卡通画

描述已自动生成

**操作系统实验报告**

**实验题目**  操作系统的启动

**学生姓名**  侯腾跃

**学 号**  2022217477

**专业班级**  计科22-4班

**指导教师**  田卫东

**完成日期**  2024.11.26

**合肥工业大学 计算机与信息学院**

1. **实验目的和任务要求**

跟踪调试 Linux 0.11在PC机上从CPU加电到完成初始化的过程。

查看Linux 0.11启动后的状态和行为，理解操作系统启动后的工作方式。

1. **实验原理**

通过跟踪和调试 Linux 0.11 操作系统在PC机上的启动过程，以及观察其启动后的状态和行为，帮助深入理解操作系统启动的工作原理。

1. **实验内容**
   1. **准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

在VSCode左侧的“文件资源管理器”窗口中打开boot文件夹中的bootsect.asm和setup.asm两个汇编文件。简单阅读一下这两个文件中的源代码和注释。

使用Task中的“生成项目”完成项目的生成过程后，使用Windows资源管理器打开项目文件夹中的boot文件夹。找到由bootsect.asm生成的软盘引导扇区程序bootsect.bin文件，确认该文件的大小为512字节（与软盘中一个扇区的大小相同）。

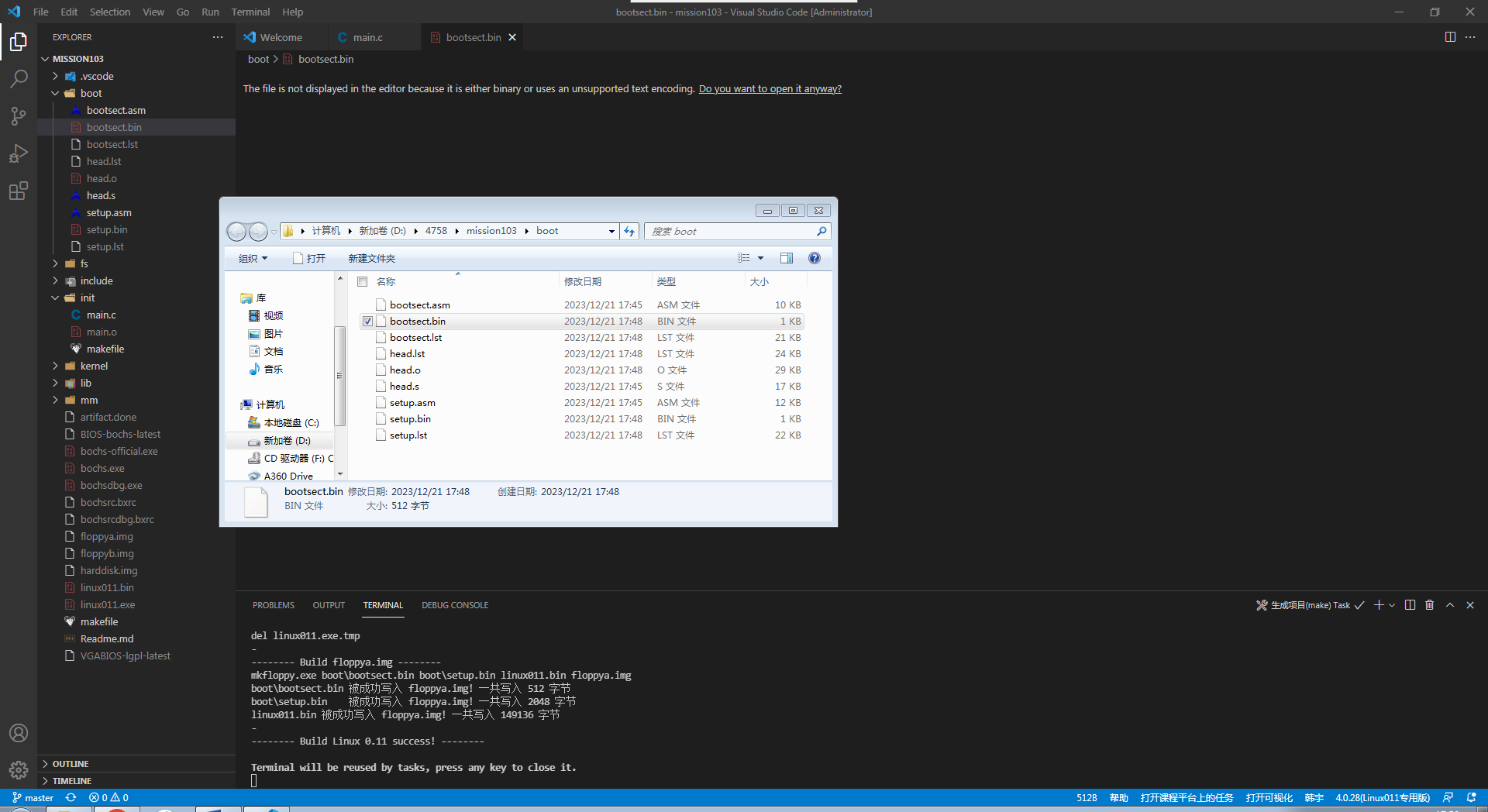


图 1 bin文件大小

* 1. **调试 Linux 0.11操作系统的启动过程**

**记录init/main.c文件中的内核入口函数start的地址**

Linux 0.11在执行完引导和加载程序后，执行的第一个内核函数就是init/main.c文件中的内核入口函数start。首先按照下面的步骤记录下start函数的地址，留待后续使用：

1. 找到start函数的代码行(init/main.c文件中的第134行)。
2. 在此代码行添加一个断点。
3. 按F5启动调试，会在刚刚添加的断点处中断。

文本

描述已自动生成

图 2 断点处中断

1. 在start函数名称上双击鼠标左键选中此名称，然后在此名称上点击右键，选择菜单中的“Add to Watch”，可以在左侧的“WATCH”窗口中查看start函数的地址，如图3所示：

文本

描述已自动生成

图 3 start函数地址

1. start函数地址为0x66b3。
2. 结束调试。

**调试BIOS程序**

1. 在Console窗口中输入调试命令sreg后按回车，显示当前CPU中各个段寄存器的值，如图4所示。其中CS寄存器信息行中的“cs:0xf000”表示CS寄存器的值为0xf000。

电脑萤幕的截图

描述已自动生成

图 4 段寄存器值

2. 在Console窗口中输入调试命令 r 后按回车，显示当前CPU中各个通用寄存器的值，如图5所示。其中“eip：0x0000fff0”表示IP寄存器的值为0xfff0。结合BIOS的第一条指令的地址，可以验证CPU将要执行的指令地址为CS:IP。

电脑萤幕的截图

描述已自动生成

图 5 验证CS:IP

3. 输入调试命令xp /1024b 0x0000，查看从地址0开始的1024个字节的物理内存。在Console中输出的这1K物理内存的值都为0，说明BIOS中断向量表还没有被加载到从地址0开始的物理内存。

电脑萤幕的截图

描述已自动生成

图 6 BIOS中断向量表内存

4. 输入调试命令xp /512b 0x7c00，查看软盘引导扇区要被加载到的物理内存。输出的内存值都为0，说明软盘引导扇区还没有被加载到从地址0x7c00开始的物理内存。

图形用户界面, 文本

描述已自动生成

图 7 软盘引导扇区内存

通过以上的实验步骤，可以验证BIOS第一条指令的逻辑地址中的段地址和CS寄存器值是一致的，偏移地址和IP寄存器的值是一致的。由于物理内存还没有被使用，所以其中的值都为0。

**查看软盘引导扇区程序生成的列表文件**

1. 在VSCode的“文件资源管理器”窗口中打开“boot”文件夹，打开其中的bootsect.lst文件。

2. 将bootsect.lst文件和bootsect.asm文件对比可以发现，此文件包含了bootsect.asm文件中所有的汇编代码，同时在代码的左侧又添加了更多的信息。

3. 在bootsect.lst中查找到软盘引导扇区程序第一条指令所在的行

32 00000000 B8C007 mov ax,BOOTSEG

此行包含的信息有：

 32是行号。

 00000000是此条指令相对于程序开始位置的偏移（第一条指令的偏移为0）。

 B8C007是此条指令的机器码，此条指令包含了3个字节。

4. 在VSCode的“文件资源管理器”窗口中打开“boot”文件夹，在其中的bootsect.bin文件上点击鼠标右键，在弹出的菜单中选择“Open With…”，会在VSCode的顶部中间位置弹出一个文件打开方式的列表，选择其中的“Hex Editor”，就会使用二进制编辑器打开此文件。可以看到bootsect.bin文件中的机器码与列表文件bootsect.lst中的机器码是完全一致的。

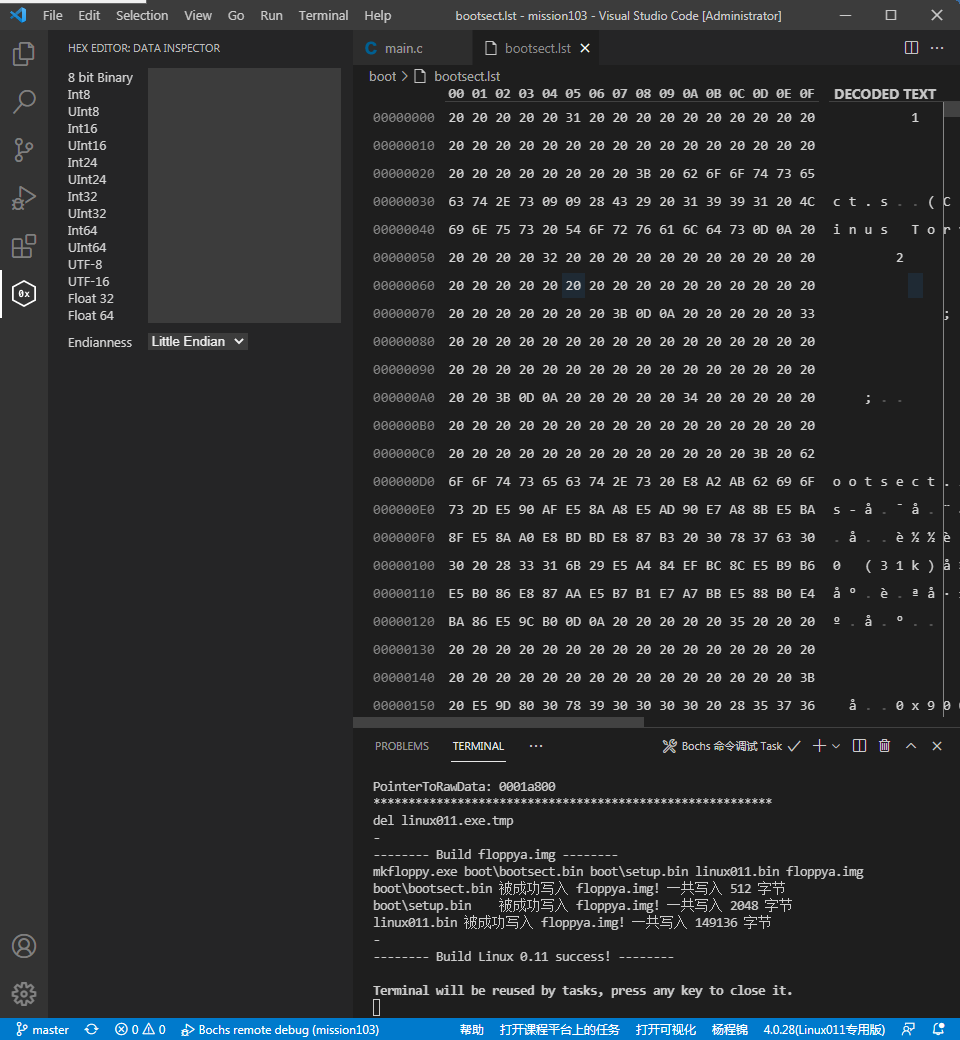


图 8 bootsect.lst

**调试软盘引导扇区程序（bootsect.asm）**

1. 在Bochs的Console窗口中输入调试命令vb 0x0000:0x7c00，这样就在逻辑地址0x0000:0x7c00（相当于物理地址0x7c00）处添加了一个断点。

2. 输入调试命令c让Bochs继续执行，会在0x7c00处的断点中断。此时会在Console窗口中输出下一个要执行的指令，即软盘引导扇区程序的第一条指令，如图2-5所示。

图形用户界面, 文本

描述已自动生成

图 9 引导扇区程序的第一条指令

3. 为了方便后面的使用，可以在纸上记录下此条指令的字节码（b8c007）。

4. 输入调试命令sreg验证CS寄存器的值（0x0000）。

图形用户界面, 文本

描述已自动生成

图 10 CS寄存器值

5. 输入调试命令r验证IP寄存器的值（0x7c00）。

图形用户界面, 文本

描述已自动生成

图 11 IP寄存器值

6. 由于BIOS程序此时已经执行完毕，输入调试命令 xp /1024b 0x0000验证此时BIOS中断向量表已经被载入。

图形用户界面, 文本

描述已自动生成

图 12 BIOS中断向量表

7. 输入xp /512b 0x7c00显示软盘引导扇区程序的所有字节码。观察此块内存最开始的三个字节分别是0xb8、0xc0和0x07，这和引导程序第一条指令的字节码是相同的。如果将Bochs的Console窗口中显示的512个字节与bootsect.lst文件中的机器码进行比较，或者与使用二进制编辑器打开的bootsect.bin文件中的内容进行比较，读者会发现它们的内容是完全相同的。此块内存最后的两个字节分别是0x55和0xaa，它们是魔数，是BIOS规定这两个字节的值必须为0x55和0xaa时才表示引导扇区是激活的，可以用来引导操作系统。这两个字节对应bootsect.asm中最后一行语句（注意，字节顺序使用Little-endian）：

dw 0xAA55

图形用户界面, 文本

描述已自动生成图形用户界面, 文本

描述已自动生成

图 13 软盘引导扇区程序字节码

接下来查看引导程序将自己复制到0x90000后的情况：

1. 输入调试命令 xp /512b 0x9000:0x0000可以验证此时引导程序还没有将自己移动到0x9000:0x0000处。

图形用户界面, 文本

描述已自动生成

图 14 验证引导程序位置

1. 输入调试命令vb 0x9000:0x0018，在0x9000:0x0018处设置一个断点。
2. 输入调试命令c继续执行，会在刚刚添加的断点处中断。

图形用户界面, 文本

描述已自动生成

图 15 设置断点并调试

4. 再次输入调试命令 xp /512b 0x9000:0x0000，并与之前的内存比较，可以知道引导程序已经将自己移到了0x90000处，并在0x9000:0x0018处中断执行了。

图形用户界面, 文本

描述已自动生成

图 16 中断执行

5. 输入调试命令 xp /512b 0x9000:0x0200，可以验证此时setup.bin模块还没有被装入内存。

图形用户界面, 文本

描述已自动生成

图 17 查看setup.bin模块对应内存

6. 根据bootsect.lst文件下面一行的内容（执行此行指令时说明setup.bin加载完毕），输入调试命令 vb 0x9000:0x0031，在逻辑地址0x9000:0x0031处设置一个断点。

75 00000031 0F830B00 jnc ok\_load\_setup ; 读取成功

7. 输入调试命令c继续执行，会在刚刚添加的断点处中断。

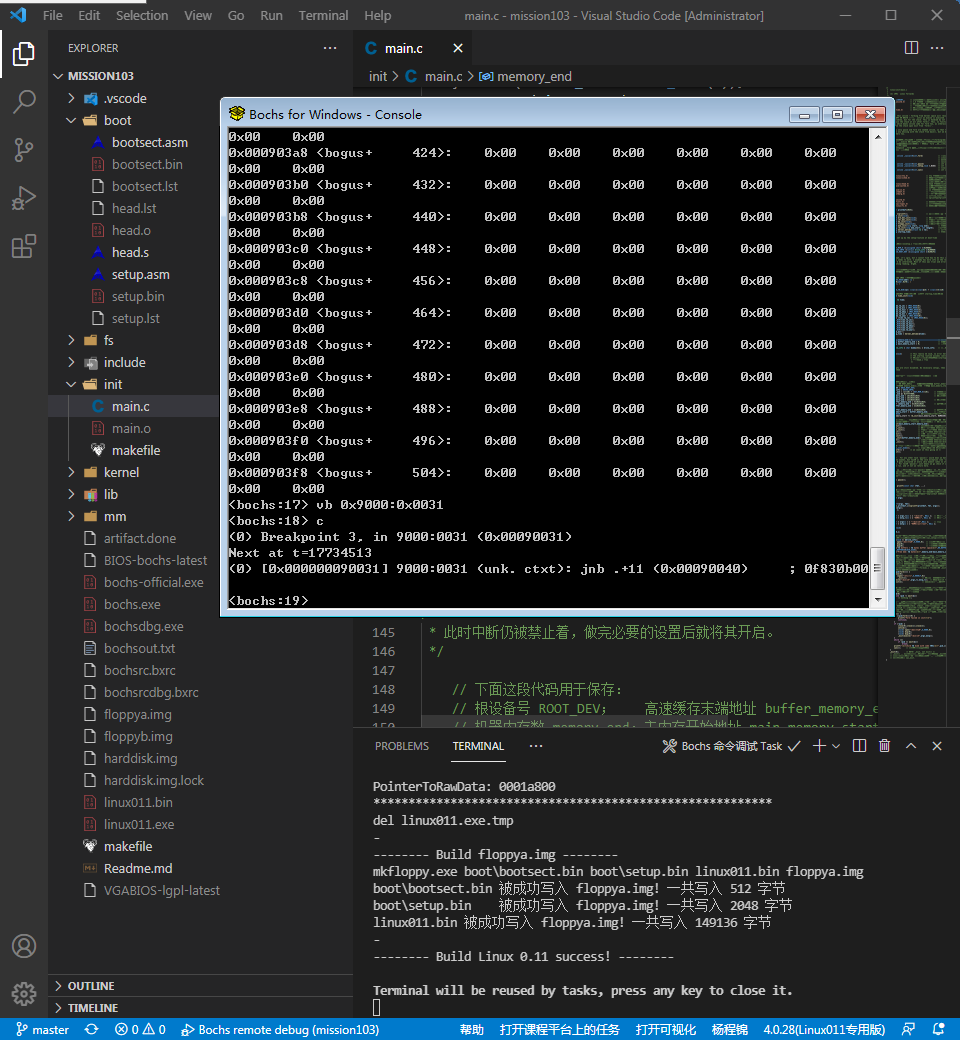


图 18 设置断点并调试

8. 输入调试命令 xp /512b 0x9000:0x0200，此块内存已经发生改变。将此块内存中的字节码与init/setup.lst文件和init/setup.bin文件中的内容比较，可以知道此时setup.bin模块已经被载入内存。

图形用户界面, 文本

描述已自动生成

图 19 查看内存

9. 输入调试命令 xp /512b 0x1000:0x0000，可以看到除前面有少量数据(BIOS自检程序残留下的)，后面全是0。可以知道此时内核模块linux011.bin还没有被装载进入内存。

图形用户界面, 文本

描述已自动生成

图 20 查看linux011.bin模块对应内存

10. 根据bootsect.lst文件下面一行的内容（执行此行指令时说明linux011.bin加载完毕，软盘驱动器中的马达被关闭了），输入调试命令 vb 0x9000:0x006e，在逻辑地址0x9000:0x006e处设置一个断点。

119 0000006E E8DB00 call kill\_motor

11. 输入调试命令c继续执行，会在刚刚添加的断点处中断。

图形用户界面, 文本

描述已自动生成

图21 设置断点并调试

12. 输入调试命令 xp /512b 0x1000:0x0000，显示此块内存中的字节码与boot/head.lst文件（由init/head.asm编译时生成）和linux011.bin文件中开始位置的机器码是完全一致的。原因是init/head.asm生成了目标文件init/head.o，此目标文件中的代码段（包括指令的机器码）会在链接时写入内核模块linux011.bin的代码段的开始位置。由此可知，此时内核模块linux011.bin已经被装入物理内存。

图形用户界面

描述已自动生成

图 22 查看内存

**调试加载程序（setup.asm）**

1. 在Bochs的Console窗口中输入调试命令 vb 0x9020:0x0000，在逻辑地址0x9020:0x0000（即setup程序的第一条指令）处设置一个断点。

2. 输入调试命令 c继续执行，可在0x9020:0x0000处中断，如图23。打开setup.lst文件，可以看到其中第一条指令的字节码与此处将要执行的指令的字节码是相同的，说明Linux 0.11将要开始执行setup模块。

图形用户界面, 文本

描述已自动生成

图 23 设置断点并调试

3. 输入调试命令 xp /2b 0x9000:0x0000，查看取得各种机器参数之前物理内存0x90000-0x901FF中前两个字节的值并记录下来，如图28。

图形用户界面, 文本

描述已自动生成

图 24 查看内存值

4. 根据setup.lst 文件下面一行的内容，输入调试命令 vb 0x9020:0x0080。

116 00000080 FA cli

5. 输入调试命令c继续运行，在刚刚添加的断点处中断。此时Linux 0.11已经将各种机器参数放入0x90000-0x901FF的物理内存中。

6. 输入调试命令 xp /2b 0x9000:0x0000，查看此处内存的值，如图2-8，并和步骤3进行对比。可以知道此时该地址处的内存值已经改变。

文本

描述已自动生成

图 25 插入断点并调试

7. 根据setup.lst 文件下面一行的内容，输入调试命令vb 0x9020:0x010b。

209 0000010B EA00000800 jmp 8:0

8. 输入调试命令 c继续运行，可以在刚刚添加的断点处中断。此时，setup.asm已经完成了它的工作，并让处理器从实模式进入了保护模式，下一条将要执行的跳转指令是工作在保护模式下的，所以要指定段选择符和偏移，才会跳转到head.s的第一条指令处执行。注意，这里的段值的8已经是保护模式下的段选择符了，用于选择描述符表和描述符表项以及所要求的特权级。

文本

描述已自动生成

图 26 插入断点并调试

**调试内核模块中的head.s**

1. 打开head.lst，找到其第一条指令所在行，如下（此条指令包含了5个字节）：

25 0000 B8100000 movl $0x10,%eax

25 00

在生成Linux 0.11内核的过程中，head.s会生成目标文件head.o，此目标文件中的代码段在链接时会写入Linxu011.bin中代码段的开始位置，并且linux011.bin会被从之前的0x10000起始的物理内存移动到0x00000起始的物理内存。所以，可以在物理地址0x0000处设置一个断点。注意，执行到这里时，CPU已经处于保护模式了，使用在物理地址添加断点的命令比较方便，所以输入的调试命令为：pb 0x0000。

2. 输入调试命令 c 继续执行，可在图 27所示的指令处中断。对比head.lst文件中第一条指令的字节码，可以确认已经进入了head.s模块。

文本

描述已自动生成

图 27 插入断点并调试

3. 根据head.lst文件下面一行的内容，输入调试命令：pb 0x540b。 该指令负责将内核入口点函数start的地址压入栈。

202 540b 68000000 pushl $\_start

202 00

4. 输入调试命令 c 继续执行，在步骤3设置的断点处中断，如图28。

将图中红框中的地址与之前记录的start函数的地址比较，可以确认这个地址就是start函数的地址。最终就是通过这个地址跳转到操作系统内核的入口点开始执行，从而结束引导过程的。

图形用户界面, 文本

描述已自动生成



图 28 插入断点并调试

5. 根据head.lst文件下面一行的内容，输入调试命令：pb 0x54a5。

316 54a5 C3 ret

6. 输入调试命令c继续执行，在ret指令处中断。然后接着输入调试命令s，单步执行此行的ret指令，可以得到图29：

图形用户界面, 文本

描述已自动生成



图 29 插入断点并调试

图中红线圈出的是IP寄存器将要执行的下一条指令。对比可以发现它和图28中圈出的值是一样的，说明接下来就开始执行start函数中的指令了。

* 1. **内核初始化**

通过对比图2-1和图2-11中的函数地址，可以知道，处理器最终跳转到文件init/main.c的start函数中去执行。说明此时已经进入了内核，但从刚进入内核到系统最终稳定下来，等待用户输入命令，还有很多初始化工作要做。

在Bochs的Console窗口中输入调试命令 c 让Bochs虚拟机继续执行。当Linux 0.11操作系统完成启动后，打开Bochs的Display窗口，然后按F1键，会列出当前的进程信息，如图30所示。这些进程信息显示出当系统启动完毕，开始等待用户输入命令时，有进程号（pid）分别为0、1、4、3的四个进程。

文本

描述已自动生成

图 30 Linux 0.11完成初始化后的进程信息

显然，在终端显示的进程列表信息量比较小，读者此时可以使用可视化窗口查看Linux内核初始化过程中的更多信息。步骤如下：

1. 结束之前的调试，关闭Bochs虚拟机。

2.使用VSCode打开在3.1中得到的Linux内核项目。选择“Run”菜单中的“Remove All Breakpoints”将所有断点清除。

3. 接下来在源代码中一个合适的位置添加断点，从而查看Linux内核初始化完毕后的状态。之前介绍过，当Linux内核完成初始化后，就开始等待用户输入命令了。显然在处理等待用户输入命令的代码处添加这个断点最合适。所以，读者可以在kernel\chr\_drv\tty\_io.c文件的tty\_read函数中（第321行）添加一个断点即可。此函数就是终端进程读取用户输入的字符时调用的函数。

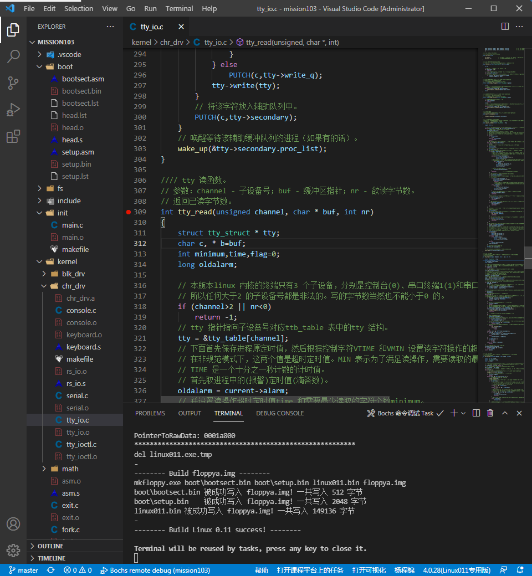


图 31 插入断点

4. 按F5启动调试，待Linxu 内核初始化完毕后，就会命中刚刚添加的断点。

5. 在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口让读者查看可视化视图。

6. 在右侧可视化视图顶部的编辑框中输入命令“#task”，就可以显示当前系统中所有进程的数据（如图32所示）。此时系统中存在四个进程，进程号分别为0、1、4、3。其中current游标是一个全局变量，用于指向当前正在运行的进程。所以，从图上可以看出，当前正在运行的进程是4号进程，也就是终端进程，这与添加断点中断执行的位置是吻合的，说明终端进程正在（运行）从键盘获取用户输入的命令。当然，如果接下来用户没有输入任何命令，终端进程就会进入阻塞状态，让出处理器，调度程序会让0号进程继续运行，直到用户输入命令时将终端进程唤醒继续执行。关于进程状态转换过程和进程调度，会在后面的实验中详细讨论。请读者根据可视化窗口中各个进程“父进程ID”字段的值，整理出所有进程的父子关系。其中，-1表示没有父进程。

文本

描述已自动生成

图 32 Linux 0.11内核初始化完毕时的进程信息

7. 在右侧可视化视图顶部的编辑框中输入命令“#sched”后按回车（需要等待较长时间完成刷新），就可以查看进程的运行轨迹。在图中每一行表示所有进程中的某一个进程进入了某个状态，并使用不同的颜色表示不同的状态（具体看图例），例如第一行表示pid为0的进程进入了运行态，后面在其运行的过程中创建了pid为1的进程。细心的读者一定会发现在每一行中最多只会有一个处于运行状态的进程，这是因为Linux 0.11内核虽然允许多个进程并发运行，但是不允许多个进程并行运行，也就是说只支持单处理器，不支持多处理器，所以某个时刻只能有一个进程在运行，请读者一定要理解并发和并行的区别。在图中最左侧是Jiffies时钟滴答，在本书后面会有详细的介绍，这里读者只需要将其理解为CPU运行的时间即可。在图中最右侧是函数名称和行号，根据这两个信息读者可以定位具体是哪一行代码让进程发生状态变化的。从图中可以看到进程号为0的进程是Linux初始化过程中创建的第一个进程，在完成加载系统后，演变为进程调度，交换以及存储管理进程。然后，以进程0为母本创建进程1，来继续完成系统的初始化，进程1是系统中所有其它进程的祖先进程。然后，以进程1为母本创建进程号为2的Shell进程。2号进程再创建进程号为3的update进程，然后进程2就结束了。update进程的主要任务是将缓冲区中的数据同步到外设(软盘和硬盘)。update进程启动后将会被阻塞，然后让进程1继续运行，此时进程1会创建一个新的Shell进程接受用户的命令，其进程号为4。进程5是进程4创建的，用于监听内核异常。当系统出现宕机，可以利用该进程将宕机时的一些堆栈信息写入指定文件，用于事后分析宕机的原因。当系统正常启动后，进程5结束。这就是Linux 0.11内核的初始化过程。

图形用户界面

描述已自动生成

图 33 进程运行轨迹

1. **实验的思考与问题分析**

1、答：

bootsect.bin（引导扇区）的功能:

（1）这个程序被存储在软盘的引导扇区，也就是软盘的第一个扇区（通常512字节大小）。

（2）它的主要任务是引导计算机启动并加载更多的启动代码。由于BIOS限制，引导扇区只有512字节，因此需要精心编写代码以适应这个大小限制。

（3）bootsect.bin 加载内存中的 setup.bin 程序。

setup.bin的功能:

（1）setup.bin 负责进行更多的启动前准备工作，例如检测系统硬件、设置内存段、初始化设备等。

（2）这个阶段可能需要更多的代码，超过了引导扇区512字节的限制。

为什么使用两个文件:

（1）由于BIOS加载引导扇区代码的限制（只有512字节），所有的引导代码无法全部放在 bootsect.bin 中。

（2）因此，将一部分启动过程的代码移至 setup.bin，以便 bootsect.bin 可以在有限的空间内完成其核心任务。

将 setup.bin 的功能合并到 bootsect.bin:

（1）如果尝试将 setup.bin 的功能合并进 bootsect.bin，则 bootsect.bin 的大小将超过512字节，这是不可行的，因为BIOS只能从软盘的第一个扇区读取512字节的引导代码。

（2）这种设计是为了遵守硬件和BIOS的限制，同时提供足够的功能来启动操作系统。

因此，Linux 0.11操作系统从软盘启动时使用 bootsect.bin 和 setup.bin 两个程序是出于必要，以适应硬件和BIOS的限制。尝试将它们合并到一个文件中会导致超出BIOS能够处理的引导扇区大小限制。

2、答：

bootsect.s（bootsect.bin）:

（1）此文件包含了引导扇区的代码。

（2）它的主要职责是加载setup.s（setup.bin）到内存，并跳转执行。

在bootsect.s中，会有代码来读取软盘上的其他扇区（包括setup.bin和内核的一部分）到内存中。

（3）它还设置了一些初始的寄存器值，为系统启动做准备。

setup.s（setup.bin）:

（1）setup.s包含了在加载实际的Linux内核之前需要执行的代码。

（2）它主要负责初始化硬件设备、设置CPU的工作模式，以及建立内存分段。

（3）这个文件还会检测系统的硬件配置，例如RAM大小、显卡类型等。

（4）setup.s将控制权转移给内核的主要部分。

head.s:

（1）head.s是内核的一部分，通常紧跟在setup.s之后加载。

（2）它负责将CPU从实模式切换到保护模式，这是运行现代操作系统所必需的。

（3）head.s还初始化了用于内核运行的各种表和数据结构，比如全局描述符表（GDT）和中断描述符表（IDT）。

3、答：

修改 mkfloppy 应用程序

假设 mkfloppy 程序中有一个明确的地方，定义了 linux011.bin 写入软盘镜像的起始扇区。你需要找到这个位置并进行修改。示例代码如下（具体代码取决于原始代码的结构）：

// 假设原代码中有类似这样的定义

#define LINUX011\_START\_SECTOR 5

// 将其修改为

#define LINUX011\_START\_SECTOR 8

在修改后，重新编译 mkfloppy 以生成新的可执行文件。

修改 boot\bootsect.asm

在 bootsect.asm 文件中，需要找到负责从软盘读取 linux011.bin 的代码部分，并修改它以从第8个扇区开始读取。示例代码可能类似于以下（具体代码取决于原始代码的结构）：

/假设原代码中有类似这样的指令

mov ax, 0x0005 ; 软盘的第5个扇区

//将其修改为

mov ax, 0x0008 ; 软盘的第8个扇区

在进行修改后，重新编译 bootsect.asm 以生成新的 bootsect.bin。

4、答：

修改 boot\bootsect.asm

对于 boot\bootsect.asm，您需要增加代码来识别和加载PE格式的文件。这需要对PE格式有深入的了解。

伪代码示例，具体实现取决于原始代码和PE格式的具体细节

假设已经将文件加载到内存

现在检查PE签名

mov ax, [内存中PE文件的位置] ; 将文件头的位置加载到ax

cmp word [ax], 0x5A4D ; 检查PE文件的"MZ"签名

jne not\_pe\_format ; 如果不是PE格式，跳转到错误处理代码

识别并加载PE文件的代码和数据段

not\_pe\_format:

错误处理代码

1. **总结和感想体会**

通过本实验我更加深入的理解了Linux0.11操作系统启动的流程，以及学会了如何用Bochs进行打断点并调试和查看内存值等技术。这次实验让我亲身经历了操作系统启动过程的细节，虽然有些繁琐，但这种深入理解是非常有价值的。通过亲自跟踪调试，我更加熟悉了底层的机制，对操作系统的启动有了更为清晰的认识。