实验1报告

练习1：理解通过make生成执行文件的过程

列出本实验各练习中对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。

在此练习中，大家需要通过静态分析代码来了解：

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

分析：为了说明这个问题，首先需要确定解答思路。

在Makefile文件中，生成ucore.img镜像文件的内容如下：

```
# create ucore.img
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)
```

根据makefile文件的基本格式可知，生成时，目标文件为UCOREIMG变量指代的文件名，而依赖文件则是生成目标文件的必须文件，在此为kernel和bootblock两个变量指代的文件。

因此，生成ucore.img，首先要得到kernel和bootblock文件。

从Makefile中，可以获得kernel的生成代码如下：

```
# create kernel target
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)
```

根据Makefile的相关语法可知，生成kernel文件的依赖文件是tools目录中的kernel.ld，以及一个KOBJS变量指定的文件。

但是，只通过这种观察Makefile文件内容的方式来获得程序执行信息是不够直观的，可以在中断中通过输入make V=指令来查看Makefile文件中生成的指令内容。

make V=输入后，从中查找与kernel可执行程序生成有关的代码，内容如下：

```
+ 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
```

上述显然是一个链接命令，ld是命令的名字表示，表示需要进行连接；kernel.ld是链接脚本，用来指导ld命令如何把诸多.o文件链接为可执行文件kernel。所以，生成kernel可执行文件首先需要一个指导链接操作的kernel.ld脚本文件，其次需要若干用于生成可执行文件的.o文件，这些.o文件很多，从init.o一直到printfmt.o。

> *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 ： 表示不使用标准库参与到链接之中

-T：指定用来制导链接的脚本文件，也就是使用kernel.ld作为脚本文件来执行链接
```

通过make V=指令，可以查看到用于生成这些.o文件的内容如下：

```
+ cc kern/init/init.c
gcc -Ikern/init/ -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
+ cc kern/libs/stdio.c
gcc -Ikern/libs/ -fno-builtin -fno-PIC -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 -Ikern/libs/ -fno-builtin -fno-PIC -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 -Ikern/debug/ -fno-builtin -fno-PIC -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
+ cc kern/debug/kdebug.c
gcc -Ikern/debug/ -fno-builtin -fno-PIC -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
+ cc kern/debug/kmonitor.c
gcc -Ikern/debug/ -fno-builtin -fno-PIC -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 -Ikern/driver/ -fno-builtin -fno-PIC -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 -Ikern/driver/ -fno-builtin -fno-PIC -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 -Ikern/driver/ -fno-builtin -fno-PIC -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 -Ikern/driver/ -fno-builtin -fno-PIC -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 -Ikern/trap/ -fno-builtin -fno-PIC -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
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -fno-builtin -fno-PIC -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 -Ikern/trap/ -fno-builtin -fno-PIC -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 -Ikern/mm/ -fno-builtin -fno-PIC -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 -Ilibs/ -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/  -c libs/string.c -o obj/libs/string.o
+ cc libs/printfmt.c
```

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

依次执行这些指令后，就可以从已有的.c文件出发，利用gcc命令编译得到.o文件；这些.o文件将被链接为需要的kenerl可执行程序。

各个命令的基本格式大同小异，其中部分参数的含义解释如下：

> *参数说明：*

```
1  -Ikern/与-Ilibs/：以大写的字母i开头并附加一个目录名，表示在进行gcc编译时指定搜
   索.c文件中头文件的位置。如果.c文件中包含了许多头文件，就通过-I指定的目录前去查
   找，从而生成正确的.o文件。
2
3  -fno-builtin：这个选项表示不使用c语言的内建函数，可以解决自定义函数名与c语言库
   中函数名重名导致的冲突问题。
4
5  -fno-PIC：据说表示不使用位置无关代码，即允许引用绝对地址。
6
7  -Wall：表示开启所有警告，可以及时显示警告信息。
8
9  -ggdb：据说可以使 GCC 为 GDB 生成比-g选项更为丰富的调试信息。
10
11 -m32：指定编译为32位的可执行程序。
12
13 -gstabs：以stabs格式声称调试信息，但是不包括gdb调试信息。
14
15 -nostdinc：表示不要在标准系统目录中寻找头文件.只搜索-I'选项指定的目录；与上文
   中的-I用法遥相呼应。
16
17 -fno-stack-protector：禁用栈保护措施
```

以上内容说明了kernel部分是如何从多个.c文件逐步生成的；考虑到为了生成ucore.img还需要bootblock文件，在Makefile中查找相应内容如下：

```
1  # create bootblock
2  bootfiles = $(call listf_cc,boot)
```

```
3  $(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -
   nostdinc))
4
5  bootblock = $(call totarget,bootblock)
6
7  $(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
8    @echo + ld $@
9    $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call
   toobj,bootblock)
10   @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
11   @$(OBJDUMP) -t $(call objfile,bootblock) | $(SED) '1,/SYMBOL TABLE/d;
   s/ .* / /; /^$$/d' > $(call symfile,bootblock)
12   @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call
   outfile,bootblock)
13   @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)
14
15 $(call create_target,bootblock)
```

从上述内容来看，为了生成bootblock，依赖包括sign和bootfiles指定的相关文件。

首先，查看bootfiles相关文件有哪些：在输入make V=后终端显示部分中可以查到如下内容：

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

可见，为了生成bootblock.o，需要的.o文件包括bootasm.o、bootmain.o。继续查看，生成这两个文件的指令如下：

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

相关参数的意义已在前文中说明。

接下来，查看sign的生成：

```
+ 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
```

> *参数说明*

```
只说明前文未解释的参数：
-O2：指定编译优化级别为2。
```

由此，就解释了kernel和bootblock的生成途径，它们将作为依赖使得ucore.img镜像的生成成为可能。

在make V=显示的、有关镜像生成的指令如下：

```
dd if=/dev/zero of=bin/ucore.img count=10000
记录了10000+0 的读入
记录了10000+0 的写出
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0182823 s, 280 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.000290929 s, 1.8 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
记录了146+1 的读入
记录了146+1 的写出
```

一共执行了三次dd指令。关于dd指令的作用和这里执行的功能如下：

dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换。

if参数指定了源文件，of参数指定了目标文件；要从源文件拷贝到目标文件。

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

这条命令的作用是拷贝10000个全部为0的块，存放到bin目录下新建的ucore.img文件之中；dev/zero提供一个零设备，用于提供无数个零。

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

这条命令的作用是将bootblock中的内容写入ucore.img之中，从第一个块开始写。

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

这条命令的作用是将kernel中的内容写入ucore.img之中，由于seek为1，跳过了第一个块，从第二个快开始写。由于bootblock占用的空间为512个字节，可以想见，ucore.img的10000个块中，第一个块存放bootblock程序，剩下的存放kernel程序。

相关原理：磁盘镜像是一个模拟的磁盘，计算机启动时需要从这里读取数据。首先，需要执行BIOS程序，这个程序会对CPU进行一定程度的初始化，并从磁盘的第一个块（也就是主引导扇区）里加载bootblock进入内存；bootblock程序的作用是修改CPU从实模式变为保护模式，同时加载磁盘中剩余块里的kernel内核代码。bootblock对应的是所谓加载程序，没有它就无法从磁盘中获取实现操作系统功能的内核代码；kernel是真正的操作系统内核程

序，bootblock将它加载入内存后，就将控制权转移给它并开始运行操作系统。

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?

根据慕课中得到的知识，磁盘中的一个块，或者说一个扇区，一般有512个字节；其中，用于存放bootblock的扇区中，数据需要满足一定的格式，从而确保这个扇区是正确的主引导扇区。

所以，大小为512个字节，且最后两个字节是55AA也就是验证字节的扇区是合格的硬盘主引导扇区。

练习2：使用qemu执行并调试lab1中的软件。

为了熟悉使用qemu和gdb进行的调试工作，我们进行如下的小练习：

1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。
2. 在初始化位置0x7c00设置实地址断点,测试断点正常。
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。
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。

分析：

CPU加电之后，将会执行BIOS也就是固化在内存中的初始代码，这一部分内容将完成一些基本的初始化并将主引导扇区的内容放入内存，然后转移控制权给那一部分代码。

1.从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

根据实验指导书附录B，修改 lab1/tools/gdbinit文件的内容为：

```
set architecture i8086

target remote :1234
```

接着，在lab1的目录下打开终端，输入make debug；之后，会弹出多个窗口，在调试窗口中输入si就可以单步跟踪执行，也可以用x指令打印特定数目的汇编代码。

2.在初始化位置0x7c00设置实地址断点,测试断点正常。

根据学堂在线的相关教学，接下来我通过执行make lab1-mon指令来完成剩余的任务。

make lab1-mon将指定执行Makefile文件中以它命名的相关指令。

```
lab1-mon: $(UCOREIMG)
  $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -monitor stdio -hda $< -serial null"
  $(V)sleep 2
  $(V)$(TERMINAL) -e "gdb -q -x tools/lab1init"
```

如图，下一条似乎是在终端输入了gdb调试命令。-q可以简化输出，不显示版权说明；-x的作用是以指定文件的内容为gdb的命令。也就是说，使用make lab1-mon可以把lab1init文件中的内容作为命令使用。

其文件中的内容如下：

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

```

含义解析：

file bin/kernel：表示加载bin目录下的可执行文件kernel作为调试对象。注意，此练习中不需要调试kernel中的内容，因为指定的断点地址不是kernel的部分，而是加载程序的部分。target remote :1234 表示链接qemu。 set architecture i8086 表示指定指令架构 b *0x7c00 设置断点 continue 继续运行直到断点出现。由于上一行设置了断点，所以执行continue后会停在断点位置。 x /2i $pc 使用examine命令查看pc变量（也就是地址寄存器中的地址）

所对应的内存地址的值。/2i的意思是，n为2，从当前地址往后显示2个内存单元的内容。x表示显示，则/2i的意思就是显示两条指令。

总结

首先进行gdb bin/kernel，加载内核程序（但是还不会执行）

然后建立与qemu的连接，指定i8086架构。

随后用b *0x7c00指定了断点。这个位置是bootloader引导加载程序的第一条指令的地址。

设置断点后执行continue，就会运行程序并停在0x7c00处。

然后用x /2i $pc，查看内存中从当前pc指令寄存器中开始的两条指令。

操作

在一个终端里执行make lab1-mon后，一下子弹出了三个终端。

调试界面显示调试的效果，已经设置了断点，并显示了当前pc中指令地址是0x7c00，且指令的名字是cli(这是靠set architecture来确定的)；同时，显示了从0x7c00开始的两条指令。

终端显示如下，可见断点设置成功。

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

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。

可以直接通过x命令查看任意数量的代码并进行比较。

距离，显示10条指令如下：

```
(gdb) x /10i $pc
=> 0x7c00:  cli
   0x7c01:  cld
   0x7c02:  xor    %ax,%ax
   0x7c04:  mov    %ax,%ds
   0x7c06:  mov    %ax,%es
   0x7c08:  mov    %ax,%ss
   0x7c0a:  in     $0x64,%al
   0x7c0c:  test   $0x2,%al
   0x7c0e:  jne    0x7c0a
   0x7c10:  mov    $0xd1,%al
(gdb)
```

接下来对照bootasm.S文件中的内容:

```
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

    # 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

```

```
20      movb $0xd1, %al                                 # 0xd1 -> port 0x64
```

可见，二者的汇编代码部分是一致的。

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。

选择0x7c0c作为断点并用c指令执行到断点为止，效果如下：

```
Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:  cli
   0x7c01:  cld
(gdb) x /17i $pc
=> 0x7c00:  cli
   0x7c01:  cld
   0x7c02:  xor    %ax,%ax
   0x7c04:  mov    %ax,%ds
   0x7c06:  mov    %ax,%es
   0x7c08:  mov    %ax,%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
   0x7c1c:  out    %al,$0x60
   0x7c1e:  lgdtw  0x7c6c
(gdb) b *0x7c0c
Breakpoint 2 at 0x7c0c
(gdb) c
Continuing.

Breakpoint 2, 0x00007c0c in ?? ()
(gdb) x /5i $pc
=> 0x7c0c:  test   $0x2,%al
```

```
   0x7c0e:  jne    0x7c0a
   0x7c10:  mov    $0xd1,%al
   0x7c12:  out    %al,$0x64
   0x7c14:  in     $0x64,%al
(gdb)
```

练习3：分析bootloader进入保护模式的过程。

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。

提示：需要阅读小节“保护模式和分段机制”和lab1/boot/bootasm.S源码。

分析：

CPU上电后，首先处于实模式之下，在这个模式中所使用的地址空间只有1M字节大小；CPU会执行固化在内存中的BIOS程序，这一部分程序会执行一些最基本的初始化功能，同时读取硬盘主引导扇区中的引导加载程序，也就是bootloader和相关程序，将其从硬盘中转移到内存里；接着，将控制权转移给引导加载程序。

bootloader执行时会切换CPU的模式为保护模式，保护模式下可以拥有32位地址空间和保护机制，然后读取磁盘并加载执行文件（这个文件是操作系统内核）。根据练习1，可以知道磁盘的第一个块里存放着512个字节的引导加载程序，从第二个块开始就是kernel操作系统内核程序。

bootloader完成的工作包括：

1、切换到保护模式，启用分段机制（在实模式下，不仅地址空间很小，而且无法执行分段机制）

2、读取磁盘中ELF执行文件格式的ucore操作系统到内存（这正是它被成为引导加载程序的原因所在，因为它可以从磁盘中读取操作系统的内核程序）

3、显示字符串信息

4、把控制权交给ucore操作系统（开始执行已经在第2步被加载到内存中的具体ucore内核程序）

根据实验指导书说明，从实模式到保护模式的转换是通过修改A20地址线实现的。

考虑到切换模式时意味着启用分段机制，需要对分段机制进行部分说明：

保护模式下，程序中使用的地址是32位的逻辑地址，逻辑地址需要转化为线性地址，最后转化成对应真正内存地址的物理地址。这一转化过程的实现就是分段机制的核心。

由于内存一共32位，对应的是是4GB大小的空间。所以，线性地址空间和物理地址空间一共有4G字节。

保护模式下，一共有两个段表（GDT和LDT，在ucore中只使用GDT）。分段机制的基本实现方式就是通过段表存放很多段描述符，每一个段描述符对应一个段，比如说内核数据段、内核代码段、用户数据段、用户代码段就是四个不同的段，每个段在内存中都占据一部分特定的空间，其起始地址和大小信息就存放在对应的段描述符中。

所以，GDT中存放的各个段描述符实际上就描述了各个段的位置信息和某些属性。保护模式下，逻辑地址中会被拆分出一部分数据作为索引，用于指定GDT中的某一个段描述符，说明当前逻辑地址对应的指令存放在内存的哪一个段之中；根据这个索引，可以从GDT中查找到相应的段描述符，然后得到段的起始地址和相关属性；接着，逻辑地址的剩余部分将作为段偏移量，用于和段的起始地址组合在一起，以得到真正的内存地址。

从上述段机制的实现思想可以看出，CPU会将逻辑地址分为两部分，段选择子selector和段偏移量offset（段选择子的内容往往对应着段寄存器，段寄存器会存放当前的段选择子内容）。段选择子作为索引查到GDT中的段描述符，得到段的起始地址，结合段偏移量offset获得线性地址。**在不使用分页机制的时候，根据逻辑地址、结合分段机制算出的线性地址就是物理地址，可以直接用来从内存中查找相应指令或数据。**

段寄存器：CS代码段；DS数据段；SS堆栈段；ES附加段。

虽然前文说逻辑地址中包含了段选择子，但实际上，根据课本知识，一个逻辑地址往往需要先被判定处于哪一个段，然后操作系统会直接从段寄存器中获得段选择子，然后去查找段描述符信息并组合偏移量得到线性地址。

综上所述，为了切换模式并同时开启段机制，需要一个GDT表用来存放段描述符；同时，需要更新段寄存器，用来说明当前段的具体信息。

A20的作用

早期8086的CPU只提供了20根地址线，因此寻址空间有1MB，但使用的寄存器却只有16位。于是，8086提供了段地址加偏移地址的寻址方式。

其中，段寄存器有16位，偏移地址也有16位；为了得到真正地址，首先会将段寄存器中的数据左移4位得到20位的段基址（也就是段的起始地址），然后用20位的段机制加上16位的偏移地址，得到20位的真正地址。

但是这种表示方法的问题在于，两个16位数据组合得到20位数据，但同样有可能超过1MB大小；也就是说，可以表示的地址范围大于了真正的地址范围。为此，一旦出现寻址大于1MB的时候，就需要进行“回卷”操作。

随着时代发展，CPU开始提供更多的地址线，可以表示的地址空间已经大于1MB了；此时，一旦访问大于1MB的地址，是不需要进行回卷的。问题在于，这导致了向下不兼容。于是，A20 Gate应运而生。

如果不打开A20，就会保留回卷机制，禁止访问大于1MB的空间，从而实现向下兼容，保留在实模式；打开A20，就会撤销回卷机制，运行访问大于1MB的空间。

修改A20的实现：

8042芯片结构中，有三个内部端口：

Input Port输入端口，记为P1；

Output Port输出端口，记为P2；

其中，输出端口P2的位1部分，也就是P21引脚可以控制A20信号的开启与否。

有四个寄存器：

只能写的8位寄存器Input buffer;

只能读的8位寄存器Output buffer；

只能读的8位寄存器Status Register；

可读可写的8位寄存器：Control Register;

程序可通过**60h**和**64h**端口操作寄存器。

直接读60h端口，可以获得output buffer寄存器中的内容;

直接写60h端口，可以写入input buffer寄存器内容。

直接读64h端口，可以读Status Register寄存器的内容。

通过向64h端口输入特定的命令，则可以读写P2端口，从而控制P21引脚:

读Output Port（P2）操作：先向64h端口发送0d0h命令，然后就可以从60h端口读取当前P2的内容;

写Output Port（P2）操作：先向64h端口发送0d1h命令，然后就可以向60h端口写入当数据，这些数据将被放入P2端口之中。

为了操作A20，就需要向P2写入数据；为此，需要先向64h端口发送0d1h命令表示即将写入，再向60h端口写入将赋值给P2的数据，进而控制P21引脚。这两个操作无论是发送命令还是写入数据，都涉及对输入缓冲寄存器的操作；在操作输入缓冲寄存器之前，必须确保输入缓存寄存器已经是空的。可以通过从64h读取状态寄存器内容，来查看当前输入缓存是否为空。

根据实验指导书说明，从64h读取的状态寄存器内容中，bit 1位可以表示输入缓存寄存器中是否已经有数据。只有等这一位说明缓存为空时，才能向64h发出指令、写入数据。

代码解析如下:

```
.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
```

首先，设置三个段寄存器内容。第一步让ax寄存器与自身异或，确保ax寄存器中有8位的0；接着，用ax寄存器的额内容赋值给ds和es和ss三个段寄存器，将其初始化为0。

```
# 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

```

接着执行向64h发送0d1h命令的操作。首先，用inb指令从64h端口读取8位数据存入al寄存器，其中内容就是状态寄存器；接着，读取状态进入al寄存器后，testb指令将0000 0010与状态寄存器的内容进行按位与；实际上，就是提取了8位中的bit1位。如果test之后得到全零，说明这一位的值是0；反之，这一位的值是1。

jnz表示在结果不为零的时候转移，实际上就是在1位为1的时候转移到这段代码的开头重新执行，直到1位为零为止。状态寄存器中的bit 1位为0，意味着input register里面没有数据，也就是说input buffer为空。

上图的循环持续到输入缓存为空为止，之后执行最后两行，也就是向64h输入命令d1h。

接下来需要向60h输入数据，这一数据将被赋值给P2端口，从而控制P21线。

```
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

```

格式与之前类似，都是先循环等待缓存为空，然后向60h端口输入给P2的值，也就行1101 1111，这一数据中的bit 1位是1，可以打开A20，从而切换到保护模式。

切换到保护模式之后，接下来需要启动分段机制：

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

上述代码中，lgdt命令的作用是加载数据到gdt段表中。这里就是在执行载入，初始化gdt段表，用于段保护机制的实现。

```
.set CR0_PE_ON,             0x1                     # protected mode
enable flag
```

考虑到CR0_PE_ON的值是0000 0001，所以上述代码将cr0寄存器中的最低位置为1，从而代开保护模式的使能位，正式进入保护模式。

最后，使用ljmp指令。

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

ljmp指令注释含义：注释含义：#跳转到下一条指令，但在32位代码段中。#将处理器切换到32位模式。

```
.set PROT_MODE_CSEG,        0x8                     # kernel code segment
selector
```

从参数可知，使用了内核代码段选择子。

ljmp执行的结果是将PROT_MODE_CSEG的值加载到CS代码段寄存器中，同时将protcseg的值加载到ip偏移量寄存器中，得到一个地址，然后ljmp无条件跳转到那里。

其中，protcseg部分代码的内容如下：

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

```
8       movw %ax, %gs                                   # -> GS
9       movw %ax, %ss                                   # -> SS: Stack
    Segment
10
11      # Set up the stack pointer and call into C. The stack region is from
    0--start(0x7c00)
12      movl $0x0, %ebp
13      movl $start, %esp
14      call bootmain
```

跳转到这里后，前一部分是在保护模式下设置了各个段寄存器的值，后一部分是开辟了栈空间；最后，用call函数调用bootmain，因为引导加载程序在转换模式后还有许多工作需要处理。bootmain函数的作用是从磁盘中读取kernel内核代码，其效果如练习4所述。

练习4：分析bootloader加载ELF格式的OS的过程。

通过阅读bootmain.c，了解bootloader如何加载ELF文件。

分析：

根据练习1中对磁盘镜像生成流程的研究，可以发现内核可执行程序是存放在磁盘中从第二个块开始的区域的（第一个块中存放的是主引导扇区中的引导加载程序）。

从练习3可以看出，在bootloader完成了模式转换、建立分段机制的工作之后，就转而执行bootmain函数；这一部分的内容完成于bootmain.c文件之中，其作用在于加载存放在磁盘中的、ELF格式的OS操作系统程序。

为了了解加载机制的实现，首先需要了解访问硬盘的相关知识。

硬盘访问概述

根据实验指导书说明，bootloader加载程序访问硬盘的方式是所谓的LBA模式下的PIO方式，即相关输出输出IO操作是通过访问硬盘的IO地址寄存器完成的。

主板一般有2个IDE通道，每个通道可以接2个IDE硬盘。

通过提供的端口可以完成对硬盘的访问。访问第一个硬盘的扇区时，可以设置0x1f0-0x1f7这些IO地址寄存器，从而实现相关功能；通过给这些寄存器赋予特定值，可以指定对硬盘的操作方法。

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，你需要表明你要读写几个扇区。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
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

如图，在LBA模式下：

0x1f0是一个16位的端口，作用是在0x1f7不忙时，作为读取数据的来源，可以通过它从磁盘中取出数据。

0x1f1是一个错误寄存器，此处没有展示。

0x1f2寄存器可以存放一个数字，用来指定需要访问磁盘的扇区数目。

0x1f3到0x1f5这三个寄存器用于存放逻辑地址的0-23位，每个寄存器存放8位；设置逻辑地址后，决定了从哪里开始访问磁盘中的数据。

0x1f6在LBA模式下，其低4位（0-3位）存放逻辑地址的24-27位；4位表示硬盘号，0则操作主盘，1则操作从盘；其高3位（5-7位）设置为111，表示处于LBA模式。

0x1f7既是命令端口也是状态端口；作为命令端口，可以通过输入0x20命令表示需要读取数据，可以从0x1f0处获得磁盘数据；作为状态端口，其7位为1时表示硬盘忙，3位为1时表示硬盘已经准备好和主机交换数据，0位为1表示前一个命令执行错误，具体情况从0x1f1端口中获取。

综上所述，既然需要从硬盘中获得ELF格式的可执行文件，就需要对上述各个IO地址寄存器进行相应的赋值，说明需要进行怎样的操作。

ELF文件格式概述：

ELF格式是目标文件格式，有三种主要类型，其中一种类型就是可执行文件。

在ELF格式的可执行文件中，最开头含有一个ELF header，它描述了整个文件的组织，包含整个文件的控制结构，描述了文件的组成。

elf.h文件中定义了ELF header的数据结构，如下所示：

```
/* file header */
struct elfhdr {
    uint32_t e_magic;     // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;      // 1=relocatable, 2=executable, 3=shared object, 4=core image
    uint16_t e_machine;   // 3=x86, 4=68K, etc.
    uint32_t e_version;   // file version, always 1
    uint32_t e_entry;     // entry point if executable
    uint32_t e_phoff;     // file position of program header or 0
    uint32_t e_shoff;     // file position of section header or 0
    uint32_t e_flags;     // architecture-specific flags, usually 0
    uint16_t e_ehsize;    // size of this elf header
    uint16_t e_phentsize; // size of an entry in program header
    uint16_t e_phnum;     // number of entries in program header or 0
    uint16_t e_shentsize; // size of an entry in section header
    uint16_t e_shnum;     // number of entries in section header or 0
    uint16_t e_shstrndx;  // section number that contains section name strings
};
```

在加载ELF文件时，首先要通过文件头来了解ELF文件的相关信息。

其中，e_magic成员变量必须等于ELF_MAGIC幻数，从而确保这是正确的可执行文件；e_entry成员变量指定了该可执行文件程序入口的虚拟地址（也就对应于逻辑地址），而e_phoff成员变量则指定了program header表的位置偏移量，可以根据它查找到这个program header。

那么，什么是program header呢？

它用来描述了一些目标文件的结构信息，这些目标文件与程序的执行直接相关；此外，它还包含了一些为程序创建进程所必须的信息。实际上，可执行文件中分为很多个部分，各个部分的具体描述就是通过program header完成的；通过ELF header获得program header的位置之后，就可以找到program header并从中获取更多的文件相关信息。

可执行文件中的“程序头部”是一个program header类型的数组，每一个元素都描述了一个段(section，理解为扇区、节区)的相关信息，包括其类型、位置等等。该数据结构类型如下：

```
/* program section header */
struct proghdr {
    uint32_t p_type;   // loadable code or data, dynamic linking info,etc.
    uint32_t p_offset; // file offset of segment
    uint32_t p_va;     // virtual address to map segment
    uint32_t p_pa;     // physical address, not used
    uint32_t p_filesz; // size of segment in file
    uint32_t p_memsz;  // size of segment in memory (bigger if contains bss)
    uint32_t p_flags;  // read/write/execute bits
    uint32_t p_align;  // required alignment, invariably hardware page size
};
```

其中，p_type成员变量描述了section类型，p_offset描述了section相对于文件头的偏移量，p_va描述了这个section的第一个字节在内存中的虚拟地址，p_memsz是这个section在内存映像中占用的字节数目。

接下来查看bootmain.c的相关内容，首先是一段注释：

```
* DISK LAYOUT
```

```
 *  * This program(bootasm.S and bootmain.c) is the bootloader.
 *    It should be stored in the first sector of the disk.
 *
 *  * The 2nd sector onward holds the kernel image.
 *
 *  * The kernel image must be in ELF format.
 *
 * BOOT UP STEPS
 *  * when the CPU boots it loads the BIOS into memory and executes it
 *
 *  * the BIOS intializes devices, sets of the interrupt routines, and
 *    reads the first sector of the boot device(e.g., hard-drive)
 *    into memory and jumps to it.
 *
 *  * Assuming this boot loader is stored in the first sector of the
 *    hard-drive, this code takes over...
 *
 *  * control starts in bootasm.S -- which sets up protected mode,
 *    and a stack so C code then run, then calls bootmain()
 *
 *  * bootmain() in this file takes over, reads in the kernel and jumps
to it.
```

注释说明，bootasm.S和bootman.c共同组成了bootloader引导加载程序，存放在磁盘的第一个sector，也就是第一个扇区之中。从第二个sector开始，存放着内核的ELF格式文件。

在上电时，CPU首先执行BIOS；BIOS负责初始化设备，设置中断例程，然后从磁盘中的第一个扇区里读取到bootloader并将其加载进入内存，然后转移控制权给它并执行它；这个过程开始于bootasm.S，先转化为保护模式、开启段基址、开辟栈空间，然后调用bootmain函数，也就是bootmain.c文件的主要内容。

```
unsigned int    SECTSIZE  =     512 ;
struct elfhdr * ELFHDR    =     ((struct elfhdr *)0x10000) ;     //
scratch space
```

在文件中，首先定义SECTSIZE扇区大小为512，即磁盘中一个扇区大小为512个字节；接着声明和初始化ELFHDR，它是elfhdr类型的指针，用于存放从磁盘中读取的文件头。

接下来看readsect函数：

```
/* 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);
}
```

该函数的作用是从磁盘中读取一个扇区大小的内容，参数为磁盘和secno（逻辑地址，可以理解为扇区编号）。dist参数是读取的数据应当被存放近的地址。

首先调用waitdisk()函数用来等待磁盘准备完毕，准备完毕之后就可以通过设置端口寄存器来指定操作。

设置0x1f2端口寄存器为1，即指定只读取1个扇区；接下来对0x1f3到0x1f4赋予参数逻辑地址的各个位；对0x1f6赋予逻辑地址的部分位，取secno的24-27位作为低4位；而0xE0是1110 0000，所以高三位的111表示在LAB模式下进行，4位的硬盘号是0，因此会选择主盘读取数据。接着对0x1f7端口输入命令0x20，表示需要读取数据。

设置完寄存器后，再次等待磁盘准备完毕，然后调用insl，通过0x1f0端口，从磁盘中读取一个sector的数据。

接下来看readseg函数：

```c
/* *
 * 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);
    }
}
```

注释说明：readseg-将@offset处的@count字节从内核读取到虚拟地址@va，可能会复制超过要求的数量。也就是说，这个函数可以从offset开始，读取指定的字节数到内存的虚拟地址之中。

readseg函数将会读取一共count个字节，存放到va到end_va的虚拟地址之中。

在磁盘中，从偏移offset个字节后的地方开始读取扇区。一个扇区有512个字节；比如说，假如offset是600，那么就从2号扇区开始读（1024-1535字节），0号（0-511字节）和1号（512-1023字节）扇区不读。

以防万一，va首先减去offse%SECTSIZE；比如offset是600，则600%512=88，secno是2，从2号扇区开始读；

同理，假如offset是0，就从1号扇区开始读。读取时，secno不断增加，表示依次读取各个扇区；由于一个扇区占据512个字节，所以va每次读完一个扇区都要加上SECTSIZE即扇区字节数，更新起始地址。

在bootmain中调用readseg函数时，参数count是8*sectsize，实际上就是8个扇区的字节数。

```
// read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
```

在bootmain中，调用readseg的方式如上，将磁盘中八个扇区的内容读取到ELFHDR指针对应的地址处；其中，ELFHDR指向的就是读取到的数据的头部。

查看主函数：

```
/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // load each program segment (ignores ph flags)
    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);
    }

    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

```
24
25  bad:
26      outw(0x8A00, 0x8A00);
27      outw(0x8A00, 0x8E00);
28
29      /* do nothing */
30      while (1);
31  }
```

主函数首先调用readseg读取了从1号扇区开始的8个扇区的内容（0号扇区不读取，因为里面存放的是主引导扇区里的bootloader），读取到ELFHDR指定的地址之中；这样一来，ELFHDR就指向整个文件的头部。

```
1      // read the 1st page off disk
2      readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
3
4      // is this a valid ELF?
5      if (ELFHDR->e_magic != ELF_MAGIC) {
6          goto bad;
7      }
```

此时，ELFHDR作为elfhdr类型的指针，就指向文件头部的ELF header了，可以用它获取成员变量；检查幻数后，如果符合要求，就继续进行操作，否则跳转。

```
1     struct proghdr *ph, *eph;
2      // load each program segment (ignores ph flags)
3      ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
4      eph = ph + ELFHDR->e_phnum;
5      for (; ph < eph; ph ++) {
6          readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
7      }
```

从ELFHDR里的e_phoff成员变量获得program header的偏移量，然后和ELFHDR也就是文件开始地址组合在一起，得到program header的起始地址，然后用程序头类型ph指向它，从而ph获得了program header表。

e_phnum说明了program header表中的条目数目，从而从ph到eph就是各个条目的具体位置。注意，虽然e_phnum变量是一个整数，表示条目的数目，但是指针ph加上它之后，得到的新地址相当于是ph地址加上了条目数目乘以该类型所占字节数。这是指针加减法的特殊规定。

使用for循环，ph不断增加，每次都指向新的条目，从而获得这一段应该被置入的虚拟地址、大小和相对头文件的偏移，然后调用readseg函数把各个部分的代码加载到内存中的正确位置中去。

```
    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

最后，用ELFHDR->e_entry 可以获得可执行文件中的程序入口地址； ((void (*)(void))开头可以跳转到这个地址并开始执行程序，也就是开始执行下载好的内核程序。

练习5：实现函数调用堆栈跟踪函数

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。在如果能够正确实现此函数，可在lab1中执行 “make qemu”后，在qemu模拟器中得到类似如下的输出：

```
……
ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094 0x00007b58 0x00100096
    kern/debug/kdebug.c:305: print_stackframe+22
ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xffff0000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000 0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
```

```
ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007c53
    kern/init/init.c:28: kern_init+88
ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknow>: -- 0x00007d72 –
……
```

请完成实验，看看输出是否与上述显示大致一致，并解释最后一行各个数值的含义。

分析：

完成此练习首先要理解函数调用时栈的相关知识。

说明：栈帧寄存器是ebp，栈顶寄存器是esp。栈是向下增长的，每次放入数据，esp里的值都会减小。

假设在main函数里通过call指令调用add函数，则main会先在自己的栈帧中保存即将传递给过程的参数以及返回地址（call指令负责将返回地址保存到栈之中），然后call之后，add会首先保存main的ebp值，其地址比栈中保存返回地址的位置更小；然后把esp的值赋值给ebp，这样一来ebp就变成了add函数的栈帧，它直接指向main函数的旧ebp值，它加4的结果就是返回地址，再往上就是函数参数。

接下来把esp进行自减，就可以开辟属于add函数的栈空间。

等函数执行完毕，先把返回值保存在eax寄存器之中，再给esp赋值ebp的值使得add函数的栈顶指针和栈帧指针一致；接着弹出栈顶的旧ebp值给ebp寄存器，恢复main函数的栈帧；再然后，ret指令把新的栈顶元素也就是保存的返回地址交给eip，从而让控制权回来。

对于被调用者而言，[ebp]处为保存的调用者的旧ebp值；[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)。由于ebp中的地址处总是"上一层函数调用时的ebp值”，而在每一层函数调用中，都能通过当时的ebp值"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。如此形成递归，直至到达栈底。这就是函数调用栈。

查看kdebug.c文件，首先翻译注释内容如下：

```
print_stackframe - print a list of the saved eip values from the nested 'call' instructions that led to the current point of execution.
print_stackframe-从导致当前执行点的嵌套“call”指令打印保存的eip值列表。
从嵌套的call指令打印保存的eip值列表，这些嵌套的call指令决定了当前执行的指令地址。
解释：函数的嵌套调用是通过不断执行call而成功的，每一次call都会在保存旧地址的基础上转移控制权，从而给eip赋予新值；由于eip寄存器被保存的值会被视为当前指令执行地址，所以call可以跳转到指定的过程调用位置。

The x86 stack pointer, namely esp, points to the lowest location on the stack that is currently in use. Everything below that location in stack is free. Pushing a value onto the stack will invole decreasing the stack pointer and then writing the value to the place that stack pointer pointes to. And popping a value do the opposite.
x86堆栈指针，即esp，指向当前正在使用的堆栈上的最低位置（也就是栈顶）。堆栈中比esp地址更低的位置全都是栈中的未使用空间。让一个数值入栈上会让指针的值减小（esp寄存器的值减少，栈向地址减小的方向增长），然后将值写入堆栈指针指向的位置。而弹出一个值则相反（会让esp的值增加）。

```

9 The ebp (base pointer) register, in contrast, is associated with the stack primarily by software convention. On entry to a C function, the function's prologue code normally saves the previous function's base pointer by pushing it onto the stack, and then copies the current esp value into ebp for the duration of the function. If all the functions in a program obey this convention, then at any given point during the program's execution, it is possible to trace back through the stack by following the chain of saved ebp pointers and determining exactly what nested sequence of function calls caused this particular point in the program to be reached. This capability can be particularly useful, for example, when a particular function causes an assert failure or panic because bad arguments were passed to it, but you aren't sure who passed the bad arguments. A stack backtrace lets you find the offending function.

10 相反，ebp（基指针）寄存器主要通过软件约定与堆栈相关联（因为对于一个函数的栈，ebp的值一般不变，esp的值随着数值的增加而减小）。进入C函数时，函数的序言代码通常通过将上一个函数的基指针推到堆栈上来保存它，然后在函数期间将当前esp值复制到ebp中。如果程序中的所有函数都遵守此约定，那么在程序执行过程中的任何给定点，都可以通过跟踪保存的ebp指针链并准确确定函数调用的嵌套序列使程序中的该特定点到达，从而追溯堆栈。此功能特别有用，例如，当某个特定函数由于传递了错误参数而导致断言失败或死机，但您不确定是谁传递了错误参数时。堆栈回溯可以让您找到有问题的函数。

11 翻译的不好，实际上就是说一个函数内的第一部分代码需要保存调用者的旧ebp值；也就是说，每一个函数的栈空间的ebp指针都指向其保存的调用者的旧ebp值。

12

13 The inline function read_ebp() can tell us the value of current ebp. And the non-inline function read_eip() is useful, it can read the value of current eip, since while calling this function, read_eip() can read the caller's eip from stack easily.

14 内联函数read_ebp（）可以告诉我们当前ebp的值。非内联函数read_eip（）非常有用，它可以读取当前eip的值，因为在调用此函数时，read_eip（）可以轻松地从堆栈中读取调用方的eip。

15 可以使用read_ebp()读取当前进程栈空间的栈帧，也就是ebp寄存器的值；read_eip()可以读取eip的值，eip的值实际上就是当前执行的指令地址，因为指令地址都保存在eip寄存器中，修改eip寄存器的值就可以转移函数的控制权。注意，假如一个函数A调用了read_eip函数，那么在运行函数A的时候，read_eip函数返回的值实际上是函数A也就是调用方的当前eip。

16

```
In print_debuginfo(), the function debuginfo_eip() can get enough information about calling-chain. Finally print_stackframe() will trace and print them for debugging.
在print_debuginfo（）中，函数debuginfo_eip（）可以获得有关调用链的足够信息。最后，print_stackframe（）将跟踪并打印它们以进行调试。
```

未实现的print_stackframe函数如下：

```
void
print_stackframe(void) {
     /* LAB1 YOUR CODE : STEP 1 */
     /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
      * (2) call read_eip() to get the value of eip. the type is (uint32_t);
      * (3) from 0 .. STACKFRAME_DEPTH
      *    (3.1) printf value of ebp, eip
      *    (3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp +2 [0..4]
      *    (3.3) cprintf("\n");
      *    (3.4) call print_debuginfo(eip-1) to print the C calling function name and line number, etc.
      *    (3.5) popup a calling stackframe
      *           NOTICE: the calling funciton's return addr eip  = ss:[ebp+4]
      *                   the calling funciton's ebp = ss:[ebp]
      */
}
```

注释说明：

```
/*LAB1您的代码：步骤1*/
/* (1) 调用read_ebp () 获取ebp的值。类型为 (uint32_t) ；
* (2) 调用read_eip () 获取eip的值。类型为 (uint32_t) ；
* (3) 从0到STACKFRAME_DERTH，开始读：
* (3.1) ebp、eip的printf值
* (3.2) (uint32_t) 调用参数[0..4]=地址中的内容 (uint32_t) ebp+2[0..4]
* (3.3) cprintf (“\n”) ；
* (3.4) 调用print_debuginfo (eip-1) 打印C调用函数名、行号等。
* (3.5) 弹出调用堆栈帧
*注意：调用函数的返回地址eip=ss:[ebp+4]
*调用函数的ebp=ss:[ebp]
```

解析和实现：

```
uint32_t ebp = read_ebp();
uint32_t eip = read_eip();
int i1;
   for (i1=0; i1<STACKFRAME_DEPTH && ebp!= 0 ; i1++)
   {
       cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
       uint32_t *argslist =(uint32_t *)ebp + 2;
       int i2;
       for (i2 = 0; i2 < 4; i2 ++)
       {
           cprintf("0x%08x ", argslist[i2]);
       }
       cprintf("\n"); print_debuginfo(eip - 1);
       eip = ((uint32_t *)ebp)[1];  ebp = ((uint32_t *)ebp)[0];
   }
```

STACKFRAME_DERTH在文件中被指定为20，它表示要进行回溯的嵌套的个数。由于函数可以嵌套调用，我不仅想要打印当前函数的相关信息，还要打印调用此函数的函数的相应信息，以及调用了调用者的函数的相应信息……如此往复。由于栈是向下增长的，所以不断反溯调用者会导致**ebp**的值不断增大；所以， **STACKFRAME_DEPTH**指定了到底要回溯多少层。

实现中，i1从0遍历到STACKFRAME_DEPTH；每次遍历，首先打印当前的ebp和eip值，以及当前调用者的参数值，作为相关信息。

```
cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        uint32_t *argslist =(uint32_t *)ebp + 2;
        int i2;
        for (i2 = 0; i2 < 4; i2 ++)
        {
            cprintf("0x%08x ", argslist[i2]);
        }
        cprintf("\n"); print_debuginfo(eip - 1);
        eip = ((uint32_t *)ebp)[1];  ebp = ((uint32_t *)ebp)[0];
```

注意一下：ebp指针指向的位置是旧的ebp值，（ebp）+4指向的位置是返回值，（ebp）+8指向的位置才是真正的参数；

由于unint32_t*类型指针占据4个字节，所以（unint32_t *）ebp+2对应的地址值就是（ebp）+8。这涉及到指针的计算——指针增加1，对应的值是指针地址加上指针相应类型的字节数。

所以，这里的argslist实际上是一个整数指针，可以理解为一个整数数组argslist[ ]；它的起始地址是（ebp）+8，也就是当前函数收到的来自调用者的参数(这方面参考计算机系统的相关知识)。

这里依次输出argslist[0] argslist[1] argslist[2] argslist[3]，实际上就是在依次输出（ebp）+8、（ebp）+12、（ebp）+16、（ebp）+20，一共四个来自调用者的参数。

输出完当前函数的ebp值、eip值和调用者传给当前函数的四个参数后，就进行换行；

根据注释要求，调用print_debuginfo(eip-1)，打印相关信息。

然后是有趣的地方：

前面说过，我不仅要打印本函数的信息，更要打印调用者的信息，以及调用者的调用者的信息，不断回溯，直到i等于STACKFRAME_DEPTH也就是常量12，或者ebp值变成0了。

那么，如何由当前函数回溯到调用者呢?

```
eip = ((uint32_t *)ebp)[1];  ebp = ((uint32_t *)ebp)[0];
```

当前函数的ebp指针可以作为“数组”。（ebp）对应的正是调用者的ebp旧值，而（ebp）+4正是该函数的返回地址，这个返回地址是执行完本函数后，调用本函数者将继续运行的地方；把ebp指针的值作为数组，就可以从ebp[0]也就是（ebp）处获得调用者的ebp值，从ebp[1]也就是（ebp）+4处获得返回地址的值。

如图，给eip和ebp赋予新的值之后（注意，先给eip遍历赋值，再给ebp变量赋值，不然ebp变化后就没法得到eip了），它们所针对的就是调用者的ebp和eip了。**注意，这里的eip和ebp只是变量而已，并不是真正的寄存器。**

总之，这个函数是很好理解的，就是从本函数开始不断回溯到调用者，打印所有调用者、调用者的调用者.....的信息。

运行效果如下：

```
ebp:0x00007b38 eip:0x00100a4c args:0x00010094 0x00010094 0x00007b68 0x00100084
kern/debug/kdebug.c:305: print_stackframe+21
ebp:0x00007b48 eip:0x00100d4c args:0xo0000000 0x00000000 0x00000000 0x00007bb8kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b68 eip:0x00100084 args:0x00000000 0x00007b90 0xffff0000 0x00007b94'kern/init/init.c:48: grade_backtrace2+19
ebp:0x00007b88 eip:0x001000a6 args:0x00000000 0xffff0000 0x00007bb4 0x00000029kern/init/init.c:53: grade_backtrace1+27
ebp:0x00007ba8 eip:0x001000c3 args:0x00000000 0x00100000 0xffff0000 0x00100043'kern/init/init.c:58: grade_ backtrace0+19
ebp:0x00007bc8 eip:0x001000e4 args:0x00000000 0x00000000 0x00000000 0x00103560'kern/init/init.c:63: grade_backtrace+26
ebp:0x00007be8 eip:0x00100050 args:0x00000000 0x00000000 0x00000000 0x00007c4f
kern/init/init.c:28: kern_init+79
ebp:0x00007bf8 eip:0x00007d72 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknow>: -- 0x00007d71 --
```

与最上面参考图的形式确实是类似的，都只有八条信息，意味着这种嵌套调用只进行了八次。

现在要求我解释最后一行的信息：

最后一行的ebp只最大，意味着它是最初的函数，是一切嵌套调用的源头。

我们在加载OS前，执行的第一个嵌套调用完成后，返回地址是0x00007d71

那么，哪一个函数最先使用了堆栈?

毫无疑问，不考虑BIOS中的那些程序，那么整个加载过程中，在保护模式下执行的第一个函数是bootloader。

bootloader这个程序是从0x7c00开始的。也就是说，bootloader程序拥有的堆栈是从0x7c00开始的。

bootloader部分函数的起始地址是0x7c00，但其在栈中的开始位置是0x7bf8。

通过练习2中的调试，可以看到如下内容：

```
-Type <return> to continue,or q <return> to quit---
    0x7d6e:   incw (%bx,%si)
    0x7d70:   call *%ax
    0x7d72:   mov  $0x8a00,%dx
```

也就是说，bootloader程序在0x7d70处使用call指令进行了第一次嵌套调用，call指令将下一条指令的地址也就是0x7d72保存在栈中。

练习6：完善中断初始化和处理

请完成编码工作和回答如下问题：

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？

2. 请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt_init。在idt_init函数中，依次对所有中断入口进行初始化。使用mmu.h中的SETGATE宏，填充idt数组内容。每个中断的入口由tools/vectors.c生成，使用trap.c中声明的vectors数组即可。
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”。

> *【注意】除了系统调用中断(T_SYSCALL)使用陷阱门描述符且权限为用户态权限以外，其它中断均使用特权级(DPL)为 0 的中断门描述符，权限为内核态权限；而ucore的应用程序处于特权级 3 ，需要采用 `int 0x80`指令操作（这种方式称为软中断，软件中断，Tra中断，在lab5会碰到）来发出系统调用请求，并要能实现从特权级 3 到特权级 0 的转换，所以系统调用中断(T_SYSCALL)所对应的中断门描述符中的特权级（DPL）需要设置为 3 。*

分析：

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？

关于中断的概念理解：

三种主要中断：外设引起的中断，异步；同步中断/内部中断，即异常；陷入中断、软中断，系统调用。

中断发生后，会产生一个中断号；中断号中的索引被提取出来，可以根据它在中断描述符表IDT中获得对应于这个中断的一个中断描述符；该中断描述符中含有一个段选择子和段偏移量，可以根据该段选择子从段表GDT中查到段描述符，根据段描述符提供的段基址，结合中断描述符中的段偏移量，获得一个地址；这个地址就是终端服务例程程序所在的地址，中断发生后将跳转到这里以执行中断服务。

根据实验指导书说明，IDT中断描述表的基本单位是“中断门”，一个中断门相当于一个中断描述符；且，中断门一共占有64位，对应8个字节。其中32位是段选择子，32位是段内偏移。

其中2-3字节是段选择子，0-1字节和6-7字节构成段偏移量，二者结合起来，参照上述流程就可以得到中断服务例程的起始地址，也就是中断处理代码的入口。

2.请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt_init。在idt_init函数中，依次对所有中断入口进行初始化。使用mmu.h中的SETGATE宏，填充idt数组内容。每个中断的入口由tools/vectors.c生成，使用trap.c中声明的vectors数组即可。

未完成的trap.c内容如下：

```
/* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S */
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!
      */
}
```

注释说明：

```
/*LAB1您的代码：步骤2*/
 （1）每个中断服务例程（ISR）的入口地址在哪里？（只有找到中断地址，才能初始化IDT表）
所有ISR的地址都存储在_vectors中。
uintptr_t___vectors[ ]在哪里?
__vectors[ ]位于kern/trap/vector.S中，由tools/vector.c生成
*（在lab1中尝试“make”命令，然后在kern/trap DIR中找到vector.S）
*您可以使用“extern uintptr_t ____vectors[ ]；”来定义此extern变量（外部变量，意味着这个数组是其他文件夹里的，只不过本文件中的函数需要使用它来初始化idt表。），该变量将在后头用到。

*（2）现在您应该在中断描述符表（IDT）中设置ISR（各个中断门）条目。
*你能在这个文件中看到idt[256]吗？是的，这个数组就是IDT中断描述符表（只要给这个数组赋值就可以初始化IDT表了）！您可以使用SETGATE宏设置IDT的各个条目。

*（3）设置IDT的内容后，您将使用“lidt”指令让CPU知道IDT在哪里。
*你不知道这个说明的意思吗？只需谷歌一下！并查看libs/x86.h以了解更多信息。
*注意：lidt的参数是idt_pd。试着找到它！
```

从注释可以看出，初始化操作需要初始化中断描述符表。在另一个文件中已经定义了vectors[]数组，这个数组里存放着将被赋值给IDT的数据；通过使用宏SETGATE，可以将vectors[]数组中的内容依次存放到IDT里。

idt数组在文件开头声明：

```
/* *
 * Interrupt descriptor table:
 *
 * Must be built at run time because shifted function addresses can't
 * be represented in relocation records.
 * */
static struct gatedesc idt[256] = {{0}};
```

因此，必须给这个idt数组进行初始化。

实现效果如下：

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int gatenum=sizeof(idt)/sizeof(struct gatedesc);
    int i;
    for(i=0; i<gatenum; i++)
    {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK],
DPL_USER);
    lidt(&idt_pd);
}
```

vectors[]是定义在外部文件的数组，所以根据注释提示，此处通过extern将此数组引入，便于使用。

接下来用gatenum保存总共的中断门数目。中断门数目就是idt数组的元素数目，可以通过其总字节数除以gatedesc类型所占字节数求出。

接下来，通过for循环对idt数组中的各个条目赋值，通过调用SETGATE宏来实现这一点。

为了理解赋值过程，首先要查看gatedesc类型的数据结构，也就是中断门（中断描述符的组成）

在mmu.h中，定义如下：

```
/* 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
};
```

SETGATE宏实现如下，作用就是给gatedesc类型的数据赋值：

```
/* *
 * 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) {            \
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;        \
    (gate).gd_ss = (sel);                                 \
    (gate).gd_args = 0;                                     \
    (gate).gd_rsv1 = 0;                                     \
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;    \
    (gate).gd_s = 0;                                    \
    (gate).gd_dpl = (dpl);                                \
    (gate).gd_p = 1;                                    \
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16;        \
}
```

该宏可以当作函数理解，一共使用了5个参数。

第一个参数gate是要赋值的对象，它明显是gatedesc类型的数据，所以正好依次赋值给gatedesc类型的数组idt[i]，表示第i个门。

第二个参数istrap可以决定gd_type的值，但是我感觉它无论取1还是0都不会有什么影响，因为宏定义给出的两个选择都是STS_IG32，所以直接赋值为0也没有关系。

第三个参数sel将决定gd_ss，也就是段选择子。从memlayout.h中可以看到，一共有五个段，从上到下依次为：

内核代码段，内核数据段，用户代码段，用户数据段，以及一个任务段。

```
//memlayout.h文件中，对于各个段的序号说明如下
/* global segment number */
#define SEG_KTEXT    1
#define SEG_KDATA    2
#define SEG_UTEXT    3
#define SEG_UDATA    4
#define SEG_TSS        5
/* 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
```

毫无疑问，中段服务例程都是操作系统写好的，所以应该使用的段选择子必然是内核的代码段；IDT需要提供段选择子，而且是16位的，那必然是下面的GD_KTEXT了，因为它描述了指向内核代码段的描述符的选择子。

第四个参数off是偏移量，用外部数组vectors[i]就可以相应赋值了。这是因为，既然段已经确定，而且所有的服务例程全部都写在了内核的代码段里，所以vector[i]提供的就是i号中断门的段偏移量，也就是地址。

第五个参数**dpl**表示的是特权级，也就是这些中断服务例程的特权级；根据memlayout.h文件中的描述，内核特权级是0，保存在DPL_KERNEL之中；用户特权级是3，保存在DPL_USER之中。由于中断只能在内核态中调用，所以参数必须选择DPL_KERNEL。

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

最后，for循环实现如下：

```
    for(i=0; i<gatenum; i++)
    {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
```

然而，并不是所有的中断描述符的特权级都必须被设定为内核态。在完成上述初始化后，需要对一个中断描述符的特权级进行调整。

trap.h文件中有内容如下：

```
/* *
 * These are arbitrarily chosen, but with care not to overlap
 * processor defined exceptions or interrupt vectors.
 * */
#define T_SWITCH_TOU                120    // user/kernel switch
#define T_SWITCH_TOK                121    // user/kernel switch
```

注释含义为：这些是任意选择的，但注意不要与处理器定义的异常或中断向量重叠。

也就是说，这两个地方的中断作用是保证可以由用户态切换到内核态，所以需要把T_SWITCH_TOK处的idt条目的特权级专门设置为DPL_USER，这样可以在用户态下完成到内核态的转换。于是，需要把相应位置的中断描述符设置的特权级设置为用户级。之所以特地这样做，是为了实现扩展练习，详见下文说明。

```
SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
```

最后，使用lidt指令：

```
lidt(&idt_pd);
```

用lidt指令加载idt_pd。它在trap.c文件中定义如下：

```
static struct pseudodesc idt_pd = {
    sizeof(idt) - 1, (uintptr_t)idt
};
```

lidt的作用参见x86.h：

```
static inline void
lidt(struct pseudodesc *pd) {
    asm volatile ("lidt (%0)" :: "r" (pd));
}
```

需要把idt_pd的内容加载到idt寄存器之中。lidt指令的参数是指针，所以需要以指针形式传入。

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”。

trap函数内容如下：

```
/* *
 * trap - handles or dispatches an exception/interrupt. if and when trap() returns,
 * the code in kern/trap/trapentry.S restores the old CPU state saved in the
 * trapframe and then uses the iret instruction to return from the exception.
 * */
void
trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
}
```

它调用了trap_dispatch函数。

```
/* 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 funciton, such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
        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");
        }
    }
}
```

需要补充的内容如下：

```
    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!
         */
        break;
```

注释内容：

```
注释内容为：
/*LAB1您的代码：步骤3*/
/*处理定时器中断*/
/*（1）计时器中断后，应使用全局变量（增加它）记录此事件，如kern/driver/clock.c中的ticks
*（2）每个TICK_NUM周期，您都可以使用一个函数打印一些信息，例如使用print_ticks()函数。
*（3）太简单？是的，我想是的！
*/
```

即，每次在此事件发生时，增加ticks变量的值；当期达到TICK_NUM这个界限的时候，使用print_ticks()函数打印相关信息。

实现如下：

```
   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;
```

如图，ticks变量需要及时增加，并在到达周期的时候打印信息。

实现效果如下：

```
++ setup timer interrupts
0: @ring 0
0:  cs = 8
```

```
0:  ds = 10
0:  es = 10
0:  ss = 10
+++ switch to  user  mode +++
1: @ring 3
1:  cs = 1b
1:  ds = 23
1:  es = 23
1:  ss = 23
+++ switch to kernel mode +++
2: @ring 0
2:  cs = 8
2:  ds = 10
2:  es = 10
2:  ss = 10
100 ticks
100 ticks
100 ticks
```

100 ticks会逐步输出。

扩展练习 Challenge 1

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。

提示： 规范一下 challenge 的流程。

kern_init 调用 switch_test，该函数如下：

```
    static void
    switch_test(void) {
        print_cur_status();          // print 当前 cs/ss/ds 等寄存器状态
        cprintf("+++ switch to  user  mode +++\n");
        switch_to_user();            // switch to user mode
        print_cur_status();
        cprintf("+++ switch to kernel mode +++\n");
        switch_to_kernel();         // switch to kernel mode
        print_cur_status();
    }
```

switch*to*** *函数建议通过 中断处理的方式实现。主要要完成的代码是在 trap 里面处理 T_SWITCH_TO* 中断，并设置好返回的状态。

在 lab1 里面完成代码以后，执行 make grade 应该能够评测结果是否正确。

分析：

为了完成此扩展练习，首先需要了解中断的实现机制和ucore对中断的处理方法。否则，无法理解需要添加的函数起到什么作用，以及如何实现状态切换。

中断与特权级保护的实现机制：

中断发生后，首先获取中断向量；然后，以中断向量为索引，去中断描述符表IDT（IDT的位置和大小存放在寄存器IDTR中，可以从这个寄存器器读出IDT处于什么地址）中获得中断描述符（中断描述符分为段选择子、段偏移量两个部分，其中段选择子部分含有一个RPL请求特权级。注意，当前执行代码有一个CPL当前程序特权级，存放在特定寄存器中），依据中断描述符的段选择子找到GDT中的段描述符（里面有段的访问特权级DPL），而后可以根据段基址和段偏移量，获得中断服务例程的位置，并跳转到那里执行。

特权级的实现：特权级在ucore中分为0和3，3表示用户态的特权级，0表示内核态的特权级；只能被内核态访问的数据不能被用户态的特权级访问，内核态特权级可以访问任意特权级的数据，因为数字越小特权级的级别越高。

每一个段都有自己专有的特权级称为DPL，它被保存在GDT段表的段描述符中，表示访问这个段所需的最低特权级（只有数字值小于等于它的、特权级更高的特权级才能访问）；

每一段正在执行的代码有自己的特权级CPL，保存在CS和SS的第0位和第1位上，表示代码所在的特权级——当程序转移到不同特权级的代码段时，CPL将被改变，表示程序本身的特权级在用户态和内核态之间切换；

中断描述符的段选择子中含有的特权级RPL是请求特权级，表示当前代码段发出了一个特定特权级的请求。比如说，一个CPL为3的用户态的程序，需要执行软中断，所以发出了一个内核态的请求，则其RPL就将是0。一般，访问段时，会取CPL和DPL中较低的特权级用来与DPL进行比对，从而判断能否访问特定段；但在系统调用中，系统调用的过程发生了特权级的变化，为了执行系统调用，CPL需要从用户态升级为内核态，数字由3变成0。

内核态和用户态使用不同的栈来执行操作，因此特权级切换的本质就是使用内核态的栈而不是用户态的栈，同时保存用户态栈的相关信息便于在中断完成后恢复到用户态。

CPU会根据CPL和DPL判断需要进行特权级转换（从用户态升级为内核态）。一旦发生，需要从**TR寄存器**中获得当前程序的**TSS信息**的地址，从TSS信息中获得**内核栈地址**，然后将**用户态的栈地址信息（SS和ESP值）**保存到内核栈中。中断打断了用户态的程序，所以还要把用户态程序的eflags、cs、eip，乃至于可能存在的errorCode信息都压入**内核栈**之中。

中断结束后，使用iret指令可以弹出内核栈中的eflags；如果中断时有特权级切换，说明内核栈中还有用户栈的SS和ESP信息，iret会把它们也弹出，以便恢复到用户态。

ucore对上述过程的实现：

在lab1的代码中，vector.S文件中保存了各个中断向量（中断号）所对应的中断服务例程入口。以部分中断向量的相应处理示意如下：

```
# handler
.text
.globl __alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp __alltraps
.globl vector1
vector1:
  pushl $0
```

```
  pushl $1
  jmp __alltraps
.globl vector2
vector2:
  pushl $0
  pushl $2
  jmp __alltraps
.globl vector3
vector3:
  pushl $0
  pushl $3
  jmp __alltraps
```

可以看出，上述中断向量对应的操作，是向栈中压入0，然后压入中断号，接着跳转到__alltraps（位于trapentry.S文件中）中断处理函数，其内容如下：

```
# vectors.S sends all traps here.
.text
.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...
```

从注释可以看出，它专门处理从vector.S文里的中断请求。为了执行中断，他会将诸多寄存器保存在栈中，如ds到gs；接着，直接调用call指令，用trap函数继续进行处理。

需要指出的是，trap函数接收的参数是tf，它是一个特殊的数据类型指针，指向trapframe数据结构类型。这个数据结构实际上是中断所保存的环境信息，需要依赖它在中断后恢复原状。该函数位于trap.c文件中。

```
/* *
 * trap - handles or dispatches an exception/interrupt. if and when trap() returns,
 * the code in kern/trap/trapentry.S restores the old CPU state saved in the
 * trapframe and then uses the iret instruction to return from the exception.
 * */
void
trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
}
```

trapframe结构实现位于trap.h文件中：

```
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));
```

其中，tf_trapno可用于判断中断号。trap_dispatch函数通过它来判断当前需要处理的是哪一种中断；tf_esp和td_ss是发生用户态中断、需要进行特权级转换时，必须额外保存的信息。

trap函数进一步调用trap_dispatch函数，它在练习6中被补充，从而可以对时钟ticks计数和输出相应信息。实际上，trap_dispatch函数会根据tf中保存的中断信息——尤其是中断号——来进行真正的中断服务。

中断服务结束后，会依赖tp中保存的环境信息恢复到中断前的状态。因此，实现扩展练习1的方式就是在中断响应函数trap_dispatch中响应关于申请状态切换的中断请求，响应方式就是修改tp信息，也就是修改中断结束后需要赖以恢复的状态信息。

具体实现：

操作系统加载后，会首先执行kern_init函数进行内核初始化。

```
int
kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
```

```
5
6       cons_init();                // init the console
7
8       const char *message = "(THU.CST) os is loading ...";
9       cprintf("%s\n\n", message);
10
11      print_kerninfo();
12
13      grade_backtrace();
14
15      pmm_init();                 // init physical memory management
16
17      pic_init();                 // init interrupt controller
18      idt_init();                 // init interrupt descriptor table
19
20      clock_init();               // init clock interrupt
21      intr_enable();              // enable irq interrupt
22
23      //LAB1: CAHLLENGE 1 If you try to do it, uncomment
    lab1_switch_test()
24      // user/kernel mode switch test
25      lab1_switch_test();
26
27      /* do nothing */
28      while (1);
29  }
```

使这个函数得以调用 switch_test，该函数如下：

```
 static void
lab1_switch_test(void) {
    print_cur_status();          // print 当前 cs/ss/ds 等寄存器状态
    cprintf("+++ switch to  user  mode +++\n");
    switch_to_user();            // switch to user mode
    print_cur_status();
    cprintf("+++ switch to kernel mode +++\n");
    switch_to_kernel();         // switch to kernel mode
    print_cur_status();
 }
```

这个函数的作用，是先执行switch_to_user(); 切换到用户态，然后执行 switch_to_kernel(); 切换到内核态。

实现这两个函数，本质上就是让它们发出中断申请——分别用于申请切换为用户态和申请切换为内核态。

trap.h文件中定义了两个专门用于实现这两种中断的中断号，如下所示

```
/* *
 * These are arbitrarily chosen, but with care not to overlap
 * processor defined exceptions or interrupt vectors.
 * */
#define T_SWITCH_TOU                120    // user/kernel switch
#define T_SWITCH_TOK                121    // user/kernel switch
```

T_SWITCH_TOU中断向量对应于申请切换到用户态。

T_SWITCH_TOK中断向量对应于申请切换到内核态。

其中，后者发生在用户态下，所以其在idt中断描述符表里，需要专门设置特权级为DPL_USER也就是用户态特权级；关于idt的初始化，参见练习6的实现。方法如下。

```
SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
```

switch_to_user()和 switch_to_kernel()内需要进行软中断，并发出响应中断号，各自实现如下：

```
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
  asm volatile (
      "sub $0x8, %%esp \n"
      "int %0 \n"
      "movl %%ebp, %%esp"
      :
      : "i"(T_SWITCH_TOU)
  );
}

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

两个函数调用内联汇编，使用int指令发出了中断号。

接着，需要在dispatch函数中设计处理这两个中断的方法：

```
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    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 |= 0x3000;

```

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

        break;
```

如上所示，为了切换到用户态，需要在tf中保存用于恢复到用户态的信息；为了切换到内核态，需要在tf中保存用于恢复到内核态的信息。

由于tp中保存了中断前的环境信息，所以中断结束后会依赖tp中保存的信息恢复原状；为了切换到用户态，就需要在tp中保存关于用户栈的信息，也就是ss和esp相关信息；为了切换到内核态，也需要对tp做出相应处理。实际上，修改了tp的信息，实际上就是修改了中断结束后被恢复成的状态。

中断结束后，就会依赖被修改的tf信息，来恢复到用户态或者内核态。tf->tf_eflags |= 0x3000;的作用是为了在切换到用户态时，修改IO的特权级。

实现后，在终端里用**make grade**检验实现效果，得分如下。

```
csy@ubuntu:~/桌面/myoslab/os_kernel_lab/labcodes/lab1$ make grade
Check Output:            (2.4s)
  -check ring 0:                             OK
  -check switch to ring 3:                   OK
  -check switch to ring 0:                   OK
  -check ticks:                              OK
Total Score: 40/40
```

扩展练习 Challenge 2

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。

注意：

1.关于调试工具，不建议用lab1_print_cur_status()来显示，要注意到寄存器的值要在中断完成后tranentry.S里面iret结束的时候才写回，所以再trap.c里面不好观察，建议用print_trapframe(tf)

2.关于内联汇编，最开始调试的时候，参数容易出现错误，可能的错误代码如下

```
    asm volatile ( "sub $0x8, %%esp \n"
      "int %0 \n"
      "movl %%ebp, %%esp"
      : )
```

要去掉参数int %0 \n这一行

3.软中断是利用了临时栈来处理的，所以有压栈和出栈的汇编语句。硬件中断本身就在内核态了，直接处理就可以了。

思路说明：

要求通过键盘输入来切换状态。在trap_dispatch函数中，有一个中断是专门用来处理键盘输入的，内容如下：

```
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
    break;
```

它的作用是接收来自键盘输入的一个字符即c（类型为字符变量），然后显示它的内容。

扩展练习1里已经熟悉了如何实现用户态和内核态的切换，现在要求根据键盘输入来切换状态，只需要对c的值进行判断，然后执行相应的切换代码就可以了。所以，对上述的中断处理修改如下：

```
case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
```

```
        cprintf("kbd [%03d] %c\n", c, c);
        if(c=='0') //切换到内核态
         {
           if (tf->tf_cs != KERNEL_CS) {
             cprintf("try to be kernel mode\n");
             tf->tf_cs = KERNEL_CS;
           tf->tf_ds = KERNEL_DS;
             tf->tf_es = KERNEL_DS;
             print_trapframe(tf);
            }
            else{
               cprintf("already in kernel mode\n");
            }
         }
        else if(c=='3') //切换到用户态
        {
           if (tf->tf_cs != USER_CS) {
               cprintf("try to be user mode\n");
               tf->tf_cs = USER_CS;
             tf->tf_ds = USER_DS;
             tf->tf_es = USER_DS;
             tf->tf_ss = USER_DS;
               tf->tf_eflags |= 0x3000;
               print_trapframe(tf);
            }
            else{
               cprintf("already in user mode\n");
            }
         }

        break;
```

如上，当输入字符为'0'的时候，如果根据tf中信息发现当前不属于内核态，就输出一行信息并开始修改tf内容；如果已经处于内核态，就输出提示信息说明已经处于内核态。

同理，当输入字符为'3'的时候，如果根据tf中信息发现当前不属于用户态，就输出一行信息并开始修改tf内容；如果已经处于用户态，就输出提示信息说明已经处于用户态。

通过用 print_trapframe(tf);函数来查看tf被修改的状况，从而确认对它的修改是否成功。

在终端里，输入3后，首先会输出这个字符，然后开始在不处于用户态的时候修改tf信息并输出，示例如下：可见，有专门的esp和ss信息为恢复到用户态做准备。

```
kbd [051] 3
try to be user mode
trapframe at 0x10fcd4
  edi  0x00000000
  esi  0x00010094
  ebp  0x00007be8
  oesp 0x0010fcf4
  ebx  0x00010094
  edx  0x001035e7
  ecx  0x00000000
  eax  0x00000003
  ds   0x----0023
  es   0x----0023
  fs   0x----0023
  gs   0x----0023
  trap 0x00000021 Hardware Interrupt
  err  0x00000000
  eip  0x00100073
  cs   0x----001b
  flag 0x00003206 PF,IF,IOPL=3
  esp  0x0010359c
  ss   0x----0023
```

在终端里，输入0后，首先会输出这个字符，然后开始在不处于内核态的时候修改tf信息并输出，示例如下：可见，没有esp和ss信息为恢复到用户态做准备，会直接恢复到内核态。

```
kbd [048] 0
try to be kernel mode
trapframe at 0x10fcd4
  edi  0x00000000
  esi  0x00010094
  ebp  0x00007be8
  oesp 0x0010fcf4
```

```
  ebx  0x00010094
  edx  0x001035e7
  ecx  0x00000000
  eax  0x00000003
  ds   0x----0010
  es   0x----0010
  fs   0x----0023
  gs   0x----0023
  trap 0x00000021 Hardware Interrupt
  err  0x00000000
  eip  0x00100073
  cs   0x----0008
  flag 0x00003206 PF,IF,IOPL=3
```

至此，实验原理基本理解，实现基本成功。

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；分析了生成文件的指令部分，但是参数部分的介绍比答案中更加详细。

练习2：

与参考答案的实现区别较大，主要根据学堂在线的视频介绍，采用了make lab1-mon的指令进行调试，并对相应指令的参数和实现方式做出了更加详细的说明。

练习3:

介绍思路和顺序和答案一致，但是补充了大量对应的原理知识点，并对各条指令的内涵做出了自己能基本理解的详细解释。

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。

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。

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大量注释。

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，主要受到学堂在线教学视频的指导。

扩展练习2

参考答案未提供参考，依靠自学交流完成。

重要知识点和对应原理

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）

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

二者关系：

本实验设计的知识是对OS原理的具体实现，在细节上非常复杂。

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