

（2） 点击 OS Lab 菜单栏中的“工具”—打 开 OS Lab 安装目录(S)”，将“Bochs-2.6.8”文件夹下的全部内容复制到项目的 Bochs 文件夹下。

（3） 查看 Bochs 配置文件 bochsrc.bxrc，其中设置了 Bochs 内存大小以及引导方式，采用 U 盘引导，并配置引导镜像 Floppy.img。

注意：当从 CodeCode 平台克隆项目并打开后，OS Lab 会自动将默认的 Bochs 文件夹恢复到项目文件夹下，这样会导致之前对 Bochs 的修改被覆盖而无法从 U 盘引导，因此还需要读者再次根据上面的操作步骤替换掉项目中的 Bochs。

因为本次任务是通过 U 盘镜像进行引导，因此需要调整项目配置，具体的操作步骤如下：

（1）右键点击“项目管理器”中的项目根节点，选择“属性”，将“配置属性”中的“远程目标机”设置为“Bochs Debug”，这样就可以在启动时进行调试。

（2）将“配置属性”中的“调试前命令”清空。

（3）在“生成事件”—》“生成后事件”—》“命令行”中添加如下命令：

```
echo 正在制作引导 U 盘映像文件...
mkimage.exe "$(OutDir)\boot.bin" "$(OutDir)\loader.bin" "$(TargetPath)" "Floppy.img"
```

命令的作用是调用写镜像工具 mkimage.exe 将二进制文件 boot.bin、loader.bin 以及内核程序 kernel.dll 依次写入到镜像 Floppy.img 中。

因为当Bochs设置U盘引导后，所用镜像大小必须是 512 字节的整数倍且镜像文件不能太小。因此使用mkimage.exe生成镜像。此工具的功能是将boot.bin，loader.bin，kernel.dll依次写入U盘镜像，且因为kernel.dll大小不是 512 字节的整数倍，需要对其进行填充，保证最后是 512 字节的整数倍，然后在最后再填充上 512 个扇区的内容，保证U盘镜像不会因为太小而无法被 Bochs 用来引导加载内核程序。mkimage 的源代码参见https://www.codecode.net/engintime/os-lab/bonus-lab/mkimage.git。

（4）以上配置完成后点击“应用”按钮并关闭属性页。

阅读 boot.asm

因为这个任务是模拟 U 盘在裸机上引导内核加载，因此在编写 boot.asm 时，需要设置硬盘分区表的内容，参照图 2-1 主引导记录 MBR 的结构对第一个分区进行设置即可。代码已经给出，请仔细阅读并理解其含义。

在 boot.asm 程序中使用 LBA(逻辑块寻址模式)模式寻址，读取 U 盘镜像扇区内容到指定位置。

调整 loader

Loader.asm 本次需要调整的内容是读 kernel.dll 到物理内存 0x10000 处，因为 kernel.dll 大小将近 300K，因此需要从 U 盘镜像读 600 个扇区，这样当扇区数较多时，可

以使用 boot.asm 程序中通过 LBA(逻辑块寻址模式)模式寻址读扇区的操作，即从第 6 个扇区开始,连续读 600 个扇区到 0x10000 处。在设置时,可以将写内存时的基址设置为 0x1000，这样当每读完一个扇区，基址增加 0x20，然后继续读下一个扇区。

内核代码调整

由于键盘设备种类很多，因此初始化键盘设备时，对其 Led 灯的控制可能存在问题，因此可先将文件 keyboard.c 文件第 291 行“更新键盘 LED 指示灯的状态”这部分代码注释，防止初始化时失败而无法进入到内核。

mempool.c 文件的 86 行、87 行注释，防止内存校验报错导致无法进入内核。

ioinit.c 文件的第 174 至第 226 行代码注释，去掉初始化软驱的内容。

启动 Bochs 调试

生成项目，完成 U 盘镜像 Floppy.img 的创建。启动调试，可执行按“c”继续运行，查看运行状态。进入到内核程序后，可输入“pm”命令统计并输出物理存储器信息。

若无法进入内核程序，可对引导过程进行调试，查看 boot、loader 功能是否正确，kernel.dll 是否被写到物理内存 0x10000 处。

若调试正常，但是依然无法进入内核程序，则可以对内核初始化部分添加输出内容，这样启动调试后，可以确定是由内核程序哪一步的初始化发生问题导致无法进入内核。添加输出内容的方法是，模仿 bugcheck.c 中 KeyBugCheck 的写法，自己完成 Kelog，其功能是在 Bochs 的 Display 窗口输出字符串，执行每一步初始化之前调用 Kelog 输出内容，帮助定位发生错误的位置。

3.3 在物理机上使用 U 盘完成 EOS 内核的引导和加载

当通过 U 盘镜像引导加载内核在 Bochs 虚拟机上可以正常运行后，下面准备通过实际的 U 盘引导,在真实的物理机上完成 EOS 内核的加载。无需对项目进行修改,直接使用任务(二)中的 U 盘镜像即可。

修改 BIOS 设置使 PC 机从 U 盘启动

警告！

修改 PC 机的 BIOS 设置可能会造成已经安装在 PC 机上的操作系统（包括 Windows）无法正常启动。所以，建议读者在修改 BIOS 前，一定要先做好下面的准备工作：

- 将重要的个人数据和软件在非系统盘中做好备份。通常 C 盘为系统盘，其他盘为非系统盘。这样，即使需要格式化系统盘并重新安装操作系统，也不会造成数据丢失。
- 在修改 BIOS 前，使用相机将 BIOS 现在的配置情况记录下来。通常情况下，恢复 BIOS 的配置，即可让 PC 机恢复正常工作。
- 为了防止他人在未经自己允许的情况下篡改自己 PC 的 BIOS 设置，可以为 BIOS 设置管理员密码。

免责声明！

以下提供的修改 BIOS 的方法仅供参考，本书不保证其内容的绝对准确性和完整性。读者无论是由于正常修改 BIOS，还是错误修改 BIOS 所导致的数据丢失，隐私泄露，操作系统无法启动，甚至硬件故障等直接、间接或偶然性、继发性的损失或破坏，本书作者绝不承担任何责任。

下面分别对几款 PC 设置 BIOS 从 U 盘启动的方法进行介绍。

1. 2019 年出厂的惠普台式机，Celeron G3900T 处理器。

目前大部分电脑都设置了 UEFI 模式启动，如果我们想让裸机从 U 盘进行引导，需要对 BIOS 的引导进行设置，使其支持旧模式引导（非 UEFI 模式）。具体操作步骤如下：

（1） 重启电脑，在重启的一刻按下“ESC”键进入启动菜单，如图 2-4 所示。

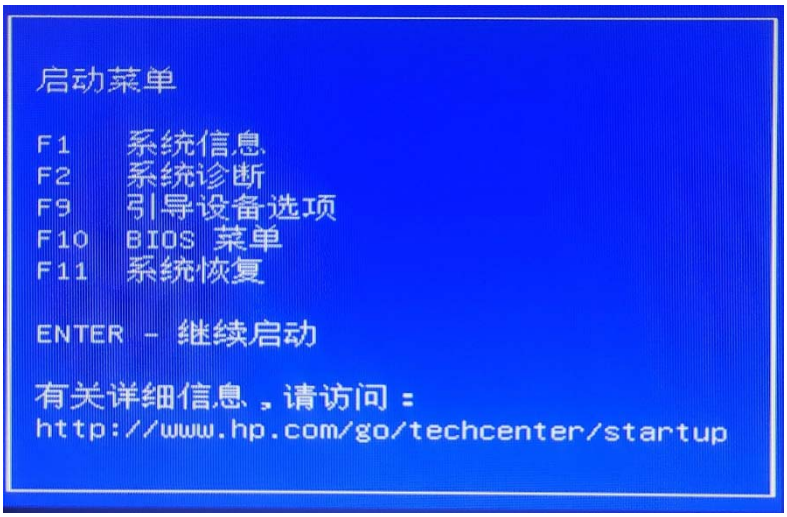


图 2-4：启动菜单

（2） 根据启动菜单的提示，按 F10 进入到 BIOS 菜单。

（3） 首先选择“安全”—》“安全引导配置”选项。如图 2-5 所示。

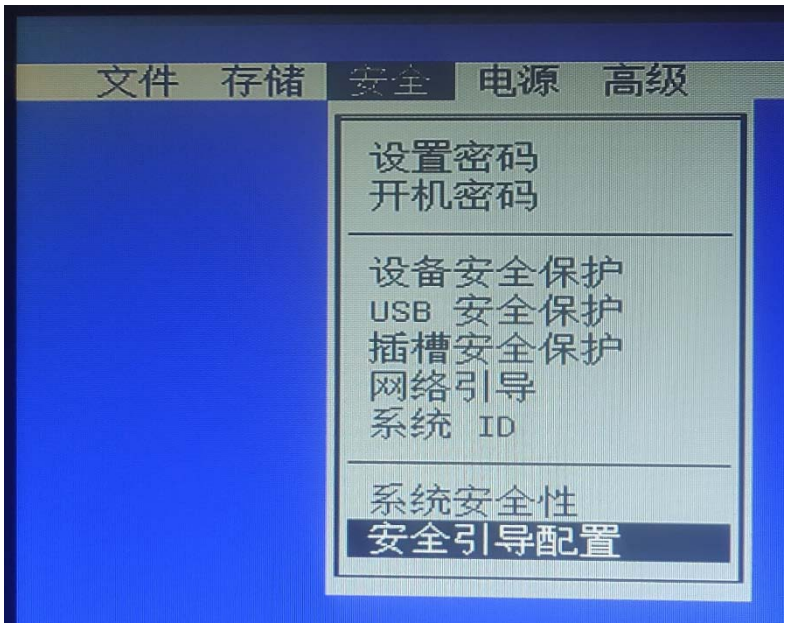


图 2-5：安全引导配置选项

（4） 在配置中将“旧支持”启动，将“安全引导”禁用。

（5） 接下来配置引导顺序，选择“存储”—》“引导顺序”，将 UEFI 引导源禁用，旧引导源启用。如图 2-6。

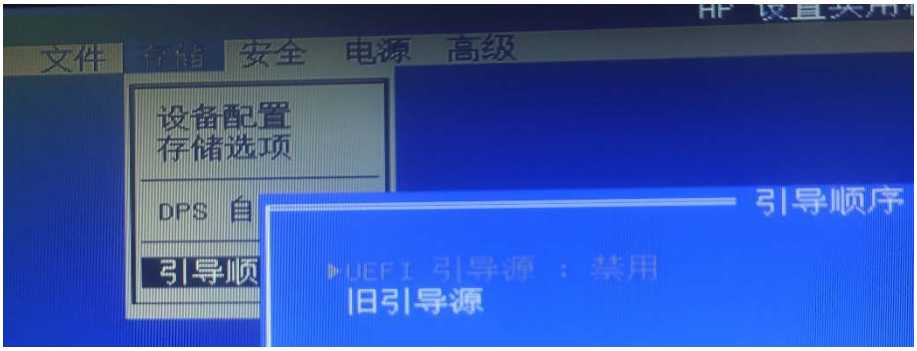


图 2-6：配置引导顺序

（6） 接下来选择“文件”—》“保存修改并退出”。

（7） 系统重新启动后，会要求用户确认修改引导模式，根据提示操作即可。如图 2-7 所示。

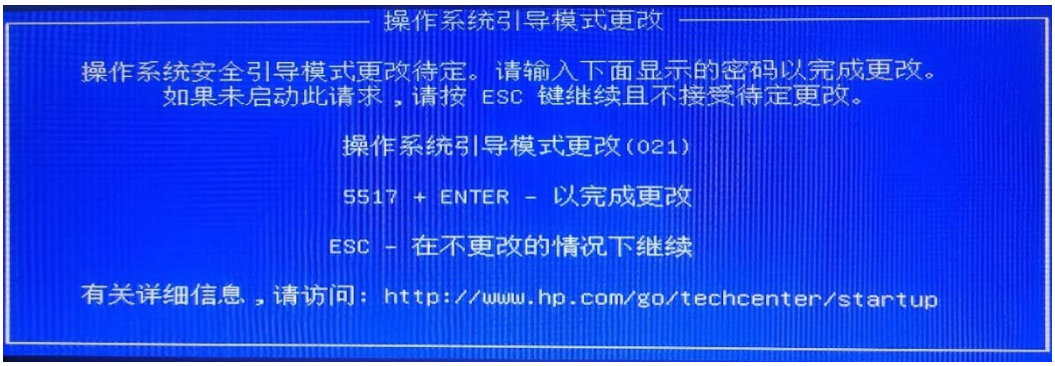


图 2-7：引导模式更改

2. 2013 年出厂的戴尔 XPS14 笔记本电脑，酷睿 i7 处理器。

（1） 开机后按 F2 进入 BIOS 设置界面，在 Advance 页中修改“USB Emulation”的值为 Enabled，使 BIOS 能够将 U 盘模拟成软盘、硬盘等设备。如图 2-8。

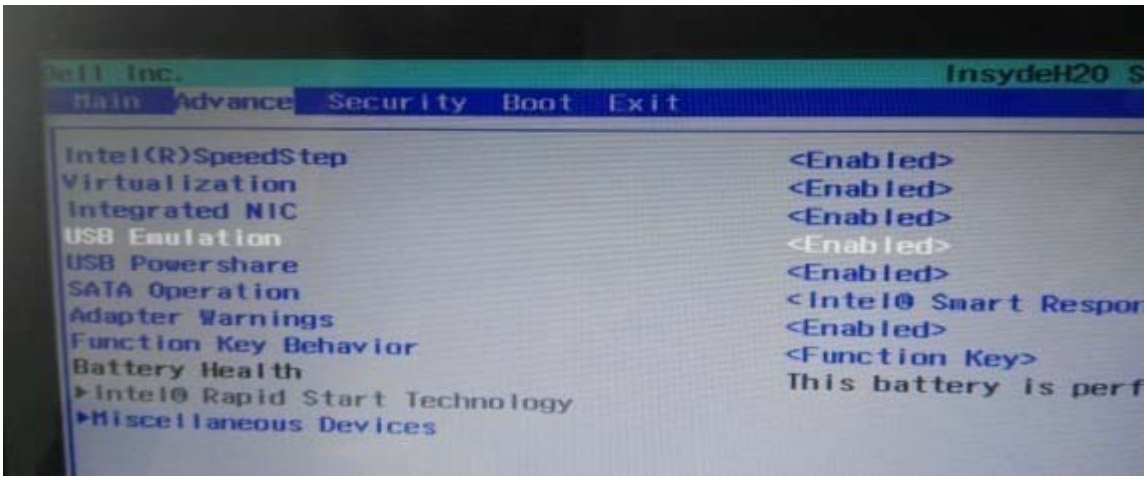


图 2-8：修改 USB Emulation

（2） 修改 Boot 页中的 Boot List Option，将其值设置为“Legacy”，如图 2-9。

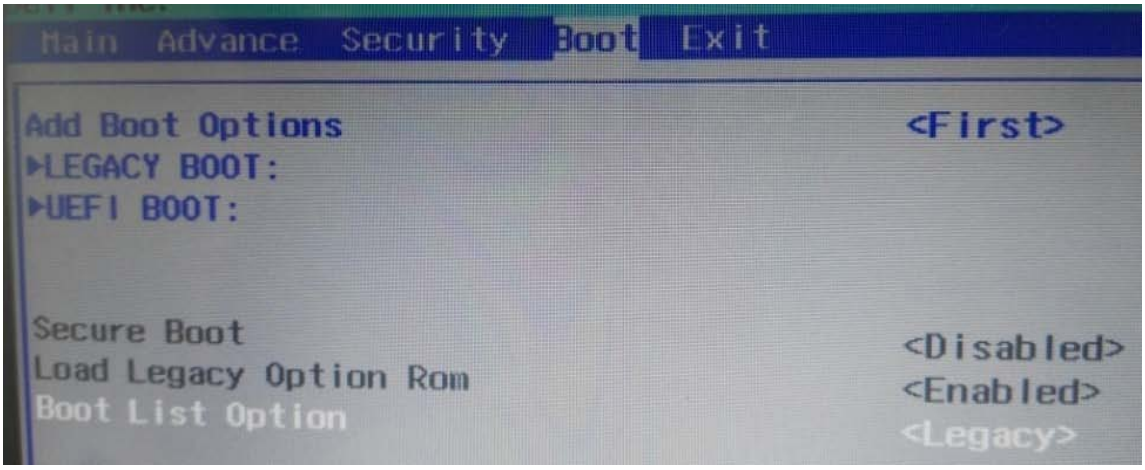


图 2-9：修改 Boot List Option

（3） 保存 BIOS 设置并重启 PC 机。

（4） 在启动时按 F12 进入启动菜单界面，选择“USB Storage Device”从 U 盘启动。如图 2-10 所示。

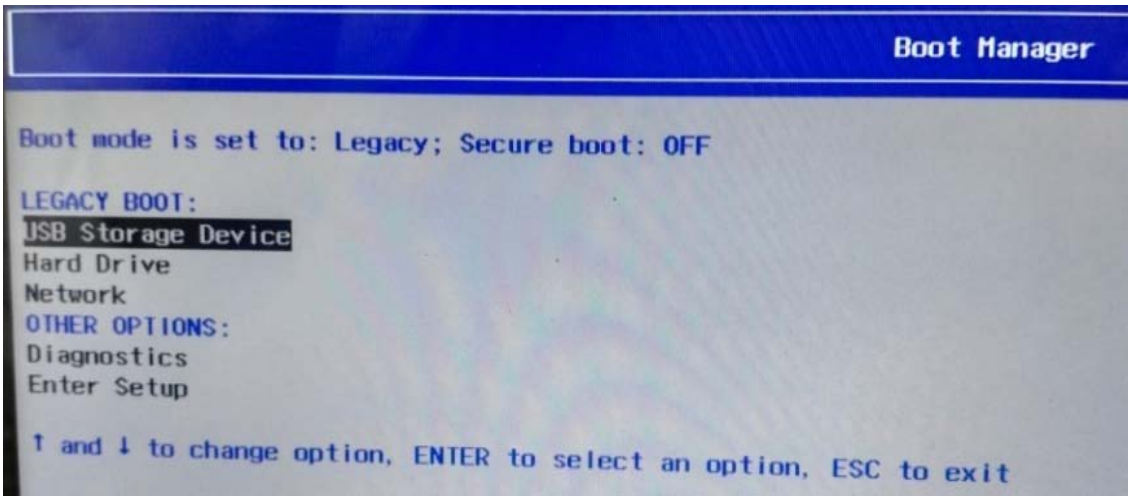


图 2-10：选择从 U 盘启动

以上内容即几款PC机BIOS设置的步骤。电脑品牌不一样，启动菜单的按键可能会不同，可自行搜索对应自己机器的操作按键。BIOS 配置的过程类似，基本选项也是一样的。

将 U 盘镜像文件写入 U 盘

警告！

将 U 盘镜像文件写入 U 盘会造成 U 盘中的所有数据丢失，并无法恢复。所以，建议读者在将 U 盘镜像文件写入 U 盘之前，一定要先将 U 盘中的重要的个人数据和软件在其他存储介质中做好备份。

免责声明！

读者由于将 U 盘镜像文件写入 U 盘所导致的数据丢失，隐私泄露，甚至硬件故障等直接、间接或偶然性、继发性的损失或破坏，本书作者绝不承担任何责任。

在 OS Lab 安装目录下的 bin 文件夹中有一个名为 ddrelease64.exe 的可执行文件，它可在 64 位操作系统上使用（32 位操作系统可使用 OS Lab 安装目录下的 bin 文件夹中的 dd.exe，操作步骤不变），主要功能就是可以将 U 盘镜像文件写入 U 盘。读者可以使用 ddrelease64.exe 将本实验任务中生成的 Floppy.img 文件写入到准备好的一个空白 U 盘中，直接在裸机上引导并加载 EOS 内核程序。具体操作步骤如下：

（1） 为了方便使用，可以首先在 D 盘根目录下创建一个文件夹 D:\usbstick，作为工作目录。

（2） 通过 OS Lab 菜单栏中的“工具”—》“打开 OS Lab 安装目录(S)”来打开 OS Lab 安装目录（一般为 C:\Program Files\Engintime\OS Lab\bin），将其中 bin 文件夹中的 ddrelease64.exe 复制到工作目录中。

（3） 将之前实验中生成的 Floppy.img 文件复制到工作目录中。

（4） 将空白 U 盘插入 PC 机上的 USB 接口。

（5） 使用管理员身份运行命令提示符（CMD）窗口。

（6） 在 Windows 命令提示符窗口中输入命令“d:”回车，然后输入命令“cd usbstick”回车，进入工作目录。

（7） 然后在 Windows 控制台中输入命令“ddrelease64.exe --list”，列出可以读写的设备。在输出结果中找到类似下面的内容，其表示的是一个大小为 16GB 的 U 盘设

备（**注意，一定要有关键字“Removable media”，表示这是一个可移除的设备而不是一个硬盘**）。

```
\\?\Device\Harddisk1\Partition1
  link to \\?\Device\HarddiskVolume12
  Removable media other than floppy. Block size = 512
  size is 15518924800 bytes
```

这个 U 盘所对应的设备名称为“\\?\Device\HarddiskVolume12”。

（8） 在 Windows 命令提示符窗口中输入命令：

```
ddrelease64.exe  if=Floppy.img  of=\\?\Device\HarddiskVolume12  bs=xxxk  --progress
```

其含义是使用工具 ddrelease64.exe，将镜像文件 Floppy.img 写入到 U 盘设备中，注意 bs 应该与 Floppy.img 的大小一致。如镜像大小 559KB，则设置 bs=559k。

（9） 使用 ddrelease64.exe 写镜像到 U 盘后，将 U 盘弹出。

（10） 查看 U 盘是否正确写入镜像。重新插入 U 盘，在 Windows 磁盘管理服务中可 U 盘此时的分区如图 2-11，表示写入镜像成功。

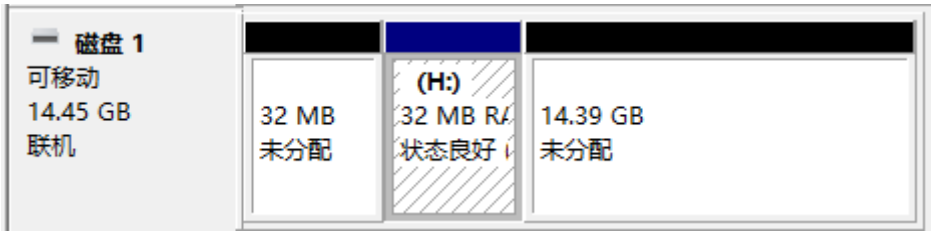


图 2-11：引导 U 盘制作完毕

注意：

- 每次将 U 盘插入 PC 机的 USB 接口后，其设备名称都会发生变化，所以必须重新执行“ddrelease64.exe --list”命令，重新获取 U 盘的设备名称。
- 将U盘镜像文件写入U盘时一定要在命令中使用正确的U盘设备名称(即带有Removable media 关键字的设备)，不要错误的写成硬盘设备的名称。否则，一旦将 U 盘镜像的内容错误的写入了硬盘，会造成整个硬盘中的数据丢失，并无法恢复。
- 保证在 PC 机上只插有一个空白 U 盘，否则在设备列表中会很难分辨哪个才是需要操作的 U 盘设备。

恢复 U 盘

将 U 盘镜像文件写入 U 盘后，默认情况下 U 盘中只有一个大小为 32MB 的分区，并且没有格式化。当 U 盘使用完毕后，如果读者希望将 U 盘并恢复成原来的大小，可以按以下步骤操作：

（1） 将本实验任务的项目中的文件 boot.asm 备份。

（2） 将 boot.asm 文件中硬盘分区表的第一个分区项中的所有字节都设置为 0。目的是清空 U 盘中的有效分区。

（3） 按 F7 生成项目。

（4） 插入 U 盘，使用管理员身份运行 Windows 控制台，输入命令“ddrelease64.exe --list”，列出可以读写的设备。在输出结果中找到类似下面的内容，其表示的是一个大小为 16GB 的U盘设备(**注意，一定要有关键字“Removable media”，表示这是一个可移除的设备而不是一个硬盘**)。

```
\\?\Device\Harddisk1\Partition0
link to \\?\Device\Harddisk1\DR4
Removable media other than floppy. Block size = 512
size is 15728640000 bytes
```

（5） 将生成的 U 盘镜像文件使用 ddrelease64.exe（32 位操作系统使用 dd.exe）写入要恢复的 U 盘。在控制台中输入命令：

```
ddrelease64.exe  if=Floppy.img  of=\\?\Device\Harddisk1\DR4  bs=xxxk  --progress
```

其含义是使用工具 ddrelease64.exe，将镜像文件 Floppy.img 写入到 U 盘设备中，注意 bs 应该与 Floppy.img 的大小一致。如镜像大小 559KB，则设置 bs=559k。

（6） 将 U 盘弹出，重新插上 U 盘。

（7） 查看 U 盘是否正确写入镜像。打开 Windows 7 的磁盘管理工具，此时 U 盘的分区如图 2-12 所示。右键点击 U 盘，对其进行格式化后即可恢复成原来的大小。

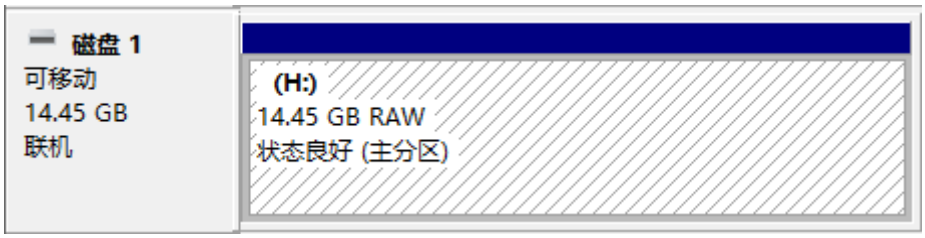


图 2-12：恢复 U 盘

（8） 将步骤（1）中备份的 boot.asm 再恢复到项目中。

在裸机上用 U 盘引导

当配置了 BIOS 从 U 盘启动且向 U 盘中写入了镜像，这时就可以插上 U 盘并启动电脑了，无需任何按键，启动后会自动从 U 盘进行引导，最终进入 EOS 内核程序。

进入到内核程序后，可输入“pm”命令统计并输出物理存储器信息。

3.4 提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建项目，并将项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。

实验三 线程调度算法改进

实验性质：验证＋设计
建议学时：2 学时
任务数：1 个

一、 实验目的

- 实现多级反馈队列调度算法

二、 实验内容

2.1 任务（一）：实现多级反馈队列调度算法

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/Project-Template/eos-kernel.git

完成实验

目前，EOS 操作系统实现了基于优先级的抢先式调度，并且读者已经在实验 6 中实现了时间片的轮转调度算法，可以让同一优先级中的就绪线程轮转执行。但是，如果考虑到有一个运行时间较长的批处理作业优先级比较高，这样较低优先级的短作业就会在一段较长的时间内无法运行，甚至无法及时响应用户。使用多级反馈队列调度算法可以有效解决此问题。

在采用多级反馈队列调度算法的操作系统中，调度算法的实施过程如下：

1. 首先设置多个线程就绪队列，并为各个队列赋予不同的优先级，并将就绪线程放入其优先级对应的就绪队列中。此部分在现有的 EOS 操作系统中已经实现了。
2. 其次，各个队列中赋予线程执行时间片的大小也各不相同。在优先级愈高的队列中，每个线程的初始时间片就规定得愈小。例如，如果规定第一队列（优先级最高）的时间片为 8ms，一般地说，第二队列的时间片要比第一队列的时间片大一个基础时间片的大小，……，第 N+1 队列的时间片比第 N 队列的大一个基础时间片的大小。
3. 第三，当一个新线程进入就绪态后，首先将它放入对应优先级队列的末尾，按 FCFS 原则排队等待调度。当轮到该线程执行时，如能在已分配的时间片内完成，便可准备结束此线程；如果在分配的时间片用完时尚未完成，需要先降低该线程的优先级并增大该线程的时间片，然后调度程序将该线程转入下一个队列的末尾，再同样地按 FCFS 原则等待调度执行；如果它在该队列中运行时用完已分配的时间片后仍未完成，再依法将它转入下一队列。如此下去，当一个长作业从第一队列降到最后一个队列后，就采取按时间片轮转的方式运行，无法再继续降低优先级了。
4. 第四，仅当第一队列空闲时，调度程序才调度第二队列中的线程运行；仅当第 1~

（N- 1）队列均为空时，才会调度第 N 队列中的线程运行。如果处理器正在第 N 队列中为某线程服务时，又有新线程进入优先权较高的队列，则此时新线程将抢占正在运行线程的处理器。

5. 第五，如果有用户交互事件（例如键盘或鼠标操作）发送到了低优先级的线程，需要提升该线程的优先级到其默认的级别，从而可以在抢占处理器后快速响应用户交互事件。

请读者按照上面的说明修改 EOS 操作系统的源代码，实现多级反馈队列调度算法。可以按照下面的步骤逐步完成任务。

1. 实现时间片轮转调度算法。
2. 修改时间片的大小 TICKS_OF_TIME_SLICE 为 100，方便观察执行后的效果。
3. 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。
4. 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。
5. 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。注意，在调用 fprintf 函数格式化字符时，需要在字符串的末尾增加一个空格，否则会导致输出异常。

```
Thread ID:34, Priority:7, InitTicks:200, UsedTicks:299, RemainderTicks:1
Thread ID:35, Priority:7, InitTicks:200, UsedTicks:158, RemainderTicks:142
Thread ID:36, Priority:8, InitTicks:100, UsedTicks:99, RemainderTicks:1
Thread ID:37, Priority:8, InitTicks:100, UsedTicks:99, RemainderTicks:1
Thread ID:38, Priority:8, InitTicks:100, UsedTicks:99, RemainderTicks:1
Thread ID:39, Priority:8, InitTicks:100, UsedTicks:99, RemainderTicks:1
Thread ID:40, Priority:8, InitTicks:100, UsedTicks:99, RemainderTicks:1
Thread ID:41, Priority:8, InitTicks:100, UsedTicks:99, RemainderTicks:1
Thread ID:42, Priority:8, InitTicks:100, UsedTicks:99, RemainderTicks:1
Thread ID:43, Priority:8, InitTicks:100, UsedTicks:99, RemainderTicks:1
```

6. 在实现多级反馈队列调度算法后（注意：数字越大，优先级越高，反之，数字越小，优先级越低），使用实验 6 中提供的“rr”命令，查看各个线程的优先级逐步降低的过程。
7. 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。关于键盘中断相关的内容可以参考实验 12。
8. 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。测试提升线程优先级命令的方法：在控制台窗口 1 执行 rr 时，可以按 Ctrl + F2 切换到控制台窗口 2，然后输入“up 24”命令，按回车执行该命令，按 Ctrl + F1 切换到控制台窗口 1，可以查看 ID 为 24 的线程优先级已

更新为 8，并且初始时间片的大小恢复为基础时间片的大小。由于在使用控制台命令提升线程优先级时按下了空格键，此时查看已提升优先级的线程会有两个。

提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的 EOS 内核项目，并将项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。

实验四 内存管理算法改进

实验性质：验证＋设计
建议学时：2 学时
任务数：3 个

一、 实验目的

- 在 EOS 操作系统中实现边界标识法。
- 在应用程序中调用 malloc 函数和 free 函数分配和释放内存。

二、 实验内容

在 EOS 操作系统内核中，已经通过 mm/mempool.c 文件中的 PoolAllocateMemory 和 PoolFreeMemory 函数实现了动态内存分配，读者可以观察一下 EOS 内核中都有哪些内容是通过动态内存分配来获取内存的，并填写下表。提示：在 EOS 操作系统内核中 PoolAllocateMemory 函数是在 MmAllocateSystemPool 函数中调用的，可以通过查找 MmAllocateSystemPool 函数的调用位置来通过动态内存分配来获取内存的位置。

文件名称	所在代码行	动态分配内存需要实现的功能
ob/obtype.c	74	创建一个对象类型

2.1. 任务（一）：实现控制台命令 dynmm,输出伙伴算法管理的内存数据。

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的

URL 为
https://www.codecode.net/engintime/os-lab/Project-Template/eos-kernel.git

完成实验

EOS 操作系统内核使用伙伴算法对动态分配的内存进行管理，EOS 操作系统内核与实验 9 中实现的伙伴算法函数的对应关系如下表，读者可以仿照实验 9 提供的模式，在 EOS 内核中添加一个控制台命令“dynmm”，将伙伴算法管理的内存数据(包括已使用内存块的数量与已使用内存块总的大小，空闲的内存块数量与空闲内存块总的大小)打印输出到屏幕上。

EOS 操作系统的伙伴算法	实验 9 的伙伴算法
PoolAllocateMemory	AllocBuddy
PoolFreeMemory	Reclaim
PoolInitialize	main 函数的前半部分
struct _MEM_BLOCK	struct _Node

```
>dynmm
freeMemBlockCount = 12, memorySize = 3085952
usedMemBlockCount = 115, memorySize = 23648
```

提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的 EOS 内核项目，并将项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。

2.2. 任务（二）：将 EOS 内核的伙伴算法替换为边界标识法。

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为
https://www.codecode.net/engintime/os-lab/Project-Template/eos-kernel.git

实现边界标识法的内存分配算法

待读者对 EOS 操作系统内核中的动态内存分配机制有一个深入的了解后，可以尝试将 EOS 操作系统中的伙伴算法修改为边界标识法，EOS 操作系统内核实现的伙伴算法与实验 9 中实现的边界标识法的函数对应关系如下表，读者只需要替换对应函数中的代码，并对代码做适当的调整即可在 EOS 操作系统内核中实现边界标识法。这里给出一些提示信息：

EOS 操作系统的伙伴算法	实验 9 的边界标识法
PoolAllocateMemory	allocBoundTag
PoolFreeMemory	reclaimBoundTag
PoolInitialize	initSpace
struct _MEM_BLOCK	struct Bound

- 可以先实现边界标识法中的内存初始化和内存分配算法，回收算法可以暂时不实现。
- 在实现边界标识法的内存分配算法时，需要注意，实际分配的内存大小为：定义的结构体的大小与需要分配的内存大小之和，返回的地址为：内存块的起始地址与定义的结构体的大小之和，如果直接返回内存块的起始地址，在写内存时会将结构体信息覆盖，破坏内存块的结构数据。
- 在实现分配算法后，尝试启动内核，如果正常启动，可以输入 pt 命令和执行应用程序 Hello.exe 进行测试，如果可以正常执行，说明已实现的内存分配算法可以正常使用。

实现控制台命令“dynmm”的相关代码，将边界标识法管理的内存数据打印输出到屏幕上。

通过实现控制台命令“dynmm”的相关代码，将边界标识法管理的内存数据打印输出到屏幕上。输出的内容可以参考任务一。

实现边界标识法的内存回收算法。

这里给出一些提示信息：

- 在 EOS 内核中，实现边界标识法的内存回收算法。
- 实现后，尝试启动内核，确保可以正常启动，并且可以执行应用程序和控制台命令。
- 在控制台中输入 dynmm 命令，查看空闲块的数量和已使用块的数量，在回收内存后，已使用块的数量和任务三相比已经减少。

提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的 EOS 内核项目，并将项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。

2.3. 任务（三）：在 EOS 应用程序中实现 malloc、calloc、realloc 和 free 函数

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

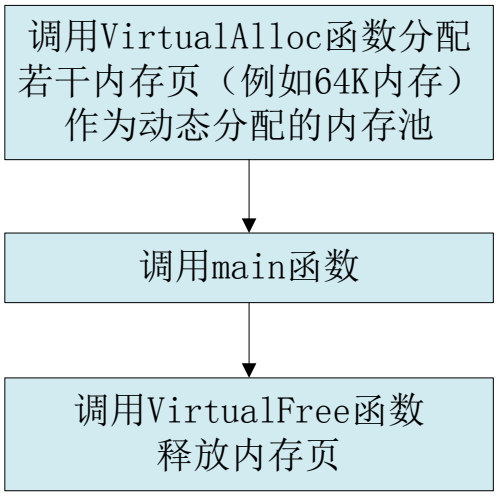
https://www.codecode.net/engintime/os-lab/Project-Template/eos-app.git

完成实验

在 EOS 应用程序提供的 C 运行时库中，还没有实现动态内存分配，也就是说无法在 EOS 应用程序的 C 源代码中调用 malloc、calloc、realloc 和 free 函数。读者可以尝试在 EOS 应用程序的 crt/src/stdlib.c 文件中实现与 C 运行时标准库一致的 malloc、calloc、realloc 和 free 函数。这里给出一些提示信息：

- 首先需要修改 EOS 应用程序 crt/src/crt0.c 文件中的标准库初始化函数_start，在其中调用 EOS API 函数 VirtualAlloc 分配若干个内存页作为动态分配的内存池，并将其基址映射到进程用户空间（低 2G）中的合适位置。注意分配内存页的代码

要写在调用 main 函数执行用户编写的功能之前，之后还需要调用 VirtualFree 函数释放内存页。可以参考下面的流程图。



- 在 crt/src/stdlib.c 文件中定义必要的数据结构，实现 malloc、calloc、realloc 和 free 函数。可以使用边界标识法，也可以使用伙伴算法等。
- 在 EOS 应用程序的 main 函数中尝试调用 malloc、calloc、realloc 函数分配内存，然后将两个整数写入已分配的内存中，求两个整数的和，并将求得的结果写入到已分配的内存中，在对内存进行读写访问后调用 free 函数释放内存，确保动态内存分配功能可以正常工作。在 EOS 应用程序的 main 函数中调用 malloc 和 free 函数并读写内存的测试代码，如下表，并以伙伴算法为例输出应用程序的内存使用数据。

```
int main(int argc, char* argv[])
{
    printf("================malloc memory================\n");

    PINT pValue = (PINT)malloc(20);
    if(NULL == pValue) {
        printf("PValue Allocate virtual memory failed!\n\n");
        return;
    }

    PINT pValue2 = (PINT)malloc(25);
    if(NULL == pValue2) {
        printf("PValue2 Allocate virtual memory failed!\n\n");
        return;
    }

    PINT pValue3 = (PINT)malloc(sizeof(INT));
    if(NULL == pValue2) {
        printf("PValue2 Allocate virtual memory failed!\n\n");
        return;
    }
    *pValue3 = 5;
```

```
    PINT pValue4 = (PINT)malloc(sizeof(INT));
    if(NULL == pValue2) {
        printf("PValue2 Allocate virtual memory failed!\n\n");
        return;
    }

    *pValue4 = 8;
    *pValue4 += *pValue3;
    printf("%d + %d = %d\n", *pValue3, *pValue4, *pValue3 + *pValue4);

    print_used();   // 输出内存的使用情况

    printf("================free memory================\n");
    free((PVOID)pValue);

    print_used();

    return 0;
}
```

```
Welcome to EOS shell
>Autorun A:\eosapp.exe
================malloc memory================
5 + 13 = 18
UsedSpace:
UsedBlockNum    UsedBlockBeginAddr      BlockSize       BlockTag(0:Free 1:Used)
    0               0x10000             32              1
    1               0x10020             32              1
    2               0x10040             8192            1
    3               0x10048             8               1

================free memory================
UsedSpace:
UsedBlockNum    UsedBlockBeginAddr      BlockSize       BlockTag(0:Free 1:Used)
    1               0x10020             32              1
    2               0x10040             8192            1
    3               0x10048             8               1
```

- 还可以进一步修改 EOS 应用程序 crt/src/crt0.c 文件中的标准库初始化函数 _start，在其中调用 EOS API 函数 VirtualAlloc 动态分配若干个内存页作为动态分配的内存池，如果动态分配的内存池内存不足之后，还可以继续调用 VirtualAlloc 动态分配若干个内存页作为动态分配的内存池，直到操作系统没有可以分配的内存页。

读者完成以上练习后，可以思考以下问题：

- 如果 EOS 应用程序只分配了内存，而未释放内存，在应用程序退出后，EOS 操作系统会回收这部分内存吗？
- 应用程序进程异常结束，杀死应用程序进程后，EOS 操作系统如何回收内存？如果 EOS 操作系统不回收这部分内存会有什么后果？
- VirtualAlloc 是在低 2G 还是在高 2G 的虚拟地址空间分配的内存？VirtualAlloc 如果在低 2G 虚拟地址空间分配的内存，可以尝试在 malloc 中直接调用 VirtualAlloc 分配内存，在 free 中调用 VirtualFree 函数释放内存。

- 由于在 malloc 中直接调用 VirtualAlloc 分配内存，会使在应用程序中每次调用 malloc 时，都会进入内核，不利于应用程序自己管理内存空间，并会导致执行效率低下。如果直接在 malloc 函数中实现动态内存分配算法（边界标识法或伙伴算法），每次调用 malloc 都会从应用层调用，执行效率会高很多。通过分别执行程序比较调用以上两种方法的执行效率。

提示：如果需要将地址和整数进行相加得到偏移地址，需要先将地址转换为无符号长整型（ULONG）后，再进行运算。

提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的 EOS 应用程序项目，并将项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。

实验五 文件系统改进

实验性质：验证＋设计
建议学时：2 学时
任务数：4 个

一、实验目的

- 掌握使用 PIO 方式读写硬盘的方法。
- 掌握文件系统 MINIX 1.0 硬盘管理方式。

二、预备知识

2.1 硬盘的 PIO（Programming Input/Output Model）模式

PIO（Programming input/output Model）模式是一种完全通过 CPU 执行 I/O 端口指令来进行数据读写的模式。PIO 模式是最早的硬盘数据传输模式，但其数据传输速率低下，CPU 占有率也很高，大量传输数据时会因为占用过多的 CPU 资源而导致系统停顿，无法进行其他操作。但是 PIO 模式的兼容性要优于 DMA 模式，一些较新的 ATA 控制器为了保持兼容性，还都提供了 PIO 模式，并且编程方法与较早的 ATA 控制器保持一致。

当使用 PIO 模式读写硬盘扇区时，需要直接对 IDE 设备的 I/O 端口命令模块和控制模块进行设置。命令模块是一个大小为 8 字节的地址空间，控制模块是一个大小为 4 字节的地址空间。I/O 地址的高端 16 位被作为全 0 译码。默认情况下命令模块和控制模块的起始地址如下：

- 初级命令模块的起始 I/O 地址：01F0H
- 初级控制模块的起始 I/O 地址：03F4H
- 次级命令模块的起始 I/O 地址：0170H
- 次级控制模块的起始 I/O 地址：0374H

图 5-1 和图 5-2 中列出了这些寄存器的功能和操作。

I/O偏移量	寄存器功能（读/写）	访问操作
00H	数据	读/写
01H	出错/特征	读/写
02H	扇区计数	读/写
03H	扇区号	读/写
04H	硬盘低端柱面	读/写
05H	硬盘高端柱面	读/写
06H	驱动器/磁头	读/写
07H	状态/命令	读/写

图 5-1：IDE I/O 端口定义：命令模块

I/O偏移量	寄存器功能（读/写）	访问操作
00H	保留	保留
01H	保留	保留
02H	Alt状态/驱动器控制	读/写
03H	朝着ISA（软盘）	读/写

图 5-2：IDE I/O 端口定义：控制模块

2.2 MINIX 1.0 文件系统

Linux 0.11操作系统启动时需要加载一个根目录，此根目录使用的是MINIX 1.0文件系统，其保存在硬盘的第一个分区中。Linux 0.11操作系统将硬盘上的两个连续的物理扇区（大小为512字节）做为一个物理盘块（大小为1024字节），而且物理盘块是从1开始计数的。硬盘上的第一个物理盘块是主引导记录块（MBR），读者已经通过实验二了解到，Linux 0.11并未使用硬盘上的主引导记录块进行引导，而是使用软盘A上的引导扇区进行引导的。在硬盘的主引导记录块中除了没有用到的引导程序外，在其包含的第一个物理扇区的尾部（引导标识0x55aa的前面）是一个64字节的分区表（典型的IBM Partition Table），其中每个分区表项占用16字节，共有4个分区表项。每个分区表项都定义了一个分区的起始物理盘块号和分区大小（占用的物理盘块数量）等信息。硬盘上的物理盘块可以按照图5-3所示进行划分。

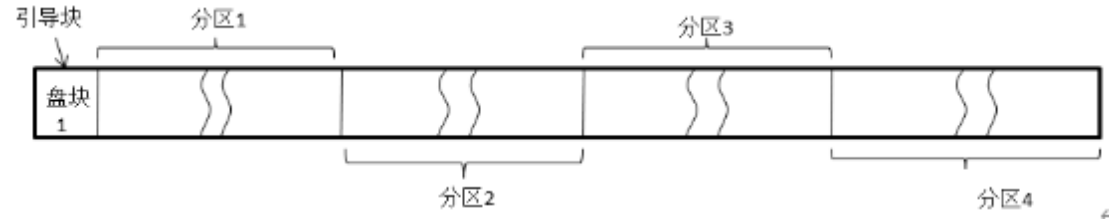


图 5-3：硬盘上物理盘块的划分

Linux 0.11使用的硬盘只包含了两个分区，在第一个分区中提供了Linux 0.11的根目录，并使用MINIX 1.0文件系统进行管理，第二个分区没有用到，内容为空。需要强调的是，在MINIX 1.0文件系统管理的硬盘分区中就不再使用物理盘块作为划分的单位了，而是将一个物理盘块作为一个逻辑块来使用，并且逻辑块号是从0开始计数的。

接下来，重点了解一下MINIX 1.0文件系统是如何管理硬盘上的第一个分区的。第一个分区包含了引导块（占用逻辑块0）、超级块（占用逻辑块1）、i节点位图（占用逻辑块2-4）、逻辑块位图（占用逻辑块5-12），以 及i节点和数据区，如图5-4所示。

图 5-4：MINIX 1.0 文件系统所管理的分区 1 的布局

分区 1 开始处的引导块同样没有被使用。其后的超级块用于存放 MINIX 1.0 文件系统的结构信息，主要用于说明各部分的起始逻辑块号和大小（包含的逻辑块数量）。超级块中各个字段的含义在表 5-1 中进行了说明。

字段名称	数据类型	说明
s_inodes	short	分区中的 i 节点总数。
s_nzones	short	分区包含的逻辑块总数。
s_imap_blocks	short	i 节点位图占用的逻辑块数。
s_zmap_blocks	short	逻辑块位图占用的逻辑块数。
s_firstdatazone	short	数据区占用的第一个逻辑块的块号。
s_log_zine_size	short	log2(逻辑块包含的物理块数量)。MINIX 1.0 文件系统的逻辑块包含一个物理块，所以其值为 0。
s_max_size	long	文件最大长度，以字节为单位。
s_magic	short	文件系统魔数，用以指明文件系统的类型。MINIX 1.0 文件系统的魔数是 0x137f。

表 5-1：MINIX 1.0 的超级块的结构

在超级块的后面是占用了 3 个逻辑块的 i 节点位图，其中的每一位用于说明对应的 i 节点是否被使用。位的值为 0 时，表示其对应的 i 节点未被使用；位的值为 1 时，表示其对应的 i 节点已经被使用。

在 i 节点位图的后面是占用了 8 个逻辑块的逻辑块位图，其中的每一位用于说明数据区中对应的逻辑块是否被使用。除第 1 位(位 0)未被使用外，逻辑块位图中每个位依次代表数据区中的一个逻辑块。因此，逻辑块位图的第 2 位（位 1）代表数据区中第一个逻辑块，第 3 位 (位 2) 代表数据区中的第二个逻辑块，依此类推。当数据区中的一个逻辑块被占用时，逻辑块位图中的对应位被置为 1，否则被置为 0。

在逻辑块位图的后面是 i 节点部分，其中存放着 MINIX 1.0 文件系统中文件或目录的索引节点（简称 i 节点)。注意，i 节点是从 1 开始计数的。每个文件或目录都有一个 i 节点，每个 i 节点结构中存放着对应文件或目录的相关信息，如文件宿主的 id(uid)、文件所属组 id(gid)、文件长度、访问修改时间以及文件数据在数据区中的位置等。i 节点共包含 32 个字节，其结构如表 5-2 所示。

字段名称	数据类型	说明
i_mode	short	文件的类型和属性(rwx 位)
i_uid	short	文件宿主的用户 id
i_size	long	文件长度(以字节为单位)
i_mtime	long	文件的修改时间(从 1970 年 1 月 1 日 0 时起，以秒为单位)
i_gid	char	文件宿主的 id
i_nlinks	char	链接数(有多少个文件目录项指向该 i 节点)
i_zone[9]	short	文件所占用的数据区中的逻辑块号数组。其中 zone[0]-zone[6]是直接块号；zone[7]是一次间接块号；zone[8]是二次(双重)间接块号。

表 5-2：MINIX 1.0 文件系统的 i 节点的结构

i_mode 字段用来保存文件的类型和访问权限属性。其位 15~12 用于保存文件类型，位 11~9 保存执行文件时设置的信息，位 8~0 表示文件的访问权限。如图 5-5 所示。 i_zone[] 数组用于存放 i 节点在数据区中对应的逻辑块号。i_zone[0]到 i_zone[6]用于存放文件开始的 7 个块号，称为直接块。例如，若文件长度小于等于 7KB，则根据其 i 节点的 i_zone[0]到 i_zone[6]可以很快找到文件数据所占用的逻辑块。若文件大一些，就需要用到一次间接块 i_zone[7]了，其对应的逻辑块中的数据又是一组逻辑块号，所以称之为一次间接块。对于 MINIX 1.0 文件系统来说，i_zone[7]对应的逻辑块中可以存放 512 个逻辑块号，因此可

以寻址 512 个逻辑块。若一次间接块提供的逻辑块仍然无法存储文件的全部数据，则需要使用二次间接块 i_zone[8]了，其可以寻址 512*512 个逻辑块。参见图 5-6。

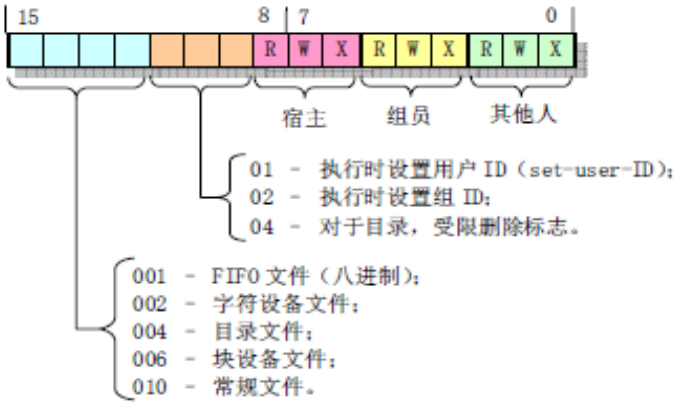


图 5-5：i 节点属性字段内容

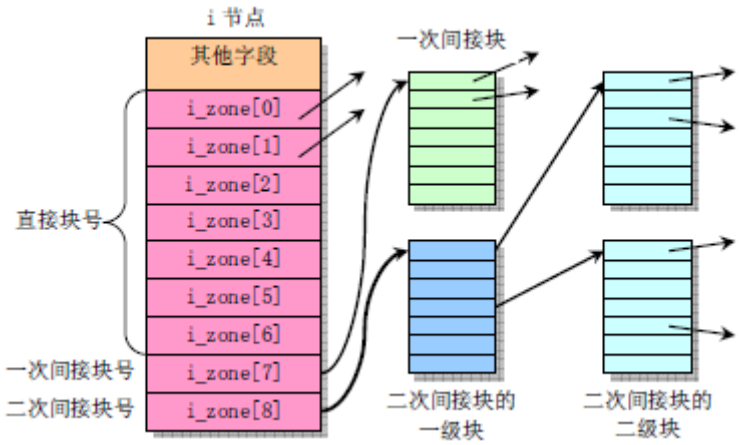


图 5-6：i 节点的逻辑块数组的功能

如果 i 节点对应的是一个文件，那么这个文件的内容（例如文本文件中的文本数据）就会保存在由 i_zone 数组指定的若干个逻辑块中。但是，如果 i 节点对应的是一个目录，这个目录又包含了若干个子文件或子目录，情况就会稍微复杂一些。如果 i 节点对应的是一个目录，则在该 i 节点的 i_zone 数组对应的逻辑块中会保存所有子文件和子目录的目录项信息。MINIX 1.0 文件系统的目录项长度为 16 字节，开始的 2 个字节指定了子文件或子目录对应的 i 节点号，接下来的 14 个字节用于保存子文件或子目录的名称。由于整个目录项的长度是 16 个字节，因此一个逻辑块可以存放 1024/16 = 64 个目录项。子文件和子目录的其他信息被保存在由 i 节点号指定的 i 节点结构中。所以，当需要访问一个文件时，需要根据文件的全路径（例如/bin/sh）从根节点找到子目录的 i 节点，然后再从子目录的 i 节点找到文件的 i 节点。

详细内容请读者阅读 OS Lab 提供的帮助文档《Linux 内核完全注释》的第 12 章。

三、实验内容

3.1 任务（一）PIO 方式读硬盘

准备实验

在 OS Lab 模板中提供了 PIO 方式读硬盘 0 号扇区到缓冲区，并且将扇区的内容进行显示的示例代码。请读者克隆模板项目到本地，对程序进行调试，掌握 PIO 方式读硬盘的操作，实验模板的 URL 为：

https://www.codecode.net/engintime/os-lab/bonus-lab/pio-read.git。

调试并验证程序

（1） 打开“准备实验”中克隆的项目。

（2） 复制硬盘镜像文件到项目中。点击“工具”—》“打开 OS Lab 安装目录(S)”，将文件夹“Linux011-harddisk”中的硬盘镜像文件 harddisk.img 复制粘贴到步骤（1）中创建的项目根目录下。

（3） 配置 Bochs。打开 Bochs 配置文件 bochsrc.bxrc，将其中 ata0-master 所在行替换为：

```
ata0-master: type=disk, path="..\harddisk.img", mode=flat, cylinders=410, heads=16, spt=38
```

这样就为 Bochs 配置了将 harddisk.img 作为硬盘挂载。

注意：当从 CodeCode 平台克隆项目并打开后，OS Lab 会自动将默认的 Bochs 文件夹恢复到项目文件夹下，这样会导致之前对 Bochs 的修改被覆盖而无法挂载硬盘，因此还需要读者再次根据上面的操作步骤修改 Bochs 配置并复制镜像文件 harddisk.img。

（4） 生成项目并开始执行不调试，在 Bochs 的显示器窗口会打印输出硬盘 0 号扇区的 512 字节信息。

（5） 打开项目文件夹下的硬盘镜像 harddisk.img 文件，其 0 号扇区的内容应该与步骤（4）中输出的内容一致。推荐使用“UltraEdit 文本/十六进制编辑器”查看镜像内容。

（6） 因为本项目启动时默认的引导方式是 floppy，因此启动 Bochs 时无需选择引导方式。

3.2 任务（二）PIO 方式写硬盘

准备实验

在 OS Lab 模板中提供了 PIO 方式写硬盘 0 号扇区的示例代码。请读者克隆模板项目到本地，对程序进行调试，掌握 PIO 方式读硬盘的操作，实验模板的 URL 为：

https://www.codecode.net/engintime/os-lab/bonus-lab/pio-write.git。

调试并验证程序

（1） 打开“准备实验”中克隆的项目。

（2） 复制硬盘镜像文件到项目中。点击“工具”—》“打开 OS Lab 安装目录(S)”，将文件夹“Linux011-harddisk”中的硬盘镜像文件 harddisk.img 复制粘贴到步骤（1）中创建的项目根目录下。

（3） 配置 Bochs。打开 Bochs 配置文件 bochsrc.bxrc，将其中 ata0-master 所在行替换为：

```
ata0-master: type=disk, path="..\harddisk.img", mode=flat, cylinders=410, heads=16, spt=38
```

这样就为 Bochs 配置了将 harddisk.img 作为硬盘挂载。

（4） 生成项目并开始执行不调试，程序的功能是将缓冲区内 512 字节的字符“0”写到硬盘的 0 号扇区，执行完毕后进入死循环。

（5） 打开项目文件夹下的硬盘镜像 harddisk.img 文件，其 0 号扇区的内容此时应该

已经被修改为 0。推荐使用“UltraEdit 文本/十六进制编辑器”查看镜像内容。

（6） 因为本项目启动时默认的引导方式是 floppy，因此启动 Bochs 时无需选择引导方式。

3.3 任务（三）模拟访问 MINIX 1.0 文件系统

准备实验

提供了以模拟的方式来访问硬盘上的 MINIX 1.0 文件系统的示例代码，即使用 C 语言编写一个 Windows 控制台程序，直接访问硬盘镜像文件 harddisk.img 中的 MINIX 1.0 文件系统。请读者克隆模板项目到本地，对程序进行调试，掌握 MINIX 1.0 文件系统的工作原理，实验模板的 URL 为：

https://www.codecode.net/engintime/os-lab/bonus-lab/minix1.0.git。

代码说明

在“准备实验”中克隆的项目的源代码文件 console.c 中，实现了打印输出 MINIX 1.0 文件系统的目录树的功能，仔细阅读其中的源代码，并着重理解下面的内容：

- 在源代码文件的开始位置定义了分区表项、超级块、i 节点和目录项的结构体，读者需要理解其中每个字段的意义。
- get_physical_block 函数的作用是将一个物理块的内容读取到缓冲区中。get_partition_logical_block 函数的作用是将第一个分区中的一个逻辑块的内容读取到缓冲区中。需要注意的是，物理块号是从 1 开始计数的，而逻辑块号是从 0 开始计数的。这两个函数实现了对硬盘镜像文件物理层和逻辑层的分层访问，从而可以模拟出访问硬盘的过程。
- load_inode_bitmap 函数将硬盘镜像文件中的整个 i 节点位图都读入到了内存中。is_inode_valid 函数根据 i 节点位图中的内容判断一个 i 节点是否有效。需要注意的是，i 节点是从 1 开始计数的。get_inode 函数根据 i 结点的 id 读取相对应的 i 结点。
- print_inode 函数递归打印目录树。首先读取 i 节点位图的第一位，并找到根节点。然后判断 i 节点是否为目录，若是目录，则打印目录名(名称前加 Tab 键是为了有树的层次感)，然后递归打印其子目录和子文件；如果是常规文件则直接打印文件名。

调试并验证程序

（1） 打开“准备实验”中克隆的项目。

（2） 复制硬盘镜像文件到项目中。点击“工具”—》“打开 OS Lab 安装目录(S)”，将文件夹“Linux011-harddisk”中的硬盘镜像文件 harddisk.img 复制粘贴到步骤（1）中创建的项目根目录下。

（3） 按 F7 生成项目，确保没有语法错误。

（4） 打开本项目的文件夹，将 Debug 文件夹下的 console.exe 文件复制到项目的根文件夹下。

（5） 启动 Windows 控制台，进入步骤（1）中创建项目的根目录，然后执行命令：“console.exe > a.txt”。此命令会将应用程序打印输出的目录树重定向到文本文件 a.txt 中。

（6） 打开 a.txt 文件查看 MINIX 1.0 文件系统的目录树。

注意：如果是在 PowerShell 控制台中，请输入命令“.\console.exe > a.txt”。

3.4 任务（四）使 EOS 内核可以访问硬盘中的 MINIX 1.0 文件系统

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，实验模板的 URL 为 https://www.codecode.net/engintime/os-lab/Project-Template/eos-kernel.git

按以下步骤准备项目：

（1） 打开“准备实验”中克隆的项目。

（2） 复制硬盘镜像文件到项目中。点击“工具”—》“打开 OS Lab 安装目录(S)”，将文件夹“Linux011-harddisk”中的硬盘镜像文件 harddisk.img 复制粘贴到步骤（1）中创建的项目根目录下。

（3） 配置 Bochs。打开 Bochs 配置文件 bochsrc.bxrc，将其中 ata0-master 所在行替换为:

```
ata0-master: type=disk, path="..\harddisk.img", mode=flat, cylinders=410, heads=16, spt=38
```

这样就为 Bochs 配置了将 harddisk.img 作为硬盘挂载。

设计代码

本实验任务的目标是可以实现进入硬盘根目录并且输入“ls”命令可以查看目录中文件的功能。

在前面的三个实验任务中，读者已经了解了 PIO 模式读写硬盘的方法和 MINIX 1.0 文件系统的原理，下面的工作是结合 EOS 内核的代码规则，在 EOS 内核中实现 MINIX 1.0 文件系统的支持，并且可以对文件进行读写操作。需要注意以下几点：

（1） 在改写代码前，可先阅读《EOS 内核实验指导》的参考内容第 7 章 I/O 管理，在其中说明了 EOS 内核对驱动程序以及磁盘读写的原理和过程。

（2） EOS 中的 I/O 管理基于对象管理模块提供的对象管理功能，需要对应的驱动程序对象和设备对象。

（3） EOS 应用程序在调用 CreateFile 函数打开文件后，就可以通过调用 EOS API 函数 ReadFile 和 WriteFile 来读写文件了。

（4） 系统初始化时，会创建所需的驱动程序对象和设备对象，并将这两类对象进行绑定，从而保证设备的有效管理。

（5） 读写文件时，文件系统驱动将读写请求转换为对磁盘扇区的读写请求，最终由磁盘驱动程序完成对磁盘扇区的读写。

（6） 设备编号 0 对应名称“A”，设备编号 1 对应名称“B”…最多可以有 26 个文件系统设备。可将硬盘设备名称定为“C”。

可以参照 EOS 内核程序中软盘驱动程序的实现来完成硬盘驱动程序的设计。详见表 5-3。

函数名	位置	功能	调用
FloppyIsr	io/driver/floppy.c 第 87 行	设置软驱中断事件为有效，从而通知驱动服务线程中断到达	FlopyAddDevice 中设置中断向量

FloppyWriteFdc	io/driver/floppy.c 第 234 行	写数据到 FDC_DATA 寄存器	FloppyResetFDC
FloppyReadFdc	io/driver/floppy.c 第 299 行	读取 FDC 执行命令的结果字节序列	FloppyResetFDC
FloppyResetFDC	io/driver/floppy.c 第 381 行	复位软驱控制器	FloppyRecalibrate 中，FDC 逻辑错误，复位软驱控制器然后重新校正
FloppyRecalibrate	io/driver/floppy.c 第 459 行	校正磁头位置，将磁头强制归位到 0 磁道	FloppyRw 中，磁盘读写错误，增加错误计数器，校正磁头位置后重试
FloppyRw	io/driver/floppy.c 第 518 行	读写磁盘扇区	FloppyWrit FloppyRead
FloppyRead	io/driver/floppy.c 第 702 行	读取指定的一个扇区到指定的缓冲区中	FloppyInitializeDriver
FloppyWrite	io/driver/floppy.c 第 733 行	写缓冲区中的数据到指定的一个扇区中	FloppyInitializeDriver
FlopyAddDevice	io/driver/floppy.c 第 764 行	添加驱动设备	FloppyInitializeDriver
FloppyInitializeDriver	io/driver/floppy.c 第 813 行	驱动器初始化	IoInitializeSystem2，初始化软驱驱动程序对象

表 5-3：EOS 内核中软盘驱动程序的实现

对应软盘驱动程序的实现，硬盘驱动程序至少需要完成以下函数的设计，见表 5-4。

函数名称	功能
HarddiskRw	读写硬盘扇区
HarddiskRead	读取指定的一个扇区到指定的缓冲区中
HarddiskWrite	写缓冲区中的数据到指定的一个扇区中
HarddiskAddDevice	添加驱动设备
HarddiskInitializeDriver	驱动器初始化

表 5-4：硬盘驱动程序需要实现的功能

可以参照 FAT12 文件系统的源代码来设计 MINIX 文件系统。详见表 5-5。

函数名	位置	功能	调用
FatInitializeDriver	io/driver/fat12.c 第 20 行	初始化驱动器	IoInitializeSystem2 中初始化 FAT12 文件系统驱动程序对象
FatAddDevice	io/driver/fat12.c 第 37 行		FatInitializeDriver
FatCreate	io/driver/fat12.c 第 153 行		FatInitializeDriver
FatClose	io/driver/fat12.c		FatInitializeDriver

	第 196 行		
FatRead	io/driver/fat12.c 第 208 行		FatInitializeDriver
FatWrite	io/driver/fat12.c 第 228 行		FatInitializeDriver
FatQuery	io/driver/fat12.c 第 252 行		FatInitializeDriver
FatGetFatEntryValue	io/driver/fat12.c 第 307 行	读 FAT 中指定项的值	
FatSetFatEntryValue	io/driver/fat12.c 第 341 行	写 FAT 中指定项的值	
FatCheckPath	io/driver/fat12.c 第 443 行	检查路径字符串是否有效	
FatConvertDirName ToFileName	io/driver/fat12.c 第 576 行	将目录项中的 11 字符文件名转换为以 0 结尾的字符串	
FatConvertFile NameToDirName	io/driver/fat12.c 第 625 行	文件名称转换	
FatAllocateOneCluster	io/driver/fat12.c 第 661 行	分配一个空闲簇	
FatReadFile	io/driver/fat12.c 第 704 行	在文件的指定偏移处读取指定字节的数据	
FatWriteFile	io/driver/fat12.c 第 824 行	在文件指定的偏移位置开始写数据	
FatOpenFileInDirectory	io/driver/fat12.c 第 861 行	在指定的目录中打开指定名称的数据文件或者目录文件	
FatOpenFile	io/driver/fat12.c 第 1118 行	打开给定路径的数据文件或目录文件	
FatOpenExistingFile	io/driver/fat12.c 第 1224 行	打开已有文件	
FatWriteDirEntry	io/driver/fat12.c 第 1296 行	写文件对应的 DIRENT 结构体到磁盘	
FatCloseFile	io/driver/fat12.c 第 1367 行	关闭文件控制块	

表 5-5：EOS 内核中 FAT12 文件系统的实现

对应 FAT12 文件系统的实现，加载 MINIX1.0 文件系统至少需要完成以下函数的设计，见表 5-6。

函数名称	功能
MinixInitializeDriver	文件系统驱动初始化
MinixAddDevice	添加对象
MinixCheckPath	检查路径是否有效
MinixReadFile	读文件
MinixWriteFile	写文件
MinixOpenFile	打开文件
MinixCloseFile	关闭文件

表 5-6：MINIX 文件系统需要完成的函数

3.5 改进 EOS 内核，丰富文件操作功能

1. 将“/usr/root”文件夹下的“hello.c”文件的内容打印输出。
2. 将“/usr/src/linux-0.11.org”文件夹下的“memory.c”的内容打印输出。注意，此文件大于 7KB，所以需要使用二次间接块才能访问所有的数据。
3. 删除“/usr/root/hello.c”文件。
4. 删除“/usr/root/shoe”文件夹。
5. 新建“/usr/root/dir”文件夹。
6. 新建“/usr/root/file.txt”文件，并设置初始大小为 10KB。

3.6 提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的项目，并将项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。

实验六 多核处理器改进

实验性质：验证＋设计
建议学时：2 学时
任务数：1 个

一、实验目的

- 实现支持用多处理机的信号量。

二、实验内容

2.1 任务（一）：使 EOS 操作系统可以运行在多处理器上

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/Project-Template/eos-kernel.git

完成实验

EOS 操作系统可以运行在单核处理器下，通过关中断实现互斥信号量。本实验通过修改 EOS 内核，可以使 EOS 操作系统运行在多核处理器。MP（多核处理器）初始化协议定义了两类处理器：引导序列处理器（BSP）和应用处理器（AP）。在一个 MP 系统的一次上电或 RESET 之后，系统硬件动态地选择系统总线上的其中一个处理器作为 BSP。剩下的处理器被指派为 AP。多核处理器采用总线通信方式 APIC 将原本单个中断控制器拆解为两部分，分别是位于处理器内的 Local APIC 和位于主板芯片组中的 I/O APIC。这两部分控制器通过总线相连并在其上进行通信。使 EOS 操作系统运行在多核处理器下，之前实现互斥信号量的方式已不再适用，需要在 EOS 内核中实现实现自旋锁，通过自旋锁实现信号量的互斥。

Bochs 虚拟机可以提供多核处理器的支持，但是需要读者自己编译支持多核处理器的版本，Bochs 虚拟机的编译过程可以参考其官网。读者也可以参考上面的实验二，直接将 EOS 操作系统运行在支持多核处理器的物理机上。

在修改EOS内核使其可以运行在多处理器上时，读者还可以参考xv6操作系统。xv6是由麻省理工学院研发，用于操作系统课程，使用ANSI标准的 C语言编写，总共8000多行代码，可以运行在x86处理器上，还可以进行移植或部署到智能设备上，适合于教学和个人对操作系统的研究。xv6的源代码的链接为https://github.com/mit-pdos/xv6-public。xv6的使用手册的链接为https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf。若以上链接无法访问，可以尝试以下链接：

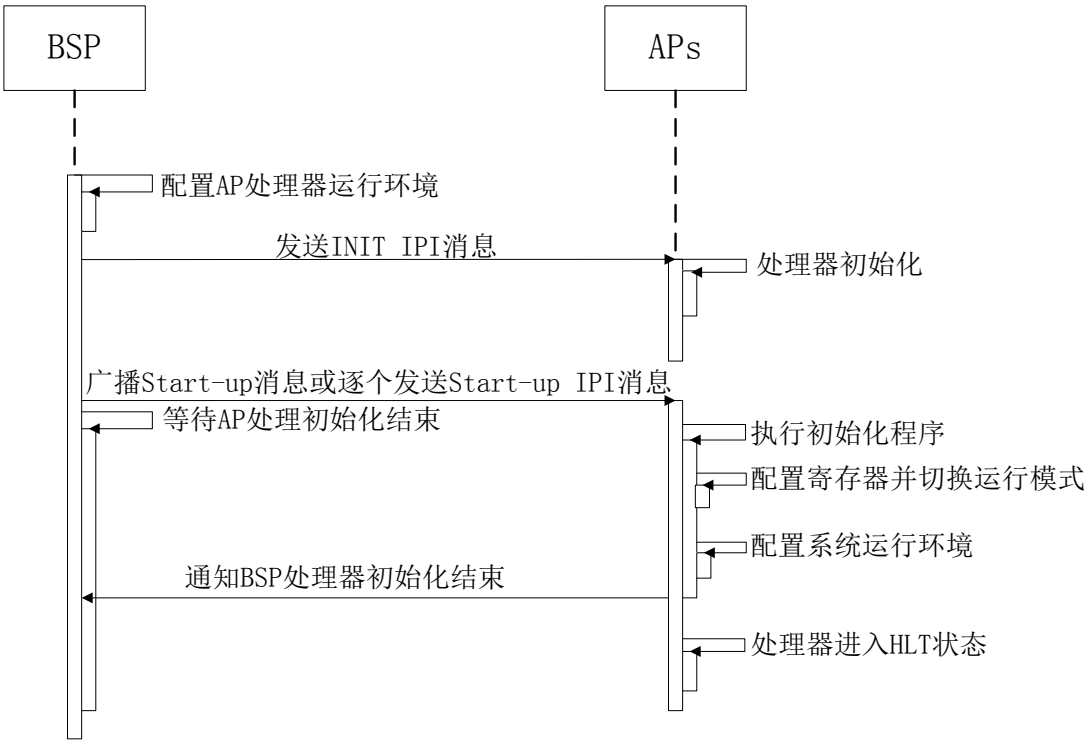
xv6的源代码克隆地址：
https://www.codecode.net/engintime/os-lab/bonus-lab/xv6-public
xv6的使用手册下载地址：
https://share.weiyun.com/5lLhCTT

关于多处理器更多的内容，可以参考《Intel 处理器系统编程手册》，打开 OS Lab 集成实验环境，选择“帮助”菜单中的“其它帮助文档”中的《Intel 处理器系统编程手册》，可以打开此手册，其中的第 8 章和第 10 章分别介绍了 APIC 和多核处理器。还可以参考 ACPI 文档，选择“帮助”菜单中的“其它帮助文档”中的“ACPI 手册”，可以打开此文档。

1. **修改 EOS 内核。**

(1) 多核处理概述

多核处理器间的 IPI 通信机制是以 Local APIC 和 I/O APIC 为载体，借助中断投递方式与与其他处理器通信。多核处理器为 IPI 通信机制提供了一系列寄存器，以实现不同种中断投递方式的处理器间通信。整个中断投递过程主要围绕 ICR（中断命令寄存器）展开，处理器会根据 ICR 寄存器的标志位配置情况，有选择地使用其它附属寄存器将 IPI 消息发送至目标处理器。SMP 系统结构将多个处理器应用于同一操作系统内，以增加多任务的秉性处理速度，当 SMP 系统初始化结束后，其系统结构下的处理器将平等使用系统中的所有资源，SMP 的启动时序如下图。根据 SMP 系统结构启动时序来引导 AP 处理器，其中将涉及 ICR 和其它附属寄存器配置过程。



多核处理器的引导过程可以分为以下几个阶段：

- 向 AP 处理器（应用处理器）发送 INIT IPI 消息和 Start-up IPI 消息
- 配置 AP 处理器执行环境并切换运行模式。
- 配置多核操作系统运行环境。

关于多核处理器更多的内容可以参考 Intel 多处理器初始化文档（8.4 MULTIPLE-PROCESSOR(MR) INITIALIZATION），将 EOS 内核修改为可以支持多处理器

的操作系统。

(2) APIC

APIC 是一个与 8259A 兼容的高级中断处理器。它实现了中断处理的功能，还提供了与中断相关的设备通讯，提供多处理器之间的共享与中断通讯。并且 xv6 很大一部分通讯处理都是通过 APIC 来实现的。

IO ACPI 是用来与外部设备通讯的。它完成了 APIC 最主要的功能，中断处理。I/O APIC 提供了两个模式，普通模式和 8259A 兼容模式。xv6 在单核环境下，会选择使用 8259A 兼容模式，而在多处理器环境下，则会使用普通模式。

ioapic.c 文件提供了 xv6 与 I/O APIC 使用普通模式通讯的函数。包括：

ioapicinit:初始化 I/O APIC

ioapicread:读 I/O APIC

ioapicwrite：写 I/O APIC

ioapicenable：包含两个参数 irq 和 cpunum，用来在 cpunum 号核上打开 ipq 所指的中断。在整个 xv6 中，该函数被调用了两次，分别是打开键盘在第一个核上的中断和打开 IDE 在最后一个核上的中断。

Local APIC 是 APIC 的顶层，每个核都有一个对应的 APIC。它负责进行多处理器间的中断传输，屏蔽中断，还提供了一个可编程的 Timer。由于 I/O APIC 已经提供了中断处理的功能。Local APIC 只是起辅助作用。

xv6 在单核环境中，Local APIC 被屏蔽不用，使用 8253PIT 来实现时钟中断；而只有在多处理器环境中，Local APIC 才被打开，完成初始化每个核、开关中断、Timer 等功能，每个核使用其对应的 Local APIC 提供的 Timer。

在多处理器的环境下，I/O APIC 开中断之后，指定的核并不能收到指定的中断。只有在指定的核的 Local APIC 中开中断后，才能收到指定的中断。其中，在 lapic.c 文件中提供了 xv6 与 Local APIC 通讯的函数。

(3) 初始化

在修改EOS内核的多核处理器的初始化过程中,可以参考xv6操作系统的相关代码。在 xv6 操作系统的源代码的 main.c 文件中,多核处理器的初始化过程是在 main 函数中完成的，在 main 函数中调用了以下 3 个关键函数，分别为：mpinit 函数用于判断是否是多核处理器；lapicinit 函数用于初始化 Local APIC；ioapicinit 函数用于初始化 I/O APIC。读者可以在理解了 xv6 多处理器初始化这部分代码的基础上，然后对 EOS 内核进行修改。

2. 实现自旋锁。

多核处理器间加锁的目的主要是为了防止多个处理器间同时操作共享数据、导致程序执行混乱。

(1) 自旋过程

自旋锁是一种广泛运用的底层同步机制。自旋锁是一个互斥设备，它只有两个值：“锁定”和“解锁”。如果锁可用，则“锁定”被设置，而代码继续进入临界区；相反，当锁被其它线程占有时，未获取锁的线程便会进入自旋，则代码进入忙循环（而不是休眠，这也是自旋锁和一般锁的区别）并不断检测自旋锁的状态，一旦自旋锁被释放，线程便结束自旋，得到自旋锁的线程便可以执行临界区的代码。对于临界区的代码必须短小，否则其它线程会一直受到阻塞，这也是要求锁的持有时间尽量短的原因。

(2) 自旋锁的特点：

- 用于临界区互斥
- 在任何时刻最多只能有一个执行单元获得锁

- 要求持有锁的处理器所占用的时间尽可能短
- 等待锁的线程进入忙循环。

(3) 在 EOS 内核中，实现自旋锁。

可以按照下面的步骤在 EOS 内核中添加自旋锁：

- 定义自旋锁结构体 spinlock。

注意：spinlock 实际上是一个操作系统相关的无符号整数。

```
struct spinlock {
uintlocked;       // Is the lock held?

  // For debugging:
  char *name;         // Name of lock.
  struct cpu *cpu;    // The cpu holding the lock.
uintpcs[10];      // The call stack (an array of program counters)
                    // that locked the lock.
};
```

- 实现初始化自旋锁函数

```
void initlock(struct spinlock *lk, char *name);
```

- 实现申请自旋锁函数

```
void acquire(struct spinlock *lk)
// lk 指向经过 spinlock 的结构体
```

- 实现释放自旋锁函数

```
void release(struct spinlock *lk)
// 指向经过 spinlock 的结构体
```

提示：关于自旋锁相关函数的实现，可以参考 xv6-public 的 spinlock.c 文件中的相关代码。

提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的 EOS 内核项目，并将项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。

实验七 搭建自己的操作系统

实验性质：设计
建议学时：2 学时
任务数：1 个

一、实验目的

- 搭建一个可以运行的操作系统。

二、实验内容

2.1 任务（一）：搭建一个可以运行的操作系统

准备实验

请读者按照下面方法之一在本地创建项目，用于完成本次实验任务：

方法一：从 CodeCode.net 平台领取任务

读者需要首先登录 CodeCode.net 平台领取本次实验对应的任务，从而在 CodeCode.net 平台上创建了个人项目，然后使用 OS Lab 提供的“从 Git 远程库新建项目”功能将这个个人项目克隆到本地磁盘中。

方法二：不从 CodeCode.net 平台领取任务

如果读者不使用 CodeCode.net 平台，可以使用 OS Lab 提供的“从 Git 远程库新建项目”功能直接将 EOS 内核实验模板克隆到本地磁盘中，创建一个 EOS 内核项目，实验模板的 URL 为

https://www.codecode.net/engintime/os-lab/bonus-lab/Lab006.git

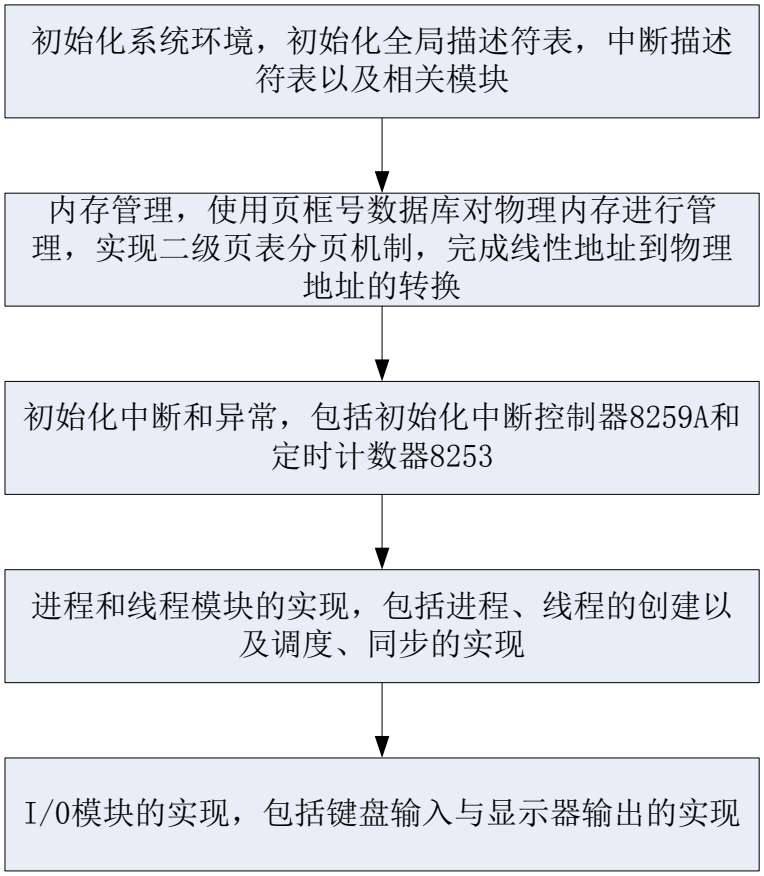
完成实验

本实验的目的是学生可以自己搭建一个操作系统，目前实验模板中已经提供了引导程序 boot.asm、加载程序 loader.asm 以及内核加载后的系统入口点 start.c。在 start.c 中目前仅给出了在控制台中输出文本的函数 KeLog 供读者调试使用，需要完成各个模块的初始化操作以及创建系统启动进程，最终目的是可以通过自己搭建的操作系统完成指定操作。

首先需要完成各部的初始化操作，在文件 start.c 中初始化全局描述符表、中断描述符表、可编程中断控制器 PIC、可编程定时计数器 PIT 以及相关模块的的初始化。函数 KiSystemStartup 是系统的入口点，Kernel.dll 被 Loader 加载到内存后从这里开始执行，其参数 LoaderBlock 为 Loader 传递的加载参数块结构体指针，内存管理器要使用。其结构体中包含以下内容：

PhysicalMemorySize	物理内存大小
MappedMemorySize	映射内存大小
SystemVirtualBase	系统虚拟基址
PageTableVirtualBase	页表虚拟基址
FirstFreePageFrame	空闲物理页框号
ImageVirtualBase	内核映像虚拟基址
ImageSize	内核映像大小

读者可按下图的步骤设计自己的操作系统：



提交作业

如果读者是通过从 CodeCode.net 平台领取任务创建的 EOS 内核项目，并将项目克隆到本地进行实验，实验结束后可以将本地已更改的项目再推送到 CodeCode.net 平台的个人项目中，方便教师通过 CodeCode.net 平台查看读者提交的作业。