**练习1**：理解通过make生成执行文件的过程（要求在报告中写出对下述问题的回答）。

在此练习中，大家需要通过静态代码分析来了解如下内容。

（1）操作系统镜像文件ucore.img是如何一步一步生成的（需要比较详细地解释 Makefile 中每一条相关命令和命令参数的含义，以及说明命令导致的结果）？

（2）一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

补充材料：如何调试Makefile？

当执行make时，一般只会显示输出，不会显示make到底执行了哪些命令。

如想了解make执行了哪些命令，可以执行：

$make ”V=”

要获取更多有关make的信息，可上网查询，并请执行：

$man make

**答：**

（1）开始编译之前lab1文件夹的目录结构如下：

|  |
| --- |
| .  ├── boot  │   ├── asm.h  │   ├── bootasm.S  │   └── bootmain.c  ├── kern  │   ├── debug  │   │   ├── assert.h  │   │   ├── kdebug.c  │   │   ├── kdebug.h  │   │   ├── kmonitor.c  │   │   ├── kmonitor.h  │   │   ├── panic.c  │   │   └── stab.h  │   ├── driver  │   │   ├── clock.c  │   │   ├── clock.h  │   │   ├── console.c  │   │   ├── console.h  │   │   ├── intr.c  │   │   ├── intr.h  │   │   ├── kbdreg.h  │   │   ├── picirq.c  │   │   └── picirq.h  │   ├── init  │   │   └── init.c  │   ├── libs  │   │   ├── readline.c  │   │   └── stdio.c  │   ├── mm  │   │   ├── memlayout.h  │   │   ├── mmu.h  │   │   ├── pmm.c  │   │   └── pmm.h  │   └── trap  │   ├── trap.c  │   ├── trapentry.S  │   ├── trap.h  │   └── vectors.S  ├── libs  │   ├── defs.h  │   ├── elf.h  │   ├── error.h  │   ├── printfmt.c  │   ├── stdarg.h  │   ├── stdio.h  │   ├── string.c  │   ├── string.h  │   └── x86.h  ├── Makefile  └── tools  ├── function.mk  ├── gdbinit  ├── grade.sh  ├── kernel.ld  ├── sign.c  └── vector.c |

由于kernel生成过程中出现的gcc编译过程中均使用了“–IPATH -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector”这些参数，为了看起来更加清楚，在这里列出的编译过程中省略了这些参数，参数具体的含义参见附录1。

下面一部分输出展示了生成kernel所需的全部目标文件：

|  |
| --- |
| # 编译kern/init文件夹下的文件  + cc kern/init/init.c # 编译init.c  gcc -c kern/init/init.c -o obj/kern/init/init.o  # 编译kern/libs文件夹下的文件  + cc kern/libs/readline.c # 编译readline.c  gcc -c kern/libs/readline.c -o obj/kern/libs/readline.o  + cc kern/libs/stdio.c # 编译stdio.c  gcc -c kern/libs/stdio.c -o obj/kern/libs/stdio.o  # 编译kern/debug文件夹下的文件  + cc kern/debug/kdebug.c # 编译kdebug.c  gcc -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o  + cc kern/debug/kmonitor.c # 编译kmonitor.c  gcc -c kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o  + cc kern/debug/panic.c # 编译panic.c  gcc -c kern/debug/panic.c -o obj/kern/debug/panic.o  # 编译kern/driver文件夹下的文件  + cc kern/driver/clock.c # 编译clock.c  gcc -c kern/driver/clock.c -o obj/kern/driver/clock.o  + cc kern/driver/console.c # 编译console.c  gcc -c kern/driver/console.c -o obj/kern/driver/console.o  + cc kern/driver/intr.c # 编译intr.c  gcc -c kern/driver/intr.c -o obj/kern/driver/intr.o  + cc kern/driver/picirq.c # 编译picirq.c  gcc -c kern/driver/picirq.c -o obj/kern/driver/picirq.o  # 编译kern/trap文件夹下的文件  + cc kern/trap/trap.c # 编译trap.c  gcc -c kern/trap/trap.c -o obj/kern/trap/trap.o  + cc kern/trap/trapentry.S # 编译trapentry.S  gcc -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o  + cc kern/trap/vectors.S # 编译vectors.S  gcc -c kern/trap/vectors.S -o obj/kern/trap/vectors.o  # 编译kern/mm文件夹下的文件  + cc kern/mm/pmm.c # 编译pmm.c  gcc -c kern/mm/pmm.c -o obj/kern/mm/pmm.o  # 编译libs文件夹下的文件  + cc libs/printfmt.c # 编译printfmt.c  gcc -c libs/printfmt.c -o obj/libs/printfmt.o  + cc libs/string.c # 编译string.c  gcc -c libs/string.c -o obj/libs/string.o |

编译生成目标文件（.o）之后，需要对kernel的目标文件使用ld命令进行链接得到二进制文件，这里用到的ld命令的参数的具体含义参见附录2，下面的输出信息展示了链接生成kernel二进制文件的过程：

|  |
| --- |
| + ld bin/kernel  ld -m elf\_i386 -nostdlib -T tools/kernel.ld -o bin/kernel \  obj/kern/init/init.o \  obj/kern/libs/readline.o \  obj/kern/libs/stdio.o \  obj/kern/debug/kdebug.o \  obj/kern/debug/kmonitor.o \  obj/kern/debug/panic.o \  obj/kern/driver/clock.o \  obj/kern/driver/console.o \  obj/kern/driver/intr.o \  obj/kern/driver/picirq.o \  obj/kern/trap/trap.o \  obj/kern/trap/trapentry.o \  obj/kern/trap/vectors.o \  obj/kern/mm/pmm.o \  obj/libs/printfmt.o \  obj/libs/string.o |

下面进行了bootblock和所需的sign工具的编译：

|  |
| --- |
| + cc boot/bootasm.S  gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc \  -c boot/bootasm.S -o obj/boot/bootasm.o  + cc boot/bootmain.c  gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc \  -c boot/bootmain.c -o obj/boot/bootmain.o  + cc tools/sign.c  gcc -Itools/ -g -Wall -O2 \  -c tools/sign.c -o obj/sign/tools/sign.o  gcc -g -Wall -O2 \  obj/sign/tools/sign.o -o bin/sign |

下面的输出是通过sign工具来修饰编译生成的BootLoader：

|  |
| --- |
| + ld bin/bootblock  #将目标文件bootasm.o、bootmain.o链接到一起，使用start符号作为入口，并指定text段在程序中的绝对位置是0x7c00（操作系统开始加载的地址是0x7c00）  ld -m elf\_i386 -nostdlib -N -e start -Ttext 0x7C00 \  obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o |

上述命令结束后，sign工具输出了如下两条信息，表示成功：

|  |
| --- |
| 'obj/bootblock.out' size: 496 bytes  build 512 bytes boot sector: 'bin/bootblock' success! |

最后，使用dd命令（用法参见附录3）把编译出的文件写入ucore.img：

|  |
| --- |
| # 创建大小为10000个块的ucore.img文件，每个块的大小为512字节，全部初始化为0  dd if=/dev/zero of=bin/ucore.img count=10000  10000+0 records in  10000+0 records out  5120000 bytes (5.1 MB) copied, 0.009409 s, 544 MB/s  # 把bootblock的内容写入第一个块  dd if=bin/bootblock of=bin/ucore.img conv=notrunc  1+0 records in  1+0 records out  512 bytes (512 B) copied, 2.2787e-05 s, 22.5 MB/s  # 从第二个块开始写入kernel的内容  dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc  146+1 records in  146+1 records out  74862 bytes (75 kB) copied, 0.000184299 s, 406 MB/s |

（2）摘自tools/sign.c中的部分代码如下：

|  |
| --- |
| char buf[512];  memset(buf, 0, sizeof(buf));  FILE \*ifp = fopen(argv[1], "rb");  int size = fread(buf, 1, st.st\_size, ifp);  if (size != st.st\_size) {  fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);  return -1;  }  fclose(ifp);  buf[510] = 0x55;  buf[511] = 0xAA;  FILE \*ofp = fopen(argv[2], "wb+");  size = fwrite(buf, 1, 512, ofp);  if (size != 512) {  fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);  return -1;  }  fclose(ofp);  printf("build 512 bytes boot sector: '%s' success!\n", argv[2]); |

从代码中可以看出一个符合规范的硬盘主引导的特征是：

* 大小为512字节
* 最后两个字节为0x55AA

**附录：**

**1．gcc****命令部分参数及其含义：**

-IPATH：注意I与PATH之间没有空格，作用是除了默认的头文件搜索路径（如/usr/include等）外，同时在PATH下搜索需要呗引用的头文件

-fno-builtin：除非以“\_\_builtin\_”开头进行引用，否则不识别所有内建函数

-Wall：编译时显示全部警告提示信息

-ggdb：产生 GDB 所需的调试信息。这意味着将会使用可用的、最具表达力的格式（DWARF2、stabs，或者在前两者不支持情况下的其他本地格式），如果可能的话还会包含 GDB 扩展信息

-g：产生原生格式的调试信息，可用于其他调试器

-m32：为32位环境生成代码，int、long和指针都是32位

-gstab：产生 stabs 格式的调试信息（如果支持的话），并且会使用只有 GNU 调试器（GDB）才理解的GNU扩展。这些扩展的使用可能会导致其他调试器崩溃或者拒绝读取编译出来的可执行程序

-nostdinc：不扫描标准系统头文件，只在-I指令指定的目录中扫描

-fno-stack-protector：禁用栈保护措施，生成用于检查栈溢出的额外代码，如果发生错误，则打印错误信息并退出

-O2：编译器对代码进行二级优化，进行大部分不以空间换时间的优化

-Os：对输出文件大小进行优化，开启全部不增加代码大小的-O2优化

-c：编译源文件但不进行链接

-o：结果的输出文件

**2．ld命令部分参数及其含义：**

-m elf\_i386：链接为Intel i386 PC上的程序

-N：设置代码段和数据段均可读写

-e start：将start符号置为程序起始点

-Ttext：指定代码段的起始位置

-nostdlib：不链接系统标准启动文件和标准库文件，只把指定的文件传递给链接器

-T tools/kernel.ld：将tools/kernel.ld作为链接器脚本

-o bin/kernel：输出到bin/kernel文件

**3．dd命令部分参数及其含义：**

if=文件名：输入文件名，缺省为标准输入。即指定源文件

of=文件名：输出文件名，缺省为标准输出。即指定目的文件

seek=blocks：从输出文件开头跳过blocks个块后再开始复制

count=blocks：仅拷贝blocks个块，块大小等于ibs指定的字节数

conv=notrunc：不截短输出文件

/dev/zero：该设备无穷尽地提供0，可以用于向设备或文件写入字符0