1.1
make "V=" :

```
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
```

-fno-builtin：禁用了内建函数，
-nostdinc：不在标准系统目录中搜索头文件
-fno-stack-protector：禁用堆栈保护
-I：指定链接的库
-c：只编译不链接
-m32：产生32位程序
-ggdb、-gstabs：与gdb调试相关
编译kern/init/init.c产生目标文件obj/kern/init/init.o

```
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
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/stdio.c -o obj/kern/libs/stdio.o
gcc -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
gcc -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
gcc -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
gcc -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
gcc -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
gcc -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
gcc -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
gcc -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
gcc -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
gcc -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
gcc -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
gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-
stack-protector -Ilibs/  -c libs/printfmt.c -o obj/libs/printfmt.o
gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-
stack-protector -Ilibs/  -c libs/string.c -o obj/libs/string.o
```

编译kern/libs/readline.c、kern/libs/stdio.c、kern/debug/kdebug.c、kern/debug/kmonitor.c、kern/debug/panic.c、kern/driver/clock.c、kern/driver/console.c、kern/driver/intr.c、kern/driver/picirq.c、kern/trap/trap.c、kern/trap/trapentry.S、kern/trap/vectors.S、kern/mm/pmm.c、printfmt.c、string.c生成目标文件

```
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
```

-m：指定运行程序格式（此处使用elf_i386格式）
-nostdlib：不链接标准库
-T：指定链接脚本（此处使用tools/kernel.ld）
链接刚编译生成的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，产生bin/kernel

```
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-
stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/
bootasm.o
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-
stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/
boot/bootmain.o
```

-Os：开启编译优化
编译boot/bootasm.S、boot/bootmain.c生成目标文件

```
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
```

-O2：开启编译优化
编译tools/sign.c生成目标文件，并最终生成运行程序bin/sign

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

-N：设置text和data段可读写
-e：指定程序入口点
-Ttext：指定链接时初始重定向地址（此处指定为0x7C00）

链接obj/boot/bootasm.o obj/boot/bootmain.o生成目标文件obj/bootblock.o

cgtype = $(patsubst %.$(2),%.$(3),$(1))
outfile = $(call cgtype,$(call toobj,$(1)),o,out)
@$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
objcopy -S -O binary bootblock.o bootblock.out
-S：删掉包含调试信息的部分
-O binary：生成原始二进制文件
通过obj/bootblock.o生成obj/bootblock.out

@$(call totarget,sign) $(call outfile,bootblock) $(bootblock)
'obj/bootblock.out' size: 472 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
通过刚才编译生成的bin/sign将obj/bootblock.out生成bin/bootblock

dd if=/dev/zero of=bin/ucore.img count=10000
生成10000个块大小，即5120000字节的bin/ucore.img，内容全为0

dd if=bootblock of=bin/ucore.img conv=notrunc
conv=notrunc意味着不缩减输出文件,也就是说,如果输出文件已经存在,只改变指定的字节，然后退出，并保留输出文件的剩余部分
将bin/bootblock拷贝到bin/ucore.img的前512个字节

dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
seek=1意味着跳过输出文件的一个块后开始复制
将bin/kernel拷贝到bin/ucore.img第513个字节开始的位置，实际拷贝大小为70783字节

Makefile:
1-137：定义各种常量和函数
138-153：生成bin/kernel
154-170：生成bin/bootblock
171-176：生成bin/sign
177-188：生成bin/ucore.img
189-202：定义常量
203-267：定义各种make目标

1.2
根据上面对make "V="的分析，可以看出主引导扇区大小为512个字节，同时查看sign.c可以看出，主引导扇区要以0x55AA结尾

2.1
make debug之后，程序暂停在：
0x0000fff0 in ?? ()
Breakpoint 1 at 0x100000: file kern/init/init.c, line 17.
即程序已经进入了ucore操作系统，单步可发现程序在初始化console、内存管理、各中断后，即进入while (1);死循环。

由于已知第一条指令在0xffffff0处，所以将tools/gdbinit修改为：

```
file bin/kernel
target remote :1234
break kern_init
```

即删除continue，再用make debug启动，会发现程序停在0xf000:0xffff，由于还在8086模式下，故0xffffff0即相当于是0xfffff，此处命令为：

```
   0xffffff0:  ljmp   $0xf000,$0xe05b
```

然后si，会跳转到0xf000:0xe05b，此处即为BIOS
然后再break *0x7c00并continue，会发现程序暂停在0x7c00，查看0x100000处的代码：

```
   0x100000 <kern_init>:        add    %al,(%eax)
   0x100002 <kern_init+2>:      add    %al,(%eax)
   0x100004 <kern_init+4>:      add    %al,(%eax)
```

即由0x7c00处的bootloader首先执行，载入ucore到0x100000，然后跳转到0x100000执行启动操作系统

2.2
在gdb中，b *0x7c00，然后c：

```
(gdb) c
Continuing.

Breakpoint 2, 0x00007c00 in ?? ()
```

程序成功暂停在0x7c00处

2.3
通过单步跟踪以及x/i、x/x发现，反汇编得到的结果与bootblock.S、bootblock.asm的代码基本一致，bootblock.asm比bootblock.S更详细，但是有个别语句不一致，但16进制内容一致，如0x7c1e-0x7c25：
反汇编结果：

```
    0x7c1e  lgdtw  0x7c6c
    0x7c23  mov    %cr0,%eax
    0x7c26  or     $0x1,%eax
```

bootblock.asm：

```
    lgdt gdtdesc
    7c1e:   0f 01 16                lgdtl  (%esi)
    7c21:   6c                      insb   (%dx),%es:(%edi)
    7c22:   7c 0f                   jl     7c33 <protcseg+0x1>
    movl %cr0, %eax
    7c24:   20 c0                   and    %al,%al
    orl $CR0_PE_ON, %eax
    7c26:   66 83 c8 01             or     $0x1,%ax
```

发现是由于gdb中设置了set architecture i8086导致，没设置时两边代码一致

2.4
测试memset，首先b memset，然后c：

```
(gdb) c
Continuing.

Breakpoint 3, memset (s=0x10da16, c=0 '\000', n=4874) at libs/
string.c:273
```

然后查看代码如下：

```
    270     void *
```

```
   271     memset(void *s, char c, size_t n)
{
   272     #ifdef __HAVE_ARCH_MEMSET
B+>273         return __memset(s, c, n);
   274     #else
   275         char *p = s;
   276         while (n -- > 0)
{
   277             *p ++ = c;
   278         }
   279         return s;
   280     #endif /* __HAVE_ARCH_MEMSET */
   281     }
```

继续s：

```
   140     static inline void *
   141     __memset(void *s, char c, size_t n)
{
   142         int d0, d1;
  >143         asm volatile
(
   144                 "rep; stosb;"
   145                 : "=&c" (d0), "=&D" (d1)
   146                 : "0" (n), "a" (c), "1" (s)
   147                 : "memory");
   148         return s;
   149     }
```

同时可得反汇编代码为：

```
  >0x1030a0 <memset+32>    mov    -0x10(%di),%cx
   0x1030a3 <memset+35>    movzbw -0x9(%di),%ax
   0x1030a7 <memset+39>    mov    -0x8(%di),%dx
   0x1030aa <memset+42>    mov    %dx,%di
   0x1030ac <memset+44>    rep stos %al,%es:(%di)
   0x1030ae <memset+46>    mov    %di,%dx
```

```
   |0x1030b0 <memset+48>    mov    %cx,-0x14(%di)
|
   |0x1030b3 <memset+51>    mov    %dx,-0x18(%di)
|
   |0x1030b6 <memset+54>    mov    -0x8(%di),%ax
|
   |0x1030b9 <memset+57>    add    $0x24,%sp
|
   |0x1030bc <memset+60>    pop    %di
|
   |0x1030bd <memset+61>    pop    %bp
|
   |0x1030be <memset+62>    ret
```

3.
bootloader首先关中断、初始化段寄存器
然后打开A20 Gate（等待8042 input buffer empty后，向0x64端口写入0xd1，然后再等待8042 input buffer empty后，向0x60端口写入0xdf打开A20 Gate）
接着将全局描述符表入口加载到寄存器后，修改控制寄存器CR0的第0位（PE位）转换模式（为0表示实模式，为1表示保护模式）
最后通过ljmp以32位的格式跳转到下一条命令完成实模式到保护模式的切换

4.1
读取扇区：readsect首先等待硬盘就位，然后向IO地址为0x1F2-0x1F7写入对应参数预备读取，然后再等待硬盘就位后，就可以读取对应扇区（读取扇区采取LBA模式）

4.2
bootloader通过调用readseg读入一个页作为elf header，readseg通过调用readsect读入特定编号的扇区
然后验证elf header的e_magic为ELF_MAGIC，接着按照elf header中的e_phoff获取program header指针，并按照program header得到程序的各个段信息，通过readseg读入各个段
最后依照elf header的e_entry最后跳转到程序的入口地址处，将运行权交给载入的ELF格式的OS

5.
实现时，首先按照注释，输出当前栈桢的情况和参数，不循环
然后观察bootblock.asm，发现在调用bootmain之前，ebp被清0，esp被置为0x7c00，故可以将ebp == 0作为循环的终止条件
然后实现循环之后发现最后会多出一行：
```
    <unknow>: -- 0x00007d67 --
```
那么，修改代码，最后一次的print_stackframe不要调用

最后一行为：ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
各数据分别意味着：
由bootblock的protcseg调用进入bootmain后，bootmain的栈桢为0x00007bf8
而bootmain调用调用kernel init启动ucore时，应该是运行到了0x00007d68的前面一条命令处，即0x00007d66处的call   *%eax
而args对应的4个参数并不是真正的意义上的参数，因为bootmain调用kernel init的时候是没有参数的，这4个参数实际是只是栈顶之上的内容，而栈顶之上为0x7c00，也就是

bootloader的代码，也就是说这16个字节是bootloader的代码的前16个字节：

```
(gdb) x/16bx 0x7c00
0x7c00: 0xfa    0xfc    0x31    0xc0    0x8e    0xd8    0x8e    0xc0
0x7c08: 0x8e    0xd0    0xe4    0x64    0xa8    0x02    0x75    0xfa
```

6.1

中断向量表一个表项占8个字节，0-15位代表offset的0-15位，48-63位代表offset的16-31位，16-31位为段选择子，通过段选择子从GDT从取得段基址，段基址加上偏移offset才是中断处理代码的入口

6.2

首先extern声明__vectors

然后查看SETGATE宏的定义，发现注释：

```
/* *
 * Set up a normal interrupt/trap gate descriptor
 *   - istrap: 1 for a trap (= exception) gate, 0 for an interrupt
gate
 *   - sel: Code segment selector for interrupt/trap handler
 *   - off: Offset in code segment for interrupt/trap handler
 *   - dpl: Descriptor Privilege Level - the privilege level
required
 *          for software to invoke this interrupt/trap gate
explicitly
 *          using an int instruction.
 * */
#define SETGATE(gate, istrap, sel, off, dpl)
```

那么我们就应该是SETGATE(idt[i], 0, sel, __vectors[i], 0)，只差sel了

查看GDT的初始化：

```
static struct segdesc gdt[] = {
    SEG_NULL,
    [SEG_KTEXT] = SEG(STA_X | STA_R, 0x0, 0xFFFFFFFF, DPL_KERNEL),
    [SEG_KDATA] = SEG(STA_W, 0x0, 0xFFFFFFFF, DPL_KERNEL),
    [SEG_UTEXT] = SEG(STA_X | STA_R, 0x0, 0xFFFFFFFF, DPL_USER),
    [SEG_UDATA] = SEG(STA_W, 0x0, 0xFFFFFFFF, DPL_USER),
    [SEG_TSS]    = SEG_NULL,
};
```

那么此处sel应该对应于[SEG_KTEXT]这个段，那么sel的值就应该是SEG_KTEXT << 3

于是乎，初始化idt，并单独设置idt[T_SYSCALL]，最后再lidt一下即可

至此程序已经完成，但查找了一下SEG_KTEXT的定义，发现：

```
/* global segment number */
#define SEG_KTEXT    1
#define SEG_KDATA    2
#define SEG_UTEXT    3
#define SEG_UDATA    4
#define SEG_TSS        5

/* global descrptor 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

```
#define DPL_KERNEL    (0)
#define DPL_USER    (3)

#define KERNEL_CS    ((GD_KTEXT) | DPL_KERNEL)
#define KERNEL_DS    ((GD_KDATA) | DPL_KERNEL)
#define USER_CS        ((GD_UTEXT) | DPL_USER)
#define USER_DS        ((GD_UDATA) | DPL_USER)
```

故我们可以将SEG_KTEXT << 3替换成KERNEL_CS，将dpl使用DPL_KERNEL

6.3

ticks在include的clock.h中已经声明，故不用再声明

然后，每次运行到该case中就意味着一次时钟中断，于是++ticks，然后判断ticks为TICK_NUM的整数倍意味着一轮输出

Challenge:

题目中要求写syscall，可查看switch_test却可发现，题目实际让我们实现的不是syscall，而是权限切换的函数。

首先是内核态切到用户态的函数:

题目建议通过中断处理实现，于是乎在trap.h中查找中断号，发现:

```
#define T_SWITCH_TOU                120    // user/kernel switch
#define T_SWITCH_TOK                121    // user/kernel switch
```

然后，查看中断处理函数入口(vector.S)，会发现所有中断处理函数都是trapentry.S中的__alltraps函数:

而__alltraps就是将各种寄存器压栈保存，然后调用trap.c中的trap函数，最后恢复各寄存器并返回，在trap.h中可发现压栈保存的数据结构:

```
/* registers as pushed by pushal */
struct pushregs {
    uint32_t reg_edi;
    uint32_t reg_esi;
    uint32_t reg_ebp;
    uint32_t reg_oesp;            /* Useless */
    uint32_t reg_ebx;
    uint32_t reg_edx;
    uint32_t reg_ecx;
    uint32_t reg_eax;
};

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));
```

于是可发现，我们想要做特权级的切换，就是要修改cs、ds、es等段寄存器，而通过中断修改，最好的办法就是直接将保存在栈中这些数据修改，等返回时就会将寄存器调整为我们需要的值，实现权限切换。
然后需要注意的是，在由内核态切换成用户态的时候，一开始调用中断时，由于是从内核态调用的，没有权限切换，故ss、esp没有压栈，而iret返回时，是返回到用户态，故ss、esp会出栈，于是为了保证栈的正确性，需要在调用中断前将esp减8以预留空间，中断返回后，由于esp被修改，还需要手动恢复esp为正确值。
这样之后，系统特权级已经成功切换，但是由于切换到了用户态，导致IO操作没有权限，故之后的printf无法成功输出，为了能够正常输出，我们需要将eflags中的IOPL设成用户级别，即3，同样也是通过修改栈中值来达到修改的目的。

接下来是从用户态切换到内核态，此时由于调用中断时，是从用户态调用的，故会将ss、esp压栈，但是iret返回时，是返回到内核态，故ss、esp不会出栈，所以此时需要手动出栈，而ss由于维持了中断内部的ss，即已经是kernel，无需修改，故直接出栈esp即可。最后，为了能够让用户态的时候可以调用此中断，需要在idt的初始化函数中将该中断的DPL改成USER级别。

最后，再实现一下syscall，获得时钟计数值，syscall number采取255，于是便可以得到：

```
static int get_ticks(void) {
    asm("mov $0xff, %eax");
    asm("int $0x80");
}
```

最后在lab1_switch_test里面的kernel切到user后，再切回kernel前进行测试，发现可正确获取ticks。

PS：这个challenge的题目描述实在是给跪了，语句严重不通顺，实在无力吐槽；然后user态怎么可以随便通过一个int中断调用，就直接获取完全的kernel权限，这样的操作系统还有安全性可言么，实在不能理解！！！