实验一：系统软件启动过程

**实验目的：**

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软甲boot loader来完成这些工作。为此我们需要完成一个能够切换到X86的保护模式并显示字符的boot loader，为启动操作系统ucore做准备。Lab1提供了一个非常小的boot loader和ucore OS，整个boot loader执行代码小于512个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个boot loader和ucoreOS ，读者可以了解到：

计算机原理、boot loader软件和ucore OS 软件。

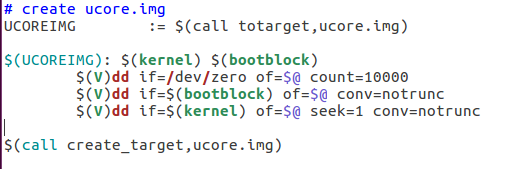
**实验内容:**

Lab1中包含一个boot loader和一个OS。这个boot loader可以切换到X86保护模式，能够读磁盘并加载elf执行文件格式，并显示字符。而这lab1中的OS只是一个可以处理时钟中断和显示字符的幼儿园级别OS。

练习1 理解通过make生成执行文件的过程。

[练习1.1] 操作系统镜像文件 ucore.img 是如何一步一步生成的?

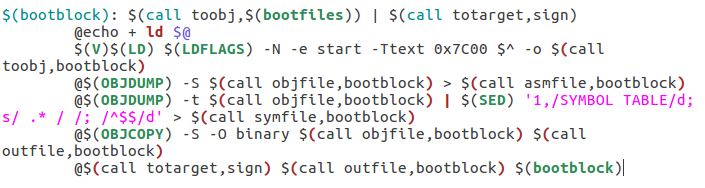
**生成ucore.img的相关代码为**

（dd是连接文件的，要申请一个空白的，大小为10000的空间，放入ucore.img）

Ucore.img依赖于两个文件，一个是bootblock、kernel，所以为了生成ucore.img，首先需要生成bootblock、kernel。（但是操作系统运行不了kernel，还要有一个加载程序，要将其包装成一个镜像，电脑才可以启动它）

**首先是bin/bootblock**

生成bootblock的相关代码为



为了生成bootblock，首先需要生成bootasm.o、bootmain.o、sign

生成bootasm.o,bootmain.o的相关makefile代码为



实际代码由宏批量生成

生成bootasm.o需要bootasm.S

实际命令为

其中关键的参数为

-ggdb 生成可供gdb使用的调试信息。这样才能用qemu+gdb来调试bootloader or ucore。

-m32 生成适用于32位环境的代码。我们用的模拟硬件是32bit的80386，所以ucore也要是32位的软件。

-gstabs 生成stabs格式的调试信息。这样要ucore的monitor可以显示出便于开发者阅读的函数调用栈信息

-nostdinc 不使用标准库。标准库是给应用程序用的，我们是编译ucore内核，OS内核是提供服务的，所以所有的服务要自给自足。

-fno-stack-protector 不生成用于检测缓冲区溢出的代码。这是for 应用程序的，我们是编译内核，ucore内核好像还用不到此功能。

-Os 为减小代码大小而进行优化。根据硬件spec，主引导扇区只有512字节，我们写的简单bootloader的最终大小不能大于510字节。

-I<dir> 添加搜索头文件的路径

生成bootmain.o需要bootmain.c

实际命令为：



新出现的关键参数有

-fno-builtin 除非用\_\_builtin\_前缀，

否则不进行builtin函数的优化

最后是bin/sign

生成sign工具的makefile代码为



实际命令为



首先生成bootblock.o



其中关键的参数为

-m <emulation> 模拟为i386上的连接器

-nostdlib 不使用标准库

-N 设置代码段和数据段均可读写

-e <entry> 指定入口

-Ttext 制定代码段开始位置

拷贝二进制代码bootblock.o到bootblock.out

objcopy -S -O binary obj/bootblock.o obj/bootblock.out

其中关键的参数为

-S 移除所有符号和重定位信息

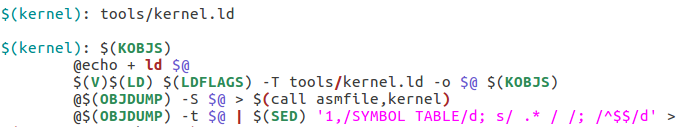
-O <bfdname> 指定输出格式

使用sign工具处理bootblock.out，生成bootblock

bin/sign obj/bootblock.out bin/bootblock

**然后生成bin/kernel**

生成kernel的相关代码为

为了生成kernel，首先需要 kernel.ld init.o readline.o stdio.o kdebug.o

kmonitor.o panic.o clock.o console.o intr.o picirq.o trap.o

trapentry.o vectors.o pmm.o printfmt.o string.o

kernel.ld已存在

生成obj/kern/\*/\*.o

生成这些.o文件的相关makefile代码为

$(call add\_files\_cc,$(call listf\_cc,$(KSRCDIR)),kernel,\

$(KCFLAGS))

这些.o生成方式和参数均类似，仅举init.o为例，其余不赘述

obj/kern/init/init.o

编译需要init.c

实际命令为



生成kernel时，makefile的几条指令中有@前缀的都不必需

必需的命令只有

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

其中新出现的关键参数为

-T <scriptfile> 让连接器使用指定的脚本

**最后生成一个有10000个块的文件，每个块默认512字节，用0填充**

dd if=/dev/zero of=bin/ucore.img count=10000

把bootblock中的内容写到第一个块

dd if=bin/bootblock of=bin/ucore.img conv=notrunc

从第二个块开始写kernel中的内容

dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc



练习2 使用qemu执行并调试lab1中的软件。

[练习2.1] 从 CPU 加电后执行的第一条指令开始,单步跟踪 BIOS 的执行。

（1）修改 lab1/tools/gdbinit,内容为:

```

set architecture i8086

target remote :1234

```

（2） 在 lab1目录下，执行

```

make debug

```

（3）在看到gdb的调试界面(gdb)后，在gdb调试界面下执行如下命令

```

si

```

即可单步跟踪BIOS了。

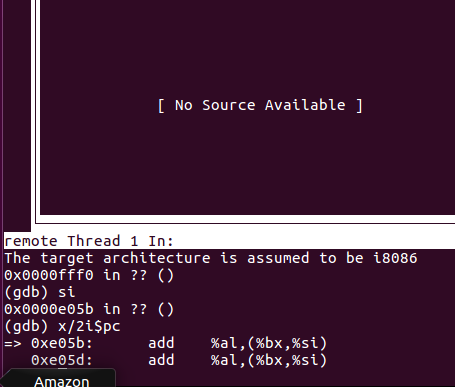
（4）在gdb界面下，可通过如下命令来看BIOS的代码

```

x /2i $pc //显示当前eip处的汇编指令

```

结果如图：



[练习2.2] 在初始化位置0x7c00 设置实地址断点,测试断点正常。

在tools/gdbinit结尾加上

```

set architecture i8086 //设置当前调试的CPU是8086

b \*0x7c00 //在0x7c00处设置断点。此地址是bootloader入口点地址，可看boot/bootasm.S的start地址处

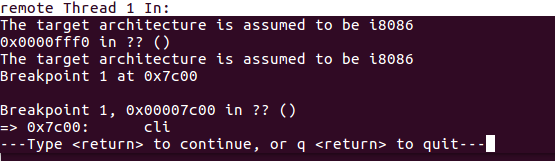
c //continue简称，表示继续执行

x /2i $pc //显示当前eip处的汇编指令

set architecture i386 //设置当前调试的CPU是80386

```

运行"make debug"便可得到



[练习2.3] 在调用qemu 时增加-d in\_asm -D q.log 参数，便可以将运行的汇编指令保存在q.log 中。

将执行的汇编代码与bootasm.S 和 bootblock.asm 进行比较，看看二者是否一致。

在tools/gdbinit结尾加上

```

b \*0x7c00

c

x /10i $pc

```

便可以在q.log中读到"call bootmain"前执行的命令

```

----------------

IN:

0x00007c00: cli

----------------

IN:

0x00007c01: cld

0x00007c02: xor %ax,%ax

0x00007c04: mov %ax,%ds

0x00007c06: mov %ax,%es

0x00007c08: mov %ax,%ss

----------------

IN:

0x00007c0a: in $0x64,%al

----------------

IN:

0x00007c0c: test $0x2,%al

0x00007c0e: jne 0x7c0a

----------------

IN:

0x00007c10: mov $0xd1,%al

0x00007c12: out %al,$0x64

0x00007c14: in $0x64,%al

0x00007c16: test $0x2,%al

0x00007c18: jne 0x7c14

----------------

IN:

0x00007c1a: mov $0xdf,%al

0x00007c1c: out %al,$0x60

0x00007c1e: lgdtw 0x7c6c

0x00007c23: mov %cr0,%eax

0x00007c26: or $0x1,%eax

0x00007c2a: mov %eax,%cr0

----------------

IN:

0x00007c2d: ljmp $0x8,$0x7c32

----------------

IN:

0x00007c32: mov $0x10,%ax

0x00007c36: mov %eax,%ds

----------------

IN:

0x00007c38: mov %eax,%es

----------------

IN:

0x00007c3a: mov %eax,%fs

0x00007c3c: mov %eax,%gs

0x00007c3e: mov %eax,%ss

----------------

IN:

0x00007c40: mov $0x0,%ebp

----------------

IN:

0x00007c45: mov $0x7c00,%esp

0x00007c4a: call 0x7d0d

----------------

IN:

0x00007d0d: push %ebp

```

其与bootasm.S和bootblock.asm中的代码相同。