**读取硬盘数据**

**目标**

我们会构造一个1KB大小的磁盘映像文件，第一个扇区，即前512字节，保存我们的引导扇区程序，第二个扇区，即后512字节，保存一个文本文件。通过引导扇区的程序将第二个扇区的文本文件打印在显示器上。

大体上来说完成这个目标需要两个步骤。第一步，将数据从磁盘读取到内存；第二步，将数据打印到显示器上。显然，第一步需要进行磁盘的 I/O 操作，第二步则相对简单。

**目录结构**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $ tree .  .  ├── boot.S  ├── main.c  ├── Makefile  ├── message.data  ├── mmu.h  ├── types.h  └── x86.h |

其中boot.S、mmu.h内容与前一篇文章相同，不再重复介绍。完整的代码戳[这里](https://github.com/kviccn/asm-boooom/tree/master/0x0D)。

先来看types.h。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #ifndef \_\_TYPES\_H\_  #define \_\_TYPES\_H\_  typedef \_\_signed char int8\_t;  typedef unsigned char uint8\_t;  typedef short int16\_t;  typedef unsigned short uint16\_t;  typedef int int32\_t;  typedef unsigned int uint32\_t;  typedef long long int64\_t;  typedef unsigned long long uint64\_t;  #endif |

我们定义了一些类型，可以让我们少打几个字。

x86.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #include "types.h"  static inline uint8\_t  inb(uint16\_t port)  {  uint8\_t data;  asm volatile("inb %1,%0"  : "=a"(data)  : "d"(port));  return data;  }  static inline void  outb(uint16\_t port, uint8\_t data)  {  asm volatile("outb %0,%1"  :  : "a"(data), "d"(port));  }  static inline void  insl(int port, void \*addr, int cnt)  {  asm volatile("cld; rep insl"  : "=D"(addr), "=c"(cnt)  : "d"(port), "0"(addr), "1"(cnt)  : "memory", "cc");  } |

这里我们使用内联汇编定义了三个函数，这三个函数使用static、inline修饰。static保证函数只在声明它的文件中可见，避免和其他相同名称的函数冲突。inline告诉编译器尽可能将函数内联到调用它的地方，这样可以减少函数调用次数，提高效率，但这不是必然。

inb函数内联了inb指令，用于从指定端口读取1字节数据。

outb函数内联了outb指令，用于向指定端口写入1字节数据。

insl函数内联了cld; rep insl指令，cld用于清除方向标志，使偏移量向正方向移动，这个偏移量其实就是传入的addr，会被关联到edi，反汇编的结果中可以看到，请大家自己实验。rep前缀用于重复执行insl，重复的次数由ecx决定，即传入的参数cnt。最终数据会被连续读取到addr指向的内存处。

main.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | #include "x86.h"  void readsect(void \*dst, uint32\_t offset);  void bootmain(void)  {  readsect((void \*)0xb8000, 1);  while (1)  ;  }  void waitdisk(void)  {  while ((inb(0x1F7) & 0xC0) != 0x40)  ;  }  void readsect(void \*dst, uint32\_t offset)  {  waitdisk();  outb(0x1F2, 1);  outb(0x1F3, offset);  outb(0x1F4, offset >> 8);  outb(0x1F5, offset >> 16);  outb(0x1F6, (offset >> 24) | 0xE0);  outb(0x1F7, 0x20);  waitdisk();  insl(0x1F0, dst, 512 / 4);  } |

第3行声明函数readsect，用于从磁盘读取一个扇区。参数dst指定目的内存位置，参数offset指定要读取的扇区的偏移量。我们将使用LBA模式访问磁盘，该模式从0开始编号数据块，第一个区块LBA=0，第二个区块LBA=1，以此类推。

第7行调用readsect，将偏移量为1的扇区，即第二个扇区的数据读取到内存0xb8000处。因为我们将向该扇区写入ASCII编码的文本，所以可以直接将数据读取到显存对应的内存处，以直接打印文本。

第13~17行定义函数waitdisk，采用忙等的方式等待磁盘准备好进行数据传输。端口1F7既是命令端口，又是状态端口。作为状态端口时，每一位含义如下：

* 第7位 控制器忙碌
* 第6位 磁盘驱动器已准备好
* 第5位 写入错误
* 第4位 搜索完成
* 第3位 为1时扇区缓冲区没有准备好
* 第2位 是否正确读取磁盘数据
* 第1位 磁盘每转一周将此位设为1
* 第0位 之前的命令因发生错误而结束

inb(0x1F7)从端口0x1F7读取出状态，与0xC0做&运算，只保留高两位，即第7位和第6位，如果不等于0x40（控制器不忙且已准备好交互），则继续等待、测试。

第19~33行定义函数readsect。

第21行，函数首先调用waitdisk以确保磁盘准备好交互。

第23行，向端口0x1F2写入1，指定读取的扇区数量为1。

第24~27行，向端口0x1F3、0x1F4、0x1F5、0x1F6写入28位的逻辑扇区编号，其中端口0x1F6的高四位写入0xE，表示以LBA模式访问主硬盘。

第28行，端口0x1F7做为命令端口，向其写入0x20表示请求读硬盘。

第30行，继续等待硬盘准备好数据。

第32行，调用函数insl从端口0x1F0读取数据到dst，0x1F0是数据端口。读取的次数是512 / 4，因为一个扇区包含512个字节，而insl指令一次可以读取4个字节。

message.data

H

e

l

l

o

l

a

o

l

i

!

每一个字符换一行，因为换行符的ASCII码为0a，正好等于浅绿色的字符显示属性，所以我们可以直接将其与字符一起读入显存对应的内存处，做为字符的显示属性。既可以说是偷懒，也可以说是个小技巧。因为我们的目的是演示如何使用 C 语言读写磁盘。message.data的底层内容如下：

|  |  |
| --- | --- |
| 1  2  3 | $ xxd -a message.data  00000000: 480a 650a 6c0a 6c0a 6f0a 200a 6c0a 610a H.e.l.l.o. .l.a.  00000010: 6f0a 6c0a 690a 210a o.l.i.!. |

**编译链接**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ cc -m32 -c -o boot.o boot.S  $ cc -m32 -fno-builtin -fno-pic -nostdinc -c -o main.o main.c  $ ld -N -e start -Ttext=0x7c00 -m elf\_i386 -o boot.elf boot.o main.o  $ objcopy -S -O binary -j .text boot.elf boot.bin  $ cp boot.bin boot  $ ./sign boot |

**制作磁盘映像**

|  |  |
| --- | --- |
| 1  2  3 | $ dd if=/dev/zero of=boot.img bs=512 count=2  $ dd if=boot of=boot.img conv=notrunc  $ dd if=message.data of=boot.img seek=1 conv=notrunc |

**运行**

|  |  |
| --- | --- |
| 1 | $ qemu-system-i386 -drive file=boot.img,format=raw -monitor stdio |

结果如下:

可以看到从!到e中间的部分是黑掉的。因为我们没有那么多数据，但是readsect还是会读取512个字节，而剩下的字节都是0。

**加载“内核”**

其实我们上一个栗子的代码已经相当于一个操作系统的bootloader了。我们从磁盘读取了一段数据并显示在屏幕上，如果我们读取一段程序并执行它呢？这可不就是一个bootloader加载内核的过程吗。你品，你细品。

接下来老李就带大家撸一个Hello world 内核加载到内存并运行起来。

**目录结构**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $ tree .  .  ├── boot.S  ├── kernel.c  ├── main.c  ├── Makefile  ├── mmu.h  ├── types.h  └── x86.h |

还是挑有变化的来讲，除了main.c和kernel.c，其他内容都与之前相同。完整的代码戳[这里](https://github.com/kviccn/asm-boooom/tree/master/0x0E)。

main.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | #include "x86.h"  void readsect(void \*dst, uint32\_t offset);  void bootmain(void)  {  readsect((void \*)0x10000, 1);  ((void (\*)(void))(0x10000))();  while (1)  ;  }  void waitdisk(void)  {  while ((inb(0x1F7) & 0xC0) != 0x40)  ;  }  void readsect(void \*dst, uint32\_t offset)  {  waitdisk();  outb(0x1F2, 1);  outb(0x1F3, offset);  outb(0x1F4, offset >> 8);  outb(0x1F5, offset >> 16);  outb(0x1F6, (offset >> 24) | 0xE0);  outb(0x1F7, 0x20);  waitdisk();  insl(0x1F0, dst, 512 / 4);  } |

不同处在第7、9行。

第7行，这次我们将数据读取到了内存0x10000处。

第9行，通过强制类型转换，将0x10000处开始的内容转换成了一个函数并调用，函数的类型是void (\*)(void)。如果像下面这样写可能会好理解一点：

|  |  |
| --- | --- |
| 1  2  3 | void (\*entry)(void);  entry = (void (\*)(void))(0x10000);  entry(); |

kernel.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include "types.h"  void entry(void)  {  uint16\_t \*video\_buffer = (uint16\_t \*)0xb8000;  for (int i = 0; i < 80 \* 25; i++)  {  video\_buffer[i] = (video\_buffer[i] & 0xff00) | ' ';  }  video\_buffer[0] = 0x0700 | 'l';  video\_buffer[1] = 0x0700 | 'a';  video\_buffer[2] = 0x0700 | 'o';  video\_buffer[3] = 0x0700 | 'l';  video\_buffer[4] = 0x0700 | 'i';  video\_buffer[5] = 0x0700 | '!';  } |

常规操作，清屏，打印字符，大家应该已经轻车熟路了。

**编译链接**

**制作 bootloader**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ cc -m32 -c -o boot.o boot.S  $ cc -m32 -fno-builtin -fno-pic -nostdinc -c -o main.o main.c  $ ld -N -e start -Ttext=0x7c00 -m elf\_i386 -o boot.elf boot.o main.o  $ objcopy -S -O binary -j .text boot.elf boot.bin  $ cp boot.bin boot  $ ./sign boot |

**制作 kernel**

|  |  |
| --- | --- |
| 1  2 | $ cc -m32 -fno-builtin -fno-pic -nostdinc -c -o kernel.o kernel.c  $ objcopy -S -O binary -j .text kernel.o kernel |

**制作磁盘映像**

|  |  |
| --- | --- |
| 1  2  3 | $ dd if=/dev/zero of=boot.img bs=512 count=2  $ dd if=boot of=boot.img conv=notrunc  $ dd if=kernel of=boot.img seek=1 conv=notrunc |

**运行**

|  |  |
| --- | --- |
| 1 | $ qemu-system-i386 -drive file=boot.img,format=raw -monitor stdio |

结果如下:

**总结**

先来说一下上面所谓的“内核”潜在的一些问题。我们只是简单的从磁盘读取了512字节的数据，事实上真正的“内核”的大小是变化的，可能小于512字节，但更多的可能是大于512字节。所以我们需要将“内核”的大小写在某个地方，让bootloader知道应该读取多少扇区。

再来说一下写汇编语言这个系列的初衷，是为了开发操作系统做一些准备工作。如果只是学习操作系统的理论知识那完全可以不学习汇编语言，但如果想开发一个操作系统，那汇编语言的知识就是必不可少的。因为不可避免的要和硬件打交道，不论是x86、arm还是其它的体系结构。