操作系统课程设计实验报告

实验名称： 系统软件启动过程

姓名/学号：宋尚儒/1120180717

实验目的

操作系统是一个软件，也需要通过某种机制加载并运行它。

学习通过另外一个更加简单的软件bootloader来完成这些工作。

为此，需要完成一个能够切换到 x86 的保护模式并显示字符的 bootloader，为启动操作系统 ucore 做准备。

lab1 提供了一个非常小的 bootloader 和 ucore OS，整个 bootloader 执行代码小于 512 个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个 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：与软件相关的中断处理
  - 外设管理：时钟

实验内容

该实验中包含一个 bootloader 和一个 OS。这个 bootloader 可以切换到 X86 保护模式，能够读磁盘并加载 ELF 执行文件格式，并显示字符。而这 lab1 中的 OS 只是一个可以处理时钟中断和显示字符的简单 OS。

为了实现实验的目标，提供了 6 个基本练习，要求完成实验报告。

实验环境

- Windows10
- VMWare15.5
- Ubuntu-20.04
- ucore lab1
- qemu

程序设计与实现

练习1

> 理解通过make生成执行文件的过程

练习1.1

> 操作系统镜像文件ucore.img是如何一步一步生成的?

先看一下`make`命令执行过程

```
$ make "V="
+ cc kern/init/init.c
gcc -Ikern/init/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/init/init.c -o obj/kern/init/init.o
+ cc kern/libs/stdio.c
gcc -Ikern/libs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/libs/stdio.c -o obj/kern/libs/stdio.o
+ cc kern/libs/readline.c
gcc -Ikern/libs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/libs/readline.c -o obj/kern/libs/readline.o
cu+ cc kern/debug/kdebug.cggdb -m32 -gstabs -nostdinc  -fno-stack-protector -
Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c
kern/debug/kdebug.c -o
+ cc kern/debug/kmonitor.c
gcc -Ikern/debug/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o
+ cc kern/driver/clock.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/driver/clock.c -o obj/kern/driver/clock.o
+ cc kern/driver/console.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/driver/console.c -o obj/kern/driver/console.o
+ cc kern/driver/picirq.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/driver/picirq.c -o obj/kern/driver/picirq.o
+ cc kern/driver/intr.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/driver/intr.c -o obj/kern/driver/intr.o
+ cc kern/trap/trap.c
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/trap/trap.c -o obj/kern/trap/trap.o
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/trap/vectors.S -o obj/kern/trap/vectors.o
+ cc kern/trap/trapentry.S
```

```
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o
+ cc kern/mm/pmm.c
gcc -Ikern/mm/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/mm/pmm.c -o obj/kern/mm/pmm.o
+ cc libs/string.c
gcc -Ilibs/ -march=i686 -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
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -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 -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
 -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o
obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
 -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o
obj/boot/bootmain.o
+ cc tools/sign.c
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
+ ld bin/bootblock
ld -m    elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o
obj/boot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 496 bytes
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, 0.0443173 s, 116 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000200773 s, 2.6 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
154+1 records in
154+1 records out
78912 bytes (79 kB, 77 KiB) copied, 0.000531112 s, 149 MB/s
```

该过程可按顺序概括为：

- `gcc` 编译 `kern` 目录下的 `.c` 文件为 `obj/kern` 目录下的 `.o` 文件
- `ld` 通过 `tools/kernel.ld` 配置，将 `obj/kern` 目录下的文件连接成 `bin/kern`
- `gcc` 编译 `boot` 目录下的 `.c` 文件为 `obj/boot` 目录下的 `.o` 文件
- `gcc` 编译 `tools` 目录下的 `sign.c` 文件为 `obj/sign/tools/sign.o`

- `ld`将`obj/kern`目录下的文件连接成`obj/bootblock.o`
- 使用`obj/sign/tools/sign.o`，将`obj/bootblock.o`转化为`bin/bootblock`
- `dd`创建`bin/ucore.img`
- `dd`向`bin/ucore.img`中写入`bin/bootblock`
- `dd`向`bin/ucore.img`中写入`bin/kernel`

接下来查看`Makefile`内部，重要命令均已按照自己理解加上注释

需要注意的是`tools/function.mk`中定义了一些`Makefile`需要用到的函数，看到`call totarget`类似形式的语句的可以在该文件或`Makefile`中查看

查看和`ucore.img`创建有关的代码

```
#在ucore.img前加上bin/
UCOREIMG    := $(call totarget,ucore.img)

$(UCOREIMG): $(kernel) $(bootblock)
    #生成一个具有10000个块的文件
    $(V)dd if=/dev/zero of=$@ count=10000
    #将bootblock的内容写到第一个块
    $(V)dd if=$(bootblock) of=$@ conv=notrunc
    #从第二个块开始写kernel的内容
    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

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

生成`ucore.img`需要先生成`bootblock`和`kernel`，这与make执行过程顺序相符

kernel

对于`kernel`

```
# kernel

#都是kern下的子目录
KINCLUDE    += kern/debug/ \
               kern/driver/ \
               kern/trap/ \
               kern/mm/

KSRCDIR     += kern/init \
               kern/libs \
               kern/debug \
               kern/driver \
               kern/trap \
               kern/mm

#在编译选项下添加头文件所在目录
KCFLAGS     += $(addprefix -I,$(KINCLUDE))

#编译生成所有.o文件
$(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))

#打包.o文件
KOBJS   = $(call read_packet,kernel libs)

# create kernel target
```

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

#依赖于tools/kernel.ld文件
$(kernel): tools/kernel.ld

#依赖于之前编译生成的.o文件包
$(kernel): $(KOBJS)
    #输出"+ ld bin/kernel"到控制台
    @echo + ld $@
    #连接命令
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
    #将.o文件反编译为汇编文件
    @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
    @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call
symfile,kernel)

$(call create_target,kernel)
```

生成`kernel`需要先编译生成`.o`文件，这里以生成`init.o`命令为例

```
+ cc kern/init/init.c
gcc -Ikern/init/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc  -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/
-Ikern/mm/ -c kern/init/init.c -o obj/kern/init/init.o
#关键参数说明：
#-fno-builtin 除非用__builtin_前缀，否则不进行builtin函数优化
#-ggdb 生成可供gdb使用的调试信息
#-m32 生成适用于32位环境的代码
#-gstabs 生成stabs格式的调试信息
#-nostdinc 不使用标准库
#-fno-stack-protector 不生成用于检测缓冲区溢出的代码
```

生成`kernel`需要使用连接命令

```
+ 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
#关键参数说明：
#-m elf_i386 模拟为i386上的连接器
#-nostdlib 不使用标准库
#-T tools/kernel.ld 让连接器使用指定脚本
```

bootblock

对于`bootblock`

```
# create bootblock
#获取boot文件夹下的全部文件列表
bootfiles = $(call listf_cc,boot)
#编译boot文件夹下全部文件
```

```
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))

#在bootblock前加上bin/
bootblock = $(call totarget,bootblock)

#bootblock目标依赖项为源文件对应的obj文件和bin/sign
$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
    #输出"+ ld bin/bootblock"到控制台
    @echo + ld $@
    #链接OBJ文件为bin/bootblock
    $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
    #反编译bin/bootblock文件
    @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
    #将bin/bootblock转换为bin/bootblock.out二进制文件
    @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
    #用链接出的sign工具将bin/bootblock.out文件再转换回bin/bootblock
    @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)

$(call create_target,bootblock)
```

生成`bootblock`需要先生成编译`boot`目录下的文件生成`.o`文件，分别是`bootasm.o`和`bootmain.o`，需要分别编译`bootasm.S`，和`bootmain.c`，生成命令如下所示

```
+ cc boot/bootasm.S
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
 -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o
obj/boot/bootasm.o
#关键参数说明：
#-fno-builtin 除非用__builtin_前缀，否则不进行builtin函数优化
#-ggdb 生成可供gdb使用的调试信息
#-m32 生成适用于32位环境的代码
#-gstabs 生成stabs格式的调试信息
#-nostdinc 不使用标准库
#-fno-stack-protector 不生成用于检测缓冲区溢出的代码
#-Os 为减小代码量进行优化，主引导扇区只有512字节

+ cc boot/bootmain.c
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
 -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o
obj/boot/bootmain.o
#关键参数同上
```

对于`sign`工具

```
# create 'sign' tools
#编译生成.o文件
$(call add_files_host,tools/sign.c,sign,sign)
#生成sign文件
$(call create_target_host,sign,sign)
```

生成`sign`的命令

```
+ 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
#关键参数说明：
#-g 生成操作系统原生调试信息
#-O2 进行不以空间换时间的优化
```

链接生成`bootblock`文件命令

```
+ ld bin/bootblock
ld -m    elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o
obj/boot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 496 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
#关键参数说明：
#-N 设置代码段和数据段均可读写
#-e start 将start符号设为程序入口
#-Ttext 0x7C00 指定代码段开始位置
```

练习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？

查看并解读`tools/sign.c`代码

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>

int
main(int argc, char *argv[]) {
    struct stat st;
    //检查输入
    if (argc != 3) {
        fprintf(stderr, "Usage: <input filename> <output filename>\n");
        return -1;
    }
    //打开文件
    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;
    }
    //申请长度位512的buff数组
    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);
    //将buff最后两位置为0x55和0xAA
    buf[510] = 0x55;
    buf[511] = 0xAA;
    FILE *ofp = fopen(argv[2], "wb+");
    //读文件到buf内
    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;
}
```

可见主引导扇区特征为：

- 大小512字节
- 倒数第二个字节为0x55
- 倒数第一个字节为0xAA

练习2

> 熟悉使用 qemu 和 gdb 进行调试工作

练习2.1

> 从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS 的执行。

修改`lab1/tools/gdbinit`内容

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

再lab1目录下，终端执行

```
make debug
```

进入gdb调试界面，可输入`si`单步追踪，以下为查看当前eip处汇编指令

```
0x0000fff0 in ?? ()
(gdb) si
0x0000e05b in ?? ()
(gdb) x /2i $pc
=> 0xe05b:      add    %al,(%eax)
   0xe05d:      add    %al,(%eax)
(gdb) si
0x0000e062 in ?? ()
(gdb)
```

练习2.2

> 在初始化位置 0x7c00 设置实地址断点,测试断点正常。

再gdb中执行以下命令

```
(gdb) b * 0x7c00
Breakpoint 1 at 0x7c00
```

输入`c`，程序会持续执行，直到断点处

```
(gdb) c
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x /10i $pc
=> 0x7c00:      cli
   0x7c01:      cld
   0x7c02:      xor    %eax,%eax
   0x7c04:      mov    %eax,%ds
   0x7c06:      mov    %eax,%es
   0x7c08:      mov    %eax,%ss
   0x7c0a:      in     $0x64,%al
   0x7c0c:      test   $0x2,%al
   0x7c0e:      jne    0x7c0a
   0x7c10:      mov    $0xd1,%al
```

练习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。

衔接上一练习的操作，查看`0x7c00`开始的10条指令

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

在练习一中，得知`bootblock`将start符号设为程序入口，指定代码段开始位置0x7C00，而文件`boot/bootasm.S`中从`start`符号开始对应部分内容为

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

```
    cld                                             # String operations
increment

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

    # 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
```

可以看到，虽然在汇编语言与寄存器命名规范上略有不同，但前十个指令完全一致，在`bootblock.asm`中也是如此

练习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。

选择在内核中设置断点，对`tools/gdbinit`修改如下

```
file bin/kernel
target remote: 1234
b cons_init
continue
```

执行`make debug`，gdb面板如下所示，可见成功设置了在函数`cons_init`处的断点并进行了单步调试

```
┌──kern/driver/console.c──────────────────────────────────────────────────────┐
|B+>425         cons_init(void) {                                             |
|   426             cga_init();                                               |
|   427             serial_init();                                            |
|   428             kbd_init();                                               |
|   429             if (!serial_exists) {                                     |
|   430                 cprintf("serial port does not exist!!\n");            |
┌─────────────────────────────────────────────────────────────────────────────┐
|B+ 0x10161e <cons_init>    endbr32                                           |
|  >0x101622 <cons_init+4>  push   %ebp                                       |
|   0x101623 <cons_init+5>  mov    %esp,%ebp                                  |
|   0x101625 <cons_init+7>  sub    $0x18,%esp                                 |
|   0x101628 <cons_init+10> call   0x100e5b <cga_init>                        |
|   0x10162d <cons_init+15> call   0x100f45 <serial_init>                     |
└─────────────────────────────────────────────────────────────────────────────┘
remote Thread 1.1 In: cons_init                          L425  PC: 0x101622
0x0000fff0 in ?? ()
```

```
Breakpoint 1 at 0x10161e: file kern/driver/console.c, line 425.

Breakpoint 1, cons_init () at kern/driver/console.c:425
(gdb) layout split
(gdb) si
(gdb)
```

练习3

> 分析 bootloader 进入保护模式的过程

进入保护模式过程如下：

- 激活A20地址线
- 加载全局描述符表gdt
- 打开保护模式

详细过程如下：

从入口start处开始分析`bootblock.S`各段，

清理环境

```
.code16                                             # Assemble for 16-bit mode
    cli                                             # 禁用中断
    cld                                             # 复位操作方向标志清零

    # 重要段寄存器置零
    xorw %ax, %ax                                   # ax置零
    movw %ax, %ds                                   # ds置零
    movw %ax, %es                                   # es置零
    movw %ax, %ss                                   # ss置零
```

在默认情况下A20地址线是关闭的，需要开启A20：

- 等待8042控制器不忙
- 送写8042P2命令到8042控制器
- 等待8042控制器不忙
- 将8042（P2）得到字节的第2位置1，然后写入8042 Input buffer；

```
seta20.1:
    inb $0x64, %al                  # 等待8042控制器不忙(8042 Input buffer为空)
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al                 # 发送写8042P2命令到8042控制器
    outb %al, $0x64                 #

seta20.2:
    inb $0x64, %al                  # 等待8042控制器不忙(8042 Input buffer为空)
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al                 # 将具体设置值发送给8042控制器
    outb %al, $0x60                 # 该值意为打开A20
```

初始化gdt表并加载，可通过以下命令载入全局描述符表

```
    lgdt gdtdesc
```

而载入的GDT已在bootasm.S中被描述，具体如下

```
# Bootstrap GDT
.p2align 2                                          # force 4 byte alignment
gdt:
    SEG_NULLASM                                     # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)           # code seg for bootloader and
kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)                 # data seg for bootloader and
kernel

gdtdesc:
    .word 0x17                                      # sizeof(gdt) - 1
    .long gdt
```

载入gdt表后，将cr0寄存器PE位置为1，这里使用按位与的方式，即可开启保护模式

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

保护模式打开完成

练习4

> 分析 bootloader 加载 ELF 格式的 OS 的过程

`bootmain.c`中包含多个函数，逐函数进行分析，重要步骤均已注释

练习4.1

> bootloader 如何读取硬盘扇区的?

查看对应函数`readsect`

```
/* readsect - read a single sector at @secno into @dst */
//读取单个扇区，从secno扇区读取数据到dst位置
static void
readsect(void *dst, uint32_t secno) {
    //等待磁盘不忙
    waitdisk();

    outb(0x1F2, 1);                         // 0x1f2，设置每次读写的扇区数为1
    /*
    0x1F3-0x1F5设置0-23位为secno的24位偏移量，
    0x1F6设置24-27位的secno偏移量，并设置28位位0，表示访问磁盘0，剩余位设为1
    */
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);                      // 0x20命令，表示读取扇区

    //等待磁盘不忙
```

```
    waitdisk();

    // 读取IO地址为0x1F0处的数据到dst位置
    insl(0x1F0, dst, SECTSIZE / 4);
}
```

总结读取某磁盘扇区流程如下：

- 等待磁盘不忙
- 设置0x1F2-0x1F6为读取扇区所需参数
- 设置0x1F7端口为读命令0x20
- 等待扇区不忙
- 从0x1F0处读取数据到指定内存

此外，`readseg`函数对`readsect`做进一步封装，可以从偏移地址offset处开始读取指定字节到指定内存。

```
//利用readsect函数从偏移地址位offset处开始，读取count字节的内容到虚拟地址va
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    va -= offset % SECTSIZE;

    //ELF文件从1扇区开始，0扇区被引导占用
    uint32_t secno = (offset / SECTSIZE) + 1;

    for (; va < end_va; va += SECTSIZE, secno ++) {
        readsect((void *)va, secno);
    }
}
```

练习4.2

> bootloader是如何加载ELF格式的OS?

重点关注`bootmain`函数

```
//bootloader的入口
void
bootmain(void) {
    // 读取第一个page
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // 根据读取的第一个page判断ELF是否合法
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    //载入ELF程序段
    //将ELF头部描述表的头地址存在ph
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    //遍历该表的每一项，按描述表的段的偏移量依次载入段
    for (; ph < eph; ph ++) {
```

```
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
    }
    //完成将OS加载到内存

    //用函数调用的方式跳转到kernel入口，完成ELF格式的OS的加载
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
```

练习5

> 实现函数调用堆栈跟踪函数

修改`kdebug.c`中函数`print_stackframe`如下

```
void
print_stackframe(void) {
    uint32_t ebp,eip;
    //读取ebp和eip
    ebp=read_ebp();
    eip=read_eip();
    //扫描栈上的函数
    for(int i=0;i<STACKFRAME_DEPTH&&ebp!=0;i++)
    {
        //输出ebp和eip值
        cprintf("ebp: 0x%08x\neip: 0x%08x\narg:",ebp,eip);
        //打印保存在栈上的最多4个参数
        for(int j=0;j<4;j++)
        {
            cprintf(" 0x%08x",((uint32_t*)(ebp + 2))[j]);
        }
        cprintf("\n");
        print_debuginfo(eip - 1);
        //指向栈上上一层函数（调用者），在下一循环中会跳转到调用者栈帧
        eip = *((uint32_t*) ebp + 1);
        ebp = *((uint32_t*) ebp);
    }
}
```

输入`make qemu`,结果如下

```
Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x00103414 (phys)
  edata  0x0010fa16 (phys)
  end    0x00110d20 (phys)
Kernel executable memory footprint: 68KB
ebp: 0x00007b28
eip: 0x00100ab3
arg: 0x0dac0000 0x00940010 0x00940001 0x7b580001
    kern/debug/kdebug.c:296: print_stackframe+25
ebp: 0x00007b38
eip: 0x00100dac
arg: 0x00960000 0x00000010 0x00000000 0x00000000
    kern/debug/kmonitor.c:125: mon_backtrace+14
```

```
ebp: 0x00007b58
eip: 0x00100096
arg: 0x00c40000 0x00000010 0x7b800000 0x00000000
    kern/init/init.c:48: grade_backtrace2+37
ebp: 0x00007b78
eip: 0x001000c4
arg: 0x00e70000 0x00000010 0x00000000 0x7ba4ffff
    kern/init/init.c:53: grade_backtrace1+42
ebp: 0x00007b98
eip: 0x001000e7
arg: 0x01110000 0x00000010 0x00000000 0x00000010
    kern/init/init.c:58: grade_backtrace0+27
ebp: 0x00007bb8
eip: 0x00100111
arg: 0x00550000 0x343c0010 0x34200010 0x130a0010
    kern/init/init.c:63: grade_backtrace+38
ebp: 0x00007be8
eip: 0x00100055
arg: 0x7d740000 0x00000000 0x00000000 0x00000000
    kern/init/init.c:28: kern_init+84
ebp: 0x00007bf8
eip: 0x00007d74
arg: 0x7c4f0000 0xfcfa0000 0xd88ec031 0xd08ec08e
    <unknow>: -- 0x00007d73 --
```

分析最后一组数据

- `ebp: 0x00007bf8`是第一个被调用函数栈帧的栈底指针
- `eip: 0x00007d74`是在该栈帧对应函数中调用下一栈帧对应函数指令的下一条指令地址
- `arg`四个值是传递给这第一个被调用函数的参数
- `<unknow>: -- 0x00007d73 --`：该函数内调用OS kernel入口函数的指令地址

对应的是第一个使用堆栈的函数，即`bootmain.c`中的`bootmain`

为验证该函数的堆栈地址，查看`bootasm.S`

```
# 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
```

`bootloader`设置的堆栈从`0x7c00`开始，使用"call bootmain"转入bootmain函数，所以call指令对应的ebp为0x00007bf8，等于最后一组数据对应值，验证成功。

练习6

> 完善中断初始化和处理

练习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？

中断描述符表IDT中表项占8字节

- 0-1字节：偏移量的低位

- 2-3字节：段选择子
- 6-7字节：偏移量的高位

使用段选择子在GDT表中查找到对应段的基址，加上偏移量就是中断处理代码的入口

练习6.2

> 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

```
void
idt_init(void) {
    //获得外部变量vector，中断入口
    extern uintptr_t __vectors[];
    //使用SETGATE宏，对中断描述符表中每个表项进行设置
    for (int i = 0; i < 256; i++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    //设置核心态和用户态转换的中断
    SETGATE(idt[T_SWITCH_TOU], 0, GD_KTEXT, __vectors[T_SWITCH_TOU], DPL_USER);
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
    //载入中断描述符表
    lidt(&idt_pd);
}
```

练习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”

在函数体外声明静态变量`Ticks`

```
static int32_t Ticks = 0;
```

然后在函数`trap_dispatch`待填的case中添加

```
Ticks++;
if (0 == (Ticks % TICK_NUM)) {
    print_ticks();
}
```

修改完成后，在终端输入命令`make qemu`，结果如下（仅截取部分），符合要求

```
    kern/init/init.c:28: kern_init+84
ebp: 0x00007bf8
eip: 0x00007d74
arg: 0x7c4f0000 0xfcfa0000 0xd88ec031 0xd08ec08e
    <unknow>: -- 0x00007d73 --
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

```
100 ticks
```

参考答案分析

- 练习1-4因编程部分不多，与参考答案基本一致
- 练习5的解答与参考答案解法基本一致，使用栈上保存的动态链信息查找栈上栈帧的内容，只有指针和数组的索引方式略有不同
- 练习6的解答与参考答案在思路上基本一致，区别在于全局变量在参考答案中被声明于另一个文件中

实验收获与体会

本次实验主要学习了一个操作系统启动的主要流程，不得不说这实验如果完整学习所需的时间还是挺长的，如果一步一步分析原码工作量也相当大，所幸资料比较丰富，如果在理解上有困难也能得到解答。

直接看原码可以较大提升对操作系统的理解，但要求的基础知识相对会更多，比如汇编课程就尚未讲授，只能先自学部分内容，这样在理解调试、堆栈等方面可能会不是很全面。

整体复杂程度比起之前的实验提升了很多，并且与之前的实验在所需知识上没有直接联系（之前主要是学习各类系统调用），知识偏向底层，时间又放在学期末尾，可能给学生带来相当的压力。如果希望进行ucore系列实验，建议设置在课程开始阶段，或者在开始阶段就给出要求，让学生有较长的时间进行进行学习。