

操作系统 Lab1 系统软件启动过程 实验报告

黄志鹏 PB16150288

- [实验目的](#)
- [实验前的准备](#)
 - [观察makefile的结构](#)
 - [对makefile精细的分析](#)
 - [第一次编译时候遇到的问题](#)和解决方法
- [基本练习](#)
 - [练习1:理解通过make生成执行文件的过程](#)
 - [练习2：使用qemu执行并调试lab1中的软件](#)
 - [练习3：分析bootloader进入保护模式的过程](#)
 - [扩展练习:Challenge1\(需要编程\)](#)
 - [扩展练习:Challenge2\(需要编程\)](#)
- [实验中涉及的知识点列举](#)

实验目的

通过分析和实现这个bootloader和ucore OS,读者可以了解到:

- 计算机原理

CPU的编址与寻址: 基于分段机制的内存管理 CPU的中断机制 外设:串口/并口/CGA,时钟,硬盘

- Bootloader软件

编译运行bootloader的过程 调试bootloader的方法 PC启动bootloader的过程 ELF执行文件的格式和加载 外设访问:读硬盘,在CGA上显示字符串

- ucore OS软件

编译运行ucore OS的过程 ucore OS的启动过程 调试ucore OS的方法 函数调用关系:在汇编级了解函数调用栈的结构和处理过程 中断管理:与软件相关的 外设管理:时钟

实验前的准备

观察makefile的结构

1~138行：定义各种变量/函数，设置参数，进行准备工作。

- 140~153行：生成bin/kernel
- 155~170行：生成bin/bootblock
- 172~176行：生成bin/sign
- 178~188行：生成bin/ucore.img
- 189~201行：收尾工作/定义变量
- 203~269行：定义各种make目标

对makefile精细的分析

```
V := @
```

变量V=@，后面大量使用了V

@的作用是不输出后面的命令，只输出结果

在这里修改V即可调整输出的内容

也可以 make "V=" 来完整输出

function.mk中定义了大量的函数。.mk中每个函数都有注释。

```
include tools/function.mk
```

call函数：call func, 变量1, 变量2,...

listf函数在function.mk中定义，列出某地址（变量1）下某些类型（变量2）文件

listf_cc函数即列出某地址（变量1）下.c与.S文件

```
listf_cc = $(call listf,$(1),$(CTYPE))
```

第一次编译时候遇到的问题解决方法

在用make编译lab1_result时候遇到了一下问题

```
'obj/bootblock.out' size: 620 bytes
620 >> 510!!
Makefile:152: recipe for target 'bin/bootblock' failed
```

然后make中止，不能继续运行，之后的make grade等操作也同样不能运行

从这两句的输出来看，是可以看出是来源在sign.c文件中定义的st的大小大于510字节导致的
查看tools/ 目录下的sign.c文件，可以看到如下代码

```
int main(int argc, char *argv[]) {
    struct stat st;
    if (argc != 3) {
        fprintf(stderr, "Usage: <input filename> <output filename>\n");
        return -1;
    }
    if (stat(argv[1], &st) != 0) {
        fprintf(stderr, "Error opening file '%s': %s\n", argv[1], strerror(errno));
        return -1;
    }
    printf("'%' size: %lld bytes\n", argv[1], (long long)st.st_size);
    if (st.st_size > 510) {
        fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
        return -1;
    }
}
```

```
}
```

可以看出这里定义了一个结构体变量 st。
我们转到这个结构体定义的地方

```
struct stat
{
    __dev_t st_dev;      /* Device.  */
#ifdef __x86_64__
    unsigned short int __pad1;
#endif
#ifdef defined __x86_64__ || !defined __USE_FILE_OFFSET64
    __ino_t st_ino;      /* File serial number.  */
#else
    __ino_t __st_ino;     /* 32bit file serial number.  */
#endif
#ifdef __x86_64__
    __mode_t st_mode;     /* File mode.  */
    __nlink_t st_nlink;   /* Link count.  */
#else
    __nlink_t st_nlink;   /* Link count.  */
    __mode_t st_mode;
    .....
}
```

我们可以看出这里的结构体定义和系统还有gcc版本有关和过程中的环境声明有关，考虑到这个lab1_result的代码在老师给的实验虚拟机上运行是ok的，所以猜测是gcc的版本问题，我的debian系统上用的是比较新的gcc-6，而实验的环境中的gcc是4.9版本。

故这里尝试使用低版本的gcc

首先，安装

在/etc/apt/source.list 文件后面增加

```
deb http://ftp.us.debian.org/debian/ jessie main contrib non-free
deb-src http://ftp.us.debian.org/debian/ jessie main contrib non-free
```

然后运行

```
$sudo apt install gcc-4.9
```

将lab1工程中的Makefile 中的gcc配置进行修改

```
CC:= $(GCCPREFIX)gcc-4.9
```

再次运行

```
$make
$make clean
```

success!

现在实验环境已经就绪!!

基本练习

练习 1:理解通过make生成执行文件的过程

1.1 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?

使用make "V=" 命令的打印结果了解make编译的大致过程 这里使用的是make "V=" 命令, 将这里的编译过程全部在终端中展示出来 如图所示

```
huangzp@localhost: ~/code/OSexperiments/ucore_os_lab/labcodes/lab1
File Edit View Search Terminal Tabs Help
huangzp@localhost: ~/code/OSexperiments/ucore_os_lab/labcodes/lab1
huangzp@localhost:~/code/OSexperiments/ucore_os_lab/labcodes/lab1$ make "V="
+ cc kern/init/init.c
gcc-4.9 -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
kern/init/init.c:95:1: warning: 'lab1_switch_test' defined but not used [-Wunused-function]
lab1_switch_test(void) {
^
+ cc kern/libs/stdio.c
gcc-4.9 -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/libs/stdio.c -o obj/kern/libs/stdio.o
+ cc kern/libs/readline.c
gcc-4.9 -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
+ cc kern/debug/panic.c
gcc-4.9 -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/panic.c -o obj/kern/debug/panic.o
kern/debug/panic.c: In function '__panic':
kern/debug/panic.c:27:5: warning: implicit declaration of function 'print_stackframe' [-Wimplicit-function-declaration]
    print_stackframe();
    ^
+ cc kern/debug/kdebug.c
gcc-4.9 -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o
kern/debug/kdebug.c:251:1: warning: 'read_eip' defined but not used [-Wunused-function]
read_eip(void) {
^
+ cc kern/debug/kmonitor.c
gcc-4.9 -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o
+ cc kern/driver/clock.c
gcc-4.9 -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/clock.c -o obj/kern/driver/clock.o
+ cc kern/driver/console.c
gcc-4.9 -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/console.c -o obj/kern/driver/console.o
+ cc kern/driver/picirq.c
gcc-4.9 -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/picirq.c -o obj/kern/driver/picirq.o
+ cc kern/driver/intr.c
gcc-4.9 -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/intr.c -o obj/kern/driver/intr.o
+ cc kern/trap/trap.c
gcc-4.9 -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trap.c -o obj/kern/trap/trap.o
kern/trap/trap.c:14:13: warning: 'print_ticks' defined but not used [-Wunused-function]
static void print_ticks() {
^
kern/trap/trap.c:30:26: warning: 'idt_pd' defined but not used [-Wunused-variable]
static struct pseudodesc idt_pd = {
^
+ cc kern/trap/vectors.S
gcc-4.9 -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/vectors.S -o obj/kern/trap/vectors.o
+ cc kern/trap/trapentry.S
gcc-4.9 -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o
+ cc kern/mm/pmm.c
gcc-4.9 -Ikern/mm/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/mm/pmm.c -o obj/kern/mm/pmm.o
+ cc libs/string.c
gcc-4.9 -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/string.c -o obj/libs/string.o
+ cc libs/printfmt.c
gcc-4.9 -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/printfmt.c -o obj/libs/printfmt.o
+ 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
+ cc boot/bootasm.S
gcc-4.9 -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -O0 -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc-4.9 -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -O0 -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: 468 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0235992 s, 217 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 8.9516e-05 s, 5.7 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
138+1 records in
138+1 records out
70780 bytes (71 kB, 69 KiB) copied, 0.000437514 s, 162 MB/s

```

第一步, gcc编译出一系列bin/kernel所需要的目标文件

```

obj/kern/init/init.o
obj/kern/init/init.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

```

第二步, 生成 bin/kernel 使用ld命令, 将这些之前生成的目标文件, 链接成可执行文件, 即之后要用到的 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

```

第三步, 编译出 bin/bootblock 所需要的目标文件

```

obj/boot/bootasm.o
obj/boot/bootmain.o
obj/sign/tools/sign.o

```

第四步, 将上面的目标文件编译成 bin/bootblock 可执行文件

```

ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o
obj/boot/bootmain.o -o obj/bootblock.o

```

第五步, 将第三步中产生的sign.o链接成sign可执行文件

现在一共有bin/kernel, obj/sign/tools/sign, obj/bootblock.out 这三个可执行文件。

这里这个bootblock.out 的文件大小为468 Bytes

build 512 bytes boot sector: 'bin/bootblock' success!

从这句话中可以看出经过一些程序，生成了一个512 Bytes的可执行文件。

第六步, 生成ucore.img 镜像文件

我们可以看到Makefile文件中有这样一段code

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

$(UCOREIMG): $(kernel) $(bootblock)
    $(V)dd if=/dev/zero of=$@ count=10000
    $(V)dd if=$(bootblock) of=$@ conv=notrunc
    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

$(call create_target,ucore.img)
```

对应make "V=" 的打印结果

```
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0222739 s, 230 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 8.632e-05 s, 5.9 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
138+1 records in
138+1 records out
70780 bytes (71 kB, 69 KiB) copied, 0.00040367 s, 175 MB/s
```

linux dd命令用于读取、转换并输出数据。

dd可从标准输入或文件中读取数据，根据指定的格式来转换数据，再输出到文件、设备或标准输出。

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。

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

conv=notrunc:表示不截短输出文件

这里的操作就是：

先用10000block的0来初始化ucore.img, 将bootblock和kernel加入进去, 调用create_target 来生成ucore.img文件.

1.2 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

这里我们可以从sign.c这个文件中的看出答案

```

int
main(int argc, char *argv[]) {
    struct stat st;
    if (argc != 3) {
        fprintf(stderr, "Usage: <input filename> <output filename>\n");
        return -1;
    }
    if (stat(argv[1], &st) != 0) {
        fprintf(stderr, "Error opening file '%s': %s\n", argv[1], strerror(errno));
        return -1;
    }
    printf("'s' size: %lld bytes\n", argv[1], (long long)st.st_size);
    if (st.st_size > 510) {
        fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
        return -1;
    }
    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]);
    return 0;
}

```

主引导扇区在磁盘的 0 磁头 0 柱面 1 扇面, 包括硬盘主引导记录MBR(Master Boot Recode) 和分区表DPT(Disk Patition Table). 规范的主引导扇区特征如下:

1. 总大小为512字节, 由启动程序 分区表 和结束符号组成.
2. 最后由一个0x55和一个0xAA表示 启动区的结束字符
3. 由不超过466字节的启动代码和不超过64字节的硬盘分区表加上两个字节的结束符组成

练习2：使用qemu执行并调试lab1中的软件

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。

a. 在tools/gdbinit 中加入

```
set architecture i8086
```

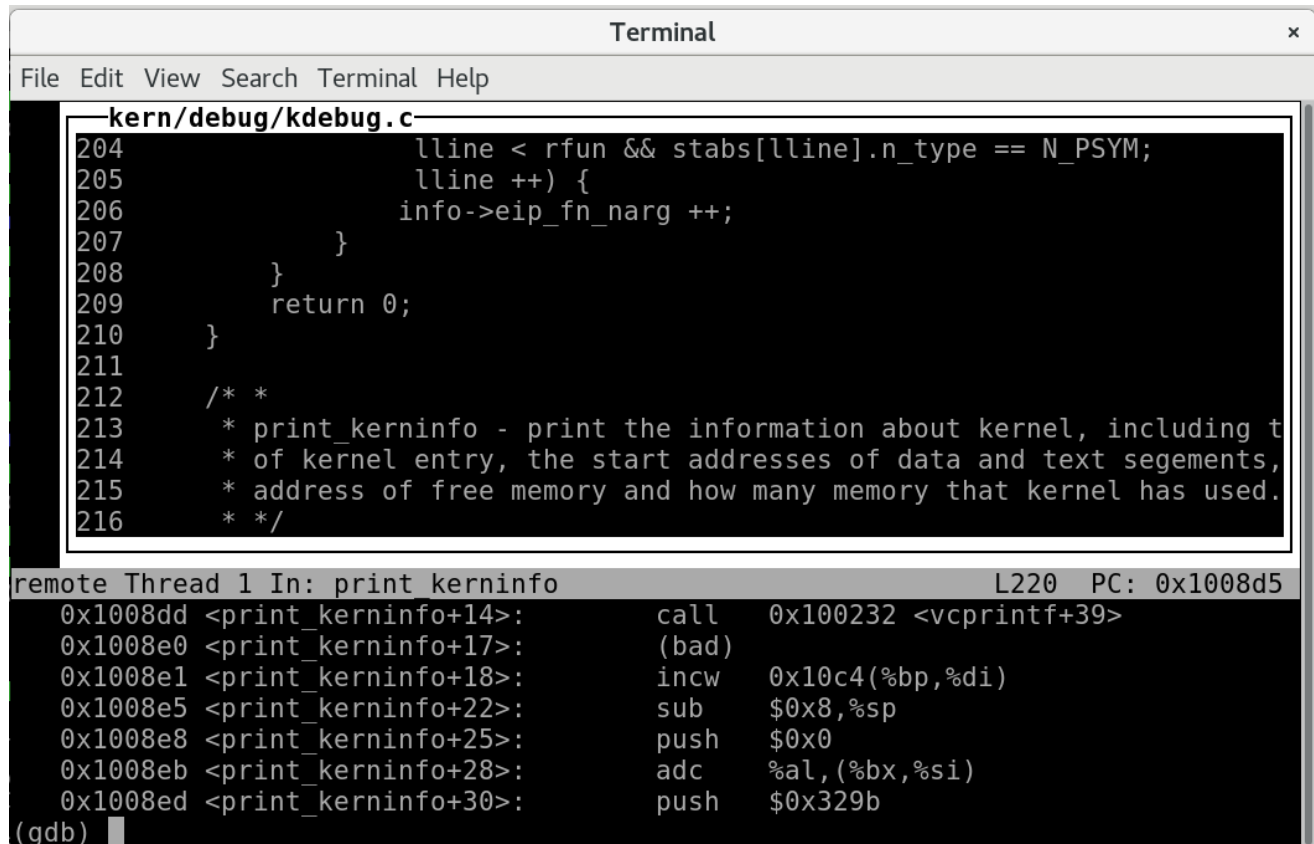
b. 随后执行debug

```
$make debug
```

c. gdb调试界面下执行 `next` 命令

d. 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2条汇编指令
```



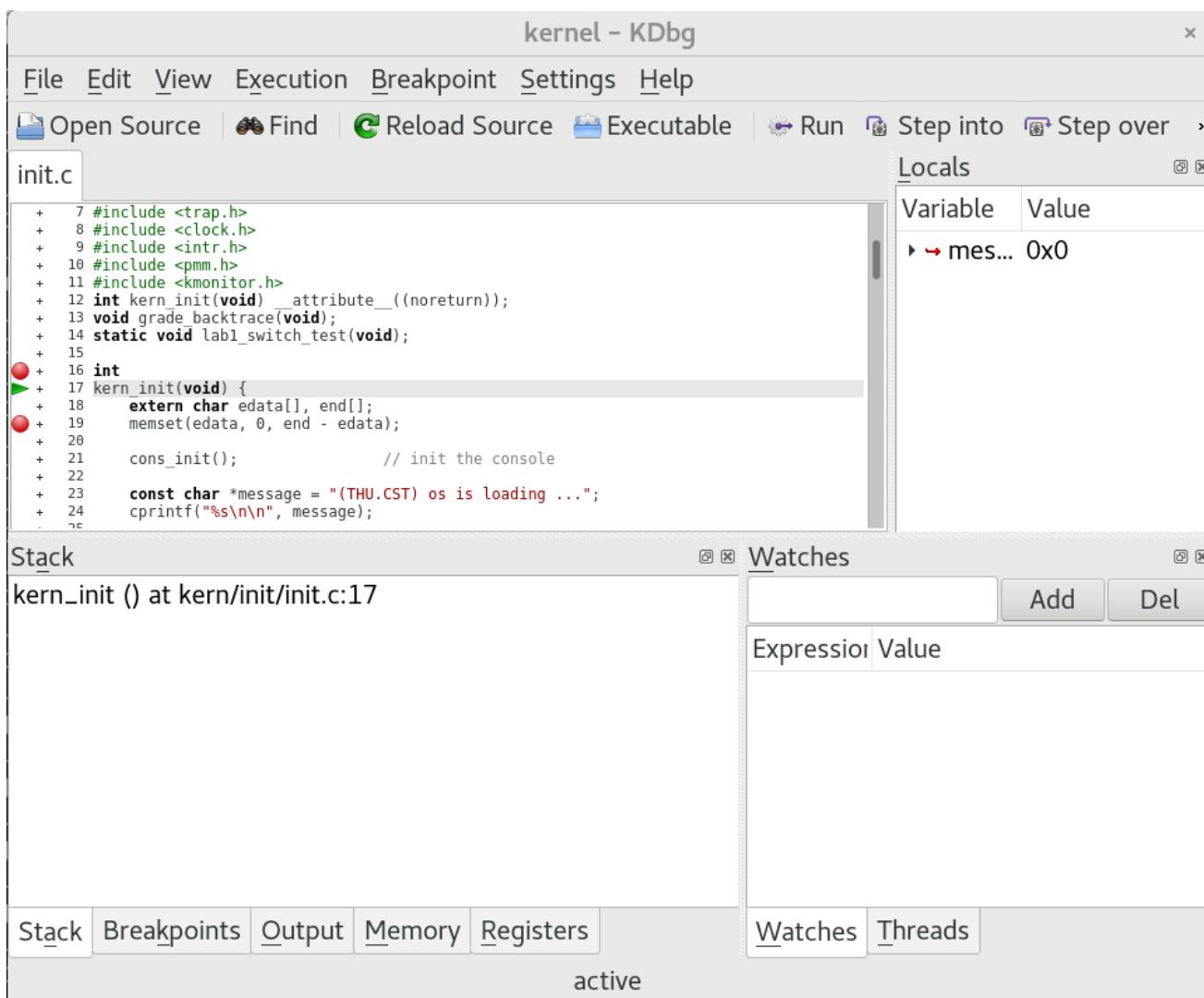
The screenshot shows a GDB terminal window titled "Terminal". The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The main window displays the source code of `kern/debug/kdebug.c` with line numbers 204 to 216. The code defines a function `print_kerninfo` that prints kernel information. Below the source code, the disassembly of the `print_kerninfo` function is shown, with addresses ranging from `0x1008dd` to `0x1008ed`. The disassembly includes instructions like `call`, `incw`, `sub`, `push`, and `adc`. The status bar at the bottom indicates "remote Thread 1 In: print_kerninfo" and "L220 PC: 0x1008d5".

```
kern/debug/kdebug.c
204     lline < rfun && stabs[lline].n_type == N_PSYM;
205     lline ++) {
206         info->eip_fn_narg ++;
207     }
208 }
209 return 0;
210 }
211
212 /* *
213  * print_kerninfo - print the information about kernel, including t
214  * of kernel entry, the start addresses of data and text segements,
215  * address of free memory and how many memory that kernel has used.
216  * */

remote Thread 1 In: print_kerninfo L220 PC: 0x1008d5
0x1008dd <print_kerninfo+14>:      call    0x100232 <vcprintf+39>
0x1008e0 <print_kerninfo+17>:      (bad)
0x1008e1 <print_kerninfo+18>:      incw    0x10c4(%bp,%di)
0x1008e5 <print_kerninfo+22>:      sub     $0x8,%sp
0x1008e8 <print_kerninfo+25>:      push    $0x0
0x1008eb <print_kerninfo+28>:      adc     %al,(%bx,%si)
0x1008ed <print_kerninfo+30>:      push    $0x329b
(gdb)
```

注：这里也可以采用kdbg等可视化调试工具来进行单步跟踪代码的执行情况

```
$qemu-system-x86_64 -S -s -d in_asm -monitor stdio -hda bin/ucore.img -serial null
$kdbg -r localhost:1234 bin/kernel
```

2.2 在初始化位置0x7c00设置实地址断点,测试断点正常。

a. 由于bios启动是为实模式, 故此时的架构需要改成i8086才能够进行调试, 修改lab1/tools/gdbinit, 内容为:

```
file bin/kernel
target remote :1234
break kern_init
continue
set architecture i8086
b *0x7c00
c
x /2i $pc
set architecture i386
```

b. make debug 后得到如下结果:

```
Breakpoint 2 at 0x7c00
Breakpoint 2, 0x00007c00 in ?? ()
=> 0x7c00:      cli
0x7c01:      cld
The target architecture is assumed to be i386
```

2.3 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。

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。

首先修改Makefile文件在调用qemu时增加-d in_asm -D q.log 参数

```
qemu: $(UCOREIMG)
    $(V)$(QEMU) -no-reboot -d in_asm -D q.log -parallel stdio -hda $< -serial null
```

然后执行

```
$make debug
```

在/lab1/bin/q.log可以找到执行的汇编指令, 与/lab1/boot/bootasm.S中的代码相同

```
-----
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
...
```

上述的反编译结果和bootblock.asm和bootasm.S比较,发现,主要的核心汇编指令是一样的,不同的地方是在,所有和执行代码无关的内容不再存在(注释、标签等)

2.4 自己找一个bootloader或内核中的代码位置，设置断点并进行测试。

```
$make debug
```

```
(gdb)break clock_init
(gdb)c
(gdb)n
(gdb)n
(gdb)n
(gdb)n
```

```
Terminal
File Edit View Search Terminal Help
kern/driver/clock.c
31  /*
32  void
33  clock_init(void) {
34      // set 8253 timer-chip
35      outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
36      outb(IO_TIMER1, TIMER_DIV(100) % 256);
37      outb(IO_TIMER1, TIMER_DIV(100) / 256);
38
39      // initialize time counter 'ticks' to zero
40      ticks = 0;
41
42      cprintf("++ setup timer interrupts\n");
43      pic_enable(IRQ_TIMER);
remote Thread 1 In: clock_init L44 PC: 0x100cd4
Breakpoint 2, clock_init () at libs/x86.h:57
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) p ticks
$1 = 0
(gdb) █
```

练习3：分析bootloader进入保护模式的过程

3.1 BIOS将通过读取硬盘主引导扇区到内存,并转跳到对应内存中的位置执行bootloader。请分析bootloader是如何完成从实模式进入保护模式的。

第一步：屏蔽中断

切换的时候当然不希望中断发生，需要使用 `cli` 来屏蔽中断。从 `cs = 0 && ip = 0x7c00` 进入bootloader启动过程，关闭中断使能，将寄存器置零

```
.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
```

第二步：开启A20

```
# 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                    # 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
    outb %al, $0x60                  # 0xdf = 11011111, means set P2's
A20 bit(the 1 bit) to 1
```

Intel早期的8086 CPU提供了20根地址线，寻址范围就是1MB，但是8086字长为16位，直接使用一个字来表示地址就不够了，所以使用段+偏移地址的方式进行寻址。段+偏移地址的最大寻址范围就是0xFFFF0+0xFFFF=0x10FFEF，这个范围大于1MB，所以如果程序访问了大于1MB的地址空间，就会发生回卷。然而随后的CPU的地址线越来越多，同时为了保证软件的兼容性，A20在实模式下被禁止（永远为0），这样就可以正常回卷了。但是在保护模式下，我们希望能够正常访问所有的内存，就必须将A20开启。

由于历史原因，开启A20由键盘控制器"8042" PS/2 Controller负责,A20的开启标志位于PS/2 Controller Output Port的第1位，程序要做的就是修改这一位。8042有两个常用的端口：

端口	访问方式	功能
0x60	读/写	数据端口
0x64	读	状态寄存器
0x64	写	命令寄存器

在发送命令或者写入数据之前，需要确认8042是否准备就绪，就绪标志在状态字的第1位。

位	意义
1	输入缓冲状态 (0 = 空, 1 = 满)(在向 0x60 或者 0x64 端口写入数据前需要确认为0)

写PS/2 Controller Output Port的命令为0xd1，所以开启过程如下：

- 等待8042 Input buffer为空；
- 发送Write Controller Output Port (0xd1) 命令到命令端口；
- 等待8042 Input buffer为空；
- 将Controller Output Port对应状态字的第1位置1，然后写入8042 Input buffer。

第三步：加载段表GDT

在保护模式下，CPU采用分段存储管理机制，初始情况下，GDT中只有内核代码段和内核数据段，这两个段在内存上的空间是相同的，只是段的权限不同。

```
# 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.
lgdt gtdesc      //初始化GDT
```

第四步：设置cr0上的保护位, 实现从实模式进入保护模式

crX寄存器是Intel x86处理器上用于控制处理器行为的寄存器，cr0的第0位用来设置CPU是否开启保护模式[\[Wikipedia\]](#)。

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

第五步：通过跳转进入32位模式

```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp $PROT_MODE_CSEG, $protcseg

.code32                                     # Assemble for 32-bit mode
protcseg:
```

第六步:设置保护模式下的数据段寄存器

```

# Set up the protected-mode data segment registers
movw $PROT_MODE_DSEG, %ax          # Our data segment selector
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

```

第七步:设置栈针, 保护模式设置完毕, 调用bootmain

```

# Set up the stack pointer and call into C. The stack region is from 0--
start(0x7c00)
movl $0x0, %ebp
movl $start, %esp
call bootmain

```

练习 4:分析 bootloader 加载 ELF 格式的 OS 的过程

4.1 通过阅读bootmain.c,了解bootloader如何加载ELF文件。通过分析源代码和通过qemu来运行并调试bootloader&OS

a. bootloader如何读取硬盘扇区的?

实验指导书中给出了磁盘 IO 地址和对应功能, 如下表所示:

端口	功能	描述
0	数据端口	读取/写入数据
2	扇区数量	读取或者写入的扇区数量
5	扇区号 / LBA低字节	
4	柱面号低字节 / LBA中字节	
5	柱面号高字节 / LBA高字节	
6	驱动器号	
7	命令端口 / 状态端口	发送命令或者读取状态

观察readseg, 和readsect函数的代码实现, 如下

```

/* readsect - read a single sector at @secno into @dst */
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

    outb(0x1F2, 1);                // count = 1
    outb(0x1F3, secno & 0xFF);
}

```

```

    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);                // cmd 0x20 - read sectors

    // wait for disk to be ready
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
}

```

可以看出, 上面的这句:

```
outb(0x1F2, 1);
```

表示选取的扇区数量为1 这四句:

```

    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);

```

上面四条指令联合制定了扇区号

在这4个字节线联合构成的32位参数中 29-31位强制设为1

28位(=0)表示访问"Disk 0"

0-27位是28位的偏移量

从上面的分析中可以看出, 扇区的读取完整流程如下 :

- 等待磁盘空闲
- 将LBA的各个部分送入对应的端口, 将读取命令0x20送入命令端口
- 等待磁盘读取完毕
- 从数据端口读取一个扇区的数据

bootloader如何读取kernel OS 的?

下面是readseg的代码:

```

/* *
 * readseg - read @count bytes at @offset from kernel into virtual address @va,
 * might copy more than asked.
 * */
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    // round down to sector boundary
    va -= offset % SECTSIZE;

```

```

// translate from bytes to sectors; kernel starts at sector 1
uint32_t secno = (offset / SECTSIZE) + 1;

// If this is too slow, we could read lots of sectors at a time.
// We'd write more to memory than asked, but it doesn't matter --
// we load in increasing order.
for (; va < end_va; va += SECTSIZE, secno++) {
    readsect((void *)va, secno);
}
}

```

从上面readseg的代码中可以看出: readseg简单包装了readsect, 可以从设备读取任意长度的内容。

在bootmain函数中,

```

void
bootmain(void) {
    // 首先读取ELF的头部
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // 通过储存在头部的e_magic判断是否是合法的ELF文件
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // ELF头部有描述ELF文件应加载到内存什么位置的描述表,
    // 先将描述表的头地址存在ph
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;

    // 按照描述表将ELF文件中数据载入内存
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
    }
    // ELF文件0x1000位置后面的0xd1ec比特被载入内存0x00100000
    // ELF文件0xf000位置后面的0x1d20比特被载入内存0x0010e000

    // 根据ELF头部储存的入口信息, 找到内核的入口
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    while (1);
}

```

练习 5: 实现函数调用堆栈跟踪函数 (需要编程)

- 函数调用的堆栈过程大致如下：


```

|_____| <- esp
|__caller's ebp__| <- callee's ebp
|_return address_|
|__argument_1__|
|__argument_2__|
|__argument_3__|
|_____..._____|

```

- 我们需要在lab1中完成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。

实现的代码如下:

```

void
print_stackframe(void) {
    uint32_t ebp, eip;
    uint32_t *args;

    //ebp指向栈底的位置, esp指向栈顶的位置。
    ebp = read_ebp();
    //eip是cpu下一次执行指令的地址
    eip = read_eip();
    int i, j;
    for (i = 0; i < STACKFRAME_DEPTH; i++) {
        cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        args = (uint32_t *)ebp + 2;
        //ebp + 1是存的是返回地址, ebp+2才是第一个参数的存储位置。
        for (j = 0; j < 4; j++) {
            cprintf("0x%08x ", args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip - 1);
        eip = ((uint32_t *)ebp)[1];
        ebp = ((uint32_t *)ebp)[0];
    }
}

```

- 在lab1中执行“make qemu”后，在qemu模拟器中得到类似如下的输出：

```

Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a6 args:0x00010094 0x00000000 0x00007b38 0x00100092
    kern/debug/kdebug.c:309: print_stackframe+21
ebp:0x00007b18 eip:0x00100c95 args:0x00000000 0x00000000 0x00000000 0x00007b88
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00000000 0x00007b60 0xffff0000 0x00007b64
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x00007b84 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x001032fc 0x001032e0 0x0000130a 0x00000000

```

```

kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00010094
kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d67 --
++ setup timer interrupts

```

ebp指向的堆栈位置储存着caller的ebp；ebp+4指向caller调用时的eip；ebp+8, +12, ...可能是调用时保存的参数，对应的是第一个使用堆栈的函数，bootmain.c中的bootmain的寄存器值

练习 6:完善中断初始化和处理

6.1 中断描述符表(也可简称为保护模式下的中断向量表)中一个表项占多少字节?其中哪几位代表中断处理代码的入口?

终端描述符的结构体定义在mmu.h中：

```

/* Gate descriptors for interrupts and traps */
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
};

```

显然占用64位，也就是8字节，中断处理代码的入口由offset和ss指定。

6.2 编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt_init。

按照题目要求给出的代码如上图所示,循环对 idt 内所有的中断入口进行初始化,完成后即可通过 LIDT 指令来加载 IDT.

代码如下:

```

void
idt_init(void) {
    /* LAB1 YOUR CODE : STEP 2 */
    /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
     *     All ISR's entry addrs are stored in __vectors. where is uintptr_t
     *     __vectors[] ?
     *     __vectors[] is in kern/trap/vector.S which is produced by tools/vector.c
     *     (try "make" command in lab1, then you will find vector.S in kern/trap
     *     DIR)
     *     You can use "extern uintptr_t __vectors[];" to define this extern
     *     variable which will be used later.

```

```

    * (2) Now you should setup the entries of ISR in Interrupt Description Table
    (IDT).

    *      Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE
macro to setup each item of IDT

    * (3) After setup the contents of IDT, you will let CPU know where is the IDT
by using 'lidt' instruction.

    *      You don't know the meaning of this instruction? just google it! and check
the libs/x86.h to know more.

    *      Notice: the argument of lidt is idt_pd. try to find it!
    */
extern uintptr_t __vectors[];
int i;
//将256个idt表项全部填充, 参数: 0表示为中断, GD_KTEXT表示kernel text,
//__vectors[i]表示对用的中断处理程序在代码段中的偏移, DPL_KERNEL表示为内核态
for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) {
    SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
}
//从用户态切换到内核态
// set for switch from user to kernel
SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
// load the IDT
//加载IDT
lidt(&idt_pd);
}

```

6.3 请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写trap函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字“100 ticks”。

添加代码如下：

```

/* trap_dispatch - dispatch based on what type of trap occurred */
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 function, such as
print_ticks().
        * (3) Too Simple? Yes, I think so!
        */
        //ticks用来记录时钟中断次数，每100次调用一次print_ticks()即可
        ticks++;
        if (ticks % TICK_NUM == 0) {
            print_ticks();
        }
        break;
    }
}

```

```
case IRQ_OFFSET + IRQ_COM1:
    . . . . .
```

运行整个系统，可以看到大约每1秒会输出一行“100 ticks”

扩展练习:Challenge1(需要编程)

扩展proj4,增加syscall功能,即增加一用户态函数(可执行一特定系统调用:获得时钟计数值),当内核初始完毕后,可从内核态返回到用户态的函数,而用户态的函数又通过系统调用得到内核态的服务(通过网络查询所需信息,可找老师咨询。如果完成,且有兴趣做代替考试的实验,可找老师商量)。需写出详细的设计和分析报告。完成出色的可获得适当加分。

从内核态切换到用户态

由于实验指导书要求使用中断处理的形式进行从内核态的用户态的切换，因此不妨考虑在ISR中进行若干对硬件保存的现场的修改，伪造出一个从用户态切换到内核态的中断的现场，然后使用iret指令进行返回，就能够实现内核态到用户态的切换，具体实现如下所示：

首先由于OS kernel一开始就是运行在内核态下的，因此使用int指令产生软中断的时候，硬件保存在stack上的信息中并不会包含原先的esp和ss寄存器的值，因此不妨在调用int指令产生软中断之前，使用pushl指令预想将设置好的esp和ss的内容push到stack上，这样就可以使得进入ISR的时候，trapframe上的形式和从用户态切换到内核态的时候保存的trapframe一致；具体代码实现为在lab1_switch_to_user函数中使用内联汇编完成，如下所示：

```
asm volatile (
    "movw %%ss, %0\n\t"
    "movl %%esp, %1"
    : "=a"(ss), "=b"(esp)
);
asm volatile (
    "pushl %0\n\t"
    "pushl %1\n\t"
    "int %2"
    :
    : "a"(ss), "b"(esp), "i"(T_SWITCH_TOU)
);
```

接下来在调用了int指令之后，会最终跳转到T_SWITCH_TOU终端号对应的ISR入口，最终跳转到trap_dispatch函数处统一处理，接下来的任务就是在处理部分修改trapframe的内容，首先为了使得程序在低CPL的情况下仍然能够使用IO，需要将eflags中对应的IOPL位置成表示用户态的3，接下来根据iret指令在ISA手册中的相关说明，可知iret认定在发生中断的时候是否发生了PL的切换，是取决于CPL和最终跳转回的地址的cs选择子对应的段描述符处的CPL（也就是发生中断前的CPL）是否相等来决定的，因此不妨将保存在trapframe中的原先的cs修改成指向用户态描述子的USER_CS，然后为了使得中断返回之后能够正常访问数据，将其他的段选择子都修改为USER_DS,然后正常中断返回；具体实现代码如下所示：

```
tf->tf_eflags |= FL_IOPL_MASK;
tf->tf_cs = USER_CS;
tf->tf_ds = tf->tf_es = tf->tf_gs = tf->tf_ss = tf->tf_fs = USER_DS;
```

事实上上述代码是具有不安全性的，这是由于在lab1中并没有完整地实现物理内存的管理，而GDT中的每一个段其实除了关于特权级的要求之外内容都是一样的，都是从0x0开始的4G空间，这就使得用户能够访问到内核栈的空间，即事实上上述代码并没有实际完成一个从内核栈到用户态栈的切换，仅仅是完成了特权级的切换而已；

至此完成了从内核态切换到用户态的要求；

从用户态切换到内核态

接下来考虑在用户态切换到内核态的情况，为了使得能够在用户态下产生中断号为T_SWITCH_TOK的软中断，需要在IDT初始化的时候，将该终端号对应的表项的DPL设置为3；接下来考虑在进行用户态切换到内核态的函数中使用int指令产生一个软中断，转到ISR，然后与切换到内核态类似的对保存的trapframe进行修改，即将trapframe中保存的cs修改为指向DPL为0的段描述子的段选择子KERNEL_CS，并且将ds, es, ss, gs, fs也相应地修改为KERNEL_DS，然后进行正常的中断返回，由于iret指令发现CPL和保存在栈上的cs的CPL均为0，因此不会进行特权级的切换，因此自然而不会切换栈并将栈上保存的ss和esp弹出。这就产生了中断返回之后，栈上的内容没能够正常恢复的问题，因此需要在中断返回之后将栈上保存的原本应当被恢复的esp给pop回到esp上去，这样才算是完整地完成了从用户态切换到内核态的要求；

具体实现的核心代码如下所示：

```
// 在ISR中修改trapframe的代码
case T_SWITCH_TOK:
    tf->tf_cs = KERNEL_CS;
    tf->tf_ds = tf->tf_es = tf->tf_gs = tf->tf_ss = tf->tf_fs = KERNEL_DS;
    break;
```

```
static void // 从用户态切换到内核态的函数
lab1_switch_to_kernel(void) {
    asm volatile (
        "int %0\n\t" // 使用int指令产生软中断
        "popl %%esp" // 恢复esp
        :
        : "i"(T_SWITCH_TOK)
    );
}
```

扩展练习:Challenge2(需要编程)

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。

拓展练习2的内容为实现“键盘输入3的时候切换到用户模式，输入0的时候进入内核模式”，该功能的实现基本思路与拓展练习1较为类似，但是具体实现却要困难需要，原因在于拓展1的软中断是故意在某一个特定的函数中触发的，因此可以在触发中断之前对堆栈进行设置以及在中断返回之后对堆栈内容进行修复，但是如果要在触发键盘中断的时候切换特权级，由于键盘中断是异步的，无法确定究竟是在哪个指令处触发了键盘中断，因此在触发中断前对堆栈的设置以及在中断返回之后对堆栈的修复也无从下手；（需要对堆栈修复的原因在于，使用iret来切换特权级的本质在于伪造一个从某个指定特权级产生中断所导致的现场对CPU进行欺骗，而是否存在特权级的切换会导致硬件是否在堆栈上额外压入ss和esp以及进行堆栈的切换，这使得两者的堆栈结构存在不同）

因此需要考虑在ISR中在修改trapframe的同时对栈进行更进一步的伪造，比如在从内核态返回到用户态的时候，在trapframe里额外插入原本不存在的ss和esp，在用户态返回到内核态的时候，将trapframe中的esp和ss删去等，更加具体的实现方法如下所示：

- 首先考虑从内核态切换到用户态的方法：
 - 从内核态切换到用户态的关键在于“欺骗”ISR中的最后一条指令iret，让CPU错以为原本该中断是发生在用户态下的，因此在最终中断返回的时候进行特权级的切换，切换到用户态，根据lab代码的内容，可以发现具体的每一个中断的处理是在trap_dispatch函数中统一进行的分类处理，而其中键盘中断的中断号为IRQ_OFFSET+IRQ_KBD，找到该中断号对应的case语句，在正常的处理流程之后，额外插入伪造栈上信息的代码，具体方法如下：
 - 将trapframe的地址保存到一个静态变量中，防止在接下来修改堆栈的时候破坏了堆栈，导致获取不到正确的trapframe地址；
 - 将整个trapframe以及trapframe以下（低地址部分）的堆栈上的内容向低地址部分平移8个字节，这使得trapframe的高地址部分空出来两个双字的空间，可以用于保存伪造的esp和ss的数值，这部分代码由于在操作过程中不能够使用到堆栈上的信息，为了保险起见，是在由汇编代码编写成的函数中完成的，具体为kern/trap/trapentry.S文件中的__move_down_stack2函数，该函数接受两个参数，分别为trapframe在高、低地址处的边界；
 - 由于上述操作对一整块区域进行向低地址部分的平移，这就会使得这块区域中保存的动态链信息出现错误（保存在栈上的ebp的数值），因此需要沿着动态链修复这些栈上的ebp的数值，具体方式为其减8；
 - 然后需要对ebp和esp寄存器分别减8，得到真正的ebp和esp的数值；
 - 最后，由于__alltraps函数在栈上保存了该函数调用trap函数前的esp数值，因此也需要将该esp数值修改成与平移过后的栈一致的数值，也就是平移过后的trapframe的低地址边界；
 - 上述三个操作为了保险起见，均使用汇编代码编写在函数__move_down_stack2中；
 - 然后在完成了堆栈平移，为伪造的ss和esp空出空间之后，按照拓展1的方法，对trapframe的内容进行修改，并且将伪造的esp和ss的数值填入其中；
 - 接下来正常中断返回，硬件由于原先的trapframe上的cs中的CPL是3，因此可以顺利切换到用户态，并且由于上述对堆栈的维护操作，在返回用户态之后仍然可以继续正常执行代码；
- 接下来考虑从用户态切换到内核态的方法：
 - 从用户态切换回内核态的关键仍然在于“伪造”一个现场来欺骗硬件，使得硬件误认为原先就是在内核态发生的中断，因此不会切换回用户态，具体实现方法如下：
 - 为了使得中断返回之后能够正常执行原先被打断的程序，不烦考虑在事实上为用户态的栈上进行现场伪造，首先将被保存在内核态上的自trapframe及以下（低地址）的所有内容都复制到原先用户态的栈上面去；（注意不要复制trapframe上的ss和esp）
 - 与切换到用户态相似的，对伪造的栈上的动态链（ebp）信息进行修复；
 - 对__alltraps函数压入栈的esp信息进行修复；
 - 上述代码为了保险期间，使用汇编语言实现，具体为trapentry.S文件的__move_up_stack2函数中；
 - 将伪造的栈上的段寄存器进行修改，使其指向DPL为0的相应段描述符；
 - 进行正常的中断返回，此时由于栈上的cs的CPL为内核态，因此硬件不会进行特权级的切换，从而使得中断返回之后也保持在内核态，从而完成了从用户态到内核态的切换；
- 实现本拓展所使用的汇编代码较为烦杂，因此未在实验报告中列出，要了解具体实现细节可以参考提交的代码文件；为了方便呈现实验效果，对init.c文件中的入口函数中的while(1)循环语句进行了修改，使得其可以在每个一段时间就打印出一次当前的CPU状态（包括特权级），然后得到的实验结果如下图所示。从图中可以看出当按下键盘数字3的时候，特权级切换到3（用户态），再按下键盘数字0的时候，特权级被切换到0（内核态）；即最终实验结果符合实验要求。

修改kern/trap.c中trap_dispatch(struct trapframe *tf)的代码如下：

```

case IRQ_OFFSET + IRQ_KBD:
    c = cons_getc();
    cprintf("kbd [%03d] %c\n", c, c);
    if (c == '0') {
        if (tf->tf_cs != KERNEL_CS) {
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_ss = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
        }
        print_trapframe(tf);
    }
    if (c == '3') {
        if (tf->tf_cs != USER_CS) {
            tf->tf_cs = USER_CS;
            tf->tf_ds = tf->tf_ss = tf->tf_es = USER_DS;
            tf->tf_eflags |= FL_IOPL_MASK;
        }
        print_trapframe(tf);
    }
    break;

```

实验中涉及的知识列举

在本实验设计到的知识点分别有：

- 基于分段的内存管理机制；
- CPU的中断机制；
- x86 CPU的保护模式；
- 计算机系统的启动过程；
- ELF文件格式；
- 读取LBA模式硬盘的方法；
- 编译ucore OS和bootloader的过程；
- GDB的使用方法；
- c函数的函数调用实现机制；

对应到的OS中的知识点分别有：

- 物理内存的管理；
- 外存的访问；
- OS的启动过程；
- OS中对中断机制的支持；
- OS使用保护机制对关键代码和数据的保护；

两者的关系为，前者硬件中的机制为OS中相应功能的实现提供了底层支持；