操作系统

LAB1实验报告

班级：人工智能2103

学号：202107030125

姓名：姚丁钰

---

lab0

QEMU

qemu:硬件模拟器，对于我们的实验而言，它可用于模拟一台 X86 计算机，让 `ucore` 能够在其上运行。

qemu 运行可以有多参数，格式如： `qemu [options] [disk_image]`

disk_image是硬盘镜像文件

部分参数说明：

```
`-hda file' `-hdb file' `-hdc file' `-hdd file'
    使用 file 作为硬盘0、1、2、3镜像。
`-fda file' `-fdb file'
    使用 file 作为软盘镜像，可以使用 /dev/fd0 作为 file 来使用主机软盘。
`-cdrom file'
    使用 file 作为光盘镜像，可以使用 /dev/cdrom 作为 file 来使用主机cd-rom。
`-boot [a|c|d]'
    从软盘(a)、光盘(c)、硬盘启动(d)，默认硬盘启动。
`-snapshot'
    写入临时文件而不写回磁盘镜像，可以使用 C-a s 来强制写回。
`-m megs'
    设置虚拟内存为 msg M字节，默认为 128M 字节。
`-smp n'
    设置为有 n 个 CPU 的 SMP 系统。以 PC 为目标机，最多支持 255 个CPU。
`-nographic'
```

```
    禁止使用图形输出。
其他：
    可用的主机设备 dev 例如：
        vc
            虚拟终端。
        null
            空设备
        /dev/XXX
            使用主机的 tty。
        file: filename
            将输出写入到文件 filename 中。
        stdio
            标准输入/输出。
        pipe: pipename
            命令管道 pipename。
        等。
    使用 dev 设备的命令如：
        `-serial dev'
            重定向虚拟串口到主机设备 dev 中。
        `-parallel dev'
            重定向虚拟并口到主机设备 dev 中。
        `-monitor dev'
            重定向 monitor 到主机设备 dev 中。
    其他参数：
        `-s'
            等待 gdb 连接到端口 1234。
        `-p port'
            改变 gdb 连接端口到 port。
        `-S'
            在启动时不启动 CPU， 需要在 monitor 中输入 'c'，才能让qemu继续模拟工作。
        `-d'
            输出日志到 qemu.log 文件。
```

例如：

```
qemu -hda ucore.img -parallel stdio // 让ucore在qemu模拟的x86硬件环境中运行
```

```
qemu -S -s -hda ucore.img -monitor stdio // 开启远程调试
```

make

在实验文件夹下使用 make 命令即可，make会按照当前目录下的 Makefile 脚本构建项目。

例如在Lab1中：

```
cd .../lab1
make
```

此时在lab1目录下的 bin 目录中会生成一系列的目标文件

- **ucore.img**：系统镜像文件
- **kernel**: ELF格式的系统内核文件，被嵌入到了**ucore.img**中
- **bootblock**: 虚拟的硬盘主引导扇区（512字节），包含了bootloader执行代码，被嵌入到了**ucore.img**中
- **sign**：小工具，用来生成符合规范的虚拟硬盘主引导扇区

调试

为了与qemu配合进行源代码级别的调试，需要先让qemu等待gdb调试器的接入并且不能让qemu中的CPU在此之前执行，因此启动qemu的时候，我们需要使用参数 -S –s 来做到这一点，这相当于在本地的1234端口 开启远程调试服务。

在使用了前面提到的参数启动qemu之后，qemu中的CPU并不会马上开始执行，这时我们启动gdb，然后在gdb命令行界面下，使用下面的命令连接到qemu：

```
//在lab1目录下
(gdb) target remote :1234
```

然后输入c（也就是continue）命令之后，qemu会继续执行下去，但是gdb由于不知道任何符号信息，并且也没有下断点，是不能进行源码级的调试的。为了让gdb获知符号信息，需要指定调试目标文件，gdb中使用file命令：

```
(gdb) file xxxx
```

之后gdb就会载入这个文件中的符号信息，这时通过gdb就可以对ucore代码进行调试了

以在lab1中调试memset函数为例：

在第一个终端中执行：

```
//此时在lab1目录下
qemu -S -s -hda ./bin/ucore.img -monitor stdio // 启动qemu运行ucore并开启远程调试服务
```

这时会弹出一个窗口，qemu已经开始等待远程gdb的连接了，接下来打开第二个终端运行gdb。（如果你在此过程中发现鼠标指针不见了，这是因为你点击到了qemu的图形窗口导致鼠标被其捕获，使用快捷键`Ctrl+Alt+G`即可重新获取鼠标控制权。）

在第二个终端中执行：

```
cd .../lab1 //切换至lab1的目录
gdb //启动gdb
(gdb) file ./bin/kernel //在gdb中载入目标文件以获取符号信息
(gdb) target remote :1234 //用gdb连接至本地的1234端口进行调试
(gdb) break memset //在memset函数处下断点
(gdb) continue //调试至断点
```

为了方便，可以将调试命令保存在脚本中，并让gdb在启动的时候载入。

以lab1为例，在 lab1/tools 目录下，执行完 make 后，我们可以修改 gdbinit 文件：

打开文件，进入文本编辑窗口，如下编辑文件内容(第5-10行)

```
file bin/kernel //在gdb中载入目标文件以获取符号信息
set architecture i8086 //设置CPU架构为i8086
target remote :1234 //用gdb连接至本地的1234端口进行调试
break kern_init //在内核初始化函数处设置断点
define hook-stop //这部分设置每次单步调试都显示出附近两行的汇编以方便调试
x/2i $pc
end
```

使用上面编辑好的脚本启动gdb：

```
cd .. //退回lab1目录
gdb -tui -x tools/gdbinit //以gdbinit脚本启动gdb并开启源代码视图
```

其他指令

```
make grade // 测试编写的实验代码是否基本正确
make handin // 如果实现基本正确（即看到上条指令输出都是OK）则生成实验打包
make qemu // 让OS实验工程在qemu上运行
make debug // 实现通过gdb远程调试OS实验工程
```

设定调试目标架构：

在调试的时候，我们也许需要调试非i386保护模式的代码，而是比如8086实模式的代码，这时我们需要

设定当前使用的架构：

```
(gdb) set architecture i8086
```

加载调试目标：

在使用qemu进行远程调试的时候，我们必须手动加载符号表，也就是在gdb中用file命令。

这样加载调试信息都是按照elf文件中制定的虚拟地址进行加载的，这在静态连接的代码中没有任何问题。但是在调试含有动态链接库的代码时，动态链接库的ELF执行文件头中指定的加载虚拟地址都是0，这个地址实际上是不正确的。从操作系统角度来看，用户态的动态链接库的加载地址都是由操作系统动态分配的，没有一个固定值。然后操作系统再把动态链接库加载到这个地址，并由用户态的库链接器（linker）把动态链接库中的地址信息重新设置，自此动态链接库才可正常运行。

由于分配地址的动态性，gdb并不知道这个分配的地址是多少，因此当我们在对这样**动态链接的代码进行调试的时候**，需要手动要求gdb将调试信息加载到指定地址。下面，我们要求gdb将linker加载到0x6fee6180这个地址上：

```
(gdb) add-symbol-file ...../linker 0x6fee6180
```

lab1基础了解

整体目录

lab1的整体目录结构如下所示：

```
>>> tree
  .
  ├── bin // =======编译后生成
=====================================
  | ├── bootblock // 是引导区
  | ├── kernel // 是操作系统内核
  | ├── sign // 用于生成一个符合规范的硬盘主引导扇区
  | └── ucore.img // ucore.img 通过dd指令，将上面我们生成的
bootblock 和kernel 的ELF文件拷贝到ucore.img
  ├── boot // =======bootloader 代码
================================
```

```
9   |  ├── asm.h // 是bootasm.S汇编文件所需要的头文件，是一些与X86保护模式的段访问方式相关的宏定义.
10  |  ├── bootasm.S // 0. 定义了最先执行的函数start，部分初始化，从实模式切换到保护模式，调用bootmain.c中的bootmain函数
11  |  └── bootmain.c // 1. 实现了bootmain函数，通过屏幕、串口和并口显示字符串,加载ucore操作系统到内存，然后跳转到ucore的入口处执行.
12  |  // 生成 bootblock.out
13  |  // 由 sign.c 在最后添加 0x55AA之后生成 规范的 512字节的
14  ├── kern // =======ucore系统部分 ====================================
15  |  ├── debug// 内核调试部分 ==================================================
16  |  |  ├── assert.h // 保证宏 assert宏，在发现错误后调用 内核监视器 kernelmonitor
17  |  |  ├── kdebug.c // 提供源码和二进制对应关系的查询功能，用于显示调用栈关系。
18  |  |  ├── kdebug.h // 其中补全print_stackframe函数是需要完成的练习。其他实现部分不必深究。
19  |  |  ├── kmonitor.c // 实现提供动态分析命令的kernel monitor，便于在ucore出现bug或问题后，
20  |  |  ├── kmonitor.h // 能够进入kernel monitor中，查看当前调用关系。
21  |  |  ├── panic.c // 内核错误(Kernel panic)是指操作系统在监测到内部的致命错误，
22  |  |  └── stab.h
23  |  ├── driver //驱动 ============================================================
24  |  |  ├── clock.c // 实现了对时钟控制器8253的初始化操作 系统时钟
25  |  |  ├── clock.h
26  |  |  ├── console.c // 实现了对串口和键盘的中断方式的处理操作 串口命令行终端
27  |  |  ├── console.h
28  |  |  ├── intr.c // 实现了通过设置CPU的eflags来屏蔽和使能中断的函数
29  |  |  ├── intr.h
30  |  |  ├── kbdreg.h //
31  |  |  ├── picirq.c // 实现了对中断控制器8259A的初始化和使能操作
32  |  |  └── picirq.h
33  |  ├── init // 系统初始化 ======================================================
34  |  |  └── init.c // ucore操作系统的初始化启动代码
35  |  ├── libs
36  |  |  ├── readline.c
37  |  |  └── stdio.c
38  |  ├── mm // 内存管理
Memorymanagement=======================================
```

```
39  |  |  ├── memlayout.h // 操作系统有关段管理（段描述符编号、段号等）的一些宏定义
40  |  |  ├── mmu.h // 内存管理单元硬件 Memory Management Unit 将线性地址映射为物理地址,包括EFLAGS寄存器等段定义
41  |  |  ├── pmm.c // 设定了ucore操作系统在段机制中要用到的全局变量
42  |  |  └── pmm.h // 任务状态段ts，全局描述符表 gdt[],加载gdt的函数lgdt，初始化函数gdt_init
43  |  └── trap // 陷阱trap 异常exception 中断interrupt 中断处理部分 ==================
44  |  ├── trap.c // 紧接着第二步初步处理后，继续完成具体的各种中断处理操作；
45  |  ├── trapentry.S // 紧接着第一步初步处理后，进一步完成第二步初步处理；
46  |  | // 并且有恢复中断上下文的处理，即中断处理完毕后的返回准备工作；
47  |  ├── trap.h // 紧接着第二步初步处理后，继续完成具体的各种中断处理操作；
48  |  └── vectors.S // 包括256个中断服务例程的入口地址和第一步初步处理实现。
49  | // 此文件是由tools/vector.c在编译ucore期间动态生成的
50  ├── libs // 公共库部分 ============================================================
51  |  ├── defs.h // 包含一些无符号整型的缩写定义
52  |  ├── elf.h
53  |  ├── error.h
54  |  ├── printfmt.c
55  |  ├── stdarg.h // argument 参数
56  |  ├── stdio.h // 标志输入输出 io
57  |  ├── string.c
58  |  ├── string.h
59  |  └── x86.h // 一些用GNU C嵌入式汇编实现的C函数
60  ├── Makefile // 指导make完成整个软件项目的编译，清除等工作。
61  └── tools // 工具部分 ============================================================
62      ├── function.mk // mk模块 指导make完成整个软件项目的编译，清除等工作。
63      ├── gdbinit // gnu debugger 调试
64      ├── grade.sh
65      ├── kernel.ld
66      ├── sign.c // 一个C语言小程序，是辅助工具，用于生成一个符合规范的硬盘主引导扇区。
67      | // 规范的硬盘主引导扇区大小为512字节，结束符为0x55AA
68      | // obj/bootblock.out( <= 500 ) + 0x55AA ->bootblock(512字节)
69      └── vector.c // 生成vectors.S 中断服务例程的入口地址和第一步初步处理实现
```

bootloader部分

- `boot/bootasm.S`：定义并实现了bootloader最先执行的函数start，此函数进行了一定的初始化，完成了从实模式到保护模式的转换，并调用bootmain.c中的bootmain函数。
- `boot/bootmain.c`：定义并实现了bootmain函数实现了通过屏幕、串口和并口显示字符串。bootmain函数加载ucore操作系统到内存，然后跳转到ucore的入口处执行。
- `boot/asm.h`：是bootasm.S汇编文件所需要的头文件，主要是一些与X86保护模式的段访问方式相关的宏定义。

ucore操作系统部分

1. 系统初始化部分：

- `kern/init/init.c`：ucore操作系统的初始化启动代码

2. 内存管理部分：**

- `kern/mm/memlayout.h`：ucore操作系统有关段管理（段描述符编号、段号等）的一些宏定义
- `kern/mm/mmu.h`：ucore操作系统有关X86 MMU等硬件相关的定义，包括EFLAGS寄存器中各位的含义，应用/系统段类型，中断门描述符定义，段描述符定义，任务状态段定义，NULL段声明的宏SEG_NULL, 特定段声明的宏SEG，设置中断门描述符的宏SETGATE
- `kern/mm/pmm.[ch]`：设定了ucore操作系统在段机制中要用到的全局变量：任务状态段ts，全局描述符表 gdt[]，加载全局描述符表寄存器的函数lgdt，临时的内核栈stack0；以及对全局描述符表和任务状态段的初始化函数gdt_init

3. 外设驱动部分：

- `kern/driver/intr.[ch]`：实现了通过设置CPU的eflags来屏蔽和使能中断的函数
- `kern/driver/picirq.[ch]`：实现了对中断控制器8259A的初始化和使能操作
- `kern/driver/clock.[ch]`：实现了对时钟控制器8253的初始化操作
- `kern/driver/console.[ch]`：实现了对串口和键盘的中断方式的处理操作

4. 中断处理部分：

- `kern/trap/vectors.S`：包括256个中断服务例程的入口地址和第一步初步处理实现。注意，此文件是由tools/vector.c在编译ucore期间动态生成的
- `kern/trap/trapentry.S`：紧接着第一步初步处理后，进一步完成第二步初步处理；并且有恢复中断上下文的处理，即中断处理完毕后的返回准备工作

- `kern/trap/trap.[ch]`：紧接着第二步初步处理后，继续完成具体的各种中断处理操作

5. 内核调试部分：

- `kern/debug/kdebug.[ch]`：提供源码和二进制对应关系的查询功能，用于显示调用栈关系。其中补全print_stackframe函数是需要完成的练习。其他实现部分不必深究。
- `kern/debug/kmonitor.[ch]`：实现提供动态分析命令的kernel monitor，便于在ucore出现bug或问题后，能够进入kernel monitor中，查看当前调用关系。实现部分不必深究。
- `kern/debug/panic.c | assert.h`：提供了panic函数和assert宏，便于在发现错误后，调用kernel monitor。大家可在编程实验中充分利用assert宏和panic函数，提高查找错误的效率。

6. 公共库部分

- `libs/defs.h`：包含一些无符号整型的缩写定义。
- `libs/x86.h`：一些用GNU C嵌入式汇编实现的C函数（由于使用了inline关键字，所以可以理解为宏）。

7. 工具部分

- `Makefile和function.mk`：指导make完成整个软件项目的编译，清除等工作。
- `sign.c`：一个C语言小程序，是辅助工具，用于生成一个符合规范的硬盘主引导扇区。
- `tools/vector.c`：生成vectors.S，此文件包含了中断向量处理的统一实现。

练习1：理解通过make生成执行文件的过程。

1：操作系统镜像文件ucore.img是如何一步一步生成的?

执行命令 `make v=`，通过阅读其输出的步骤，我们可以得知：

1. make执行将所有的源代码编译成对象文件，并分别链接形成kernel、bootblock文件。
2. 使用dd命令，将生成的两个文件的数据拷贝至img文件中，形成映像文件。

dd命令与cp命令不同，该命令针对于磁盘，功能更加底层；dd 命令主要用来进行数据备份，并且可以在备份的过程中进行格式转换

①首先把C的源代码编译为目标文件（.o文件）

比方说下面这样：

```
1 + cc kern/init/init.c
2 gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/init/init.c -o obj/kern/init/init.o
3 + cc kern/libs/readline.c
4 gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/libs/readline.c -o obj/kern/libs/readline.o
```

②ld命令将这些目标文件转变成可执行文件

比如像下面这样的：

```
1 + ld bin/kernel
2 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
```

③dd命令把bootloader放到ucore.img.count的虚拟硬盘中

```
 1 dd if=/dev/zero of=bin/ucore.img count=10000
 2 10000+0 records in
 3 10000+0 records out
 4 5120000 bytes (5.1 MB) copied, 0.0601803 s, 85.1 MB/s
 5 dd if=bin/bootblock of=bin/ucore.img conv=notrunc
 6 1+0 records in
 7 1+0 records out
 8 512 bytes (512 B) copied, 0.000141238 s, 3.6 MB/s
 9 dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
10 146+1 records in
11 146+1 records out
12 74923 bytes (75 kB) copied, 0.00356787 s, 21.0 MB/s
```

④还生成了两个软件，一个是bootloader，另一个是kernel

生成ucore.img

相关代码：

```
# create ucore.img
#
UCOREIMG := $(call totarget,ucore.img)

$(UCOREIMG): $(kernel) $(bootblock)#可见生成ucore.img需要kernel和bootblock文件
    $(V)dd if=/dev/zero of=$@ count=10000 #从/dev/zero文件中获取10000个block，每一个block为512字节，并且均为空字符，并且输出到目标文件ucore.img中
    $(V)dd if=$(bootblock) of=$@ conv=notrunc #将$(bootblock)拷贝到目标文件ucore.img中，-notruct选项表示不要对数据进行删减
    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc #将$(kernel)拷贝到目标文件ucore.img中，并且seek = 1表示跳过第一个block，输出到第二个块
```

代码详解：

从 MakeFile 里面 可以看出生成 ucore.img 首先需要生成大小为10000字的空间
然后 将 bootblock 和 kernel 依次写入到那块空间之中

利用dd命令使用 bootblock,kernel 文件来生成 ucore.img 文件

1. 创建一个10000块将/dev/zero拷贝进去(获得10000block的空间)
2. 将$(bootblock)拷贝到 ucore.img
3. 将$(kernel)拷贝到 ucore.img (从输出文件开头跳过1个块后再开始拷贝,即第二块)

生成一个叫 ucore.img count 的虚拟硬盘：

首先为目标文件ucore.img定义变量名UCOREIMG，然后指定其依赖文件为kernel和bootblock为变量名的文件。接下来使用dd命令完成ucore.img的生成。

> dd命令及其他使用的参数如下：

- if=文件名：输入文件名，默认为标准输入。即指定源文件。
- of=文件名：输出文件名，默认为标准输出。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。
- /dev/zero：提供无限的空字符(NULL, ASCII NUL, 0x00)。常见用法是产生一个特定大小的空白文件。

- conv=<关键字>，notrunc：不截短输出文件

生成kernel

相关代码

```
# 编译生成bin/kernel所需的文件
$(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))

# 链接生成bin/kernel
kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
    @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
    @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' >
    $(call symfile,kernel)
    $(call create_target,kernel)
```

在`$(call totraget,kernel)`指令中，将`bin/`前缀加到kernel中,生成`bin/kernel`

代码详解

编译获得所需要的.o文件：

```
KCFLAGS += $(addprefix -I,$(KINCLUDE))

$(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))
#该段代码的含义：生成kernel的所有子目录下包含的.s, .c文件所对应的.o文件以及.d文件;指定了若干gcc编译选项，存放在KCFLAGS变量中;
#具体而言，该命令最终生成的文件为obj/kern下子目录里的以stdio, readline, panic,kdebug, kmonitor, clock, console, picirq, intr, trap, vector, trapentry, pmm为前缀的.d, .o文件;
```

kernel的生成需要kernel.ld以及KOBJS变量指向的文件（kernel libs）。其中kernel.ld是链接脚本。

$(V)(LD) (LDFLAGS) -T tools/kernel.ld -o @ $(KOBJS) 将KOBJS指向的文件链接生成kernel文件，-T表示使用kernel.ld替代默认链接脚本。

@(OBJDUMP) -S @ > $(call asmfile,kernel) 表示使用objdump得到kernel反汇编代码，-S表示源代码和汇编代码共同显示，并重定向保存到kernel.asm文件中。

@(OBJDUMP) -t @ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > (call symfile,kernel) 表示将文件的符号表保存到kernel.sym文件中，-t表示打印出文件的符号表表项，|表示使用管道将符号表作为SED的输入，最后保存到kernel.sym文件中。（SED命令用于处理文本文件）

使用`make V=`命令可以查看生成kernel的指令，可以看到所有进行链接的文件

将 `kernel` 下面的所有文件编译生成目标文件再进行链接

```
+ ld bin/kernel
ld -m    elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel
obj/kern/init/init.o obj/kern/libs/stdio.o
obj/kern/libs/readline.o obj/kern/debug/panic.o
obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o
obj/kern/driver/clock.o obj/kern/driver/console.o
obj/kern/driver/picirq.o obj/kern/driver/intr.o
obj/kern/trap/trap.o obj/kern/trap/vectors.o
obj/kern/trap/trapentry.o obj/kern/mm/pmm.o  obj/libs/string.o
obj/libs/printfmt.o
```

可以看到kernel的生成需要链接init.o，stdio.o，readline.o等文件。使用make V=命令也可以看到这些文件由c程序经编译产生.o文件的过程。以init.o的生成为例：

```
+ cc kern/init/init.c
gcc -Ikern/init/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb
-m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/ -
Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
 kern/init/init.c -o obj/kern/init/init.o
```

其他.o文件生成类似。

也就是：

```
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
```

链接所有的目标文件生成elf-i386的内核文件：

```
kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld #表示/bin/kernel文件依赖于tools/kernel.ld文件，并且没有指定生成规则，也就是说如果没有预先准备好kernel.ld，就会在make的时候产生错误

$(kernel): $(KOBJS) #表示kernel文件的生成还依赖于上述生成的obj/libs, obj/kernels下的.o文件

    @echo + ld $@#并且生成规则为使用ld链接器将这些.o文件连接成kernel文件，
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS) #-T表示指定使用kernel.ld来替代默认的链接器脚本
    @$(OBJDUMP) -S $@ > $(call asmfile,kernel) #-S表示将源代码与汇编代码混合展示出来，这部分代码最终保存在kernel.asm文件中
    @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' >
$(call symfile,kernel) #-t表示打印出文件的符号表表项，然后通过管道将带有符号表的反汇编结果作为sed命令的标准输入进行处理，最终将符号表信息保存到kernel.sym文件中
$(call create_target,kernel)
```

> 参数说明：

- -m elf_i386：表示模拟i386的链接器来完成.o文件链接为可执行文件的过程
- -nostdlib ： 表示不链接任何标准库，kernel不需要使用标准库
- -I：添加搜索头文件的路径
- -march=i686：根据目标架构进行优化
- -fno-builtin：不使用c语言的内建函数，避免自定义函数与内建函数冲突的问题
- -fno-PIC：不使用位置无关代码
- -Wall：显示所有警告信息

- -ggdb：专门为gdb产生调试信息
- -m32：生成32位机的机器代码
- -gstabs：以stabs格式输出调试信息，不包括gdb
- -nostdinc: 不搜索默认路径头文件
- -fno-stack-protector: 禁用堆栈保护

生成bootblock

相关代码

```
# 表示将boot/文件夹下的bootasm.S, bootmain.c两个文件编译成相应的.o文件，并且生成依赖文件.d
#两个gcc编译选项含义:
#-nostdinc: 不搜索默认路径头文件;
#-Os: 针对生成代码的大小进行优化，这是因为bootloader的总大小被限制为不大于512-2=510字节;
bootfiles = $(call listf_cc,boot)
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))


bootblock = $(call totarget,bootblock)

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)#可知，bootblock依赖于bootasm.o, bootmain.o文件与sign文件
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
#使用ld链接器将依赖的.o文件链接成bootblock.o
#-N：将代码段和数据段设置为可读可写;
#-e：设置入口;
#-Ttext：设置起始地址为0X7C00;
    @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)#使用objdump将编译结果反汇编出来，保存在bootclock.asm中，-S表示将源代码与汇编代码混合表示
    @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)#使用objcopy将bootblock.o二进制拷贝到bootblock.out
#-S：表示移除符号和重定位信息;
#-O：表示指定输出格式;
    @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)#使用sign程序, 利用bootblock.out生成bootblock;

```

```
24  $(call create_target,bootblock)#利用tools/sing.c生成sign.o
```

由此可知生成bootblock，首先需要生成 `bootasm.o、bootmain.o、sign`

代码详解

① `bootflies` 表示 `boot` 下所有文件，包括

```
1  asm.h
2  bootasm.S
3  bootmain.c
```

需要把 `bootfiles` 中的`bootasm.S`,`bootmain.c` 编译成 `bootasm.o` 和 `bootmain.o`

② 生成 `bootasm.o` 和`bootmain.o` 的代码：

```
1  # 表示将boot/文件夹下的bootasm.S, bootmain.c两个文件编译成相应的.o文件，并且生成依赖文件.d
2  #两个gcc编译选项含义:
3  #-nostdinc: 不搜索默认路径头文件;
4  #-Os: 针对生成代码的大小进行优化，这是因为bootloader的总大小被限制为不大于512-2=510字节;
5  bootfiles = $(call listf_cc,boot)
6  $(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))
```

③ 生成sign的代码：

```
1  # create 'sign' tools
2  $(call add_files_host,tools/sign.c,sign,sign)#利用tools/sing.c生成sign.o
3  $(call create_target_host,sign,sign)#利用sign.o生成sign，至此bootblock所依赖的文件均生成完毕;
```

`sign.c` 是一个C语言小程序，是辅助工具，用于生成一个符合规范的硬盘主引导扇区。

④ 由 `bootasm.o`，`bootmain.o` 和`sign` 生成 `bootblock` ：

> 参数说明：

- -N：将代码段和数据段设置为可读可写
- -e：设置入口
- $^：表示所有依赖对象
- -Ttext：设置起始地址（0X7C00）

- -S：移除符号和重定位信息

`bootblock`的生成依赖于`sign`和`bootfiles`文件，执行`make V=`可以找到`bootblock`生成的部分，发现`bootfiles`包括`bootmain.o`和`bootasm.o`，可以找到`bootmain.o`和`bootasm.os`及`sign`的生成部分。

```
+ cc boot/bootasm.S
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -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/ -march=i686 -fno-builtin -fno-PIC -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
+ ld bin/bootblock
ld -m    elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 496 bytes
//-Os参数：为减小文件大小进行优化
```

首先生成bootmain.o，bootasm.o，sign，然后使用sign生成bootblock。

综上所述，ucore.img的生成过程如下：

- 生成kernel所需要的.o文件，将这些文件链接得到kernel
- 生成bootmain.o，bootasm.o链接得到bootblock.out，使用sign将bootblock.out转换为bootblock
- 创建一个有10000个512字节的块的空文件，将bootblock放入第一个块，kernel放入剩下的块中。

总结

构建 `ucore.img` 时大致进行了以下操作：

- 编译了若干内核文件，构建出内核 `kernel`，
- 生成 `bootblock`引导程序
  - 编译`bootasm.S` `bootmain.c` ，链接 `obj/bootblock.o`
  - 编译 `sign.c` 生成`sign.o`工具
  - 使用`sign.o`工具规范化 `bootblock.o` ，生成`bin/bootblock` 引导扇区
- 生成`ucore.img` 虚拟磁盘

- dd初始化一个大小为`5120000bytes`且内容为0的文件
- dd拷贝`bin/bootblock`引导文件到`ucore.img`的第一个扇区
- dd拷贝`bin/kernel`内核文件到`ucore.img`第二个扇区往后的空间

查询相关资料

1. GCC相关参数

```
-I：添加包含目录

-fno-builtin：只接受以“_builtin”开头的名称的内建函数

-Wall：开启全部警告提示

-ggdb：生成GDB需要的调试信息

-m32：为32位环境生成代码，int、long和指针都是32位

-gstab：生成stab格式的调试信息，仅用于gdb

-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：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

-c：编译源文件但不进行链接

-o：结果的输出文件
```

2. ld相关参数

```
-m elf_i386：使用elf_i386模拟器

-nostdlib：只查找命令行中明确给出的库目录，不查找链接器脚本中给出的（即使链接器脚本是在命令行中给出的）

-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. 生成bootblock和sign工具所需全部OBJ文件的相关命令参数

```
-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

-g：以操作系统原生格式输出调试信息，gdb可以处理这一信息

-O2：进行大部分不以空间换时间的优化
```

4. 链接生成bootblock二进制文件的相关命令参数为

```
-N：将文字和数据部分置为可读写，不将数据section置为与页对齐， 不链接共享库

-e start：将start符号置为程序起始点

-Ttext 0x7C00：链接时将".bss"、".data"或".text"置于绝对地址0x7C00处
```

5. 生成ucore.img的命令相关参数

```
if：输入

of：输出

count=10000：只拷贝输入的10000块

conv=notrunc：不截短输出文件

seek=1：从输出文件开头跳过1个块后再开始复制
```

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?

有以下两句说明符合规范的硬盘主引导扇区特征是最后两个字节为0x55 0xAA同时主引导扇区的大小应为512 字节

通常我们将包含**MBR(主引导记录)引导代码**的扇区称为**主引导扇区**。通常由3部分组成：

- 主引导程序（MBR，占446字节)
- 磁盘分区表项（占4×16个字节，负责说明磁盘上的分区情况)
- 结束标志位（占2个字节，其值为55 AA)。

上题中的 `sign.o` 工具可以规范化 `bootblock.o` ，生成 `bin/bootblock` 引导扇区，因此查看 `sign.c` 部分源代码进行分析：

```
// 文件大小检查，超过510字节则报错，因为最后2个字节要用作结束标志位
    if (st.st_size > 510) {
```

```
        fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
        return -1;
    }
// 写入结束标志位
    buf[510] = 0x55;
    buf[511] = 0xAA;
// 文件大小检查
    if (size != 512) {
        fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);
        return -1;
    }
    fclose(ofp); // 释放文件
```

由上可知，符合规范的硬盘主引导扇区的特征是：

① 扇区总体大小为512字节；

② 512字节的组成：

- 启动代码：不超过466字节；
- 硬盘分区表：不超过64字节；
- 两个字节的结束符

③ 磁盘最后两个字节是0x55 0xAA。

练习2：使用qemu执行并调试lab1中的软件。

1：从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

> [进一步的补充]

```
改写Makefile文件
    debug: $(UCOREIMG)
        $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -parallel stdio -hda $< -serial null"
        $(V)sleep 2
        $(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"
```

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。
为防止qemu在gdb连接后立即开始执行，删除了`tools/gdbinit`中的`continue`行。

启动后的第一条指令和BIOS完成的工作

当计算机启动时，寄存器CS，EIP中的值将被初始化，EIP=0xfff0，而CS的selector=0xf000，base=ffff0000。CS.base+EIP=0xffffffff0，这就是第一条指令的位置。（段寄存器分为可见部分和隐藏部分（描述符高速缓存器），此处CS.base是加电后直接设置的。）而这个位置的指令为一条跳转指令，跳转到0xf000e05b这个位置，这个位置就是BIOS开始的地方。BIOS将会完成硬件自检及初始化，创建中断向量表等工作，并读取第一扇区（主引导扇区或启动扇区）到内存地址0x7c00，接下来的工作交给bootloader完成。

将`lab1/tools/gdbinit`文件修改为如下内容：

```
set architecture i8086
target remote :1234
```

用如下命令调试bootloader第一条指令：

```
 $ cd labcodes_answer/lab1_result/
 $ make lab1-mon
```

接下来在该目录下`make debug`，弹出qemu及gdb调试窗口，可以查看第一条指令，并使用`si`单步跟踪BIOS的执行，使用x/i命令可以查看执行的具体汇编指令。

```
0x0000fff0 in ?? ()
(gdb) si
0x0000e05b in ?? ()
(gdb) x/i 0xffffffff0
   0xffffffff0:  ljmp   $0x3630,$0xf000e05b
```

2：在初始化位置0x7c00设置实地址断点,测试断点正常

查看`lab1init`文件，内容为调试命令

```
file bin/kernel                         //加载
target remote :1234                     //链接qemu
set architecture i8086                  //设定架构为i8086
b *0x7c00                               //断点设置在0x7c00
continue
x /2i $pc                               //查看两条指令
```

使用`make lab1-mon`命令进入调试，终端显示如下：

```
breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:    cli
   0x7c01:    cld
(gdb)
```

在0x7c00处设置断点正常，可以从这里开始正常调试`bootloader`。

3：从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较

从0x7c00开始x/15i显示接下来执行的15条指令，指令如下：

```
(gdb) x/15i 0x7c00
=> 0x7c00:    cli
   0x7c01:    cld
   0x7c02:    xor    %eax,%eax
   0x7c04:    mov    %eax,%ds
   0x7c06:    mov    %eax,%es
   0x7c08:    mov    %eax,%ss
   0x7c0a:    in     $0x64,%al
   0x7c0c:    test   $0x2,%al
   0x7c0e:    jne    0x7c0a
   0x7c10:    mov    $0xd1,%al
   0x7c12:    out    %al,$0x64
   0x7c14:    in     $0x64,%al
   0x7c16:    test   $0x2,%al
   0x7c18:    jne    0x7c14
   0x7c1a:    mov    $0xdf,%al
```

`bootasm.S`中可以找到和以上相同的汇编代码：

```
.code16                                             # Assemble for 16-bit mode
    cli                                             # Disable interrupts
    cld                                             # String operations increment
    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number zero
    movw %ax, %ds                                   # -> Data Segment
```

```
    movw %ax, %es                                   # -> Extra Segment
    movw %ax, %ss                                   # -> Stack Segment
seta20.1:
    inb $0x64, %al                                  # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.1
    movb $0xd1, %al                                 # 0xd1 -> port 0x64
    outb %al, $0x64                                 # 0xd1 means: write data to 8042's P2 port
seta20.2:
    inb $0x64, %al                                  # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.2
    movb $0xdf, %al                                 # 0xdf -> port 0x60
    ......
```

在obj文件夹下找到`bootblock.asm`，其中的代码同样和以上代码相同：

```
code16                                              # Assemble for 16-bit mode
    cli                                             # Disable interrupts
    7c00:    fa                     cli
    cld                                             # String operations increment
    7c01:    fc                     cld

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number zero
    7c02:    31 c0                  xor    %eax,%eax
    movw %ax, %ds                                   # -> Data Segment
    7c04:    8e d8                  mov    %eax,%ds
    movw %ax, %es                                   # -> Extra Segment
    7c06:    8e c0                  mov    %eax,%es
```

```
    movw %ax, %ss                                   # -> Stack Segment
    7c08:    8e d0                  mov    %eax,%ss
    ......
```

即0x7c00处开始执行的汇编指令与`bootasm.S`及`bootblock.asm`中的代码是相同的。

4：找一个bootloader或内核中的代码位置，设置断点并进行测试。

在0x7c14处设置断点并进行调试：

```
(gdb) b *0x7c14
Breakpoint 2 at 0x7c14
(gdb) c
Continuing.

Breakpoint 2, 0x00007c14 in ?? ()
(gdb) x/5i $pc
=> 0x7c14:    in     $0x64,%al
   0x7c16:    test   $0x2,%al
   0x7c18:    jne    0x7c14
   0x7c1a:    mov    $0xdf,%al
   0x7c1c:    out    %al,$0x60
(gdb)
```

练习3：分析bootloader进入保护模式的过程

相关资料

打开文件`bootasm.S`,分析bootloader：

1. 关闭中断，将各个段寄存器重置

```
# start address should be 0:7c00, in real mode, the beginning address of the running bootloader
.globl start
start:
.code16 # Assemble for 16-bit mode
    cli # Disable interrupts
    cld # String operations increment

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax # Segment number zero
    movw %ax, %ds # -> Data Segment
    movw %ax, %es # -> Extra Segment
    movw %ax, %ss # -> Stack Segment
```

关闭中断，清理环境：将flag和寄存器AX,DS,ES,SS清零

2. 开启A20

- 为什么会出现A20?

Intel早期的8086 CPU提供了20根地址线，但寄存器只有16位，因此采用段寄存器值 << 4 + 段内偏移值的方法来访问到所有内存，但按这种方式来计算出的地址的最大值为1088KB，超过20根地址线所能表示的范围，会发生“回卷”（和整数溢出有点类似）。但下一代的基于Intel 80286 CPU的计算机系统提供了24根地址线，当CPU计算出的地址超过1MB时便不会发生回卷，而这就造成了向下不兼容。为了保持完全的向下兼容性，IBM在计算机系统上加个硬件逻辑来模仿早期的回绕特征，而这就是A20 Gate.

- A20的机制

A20 Gate的方法是把A20地址线控制和键盘控制器的一个输出进行AND操作，这样来控制A20地址线的打开（使能）和关闭（屏蔽\禁止）。一开始时A20地址线控制是被屏蔽的（总为0），直到系统软件通过一定的IO操作去打开它。当A20 地址线控制禁止时，程序就像在 8086 中运行，软件可访问的物理内存空间不能超过1MB，且无法发挥Intel 80386以上级别的32位CPU的4GB内存管理能力。而开启A20,通过将键盘控制器上的A20线置于高电位1，使得全部32条地址线可用，保护模式下 A20 地址线控制打开，此时才可以访问4G内存

开启方式：

1.等待8042 Input buffer为空；

2.发送Write 8042 Output Port （P2）命令到8042 Input buffer；

3.等待8042 Input buffer为空；

4.将8042 Output Port（P2）得到字节的第2位置1，然后写入8042 Input buffer；

代码：

```
# Enable A20:
    # For backwards compatibility with the earliest PCs, physical
    # address line 20 is tied low, so that addresses higher than
    # 1MB wrap around to zero by default. This code undoes this.
seta20.1:
    inb $0x64, %al # 读取状态寄存器,等待8042键盘控制器闲置
    testb $0x2, %al # 读取到2则表明缓冲区中没有数据
    jnz seta20.1 # 如果缓冲区有数据就继续循环
# 接下来往0x64写入0xd1，表示请求修改8042的端口P2
    movb $0xd1, %al # 0xd1表示写输出端口命令，参数随后通过0x64端口写入
    outb %al, $0x64 # 0xd1 means: write data to 8042's P2 port

seta20.2:
    inb $0x64, %al # 等待8042键盘控制器闲置
    testb $0x2, %al
    jnz seta20.2

# 往0x60端口写入0xdf，表示将端口P2的位1（A20选通使能）置为1，开启A20
    movb $0xdf, %al # 0xdf -> port 0x60
    outb %al, $0x60 # 通过0x60写入数据 0xdf = 11011111, 意味着将A20置1
```

至此，A20开启，CPU进入保护模式之后可以**充分使用32位4G内存的寻址能力**

3. GDT表

什么是GDT?

GDT(Global Descriptor Table, 全局描述表)同实模式一样，在保护模式下，对内存的访问仍采用短地址加偏移地址的方式。其内存的管理方式有两种，段模式和页模式。在保护模式下，对于一个段的描述包括：Base Address（基址），Limit（段的最大长度），Access（权限），这三个数据加在一起被放在一个 64 bit 的数据结构中，被称为段描述符。而由于寄存器为 16 bit，很明显，我们无法直接通过16bit 长度的寄存器来直接使用 64 bit 的段描述符。而对此的解决方案便是将这些段描述符放入

一个全局数组中，将段寄存器中的值作为下标索引(段寄存器中的高 13 bit 的内容作为索引)来间接引用。而这个全局数组便是 **GDT**。

如何初始化GDT表?

```
# Bootstrap GDT
.p2align 2 # force 4 byte alignment向后移动位置计数器置为4字节的倍数 为了内存对齐
gdt:
    SEG_NULLASM # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) #可读可执行
    SEG_ASM(STA_W, 0x0, 0xffffffff) #可写但不可执行
gdtdesc:
    .word 0x17 # sizeof(gdt) - 1
    .long gdt # address gdt
#.long后面的参数为gdt运行时生成的值，即gdt表的地址
```

① GDT中的第一项描述符设置为空。

② GDT中的第二项描述符为**代码段**使用，设置属性为可读写可执行。

③ GDT中的第三项描述符为**数据段**使用，设置属性为可读写。

GDT的结构：全局描述符号的第一段为空段，这是intel的规定。后两个段是数据段和代码段。

怎么加载GDT表?

一个简单的GDT表和其描述符已经静态储存在引导区中，载入即可

```
    lgdt gdtdesc
```

4. 如何使能和进入保护模式?

x86 引入了几个新的**控制寄存器 (Control Registers)** cr0 cr1... cr7，每个长 32 位。这其中的某些寄存器的某些位被用来控制 CPU 的**工作模式**，其中 cr0 的最低位，就是用来控制 CPU 是否处于保护模式的。因为控制寄存器不能直接拿来运算，所以需要通过通用寄存器来进行一次存取，设置 cr0 最低位为1之后就已经进入**保护模式**。但是由于现代 CPU 的一些特性 （乱序执行和分支预测等），在转到保护模式之后 CPU 可能仍然在跑着**实模式**下的代码，这显然会造成一些问题。因此必须强制 CPU 清空一次缓冲，最有效的方法就是进行一次**long jump** 。

```
# GDT从实模式切换到保护模式， 使用GDT（全局描述表，Global Descriptor Table）和段变换，
    # 使得虚拟地址和物理地址相同，这样，切换过程中不会改变有效内存映射。
    # 将CR0的保护允许位PE(Protedted Enable)置1，开启保护模式
```

```
    lgdt gdtdesc #加载GDT表

    # 将cr0寄存器PE置1，开启保护模式
    movl %cr0, %eax #加载cr0到eax
    orl $CR0_PE_ON, %eax #将eax的第0位置为1
    movl %eax, %cr0 #将cr0的第0位置为1

    #跳转到处于32位代码块中的下一条指令，重装CS和EIP
    # Switches processor into 32-bit mode.
    ljmp $PROT_MODE_CSEG, $protcseg# # 通过长跳转更新cs的基地址

# 设置段寄存器，建立堆栈
.code32 # 32-bit模式汇编代码
protcseg:
#重装DS、ES等段寄存器
    movw $PROT_MODE_DSEG, %ax # Our data segmentselector
    movw %ax, %ds # -> DS: Data Segment
    movw %ax, %es # -> ES: Extra Segment
    movw %ax, %fs # -> FS
    movw %ax, %gs # -> GS
    movw %ax, %ss # -> SS: Stack Segment

#设置栈指针并调用C代码,进入保护模式完成,,建立堆栈,转到bootmain
#栈区是0~start（0x7c00）
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain

# If bootmain returns (it shouldn't), loop.
spin:
    jmp spin
```

如何进入保护模式?

将%cr0寄存器置1。(通过将cr0寄存器PE位置1便开启了保护模式)

bootloader 从**实模式**进入**保护模式**的过程：

1. 在开启A20之后，加载了 `GDT` 全局描述符表，它是被静态储存在引导区中的，载入即可。接着，将cr0 寄存器的`bit 0` 置为 1 ，标志着从**实模式**转换到**保护模式**。
2. 由于我们无法直接或间接 mov 一个数据到 cs 寄存器中，而刚刚开启保护模式时，cs 的影子寄存器还是实模式下的值，所以需要告诉 CPU 加载新的段信息。**长跳转**可以设置 cs 寄存器，CPU 发现了cr0 寄存器第 0 位的值是 1，就会按 GDTR 的指示找到全局描述符表GDT，然后根据索引值 把新的段描述符信息加载

到 cs 影子寄存器，当然前提是进行了一系列合法的检查。所以使用一个长跳转 `ljmp $PROT_MODE_CSEG, $protcseg` 以更新 cs 基地址，至此CPU真正进入了保护模式，拥有了32 位的处理能力。
3. 进入保护模式后，设置ds，es，fs，gs，ss段寄存器，建立堆栈（0~0x7c00），最后进入`bootmain` 函数。

- `bootasm.S`

这段代码是一个操作系统的启动程序，它将处理器从实模式（real mode）切换到保护模式（protected mode），然后跳转到C语言的代码中执行。

第一行代码 `#include <asm.h>` 是包含一个头文件，该头文件可能包含与操作系统启动相关的宏定义和函数声明。

接下来的一些 `.set` 指令定义了一些常量，如 `PROT_MODE_CSEG` 和 `CR0_PE_ON`，用于后续的代码中作为标识符使用。

`start` 是启动程序的入口点，它使用 `.globl` 指令声明为全局符号，以便其他代码可以引用它。

`.code16` 指令将汇编器的模式切换为16位模式，下面的指令将在该模式下汇编。

`cli` 和 `cld` 指令分别禁用中断和设定字符串操作方向标志。接下来一段代码用于设置重要的数据段寄存器 `DS`、`ES`、`SS` 的值为0，以便在切换到保护模式后正确访问内存。

接下来的代码使用 8042 芯片控制 A20 信号线，这个信号线可以让处理器访问超过 1MB 的内存地址。这段代码的作用是将 A20 信号线设置为开启状态，以便后续可以使用 32 位地址空间。

`lgdt` 指令加载全局描述符表（Global Descriptor Table，GDT），并使用 `orl` 指令将控制寄存器 CR0 的 PE 位（保护模式启用标志）设置为1，从而使处理器切换到保护模式。然后使用 `ljmp` 指令跳转到代码段选择子 `PROT_MODE_CSEG` 所指向的代码段，使处理器进入32位模式。

在32位模式下，将数据段寄存器 `DS`、`ES`、`FS`、`GS` 和堆栈段寄存器 `SS` 的值设置为 `PROT_MODE_DSEG`，然后使用 `movl` 和 `call` 指令将栈指针（ESP）设置为 `start` 的地址，调用 `bootmain` 函数执行操作系统的主要功能。如果 `bootmain` 函数返回（尽管这不应该发生），则使用 `jmp` 指令将处理器跳转到 `spin` 标号处，以便让处理器一直循环执行该指令，以保证操作系统的正常运行。

最后，定义了一个 GDT 的描述符，它包含一个 null 段和两个可执行和可写的代码段和数据段，以便操作系统在32位保护模式下正确运行。`gdtdesc` 指令将 GDT 的大小和起始地址

```
1 #include <asm.h>
```

```
#asm.h头文件中包含了一些宏定义，用于定义gdt，gdt是保护模式使用的全局段描述符表，其中存储着段描述符。
# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.
# 启动CPU：切换到32位保护模式，进入C语言。
# BIOS将从硬盘的第一个扇区加载此代码到物理地址0x7c00的内存中，并在实模式下启动，
# 此时%cs=0 %ip=7c00。
#此段注释说明了要完成的目的：启动保护模式，转入C函数。
#这里正好说了一下bootasm.S文件的作用。计算机加电后，由BIOS将bootasm.S生成的可执行代码从硬盘的第一个扇区复制到内存中的物理地址0x7c00处,并开始执行。
#此时系统处于实模式。可用内存不多于1M。
.set PROT_MODE_CSEG,        0x8                     # kernel code segment selector内核代码段选择器
.set PROT_MODE_DSEG,        0x10                    # kernel data segment selector内核数据段选择器
#这两个段选择子的作用其实是提供了gdt中代码段和数据段的索引
.set CR0_PE_ON,             0x1                     # protected mode enable flag保护模式启用标志
#这个变量是开启A20地址线的标志，为1是开启保护模式

# start address should be 0:7c00, in real mode, the beginning address of the running bootloader
# 启动地址应为0:7c00，在实模式下，运行引导加载程序的开始地址
.globl start
start:
#这两行代码相当于定义了C语言中的main函数，start就相当于main，BIOS调用程序时，从这里开始执行
.code16                                             # Assemble for 16-bit mode指定16位模式
    cli                                             # Disable interrupts关中断
    cld                                             # String operations increment字符串操作自增
#关中断，设置字符串操作是递增方向。cld的作用是将direct flag标志位清零，这意味着自动增加源索引和目标索引的指令(如MOVS)将同时增加它们。

    # Set up the important data segment registers (DS, ES, SS).
```

```
    # 设置重要的数据段寄存器（DS、ES、SS）
    xorw %ax, %ax                                       # Segment number zero段号为0
#ax寄存器就是eax寄存器的低十六位，使用xorw清零ax，效果相当于movw $0, %ax。 但是好像xorw性能好一些，google了一下没有得到好答案
    movw %ax, %ds                                       # -> Data Segment数据段
    movw %ax, %es                                       # -> Extra Segment扩展段
    movw %ax, %ss                                       # -> Stack Segment栈段
#将段选择子清零
    # Enable A20:
    #  For backwards compatibility with the earliest PCs, physical
    #  address line 20 is tied low, so that addresses higher than
    #  1MB wrap around to zero by default. This code undoes this.
    # 启用A20地址线：
    # 为了与最早的PC兼容，物理地址线20被绑定到低电平，
    # 这样默认情况下大于1MB的地址会被截断为0。此代码撤消了这一点。
#由于需要兼容早期pc，物理地址的第20位绑定为0，所以高于1MB的地址又回到了0x00000.
#好了，激活A20后，就可以访问所有4G内存了，就可以使用保护模式了。
#怎么激活呢，由于历史原因A20地址位由键盘控制器芯片8042管理。所以要给8042发命令激活A20
#8042有两个IO端口：0x60和0x64， 激活流程位： 发送0xd1命令到0x64端口 --> 发送0xdf到0x60，done!
seta20.1:
    inb $0x64, %al                                      # Wait for not busy(8042 input buffer empty).等待不忙（8042输入缓冲区为空）。
    testb $0x2, %al
    jnz seta20.1
#发送命令之前，要等待键盘输入缓冲区为空，这通过8042的状态寄存器的第2bit来观察，而状态寄存器的值可以读0x64端口得到。
#上面的指令的意思就是，如果状态寄存器的第2位为1，就跳到seta20.1符号处执行，知道第2位为0，代表缓冲区为空
    movb $0xd1, %al                                     # 0xd1 -> port 0x64  0xd1 -> 端口0x64
    outb %al, $0x64                                     # 0xd1 means: write data to 8042's P2 port  0xd1表示：将数据写入8042的P2端口
#发送0xd1到0x64端口
```

```
seta20.2:
    inb $0x64, %al                                          # Wait for not busy(8042 input buffer empty).  等待不忙（8042输入缓冲区为空）。
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al                                         # 0xdf -> port 0x60  0xdf -> 端口0x60
    outb %al, $0x60                                         # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1  0xdf = 11011111，表示将P2的A20位（第1位）设置为1
#到此，A20激活完成
    # Switch from real to protected mode, using a bootstrap GDT
    # and segment translation that makes virtual addresses
    # identical to physical addresses, so that the
    # effective memory map does not change during the switch.
    # 切换到保护模式，使用引导GDT和段转换，使得虚拟地址与物理地址相同，
    # 以便在切换过程中有效的内存映射不会改变。
#转入保护模式，这里需要指定一个临时的GDT，来翻译逻辑地址。这里使用的GDT通过gdtdesc段定义。它翻译得到的物理地址和虚拟地址相同，所以转换过程中内存映射不会改变
    lgdt gdtdesc
#载入gdt
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax#启用保护模式
    movl %eax, %cr0
#打开保护模式标志位，相当于按下了保护模式的开关。cr0寄存器的第0位就是这个开关，通过CR0_PE_ON或cr0寄存器，将第0位置1
    # Jump to next instruction, but in 32-bit code segment.
    # Switches processor into 32-bit mode.
    # 跳转到下一条指令，但在32位代码段中
    # 切换处理器为32位模式。
    ljmp $PROT_MODE_CSEG, $protcseg
#由于上面的代码已经打开了保护模式了，所以这里要使用逻辑地址，而不是之前实模式的地址了。
#这里用到了PROT_MODE_CSEG，他的值是0x8。根据段选择子的格式定义，0x8就翻译成：
#              INDEX                TI      CPL
#0000 0000 1       00       0
#INDEX代表GDT中的索引，TI代表使用GDTR中的GDT，  CPL代表处于特权级。
```

```
#PROT_MODE_CSEG选择子选择了GDT中的第1个段描述符。这里使用的gdt就是变量gdt。下面可以看到gdt的第1个段描述符的基地址是0x0000,所以经过映射后和转换前的内存映射的物理地址一样。
.code32# 以32位模式汇编
      # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    # 设置保护模式数据段寄存器
    movw $PROT_MODE_DSEG, %ax                       # Our data segment selector 我们的数据段选择器
    movw %ax, %ds                                   # -> DS: Data Segment数据段
    movw %ax, %es                                   # -> ES: Extra Segment附加段
    movw %ax, %fs                                   # -> FS
    movw %ax, %gs                                   # -> GS
    movw %ax, %ss                                   # -> SS: Stack Segment栈段
#重新初始化各个段寄存器。
    # Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
    # 设置堆栈指针并调用C程序。堆栈区域是从0- start(0x7c00)
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain
#栈顶设定在start处，也就是地址0x7c00处，call函数将返回地址入栈，将控制权交给bootmain
    # If bootmain returns (it shouldn't), loop.
    # 如果bootmain返回(它不应该)，则循环
spin:
    jmp spin

# Bootstrap GDT 引导GDT
.p2align 2                                          # force 4 byte alignment 强制4字节对齐
gdt:
    SEG_NULLASM                                     # null seg 空段
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)           # code seg for bootloader and kernel 引导加载程序和内核的代码段
    SEG_ASM(STA_W, 0x0, 0xffffffff)                 # data seg for bootloader and kernel 引导加载程序和内核的数据段

gdtdesc:
```

```
119     .word 0x17                                              #
    sizeof(gdt) - 1
120     .long gdt                                               #
    address gdt gdt的地址
121
```

流程

从`%cs=0 $pc=0x7c00`，进入后

1. 首先清理环境：包括将flag置0和将段寄存器置0

```
1   .code16
2       cli
3       cld
4       xorw %ax, %ax
5       movw %ax, %ds
6       movw %ax, %es
7       movw %ax, %ss
```

2. 开启A20：通过将键盘控制器上的A20线置于高电位，全部32条地址线可用，可以访问4G的内存空间。

```
1   seta20.1:               # 等待8042键盘控制器不忙
2       inb $0x64, %al      #
3       testb $0x2, %al     #
4       jnz seta20.1        #
5
6       movb $0xd1, %al     # 发送写8042输出端口的指令
7       outb %al, $0x64     #
8
9   seta20.1:               # 等待8042键盘控制器不忙
10      inb $0x64, %al      #
11      testb $0x2, %al     #
12      jnz seta20.1        #
13
14      movb $0xdf, %al     # 打开A20
15      outb %al, $0x60     #
```

3. 初始化GDT表：一个简单的GDT表和其描述符已经静态储存在引导区中，载入即可

```
1       lgdt gdtdesc
```

4. 进入保护模式：通过将cr0寄存器PE位置1便开启了保护模式

```
        movl %cr0, %eax
        orl $CR0_PE_ON, %eax
        movl %eax, %cr0
```

5. 通过长跳转更新cs的基地址

```
     ljmp $PROT_MODE_CSEG, $protcseg
    .code32
    protcseg:
```

6. 设置段寄存器，并建立堆栈

```
        movw $PROT_MODE_DSEG, %ax
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %fs
        movw %ax, %gs
        movw %ax, %ss
        movl $0x0, %ebp
        movl $start, %esp
```

7. 转到保护模式完成，进入boot主方法

```
        call bootmain
```

练习4：分析bootloader加载ELF格式的OS的过程

打开文件`bootmain.c`分析代码：

阅读其最开始的注释：

```
/*
* 这是一个非常简单的引导加载程序(bootloader)，其唯一的工作是从第一个IDE硬盘引导ELF内核映像。
*
* 磁盘格式：
* * 这个程序（bootasm.S和bootmain.c）是bootloader，
* 它应该被存储在磁盘的第一个扇区内；第二个扇区之后存储的是映像，必须是ELF格式的。
*
* BOOT步骤：
```

```
 9  * * 当 CPU 启动时，它会将 BIOS 加载到内存中并执行它。
10  *
11  * * BIOS初始化设备、设置中断程序，并将bootloader的第一个扇区（如硬盘）
    读入内存并跳转到这一部分。
12  *
13  * * 假设bootloader存储在硬盘的第一个扇区中，那么它就开始工作了
14  *
15  * * bootasm.S中的代码先开始执行，它开启保护态，并设置C代码能够运行的栈；
16  * * 最后调用本文件中的bootmain()函数；
17  * * bootmain()函数将kernel读入内存并跳转到它。
18  */
```

1.硬盘扇区读取

bootloader访问硬盘的方式是LBA模式的PIO（Program IO）方式，所有的IO操作通过CPU访问硬盘的IO地址寄存器完成（LBA模式是硬盘的逻辑块寻址模式，硬盘的IDE会把柱面，磁头等参数形成逻辑地址转换为实际物理地址）。一般主板有2个IDE通道，每个通道可以接2个IDE硬盘。访问第一个硬盘的扇区可设置IO地址寄存器0x1f0-0x1f7实现，具体参数见下表。

> 磁盘IO地址和对应功能

IO地址	功能
0x1f0	读数据，当0x1f7不为忙状态时，可以读。
0x1f2	要读写的扇区数，每次读写前，需要表明要读写几个扇区。最小是1个扇区
0x1f3	如果是LBA模式，就是LBA参数的0-7位
0x1f4	如果是LBA模式，就是LBA参数的8-15位
0x1f5	如果是LBA模式，就是LBA参数的16-23位
0x1f6	第0~3位：如果是LBA模式就是24-27位 第4位：为0主盘，为1从盘；第6位：为1=LBA模式，0 = CHS模式（Cylinder/Head/Sector）；第7位和第5位必须为1
0x1f7	状态和命令寄存器。操作时先给命令，再读取，如果不是忙状态就从0x1f0端口读数据

其中0x1f7的状态寄存器第7位表示控制器是否忙碌，第6位表示磁盘驱动器是否准备好了，在读取硬盘前需要等待硬盘就绪，然后通过设置相关参数，对0x1f7写入读数据的命令后，从0x1f0读取数据。

bootloader关于读取硬盘扇区的函数有两个，分别是readsect与readseg，其中readseg需要调用readsect。

- **readsect**:

分析main函数可得：首先是由readseg函数读取硬盘扇区，而readseg函数则循环调用了真正读取硬盘扇区的函数readsect来每次读出一个扇区。

```
void bootmain(void) {
    // 从磁盘读出kernel映像的第一页，得到ELF头
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
    ...
}
```

readsect函数有两个参数，其中一个为存放读取到数据的位置，另一个为LBA参数，通过设置参数读取一个硬盘扇区的数据到指定位置，具体的实现如下：

```
static void waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)          //等待就绪(0x40=01000000)
}//检查 0x1F7 端口 的 第7位为0则硬盘不忙
//dst：将数据读取到哪个位置，secno：LBA参数
static void    readsect(void *dst, uint32_t secno) {
    // 等待就绪
    waitdisk();
    //设置参数
        /*
    0x1F2 写入扇区数
    0x1F3 LBA 0~7
    0x1F4 LBA 8~15
    0x1F5 LBA 16~23
    0x1F6 LBA 7~4位为1110 表示LBA模式 24~27
    0x1F7 0x20 读命令
    一次从 0x1F0 读入 2个字 4个字节 读入512字节 需要 128次
    */
    outb(0x1F2, 1);                                 // 读1个扇区
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);  //0xE0=11100000
```

```
    outb(0x1F7, 0x20);                          // 读取扇区命令：0x20
    // 等待就绪
    waitdisk();
    // 读取扇区
    insl(0x1F0, dst, SECTSIZE / 4);             // 通过0x1F0端口读取数据至dst（insl：一次读四个字节）
}
```

读一个硬盘扇区流程:

① 等待磁盘准备好

② 发出读取扇区的命令

③ 等待磁盘准备好

④ 把磁盘扇区数据读到指定内存

- **readseg:**

readseg函数有三个参数，va是一个虚拟内存位置，读取到的数据从va的位置开始存放，count表示读取数据的字节数，offset表示偏移，在开始读取前va减去offset%sectsize，这表明存放数据的虚拟地址也是按照扇区大小对齐的，开始读的扇区号为offset/sectsize+1，表示从偏移offset个字节处所在的扇区开始读，并跳过主引导扇区，如offset=700，512<700<1024，则从第2个扇区开始读。每次读一个扇区的数据并更新va，直到已经读取了超过需要读取的字节数。（可能会多读取一些数据，但没有影响）

具体实现如下:

```
/* *
* readseg
* 从内核的offset处读count个字节到虚拟地址va中。
* 复制的内容可能比count个字节多。
* */
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;
    // 向下舍入到扇区边
    va -= offset % SECTSIZE;

    // 从字节转换到扇区；ELF文件从1扇区开始，因为0扇区被引导占用
    uint32_t secno = (offset / SECTSIZE) + 1;

    // 如果这个函数太慢，我们可以同时读多个扇区。
```

```
    // 我们在写到内存时会比请求的更多，但这没有关系
    // 我们是以内存递增次序加载的
    for (; va < end_va; va += SECTSIZE, secno ++) {
        readsect((void *)va, secno);
    }
}
```

2.ELF格式的OS

ELF(Executable and linking format)文件格式是Linux系统下的一种常用目标文件(object file)格式，有三种主要类型，在本实验中使用的类型为用于执行的可执行文件(executable file)，用于提供程序的进程映像，加载的内存执行。在ELF文件的开始处有一个ELF header，用于描述ELF文件的组织。在libs目录下打开elf.h，可以找到ELF header的定义：

```
#ifndef __LIBS_ELF_H__
#define __LIBS_ELF_H__
#include <defs.h>
#define ELF_MAGIC 0x464C457FU // 小端格式下"\x7FELF"
/* 文件头 */
struct elfhdr {
    uint32_t e_magic; // 必须等于ELF_MAGIC魔数
    uint8_t e_elf[12]; // 12 字节，每字节对应意义如下：
    // 0 : 1 = 32 位程序；2 = 64 位程序
    // 1 : 数据编码方式，0 = 无效；1 = 小端模式；2 = 大端模式
    // 2 : 只是版本，固定为 0x1
    // 3 : 目标操作系统架构
    // 4 : 目标操作系统版本
    // 5 ~ 11 : 固定为 0
    uint16_t e_type; // 1=可重定位， 2=可执行， 3=共享对象， 4=核心镜像
    uint16_t e_machine; // 3=x86， 4=68K， etc.
    uint32_t e_version; // 文件版本，总为1
    uint32_t e_entry; // 程序入口地址（如果可执行）
    uint32_t e_phoff; // 程序段表头相对elfhdr偏移位置
    uint32_t e_shoff; // 节头表相对elfhdr偏移量
    uint32_t e_flags; // 处理器特定标志，通常为0
    uint16_t e_ehsize; // 这个ELF头的大小
    uint16_t e_phentsize; // 程序头部长度
    uint16_t e_phnum; // 段个数 也就是表中入口数目
    uint16_t e_shentsize; // 节头部长度
    uint16_t e_shnum; // 节头部个数
    uint16_t e_shstrndx; // 节头部字符索引
};
```

```
/* 程序段表头 */
struct proghdr {
    uint32_t p_type; // 段类型
    // 1 PT_LOAD : 可载入的段
    // 2 PT_DYNAMIC : 动态链接信息
    // 3 PT_INTERP : 指定要作为解释程序调用的以空字符结尾的路径名的位置和大小
    // 4 PT_NOTE : 指定辅助信息的位置和大小
    // 5 PT_SHLIB : 保留类型，但具有未指定的语义
    // 6 PT_PHDR : 指定程序头表在文件及程序内存映像中的位置和大小
    // 7 PT_TLS : 指定线程局部存储模板
    uint32_t p_offset; // 段相对文件头的偏移值
    uint32_t p_va; // 段的第一个字节将被放到内存中的虚拟地址
    uint32_t p_pa; //段的第一个字节在内存中的物理地址
    uint32_t p_filesz; //段在文件中的长度
    uint32_t p_memsz; // 段在内存映像中占用的字节数
    uint32_t p_flags; //可读可写可执行标志位。
    uint32_t p_align; //段在文件及内存的对齐方式
};

#endif /* !__LIBS_ELF_H__ */
```

其中，e_magic成员变量必须等于ELF_MAGIC幻数，用于检验是否为合法的可执行文件。e_entry成员变量指定了该可执行文件程序入口的虚拟地址，而 e_phoff成员变量表示program header表的位置偏移量，可以根据它查找到program header。program header描述与程序执行直接相关的目标文件结构信息，用来在文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。bootloader通过ELF Header中的e_phoff在program header表中找到program header，将相应的段，读到内存中来。program header结构如下：

```
struct proghdr {
    uint32_t p_type;   // 段的类型，说明本段为代码段或数据段或其他
    uint32_t p_offset; // 段相对于文件头的偏移
    uint32_t p_va;     // 段的内容被放到内存中的虚拟地址
    uint32_t p_pa;     // physical address, not used
    uint32_t p_filesz; // size of segment in file
    uint32_t p_memsz;  // 段在内存映像中占用的字节数
    uint32_t p_flags;  // read/write/execute bits
    uint32_t p_align;  // required alignment, invariably hardware page size
};
```

bootmain.c文件中，主函数：

加载的过程为:

```
/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // 从磁盘读出kernel映像的第一页，得到ELF头
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
    // 比对ELF的magic number来判断读入的ELF文件是否正确,即判断是不是一个合法的ELF的文件
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }
    struct proghdr *ph, *eph;
    // load each program segment (ignores ph flags)
    // 先将描述表的头地址存在ph
    // ELF头部有描述ELF文件应加载到内存什么位置的描述表
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
// 按照描述表将ELF文件中每个段都加载到特定的内存地址
    for (; ph < eph; ph ++) {
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
    }

    //根据ELF头部储存的入口信息，找到内核的入口，跳转至ELF文件的程序入口点(entrypoint)。
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

bad: // ELF文件不合法
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    /* do nothing */
    while (1);
}
```

加载ELF格式的OS的大致流程为:

1. 读取磁盘上的1页（8个扇区），得到ELF头部
2. 校验e_magic字段,判断是否为合法ELF文件
3. 从ELF头中获得程序头的位置，从中获得每段的信息
4. 分别读取每段的信息，根据偏移量分别把程序段的数据读取到内存中。
5. 根据ELF头部储存的入口信息找到内核的入口并跳转

3.bootloader加载ELF格式的OS

了解了硬盘扇区的读取和ELF格式的相关内容，就可以分析bootmain.c中bootloader是如何加载ELF格式的OS的了。

进入bootmain中后，首先读取8个扇区的内容（从1号扇区开始），使ELFHDR指针指向ELF文件的头部，即指向elf header。然后通过幻数检验该文件是否是正确的ELF文件，如果是则继续后续的操作。读取扇区后，定义了program header类型的指针ph和eph，并通过e_phoff找到program header的位置，使ph指向program header，根据e_phnum使eph指向program header结束的位置。然后根据program header中的信息，调用readseg函数，通过ph中程序段的大小，相对于文件头的偏移，将各个程序段的内容放入指定的虚拟地址处，完成OS的加载。最后，根据ELFHDR的入口信息，进入并开始执行内核程序。

```
void
bootmain(void) {
    // 读取8个扇区
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
    // 检验ELF是否有效
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }
    struct proghdr *ph, *eph;
    // 加载程序段
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph ++) {
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
    }
    // 根据ELFHDR中的入口信息，进入内核程序，启动OS
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    /* do nothing */
    while (1);
}
```

练习5：实现函数调用堆栈跟踪函数

完成`kdebug.c`中函数`print_stackframe`的实现，可以通过函数`print_stackframe`来跟踪函数调用堆栈中记录的返回地址。

1. **函数堆栈的原理**

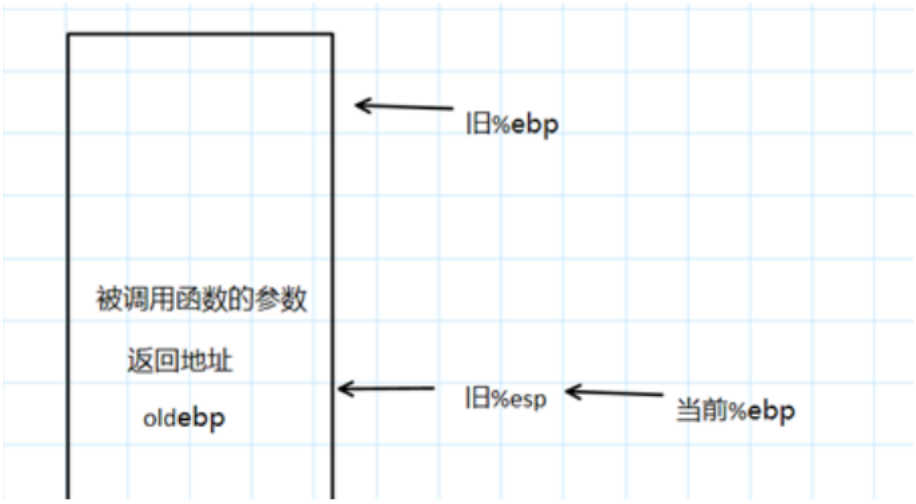
一个函数调用动作可分解为零到多个 PUSH指令（用于参数入栈）和一个 CALL 指令。CALL 指令内部其实还暗含了一个将返回地址压栈的动作，这是由硬件完成的。

函数被调用的最开始一般会出现类似如下的汇编指令

```
pushl %ebp
movl %esp,%ebp
```

这两条汇编指令：

1. 完成的操作是：首先将ebp的值入栈，然后将栈顶指针 esp 赋值给 ebp。
2. 含义是：将旧ebp值压栈（保存原来栈底的地址），然后让ebp恰恰指向旧栈顶（即ebp寄存器中存储着旧ebp入栈后的栈顶）。
3. 意义：目前ebp储存的地址为基准，向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的旧栈底的值。



函数调用大概包括以下几个步骤：

- 参数入栈：将参数从右向左依次压入系统栈中。
- 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。
- 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。
- 栈帧调整
    - 保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP入栈）。
    - 将当前栈帧切换到新栈帧（将ESP值装入EBP，更新栈帧底部）。
    - 给新栈帧分配空间（把ESP减去所需空间的大小，抬高栈顶）。

函数返回大概包括以下几个步骤：

- 保存返回值，通常将函数的返回值保存在寄存器EAX中。
- 弹出当前帧，恢复上一个栈帧。
    - 在堆栈平衡的基础上，给ESP加上栈帧的大小，降低栈顶，回收当前栈帧的空间
    - 将当前栈帧底部保存的前栈帧EBP值弹入EBP寄存器，恢复出上一个栈帧。
    - 将函数返回地址弹给EIP寄存器。
- 跳转：按照函数返回地址跳回母函数中继续执行。

因此我们可以直接根据ebp就能读取到各个栈帧的地址和值，一般而言，[ebp+4]处为返回地址，[ebp+8]处为第一个参数值（最后一个入栈的参数值，此处假设其占用4 字节内存，对应32位系统），[ebp-4]处为第一个局部变量，[ebp]处为上一层 ebp 值。

2. **print_stackframe函数的实现**

函数调用和建立栈帧的过程是从call指令开始的，call指令首先将call指令的下一条指令地址，也就是调用函数的返回地址入栈，然后将ebp入栈，以用于恢复调用者的栈帧，接下来将esp的值赋给ebp，更新帧指针，esp减小以开辟栈空间，完成调用函数的栈帧建立。另外，在使用call指令进行函数调用前，还会将参数入栈。因此在函数调用的过程中，栈帧的情况如下所示：

```
|   ...                                 高地址
|   参数2
|   参数1
|   返回地址
|   保存的[ebp]       <-------- ebp 帧指针
|   局部变量等
|   ...
|   ...               <-------- esp 栈指针
```

了解了函数调用时栈帧的建立，可以打开kdebug.c完成堆栈跟踪函数了。在注释中有关于eip，esp和ebp的一些解释及其他函数的说明，并给出了实现print_stackframe的提示，其中重要的注释如下：

打开文件`kern/debug/kdebug.c`

```
/* LAB1 你的代码：第 1 步

(1) 调用 read_ebp() 获取 ebp 的值。 类型是（uint32_t）；
(2) 调用read_eip()获取eip的值。 类型是（uint32_t）；
(3) 从 0 .. STACKFRAME_DEPTH
    (3.1) ebp, eip 的 printf 值
    (3.2) (uint32_t) 调用参数 [0..4] = 地址 (unit32_t)ebp +2 [0..4] 中的内容
    (3.3) cprintf("\n");
    (3.4)调用print_debuginfo(eip-1)打印C调用函数名和行号等
    (3.5) 弹出调用栈帧
    注意：调用函数的返回地址 eip = ss:[ebp+4]
          调用函数的 ebp = ss:[ebp]
*/
```

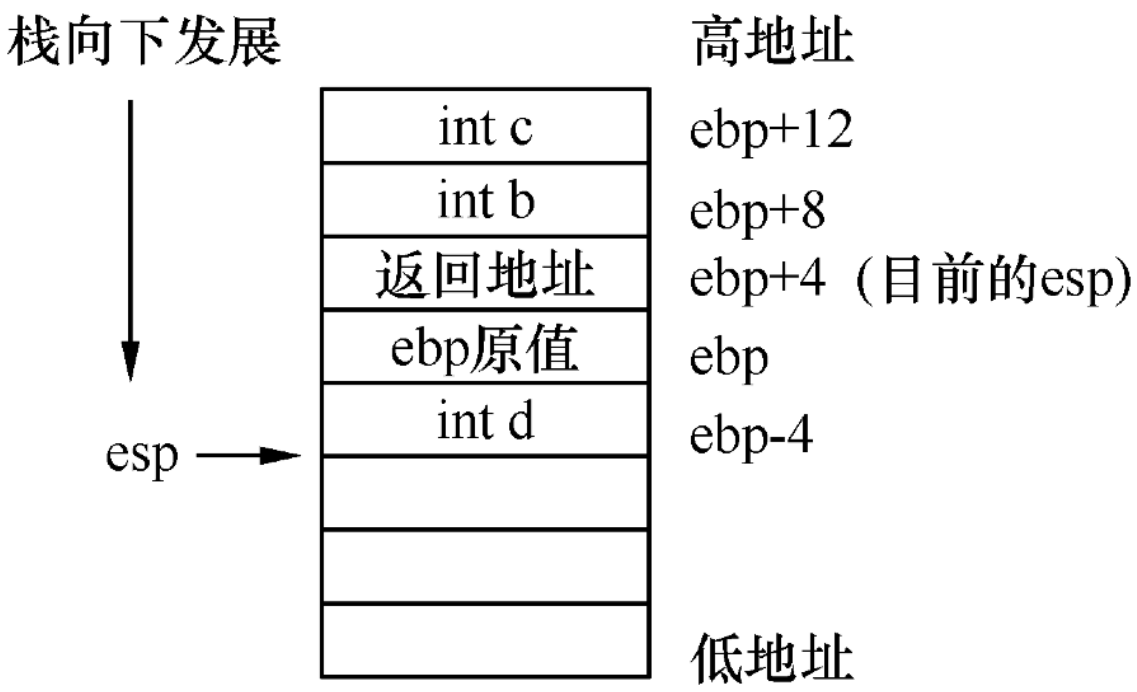
根据注释和相关知识编写成程序，如下所示：

```
void print_stackframe(void) {
    //读取当前栈帧的ebp和eip.
    uint32_t ebp=read_ebp();
    uint32_t eip=read_eip();
    int i;// from 0 .. STACKFRAME_DEPTH
    for (i=0;i<STACKFRAME_DEPTH&&ebp!=0;i++){
        // printf value of ebp, eip
        cprintf("ebp:0x%08x eip:0x%08x",ebp,eip);
//
        uint32_t *tmp=(uint32_t *)ebp+2;
//每个数组大小为4，输出数组元素
        cprintf("arg :0x%08x 0x%08x 0x%08x 0x%08x",*(tmp+0),*(tmp+1),*(tmp+2),*(tmp+3));

        cprintf("\n");
//eip指向异常指令的下一条指令，所以要减1
        print_debuginfo(eip-1);

// 将ebp 和eip设置为上一个栈帧的ebp和eip
// 注意要先设置eip后设置ebp，否则当ebp被修改后，eip就无法找到正确的位置
        eip=((uint32_t *)ebp)[1];//popup a calling stackframe
        ebp=((uint32_t *)ebp)[0];
    }
}
```

代码分析：先用read_ebp与read_eip获得最初的ebp与eip, 根据注释中的要求与要求中结果的规范，将ebp和eip输出，ebp是扩展基址指针寄存器，eip则是指令指针寄存器，因此这两个变量中储存的其实都是地址。根据注释，用arguments接收ebp+2这个地址的地址，并输出arguments[0-3]，用print_debuginfo输出调用它的函数名字与行数，用ebp指针更新下一次循环时ebp与eip的值



esp 栈顶指针
ebp 栈底指针
eip 寄存器存放的CPU下一条指令的地址
首先读取 ebp 和 eip 的值 然后 在ebp的地址上 + 2 就是第一个参数的位置

`make qemu`得到了要求实现的以下输出：

```
...
ebp:00007b28 eip:00100ab3 args:0x00010094 0x00010094
0x00007b58 0x00100096
    kern/debug/kdebug.c:306: print_stackframe+25
ebp:00007b38 eip:00100dac args:0x00000000 0x00000000
0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+14
ebp:00007b58 eip:00100096 args:0x00000000 0x00007b80
0xffff0000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+37
ebp:00007b78 eip:001000c4 args:0x00000000 0xffff0000
0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+42
ebp:00007b98 eip:001000e7 args:0x00000000 0x00100000
0xffff0000 0x0000001d
```

```
    kern/init/init.c:58: grade_backtrace0+27
ebp:00007bb8 eip:00100111 args:0x0010343c 0x00103420
0x0000130a 0x00000000
    kern/init/init.c:63: grade_backtrace+38
ebp:00007be8 eip:00100055 args:0x00000000 0x00000000
0x00000000 0x00007c4f
    kern/init/init.c:28: kern_init+84
ebp:00007bf8 eip:00007d74 args:0xc031fcfa 0xc08ed88e
0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d73 --
++ setup timer interrupts
...
```

最后一行结束后表示更新的ebp为0，即已经找到了第一个使用栈的函数。bootloader在完成实模式到保护模式的转换后建立了栈空间，将ebp设置为0，call bootmian进入bootmain函数，这就是第一次函数调用。故最后一行的ebp：0x7bf8为bootmain的帧指针，eip：0x7d74为bootmain中进入OS处。bootblock.asm中可以找到此处。

```
    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
    7d67:    a1 ec 7d 00 00           mov    0x7dec,%eax
    7d6c:    8b 40 18                 mov    0x18(%eax),%eax
    7d6f:    25 ff ff ff 00           and    $0xffffff,%eax
    7d74:    ff d0                    call   *%eax
```

在调用bootmain时没有传入参数，而ebp+8的位置为0x7c00，是bootloader代码开始的地方，故args是bootloader开始处的指令机器码，可以打开bootblock.asm，前三条指令对应机器码为fa，fc，31，c0，小端法下读出为0xc031fcfa，与args打印的第一个参数对应验证。

```
.code16                                             # Assemble
for 16-bit mode
    cli                                             # Disable
interrupts
    7c00:    fa                      cli
    cld                                             # String
operations increment
    7c01:    fc                      cld
    xorw %ax, %ax                                   # Segment
number zero
    7c02:    31 c0                   xor    %eax,%eax
...
```

练习6：完善中断初始化和处理

1.中断与中断描述符表

操作系统的中断有三种：由时钟或其他外设引起的**中断**（外部中断），由程序执行非法指令等引起的**异常**（内部中断），由程序请求系统服务的**系统调用**（陷阱中断，软中断）。而操作系统需要实现中断处理机制，对发生的中断进行处理。在保护模式下，中断是这样处理的：当CPU收到中断信号（由中断控制器发出）后，会暂停执行当前的程序，跳转到处理该中断的例程中，完成中断处理后再继续执行被打断的程序。其中，中断服务例程的信息被存放在中断描述符表（IDT）中，而中断描述符表的起始地址保存在IDTR寄存器中，CPU收到中断信号后会读取一个中断向量，以中断向量为索引，就可以在IDT表中找到相应的中断描述符，从而获得中断服务例程的起始地址并执行中断服务例程。

中断描述符表中的表项被称为**门描述符**，每个门描述符为8个字节。在`kern/mm/mmu.h`中可以找到门描述符的定义。可以看到，在8个字节共64位中，0-15位为偏移的低16位，48-63位为偏移的高16位，16-31位为段选择子。

```
struct gatedesc {
    unsigned gd_off_15_0 : 16;        // low 16 bits of offset in segment
    unsigned gd_ss : 16;              // segment selector
    unsigned gd_args : 5;             // # args, 0 for interrupt/trap gates
    unsigned gd_rsv1 : 3;             // reserved(should be zero I guess)
    unsigned gd_type : 4;             // type(STS_{TG,IG32,TG32})
    unsigned gd_s : 1;                // must be 0 (system)
    unsigned gd_dpl : 2;              // descriptor(meaning new) privilege level
    unsigned gd_p : 1;                // Present
    unsigned gd_off_31_16 : 16;       // high bits of offset in segment
};
```

根据段选择子可以在GDT表中找到段描述符，得到段的起始地址，结合段的偏移，就可以得到中断服务例程的起始地址。

中断描述符表一个表项为16+16+5+3+4+1+2+1+16=64bit 即占8字节。

① Bit 0—15: gd_odd_15_0，偏移量的低16位

② Bit 16—31: gd_ss,中断例程的段选择器，用于索引全局描述符表GDT来获取中断处理代码对应的段地址。

③ bit 47--32: 属性信息，包括DPL、P flag等

④ bit 48—63: gd_odd_31_16 ,为偏移量的16高位.

两个偏移值一起构成段内偏移量，根据段选择子和段内偏移地址就可以得出中断处理程序的地址。

2.中断向量表初始化

中断向量表的初始化需要通过idt_init()函数完成，该函数在trap.c中，需要完成其具体实现，在注释中给出了提示如下：

```
void
idt_init(void) {
     /* LAB1 YOUR CODE : STEP 2 */
     /* (1) 每个中断服务例程的起始地址被存放在__vectors数组中，这个数组是由vector.c生成，保存在vector.S当中。
      *     可以使用extern uintptr_t __vectors[];引入并使用这个数组。
      * (2) 接下来可以设置IDT中的表项了，idt[256]就是中断描述符表，可以使用SETGATE宏定义设置IDT表中的每一项。
      * (3) 设置好中断描述符表后，只需要执行lidt指令，cpu就可以找到中断描述符表了。（lidt将IDT表的起始位置和界限放入IDTR寄存器）
      */
}
```

设置IDT表中的门描述符通过SETGATE宏实现，SETGATE类似函数，有五个参数。五个参数的意义如下：

- gate：需要赋值设置的门描述符，即此处传入idt中的元素进行设置
- istrap：设置门描述符的的类型，此处无论传入什么值都将门描述符类型设置为陷阱门描述符
- sel：段选择子
- off：偏移
- dpl：描述符所指向中断服务例程的特权级

```
#define SETGATE(gate, istrap, sel, off, dpl) {
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;          //
偏移
    (gate).gd_ss = (sel);                                   //
段选择子
    (gate).gd_args = 0;
    (gate).gd_rsv1 = 0;
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;        //
描述符类型设置
    (gate).gd_s = 0;
    (gate).gd_dpl = (dpl);                                  //
特权级
    (gate).gd_p = 1;
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16;            //
偏移
}
```

接下来考虑传入的参数，第一个参数直接传入需要设置的idt[i]，第二个参数传入0或1都可以，第三个参数为段选择子，中断服务例程由操作系统执行，因此应该位于操作系统内核的代码段，在memlayout.h文件中有各个段的信息如下：

```
/* global descriptor numbers */
#define GD_KTEXT    ((SEG_KTEXT) << 3)        // kernel text
#define GD_KDATA    ((SEG_KDATA) << 3)        // kernel data
#define GD_UTEXT    ((SEG_UTEXT) << 3)        // user text
#define GD_UDATA    ((SEG_UDATA) << 3)        // user data
#define GD_TSS        ((SEG_TSS) << 3)        // task segment
selector
```

故第三个参数传入内核代码段的选择子，即GD_KTEXT。第四个参数为偏移，这个参数应该是由vector提供的，只需要传入对应的vector的元素。最后一个参数为特权级（**Descriptor Privilege Level，描述符特权级**，规定访问该段的权限级别），由于中断服务例程在内核态运行，应设置为内核特权级，在memlayout.h中定义为DPL_KERNEL(0)，需要注意系统调用是在用户态下进行的，系统调用的DPL需要设置为用户特权级，在memlayout.h中定义为DPL_USER (3)，系统调用会将特权级切换为内核特权级，在trap.h中可以找到其中断号。

```
//memlayout.h中的DPL定义
#define DPL_KERNEL    (0)
#define DPL_USER      (3)
//trap.h中定义的特权级切换中断号
#define T_SWITCH_TOU                120    // user/kernel switch
#define T_SWITCH_TOK                121    // user/kernel switch
```

初始化中断描述符表后还需要使用lidt指令将idt的起始地址和界限装载入IDTR寄存器，该函数在x86.h中定义，起始地址和界限在trap.c中定义的idt_pd中。

```
static inline void
lidt(struct pseudodesc *pd) {
    asm volatile ("lidt (%0)" :: "r" (pd));
}
static struct pseudodesc idt_pd = {
    sizeof(idt) - 1, (uintptr_t)idt
};
```

综上，可实现`idt_init()`函数如下：

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int num=sizeof(idt)/sizeof(struct gatedesc);
    for(int i=0;i<num;i++){
        SETGATE(idt[i],1,GD_KTEXT,__vectors[i],DPL_KERNEL);
    }

 SETGATE(idt[T_SWITCH_TOK],GD_KTEXT,__vectors[T_SWITCH_TOK],DPL_USER);
    lidt(&idt_pd);
}
```

3.中断处理

实验要求完成中断处理函数trap，trap函数调用了trap_dispatch()，这个函数会根据trap类型进行中断处理。

```
void
trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
}
```

其中的时钟中断处理部分，只需要使用全局变量对时钟中断进行计数，当时钟中断100次,即TICK_NUM次后，调用print_ticks()打印“100 ticks”，补充后的trap_dispatch()如下:

```
static void
trap_dispatch(struct trapframe *tf) {
    char c;

    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event using a global variable (increase it), such as ticks in kern/driver/clock.c
         * (2) Every TICK_NUM cycle, you can print some info using a funciton, such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
        ticks++;
        if(ticks%TICK_NUM==0) print_ticks();
        break;
    case IRQ_OFFSET + IRQ_COM1:
        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOU:
    case T_SWITCH_TOK:
        panic("T_SWITCH_** ??\n");
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
```

```
        /* do nothing */
        break;
    default:
        // in kernel, it must be a mistake
        if ((tf->tf_cs & 3) == 0) {
            print_trapframe(tf);
            panic("unexpected trap in kernel.\n");
        }
    }
}
```

完成后执行make debug命令，可以看到打印出的”100 ticks“。

```
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
...
```

扩展练习一

根据实验要求，需要完成kern_init.c中的lab1_switch_test()，该函数调用了lab1_switch_to_kernel() 和lab1_switch_to_user()两个函数，这两个函数分别实现了用户态到内核态的切换及内核态到用户态的切换，该函数如下：

```
static void
lab1_switch_test(void) {
    lab1_print_cur_status();
    cprintf("+++ switch to  user  mode +++\n");
    lab1_switch_to_user();
    lab1_print_cur_status();
    cprintf("+++ switch to kernel mode +++\n");
    lab1_switch_to_kernel();
    lab1_print_cur_status();
}
```

为了实现特权级的切换，需要了解以下内容：

1. 特权级

特权级：操作系统通过特权级判断是否可访问数据。在ucore中只有两个特权级，0为内核态，3为用户态。有以下三种不同的特权级：

- CPL：当前特权级，表示当前执行的代码的特权级，保存在CS或SS寄存器的特定位中，执行不同的代码时，CPL也会随之改变，表示特权级发生变化。

- DPL：描述符特权级，表示访问该段的最低特权级，特权级必须高于DPL才可以访问。保存在段描述符中。
- RPL：请求特权级，表示当前代码段发出了一个特定特权级的请求，判断能否访问时，选择 CPL 与 RPL 较大的值（即低的特权级）来判定是否有权进行访问。

2. 中断处理

ucore中发生中断后，处理的过程如下：

首先，中断发生后，以中断向量为索引，在中断描述符表IDT中获得中断描述符，根据中断描述符的段选择子找到GDT中的段描述符，通过段基址和段偏移量，找到中断服务例程的位置，并跳转执行中断服务例程。

在vector.S可以看到进入中断服务例程后，首先将中断号压栈，然后跳转进入**alltraps进行处理。在**alltraps中首先将一些寄存器压栈，并调用trap函数。向trap函数传入的参数为trapframe类型的指针，trapframe结构中保存的是进入中断前的信息，在中断处理结束后，需要利用这些信息恢复进入中断前的状态。

```
//进入中断服务例程
vector0:
  pushl $0
  pushl $0
  jmp __alltraps
//trapentry.s: __alltraps
.globl __alltraps
__alltraps:
    # push registers to build a trap frame
    # therefore make the stack look like a struct trapframe
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushal
    # load GD_KDATA into %ds and %es to set up data segments
for kernel
    movl $GD_KDATA, %eax
    movw %ax, %ds
    movw %ax, %es
    # push %esp to pass a pointer to the trapframe as an
argument to trap()
    pushl %esp
    # call trap(tf), where tf=%esp
    call trap
    # pop the pushed stack pointer
    popl %esp
```

```
    # return falls through to trapret...
```

trapframe的定义在trap.h中，其中的tf_esp和tf_ss是在发生特权级切换时需要保存的信息，如下：

```
struct trapframe {
    struct pushregs tf_regs;
    uint16_t tf_gs;
    uint16_t tf_padding0;
    uint16_t tf_fs;
    uint16_t tf_padding1;
    uint16_t tf_es;
    uint16_t tf_padding2;
    uint16_t tf_ds;
    uint16_t tf_padding3;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding4;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding5;
} __attribute__((packed));
```

需要注意的是**中断发生时的压栈**。esp寄存器和ss寄存器的值只有在发生特权级转换时才需要压栈，如从用户特权级转向内核特权级，会将当前栈由用户栈转为内核栈，需要保存esp和ss寄存器用于恢复用户栈，返回时执行iret指令，将trapframe中的信息恢复，发现栈中弹出的cs寄存器值所标记的特权级（原来的特权级）和当前cs显示的特权级不同，说明特权级发生了转换，就会对esp，ss寄存器弹栈，恢复原来的栈。不发生特权级转换时不需要进行栈的切换，这两个寄存器不会被压栈。

3. 特权级切换的实现

本练习要求的特权级切换，是通过产生中断的方式实现的。在lab1_switch_to_kernel() 和lab1_switch_to_user()两个函数中产生中断，进入中断处理例程，中断处理结束后将从trapframe恢复原状态，将trapframe中的寄存器信息进行修改，等到中断结束恢复状态的时候，就实现了状态的转换。

用函数实现内核态转为用户态时，进入中断时特权级没有发生变化，因此esp和ss没有压栈，而为了实现特权级转换，修改了trapframe的值，在弹栈时会弹出esp和ss切换栈，因此需要留出空间，并在中断处理时将值修改为用户栈的值，使中断完成后能实现栈的切换。另外，实际上es段，ss段在ucore中都是DS段，没有区别，具体实现如下：

```
//
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TO DO
    asm volatile(
    "sub $0x8,%%esp \n"             //留出ss，esp的空间
    "int %0 \n"                     //中断
    "movl %%ebp,%%esp"              //恢复栈指针
    :
    :"i"(T_SWITCH_TOU)              //中断号
    );
}
//trap_dispatch中处理中断
case T_SWITCH_TOU:
    tf->tf_cs=USER_CS;
    tf->tf_ds=USER_DS;
    tf->tf_es=USER_DS;
    tf->tf_ss=USER_DS;
    tf-tf_eflags|= FL_IOPL_MASK;  //根据答案，此处设置的flag是为了用户能正常进行IO操作
    break;
```

用函数实现用户态转为内核态时，由于中断时已经发生了特权级的变化，不需要预留空间，只需要修改trapframe。另外，由于将cs的值修改为内核态下的值（等于当前cs的值），cpu认为没有发生特权级转换，不会发生ss和esp的弹栈去恢复栈空间。

```
tatic void
lab1_switch_to_kernel(void) {
 /                /LAB1 CHALLENGE 1 : TODO
asm volatile (
     "int %0 \n"
     "movl %%ebp, %%esp \n"
     :
     : "i"(T_SWITCH_TOK)
     );
}
//trap_dispatch中处理中断
```

```
case T_SWITCH_TOK:
     tf->tf_cs = KERNEL_CS;
     tf->tf_ds = KERNEL_DS;
     tf->tf_es = KERNEL_DS;
     break;
```

扩展练习 二

用键盘实现用户模式内核模式切换。具体目标是：“键盘输入3时切换到用户模式，键盘输入0时切换到内核模式”。 基本思路是借鉴软中断(syscall功能)的代码，并且把trap.c中软中断处理的设置语句拿过来。

```
kern/trap/trap.c
static void switch_to_user() {
    asm volatile (
        "sub $0x8, %%esp \n"
        "int %0 \n"
        "movl %%ebp, %%esp"
        :
        : "i"(T_SWITCH_TOU)
    );
}
static void switch_to_kernel() {
    asm volatile (
        "int %0 \n"
        "movl %%ebp, %%esp \n"
        :
        : "i"(T_SWITCH_TOK)
    );
}

    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        if (c == '3') {
            switch_to_user();
            print_trapframe(tf);
        } else if (c == '0') {
            switch_to_kernel();
            print_trapframe(tf);
        }
        cprintf("kbd [%03d] %c\n", c, c);
        break;
```

按键的中断在 IRQ_KBD 处

实验总结

参考答案对比

- 练习1：
  在实验报告中采取的思路是从生成ucore.img的makefile命令倒推实现过程，逐步展示，条理和层次不像参考答案中那么清晰；分别从makefile依赖的两个关键的文件kernal与bootblock分析了生成文件的指令部分，并对dd指令与参数部分进行了更加详细一些的分析。

- 练习2：
  本题与参考答案的实现区别不是很大，从以下四个角度进行分析。从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。在初始化位置0x7c00设置实地址断点,测试断点正常。从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。自己找一个bootloader或内核中的代码位置，设置断点并进行测试。

- 练习3：
  介绍思路和顺序和答案基本一致，但是补充了一些对应的原理知识点，并对各条指令的内涵做出了自己能基本理解的详细解释。如为什么要开启A20以及如何开启A20的部分。再如调用bootmain函数的部分，都有较为详细的说明。

- 练习4：
  结合实验指导书提供的资料、具体的代码注释信息，对每一部分的原理和实现机制做出了比参考答案详细得多的说明。如bootloader如何读取硬盘扇区以及ELF格式的OS，再如bootmain.c的代码的解读和其中的多个函数（waitdisk、readsect、readseg、bootmain等）的实现都有更为详细的说明。

- 练习5：
  实现机制和参考答案思路基本一致。此外，对栈机制的使用做出了自己的解释和详细说明。特别是对于c指针加法的说明以及传递变量地址的说明。

- 练习6：
  进行中断初始化的方法和参考答案基本一致，主要增加了自己对相应中断原理的理解并做出注释。对于编程完善的部分也有较为独到的见解。

- 扩展练习1
  基本和参考答案一致，通过设置tf信息来完成状态切换，同时用自学的相关知识进行了理解和剖析，参考了学堂在线教学视频的指导。在补全代码的部分有较为清晰的说明。

- 扩展练习2
  参考答案未提供参考，依靠自学交流以及CSDN学习完成。由于较为简单，没有过多的说明与解释。

重要知识点和对应原理

1. 实验中的重要知识点
   makefile书写格式和实现方法
   GCC基本内联汇编语法
   linux gdb调试基础
   BIOS启动过程
   bootloader启动过程
   关于保护模式的硬件实现
   分段机制的实现原理（含特权级）
   硬盘访问实现机制
   ELF文件格式概述
   操作系统基本启动方式
   函数堆栈调用的内部原理
   中断机制的实现（含特权级）
2. 对应的OS原理知识点
   虚拟内存的实现
   分段保护机制
   过程调用与中断实现
   内存访问机制
3. 二者关系：
   本实验设计的知识是对OS原理的具体实现，在细节上很复杂。
4. 未对应的知识点
   进程调度管理
   分页管理与页调度
   并发实现机制