练习 1：理解通过make生成执行文件的过程。

习题：

Exercise 1.1：

Solution:

part 1：

赋值语句：

@ 符号的使用：

含义：

part 2：

makefile 里的 shell

代码及含义：

part 3：

part 4：

part 5：

addsuffix 函数：

if 函数：

wildcard 函数：

filter 函数：

call 函数：

function.mk -- part 1：

basename 函数：

function.mk -- part 2：

patsubst 函数：

function.mk -- part 3：

function.mk -- part 4：

function.mk -- part 5：

function.mk -- part 6：

function.mk -- part 7：

function.mk -- part 8：

function.mk -- part 9：

function.mk -- part 10：

function.mk -- part 11：

Exercise 1.2

Solution:

练习2：使用qemu执行并调试lab1中的软件

练习3：分析bootloader进入保护模式的过程

bootasm.S -- part 1：

bootasm.S -- part 2：

bootasm.S -- part 3：

bootasm.S -- part 4：

bootasm.S -- part 5：

bootasm.S -- part 6：

练习4：分析bootloader加载ELF格式的OS的过程

Exercise 4.1：

Solution:

waitdisk 函数：

readsect 函数：

Exercise 4.2

Solution:

readseg 函数：

bootmain 函数 -- part 1：

bootmain 函数 -- part 2：

bootmain 函数 -- part 3：

练习5：实现函数调用堆栈跟踪函数

练习6：完善中断初始化和处理

参考链接

练习 1：理解通过make生成执行文件的过程。

习题：

Exercise 1.1：

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)

Solution:

网上答案看得都很迷糊啊，作为小菜鸟我得先从 makefile 学起

part 1：

```
PROJ    := challenge
EMPTY    :=
SPACE    := $(EMPTY) $(EMPTY)
SLASH    := /

V       := @
```

赋值语句：

`+=` 的含义显而易见，将右值增加到左边的变量基础上，空格是自动添加的。

比如 `x = foo` 后接了一句 `x += ijk`，那么 x 的值最后就是 `xyz ijk`

`=` 的含义是遍历整个 Makefile，获得所有右值的最终值，再赋值给左边的变量。

`?=` 的含义是如果左值未被赋值过，则将右值赋值给它，不然就不赋值。

`:=` 的含义是将右值当前的值赋值给左值，不会遍历 Makefile 获取右值的最终值。

@ 符号的使用：

通常 makefile 会将其执行的命令行在执行前输出到屏幕上

如果将 @ 添加到命令行前，这个命令将不被 make 回显出来，例如：

```
@echo --compiling module----;  // 屏幕输出 --compiling module----
echo --compiling module----;   // 没有 @ 的话，屏幕输出 echo --compiling module-
---
```

含义：

这段代码的意思就是变量赋值，留着在后面用

part 2：

一上来看不懂代码，先写疑惑点

makefile 里的 shell

1. 在Makefile文件的目标项冒号后另起一行的代码才是 shell 代码

```
eg:
xx = xx1      // 这里是 makefile 代码
yy: xx = xx2  // 这里是 makefile 代码，makefile 允许变量赋值时，'='号两边留空格
yy:
    xx=xx3    // 只有这里是 shell 代码 ，shell 不允许‘=’号两边有空格哦。
```

有一个例外：

```
xx = $(shell 这里的代码也是shell代码)
```

这个例外就是我这段代码没看懂的部分

2. Makefile 中的 shell，每一行是一个进程，不同行之间变量值不能传递。所以 Makefile 中的 shell 不管多长也要写在一行

```
eg:
SUBDIR=src example
all:
    @for subdir in $(SUBDIR); \      // 这里往下是一行shell
    do \
        echo "building " $$subdir; \
    done
```

3. Makefile 中的变量以 $ 开头， 为了避免和 shell 的变量冲突，shell 的变量要以 $$ 开头

   这点没啥说的，类似于扩展内联汇编里的 `movl %%cr0,%0\n\t`

代码及含义：

```
#need llvm/cang-3.5+
#USELLVM := 1
# try to infer the correct GCCPREFX
ifndef GCCPREFIX
GCCPREFIX := $(shell if i386-elf-objdump -i 2>&1 | grep '^elf32-i386$$' >/dev/null 2>&1; \
    then echo 'i386-elf-'; \
    elif objdump -i 2>&1 | grep 'elf32-i386' >/dev/null 2>&1; \
    then echo ''; \
    else echo "***" 1>&2; \
    echo "*** Error: Couldn't find an i386-elf version of GCC/binutils." 1>&2; \
    echo "*** Is the directory with i386-elf-gcc in your PATH?" 1>&2; \
    echo "*** If your i386-elf toolchain is installed with a command" 1>&2; \
    echo "*** prefix other than 'i386-elf-', set your GCCPREFIX" 1>&2; \
    echo "*** environment variable to that prefix and run 'make' again." 1>&2; \
    echo "*** To turn off this error, run 'gmake GCCPREFIX= ...'." 1>&2; \
    echo "***" 1>&2; exit 1; fi)
endif
```

定义 GCCPREFIX 变量，利用了 linux bash 中的技巧来推断所使用的 gcc 命令是什么

在本部分首先猜测 gcc 命令的前缀是 `i386-elf-`，因此执行 `i386-elf-objdump -i` 命令

2>&1 表示将错误输出一起输出到标准输出里，然后通过管道的方式传递给下一条 bash 命令

`grep '^elf32-i386$$' >/dev/null 2>&1;` 这句话中 `>/dev/null` 这部分表示将 标准输出 输出到一个空设备里

而输入上一条命令的内容在发送给 grep 的标准输出（作为 grep 的输入）后

假如可以匹配到 `^elf32-i386$$`，则说明 i386-elf-objdump 这一命令是存在的，说明条件满足，将由 echo 输出 `i386-elf-`

由于是在 `$()` 里的 bash 命令，这个输出将会作为值被赋给 GCCPREFIX 变量

如果 i386-elf-objdump 命令不存在，则猜测使用的 gcc 命令不包含其他前缀，则继续按照上述方法，测试 objdump 这条命令是否存在

如果存在则 GCCPREFIX 为空串，否则直接报错，要求显示地提供 gcc 的前缀作为 GCCPREFIX 变量的值（可以在环境变量中指定）

part 3:

```
# try to infer the correct QEMU
ifndef QEMU
QEMU := $(shell if which qemu-system-i386 > /dev/null; \
    then echo 'qemu-system-i386'; exit; \
    elif which i386-elf-qemu > /dev/null; \
    then echo 'i386-elf-qemu'; exit; \
    elif which qemu > /dev/null; \
    then echo 'qemu'; exit; \
    else \
    echo "***" 1>&2; \
    echo "*** Error: Couldn't find a working QEMU executable." 1>&2; \
    echo "*** Is the directory containing the qemu binary in your PATH"
1>&2; \
    echo "***" 1>&2; exit 1; fi)
endif
```

跟上面 part 2 差不多，没啥可说的

part 4:

```
# eliminate default suffix rules
.SUFFIXES: .c .S .h

# delete target files if there is an error (or make is interrupted)
.DELETE_ON_ERROR:

# define compiler and flags
ifndef  USELLVM
HOSTCC        := gcc
HOSTCFLAGS    := -g -Wall -O2
CC        := $(GCCPREFIX)gcc
CFLAGS    := -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -
nostdinc $(DEFS)
CFLAGS    += $(shell $(CC) -fno-stack-protector -E -x c /dev/null
>/dev/null 2>&1 && echo -fno-stack-protector)
else
HOSTCC        := clang
```

```
HOSTCFLAGS    := -g -Wall -O2
CC        := clang
CFLAGS    := -march=i686 -fno-builtin -fno-PIC -Wall -g -m32 -nostdinc
$(DEFS)
CFLAGS    += $(shell $(CC) -fno-stack-protector -E -x c /dev/null
>/dev/null 2>&1 && echo -fno-stack-protector)
endif

CTYPE    := c S

LD      := $(GCCPREFIX)ld
LDFLAGS    := -m $(shell $(LD) -V | grep elf_i386 2>/dev/null | head -n 1)
LDFLAGS    += -nostdlib

OBJCOPY := $(GCCPREFIX)objcopy
OBJDUMP := $(GCCPREFIX)objdump

COPY    := cp
MKDIR   := mkdir -p
MV        := mv
RM        := rm -f
AWK        := awk
SED        := sed
SH        := sh
TR        := tr
TOUCH    := touch -c

OBJDIR    := obj
BINDIR    := bin

ALLOBJS    :=
ALLDEPS    :=
TARGETS    :=
```

从这一段代码里可以看出之前 GCCPREFIX 的作用：确定使用什么版本的 gcc 编译器套件

这段基本就是变量定义，内容都是命令，相当于重命名命令，方便在后边调用

part 5：

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

接下来的部分引用了 tools/function.mk 文件，因此不妨先分析一下该文件的内容

先看看不会的函数

addsuffix 函数：

```
$(addprefix <prefix>, <name1 name2 ...>)
```

把 <prefix> 加到 name 序列中的每一个元素前面，即将 <prefix> 作为序列中每个元素的前缀

注：addsuffix 用法与 addprefix 相同，只是一个是前缀，一个是后缀

实例：

```
result = $(addprefix %., c cpp)
test:
    @echo $(result)
```

输出：

```
%.c %.cpp
```

if 函数：

```
$(if <condition>, <then-part>,<else-part>)
```

<condition> 参数是 if 的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真

于是 <then-part> 会被计算，返回计算结果字符串；否则 <else- part> 会被计算，返回计算结果字符串

注：<else-part> 可以省略

实例：

```
suffix :=
result1 := $(if $(suffix), $(addprefix %.,$(suffix)), %)

suffix := c cpp
result2 := $(if $(suffix), $(addprefix %.,$(suffix)), %)

test:
    @echo result1 is $(result1)
    @echo result2 is $(result2)
```

输出：

```
result1 is %
result2 is %.c %.cpp
```

wildcard 函数：

```
$(wildcard <pattern1 pattern2 ...>)
```

展开 pattern 中的通配符

实例：

```
$(wildcard src/*)
```

输出：

得到 src 目录下的文件列表

filter 函数：

```
$(filter <pattern1 pattern2 ...>, <text>)
```

以 <pattern> 模式过滤 <text> 字符串中的单词，保留符合模式 <pattern> 的单词，可以有多个模式

实例：

```
$(filter %.c %.cpp, $(wildcard src/*))
```

输出：

src 目录下所有后缀是 .c 和 .cpp 的文件序列

call 函数：

```
$(call <expression>,<param1>,<param2>,...)
```

<expression> 中有 $(1) $(2) 这种占位符

call 函数使用 <param1> <param2> 来替换 $(1) $(2) 并计算表达式的值，返回结果字符串

实例：

```
listfile  = $(filter $(if $(2),$(addprefix %.,$(2)),%), \
                    $(wildcard $(addsuffix $(SLASH)*, $(1))))

list_cc = $(call listfile, src, c cpp)
```

输出：

src 目录下所有后缀是 .c 和 .cpp 的文件序列

function.mk -- part 1：

```
OBJPREFIX    := __objs_

.SECONDEXPANSION:
# -------------------- function begin --------------------

# list all files in some directories: (#directories, #types)
listf = $(filter $(if $(2),$(addprefix %.,$(2)),%),\
          $(wildcard $(addsuffix $(SLASH)*,$(1))))
```

`$(if $(2),$(addprefix %.,$(2)),%)` 用来判断 $(2) 是否为空字符串

如果是，则返回 %；如果不是，则返回加过 %. 前缀后的 $(2) 序列

`$(wildcard $(addsuffix $(SLASH)*,$(1)))` 用来获得 $(1) 目录下的所有文件

`filter` 函数的作用就是从 $(1) 目录下找出符合 if 函数返回值内序列 模式的所有文件

---

basename 函数：

```
$(basename <names...>)
```

从文件名序列 `<names>` 中取出各个文件名的前缀部分

可以理解为截取文件的文件名，从头开始直到最后一个 `.` 的前一个字符

返回文件名序列 `<names>` 的前缀序列，如果文件没有前缀，则返回空字串

实例：

```
$(basename src/foo.c src-1.0/bar.c hacks)
```

输出:

```
src/foo src-1.0/bar hacks
```

function.mk -- part 2:

```
# get .o obj files: (#files[, packet])
toobj = $(addprefix $(OBJDIR)$(SLASH)$(if $(2),$(2)$(SLASH)),\
        $(addsuffix .o,$(basename $(1))))
```

`$(if $(2),$(2)$(SLASH))` 先判断 `$(2)` 是否为空字符串，若不是，则返回字符串 `$(2)/`

注：`$(2)` 是未知的字符串变量名

`$(addsuffix .o,$(basename $(1)))` 用来截取所有文件的 basename，然后加上后缀 `.o`

`$(addprefix ...)` 整段代码的意思大致就是先换算出所有的 o 文件

然后设定相应的目录路径，将其设置为这些文件的前缀，以便之后创建或是修改文件的时候直接找到相应目录下的文件

---

patsubst 函数:

```
$(patsubst <pattern>,<replacement>,<text>)
```

查找 `<text>` 中的单词（单词以 空格 或 Tab 或 回车 或 换行 分隔）是否符合模式 `<pattern>`

如果匹配的话，则以 `<replacement>` 替换这里， `<pattern>` 可以包括通配符 `%` ，表示任意长度的字串

如果 `<replacement>` 中也包含 `%` ，那么 `<replacement>` 中的这个 `%` 将是 `<pattern>` 中的那个 `%` 所匹配到的字符串

（可以用 `\` 来转义，以 `\%` 来表示真实含义的 `%` 字符）

返回值为被替换过后的字符串

实例:

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

输出:

```
objects = foo.o bar.o baz.o,
```

function.mk -- part 3:

```
# get .d dependency files: (#files[, packet])
todep = $(patsubst %.o,%.d,$(call toobj,$(1),$(2)))

totarget = $(addprefix $(BINDIR)$(SLASH),$(1))
```

todep 过程

这段代码的意思是把经过 toobj 过程 而来的所有 o 文件进行模式匹配

假如匹配到 `%.o`，则将 `.o` 替换为 `.d`

看 toobj 过程 的意思，这段代码应该是不存在不匹配的情况的，应该是把所有的 o 文件都替换为 d 文件了

totarget 过程

这段代码的意思就是将 `$(1)` 序列里的字符串都加上 `bin/` 前缀

---

function.mk -- part 4:

```
# change $(name) to $(OBJPREFIX)$(name): (#names)
packetname = $(if $(1),$(addprefix $(OBJPREFIX),$(1)),$(OBJPREFIX))
```

先判断 `$(1)` 字符串是否为空字符串

如果为空，则直接返回 `__objs_` 字符串

否则返回 `$(1)` 序列加上前缀 `__objs_` 后的序列

---

function.mk -- part 5:

```
# cc compile template, generate rule for dep, obj: (file, cc[, flags, dir])
define cc_template
$$(call todep,$(1),$(4)): $(1) | $$$$(dir $$$$@)
    @$(2) -I$$(dir $(1)) $(3) -MM $$< -MT "$$(patsubst %.d,%.o,$$@) $$@"> $$@
$$(call toobj,$(1),$(4)): $(1) | $$$$(dir $$$$@)
    @echo + cc $$<
    $(V)$(2) -I$$(dir $(1)) $(3) -c $$< -o $$@
ALLOBJS += $$(call toobj,$(1),$(4))
endef
```

双 `$`是这样的：如果你要使用真实的 `$` 字符，那么你需要用`$$` 来表示。

这部分使用 define 多行定义了一个编译的模板

因为 call 是一个展开和替换的过程，会把调用函数内所有单个的 `$` 展开，因为 `$` 是一个关键字

也就是说，如果在利用 call 函数调用定义时，定义内有单个的 `$`

那么就会在 call 函数进行展开使调用 `$` 后面的函数，这里之所以会出现 `$$` 是因为后文又用了 eval 函数

实例：

```
a.o b.o x.o: a.c b.c x.c
cc -c $< -o $@
```

`$<` 表示 a.c （即所有依赖的合集的第一个文件）

`$@` 表示 a.o b.o x.o （即所用目标的合集）

cc_template 模板的前半部分是用于生成 d 文件（利用 gcc 的 -MM 选项）

后半部分则是使用 gcc 编译出 o 文件, 并且将所有 o 文件加入到 ALLOBJS 变量中

关于上述代码中的 `$(V)` 的使用，经过尝试能够发现 `make "V="` 会输出 gcc 命令的编译选项、include 目录等部分，恰好对应于上述代码中 `$(V)` 后的部分，因此猜测 `$(V)` 的使用是为了控制是否输出编译过程中的详细信息

---

function.mk -- part 6:

```
# compile file: (#files, cc[, flags, dir])
define do_cc_compile
$$(foreach f,$(1),$$(eval $$(call cc_template,$$(f),$(2),$(3),$(4))))
endef
```

表示将传入的文件列表中的每一个文件都使用 cc_template 生成编译模板

---

function.mk -- part 7:

```
# add files to packet: (#files, cc[, flags, packet, dir])
define do_add_files_to_packet
__temp_packet__ := $(call packetname,$(4))
ifeq ($$(origin $$(__temp_packet__)),undefined)
$$(__temp_packet__) :=
endif
__temp_objs__ := $(call toobj,$(1),$(5))
$$(foreach f,$(1),$$(eval $$(call cc_template,$$(f),$(2),$(3),$(5))))
$$(__temp_packet__) += $$(__temp_objs__)
endef
```

origin 函数不像其它的函数，他并不操作变量的值，他只是告诉你你的这个变量是哪里来的

先使用 call packetname 生成出某一个 packetname 对应的 makefile 中变量的名字

然后使用 origin 查询这个变量是否已经定义过，如果没定义过，则初始化该变量为空

之后使用 toobj 生成出该 packet 中所需要的生成的 o 文件的文件名列表

然后将其添加到以 __temp_packet__ 这个变量中所存的值作为名字的变量中去

并且使用 cc_template 生成出该 packet 生成 d 文件和 o 文件的代码

---

function.mk -- part 8:

```
# add objs to packet: (#objs, packet)
define do_add_objs_to_packet
__temp_packet__ := $(call packetname,$(2))
ifeq ($$(origin $$(__temp_packet__)),undefined)
$$(__temp_packet__) :=
endif
$$(__temp_packet__) += $(1)
endef
```

上述代码表示将某一个 o 文件添加到某一个 packet 对应的 makefile 中的变量中的文件列表中去

比方说，如果要添加 a.o 文件到名为 pack 的一个 packet 中

则结果就是 __objs_ 这个字符串变量会执行 `__objs_pack += a.o` 操作；

---

function.mk -- part 9:

```
# add packets and objs to target (target, #packes, #objs[, cc, flags])
define do_create_target
__temp_target__ = $(call totarget,$(1))
__temp_objs__ = $$(foreach p,$(call packetname,$(2)),$$($$(p))) $(3)
TARGETS += $$(__temp_target__)
ifneq ($(4),)
$$(__temp_target__): $$(__temp_objs__) | $$$$(dir $$$$@)
    $(V)$(4) $(5) $$^ -o $$@
else
$$(__temp_target__): $$(__temp_objs__) | $$$$(dir $$$$@)
endif
endef
```

这里将第一个参数传入的 binary targets 和第三个参数传入的 object 文件均添加到 TARGETS 变量中去

之后根据第 4 个参数是否传入 gcc 编译命令来确定是否生成编译的规则

---

function.mk -- part 10:

```
# finish all
define do_finish_all
ALLDEPS = $$(ALLOBJS:.o=.d)
$$(sort $$(dir $$(ALLOBJS)) $(BINDIR)$(SLASH) $(OBJDIR)$(SLASH)):
    @$(MKDIR) $$@
endef
```

创建编译过程中所需要的子目录

---

function.mk -- part 11:

```
# --------------------  function end  --------------------
# compile file: (#files, cc[, flags, dir])
cc_compile = $(eval $(call do_cc_compile,$(1),$(2),$(3),$(4)))

# add files to packet: (#files, cc[, flags, packet, dir])
add_files = $(eval $(call do_add_files_to_packet,$(1),$(2),$(3),$(4),$(5)))
```

```
7
8  # add objs to packet: (#objs, packet)
9  add_objs = $(eval $(call do_add_objs_to_packet,$(1),$(2)))
10
11 # add packets and objs to target (target, #packes, #objs, cc, [, flags])
12 create_target = $(eval $(call do_create_target,$(1),$(2),$(3),$(4),$(5)))
13
14 read_packet = $(foreach p,$(call packetname,$(1)),$($(p)))
15
16 add_dependency = $(eval $(1): $(2))
17
18 finish_all = $(eval $(call do_finish_all))
```

接下来这部分则是使用 eval 来进一步将原先设计好的编译代码中表达式的变量替换为变量的数值

Exercise 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?

Solution:

该扇区最后两个字节为 0X55AA，即该扇区有 512 个字节

练习2：使用qemu执行并调试lab1中的软件

没啥说的，就说说意外情况吧

1. 可以用 pwndbg，指令能对的上
2. 进入 qemu 了可以利用 `ctrl+alt+g` 来释放鼠标到正常的虚拟机系统
3. 利用 debug-nox 参数的话，关闭执行 qemu 的终端后（非标注的 qemu 终端，在原终端运行的那种）

   需要利用 `ps auxwww` 寻找 qemu 终端，用 kill 命令结束掉这个进程

   否则会报这个错误：`qemu-system-i386: Initialization of device ide-hd failed: Failed`

练习3：分析bootloader进入保护模式的过程

bootasm.S -- part 1：

```
#include <asm.h>

# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.set PROT_MODE_CSEG,        0x8                     # kernel code segment selector
.set PROT_MODE_DSEG,        0x10                    # kernel data segment selector
.set CR0_PE_ON,             0x1                     # protected mode enable flag

# start address should be 0:7c00, in real mode, the beginning address of the running bootloader
```

bootloader 入口为 start，根据 bootloader 的相关知识可知：

bootloader 会被 BIOS 加载到内存的 0x7c00 处，此时 cs = 0，eip = 0x7c00

这部分就是一些常量定义

bootasm.S -- part 2：

```
.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
```

在刚进入 bootloader 的时候，最先执行的操作分别为关闭中断、清除 EFLAGS 的 DF 位

以及将 ax，ds，es，ss 这四个寄存器初始化为 0

bootasm.S -- part 3：

```
    # 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
```

这段代码的过程就是开启 A20 的过程，下面一个一个指令说：

`inb $0x64, %al` 将 0x64 地址上的内容写入 al 寄存器里

`testb $0x2, %al` 将 0x64 地址上的内容和 0x2 地址上的内容比较，判断两者是否相等

本人测试如下：

```
pwndbg> x/4bx 0
0x0:    0x53    0xff    0x00    0xf0
```

所以这句应该是判断 0x64 地址上的内容是否为 0

`jnz seta20.1` 假如上一句汇编满足相等条件就向下执行，不满足就退回到 `inb $0x64, %al` 继续循环运行

直到吸收完所有的字符，也就是最后在 test 那句满足相等条件的时候，缓冲区是无字符的

即 8042 的 Input Buffer 寄存器已经被清空了

`movb $0xd1, %al` 和 `outb %al, $0x64` 这两句的直面翻译是把 0xd1 地址存入 0x64 地址

注意存的是地址不是地址上的值，具体的意思需要知道 i8042 键盘控制器的知识：

0xD1：准备写 Output 端口，随后通过 0x60 端口写入的字节，会被放置在 Output Port 中

0x60：驱动中把 0x60 叫数据端口

0x64：驱动中把 0x64 叫命令端口

所以这两句汇编的意思就是往 0x64 命令端口写入 0xD1 命令，表示修改 8042 的 P2 port

8042 有三个 port：
Input Port Pl, Output Port P2，Test Port T

System Reset 和 A20 Gate 是两个与键盘无关的重要的控制位
GateA20：决定 CPU 是否可以访问以 MB 为单位的偶数内存
  0：CPU 工作于 DOS 的实模式
  1：CPU 可进入保护模式
KRST*：向系统发送 Reset 信号，可让主机重新启动
  0：normal
  1：reset computer

bootasm.S -- part 4:

```
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
```

继续等待 Input Buffer 为空，之后向 0x60 端口里输入数值 0xdf，开启 A20

现在内核进入保护模式之后可以充分使用 4G 的寻址能力

bootasm.S -- part 5:

```
    # 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

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

`lgdt gdtdesc` 用来初始化GDT表，一个简单的GDT表和其描述符已经静态储存在引导区中，载入即可

下面三行代码的作用是进入保护模式：通过修改 cr0 寄存器的 PE 位，将其置 1 便开启了保护模式

`ljmp $PROT_MODE_CSEG, $protcseg` 使用长跳转指令，将cs修改为32位段寄存器

并跳转到 protcseg 这一 32 位代码入口处，此时 CPU 进入 32 位模式

bootasm.S -- part 6:

```
.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
    movw %ax, %gs                                   # -> GS
    movw %ax, %ss                                   # -> SS: Stack Segment

    # 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

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

设置 ds，es，fs，gs，ss 这几个段寄存器的值，gdb 调试如下：

```
cs             0x8     8
ss             0x10    16
ds             0x10    16
es             0x10    16
fs             0x10    16
gs             0x10    16
```

之后初始化栈的 frame pointer 和 stack pointer，再调用 C 语言编写的 bootmain 函数，加载操作系统的内核

至此，bootloader 已经完成了从实模式进入到保护模式的任务

练习4：分析bootloader加载ELF格式的OS的过程

这里分析 bootmain.c 文件

Exercise 4.1：

bootloader如何读取硬盘扇区的?

Solution:

waitdisk 函数：

```
#include <defs.h>
#include <x86.h>
#include <elf.h>

#define SECTSIZE        512
#define ELFHDR          ((struct elfhdr *)0x10000)      // scratch space

/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}
```

该函数的作用是从 0x1F7 地址连续不断地读取磁盘的状态，直到磁盘不忙为止

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);
}
```

流程如下：

`waitdisk();` 等待磁盘不忙

`outb(0x1F2, 1);` 往 0X1F2 地址中写入要读取的扇区数，由于此处需要读一个扇区，因此参数为 1

`outb(0x1F3, secno & 0xFF);` 输入 LBA 参数的 0 - 7 位

`outb(0x1F4, (secno >> 8) & 0xFF);` 输入 LBA 参数的 8 - 15 位

`outb(0x1F5, (secno >> 16) & 0xFF);` 输入 LBA 参数的 16 - 23 位

`outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);`

输入 LBA 参数的 24 - 27 位（对应到 0 - 3 位），第 4 位为 0 表示从主盘读取，其余位被强制置为 1

`outb(0x1F7, 0x20);` 向磁盘发出读命令 0x20（上文提到写命令是 0x60）

`waitdisk();` 等待磁盘不忙（完成读取操作）

`insl(0x1F0, dst, SECTSIZE / 4);`

从数据端口 0x1F0 读取数据，除以 4 是因为此处是以 4 个字节为单位的

从这个 insl 指令是以 l(long) 结尾这点可以推测出来

PS：

```
0x1f0 读数据，当 0x1f7 不为忙状态时，可以读。
0x1F3 R/W，数据寄存器
0x1F2 R/W，扇区数寄存器，记录操作的扇区数
0x1F3 R/W，扇区号寄存器，记录操作的起始扇区号
0x1F4 R/W，柱面号寄存器，记录柱面号的低 8 位
0x1F5 R/W，柱面号寄存器，记录柱面号的高 8 位
0x1F6 R/W，驱动器/磁头寄存器，记录操作的磁头号，驱动器号，和寻道方式，前 4 位代表逻辑扇区号的高 4 位，DRV = 0/1 代表主/从驱动器，LBA = 0/1 代表 CHS/LBA 方式
0x1F7 R，状态寄存器，第 6、7 位分别代表驱动器准备好，驱动器忙
0x1F8 W，命令寄存器，0x20 命令代表读取扇区
```

Exercise 4.2

bootloader是如何加载ELF格式的OS?

Solution:

bootloader 加载 ELF 格式的 OS 的代码位于 bootmain.c 中的 bootmain 函数中

接下来不妨分析这部分代码来描述加载 ELF 格式 OS 的过程：

readseg 函数：

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

bootmain 函数 -- part 1：

```
/* 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;
    }
```

`readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);`

首先，从磁盘的第 1 个扇区（第 0 个扇区为 bootloader）中读取 OS kenerl 最开始的 4 kB 代码

`if (ELFHDR->e_magic != ELF_MAGIC)`

然后判断其最开始 4 个字节是否等于指定的 ELF_MAGIC，用于判断该 ELF header 是否合法

bootmain 函数 -- part 2：

```
    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);
    }
```

`ph = (struct proghdr *) ...` 接下来从 ELF 头文件中获取 program header 表的位置（描述表的头地址）

`eph = ph + ELFHDR->e_phnum;` 以及该表的入口数目，然后遍历该表的每一项

`for (; ph < eph; ph ++) ...`

然后获取每一个 program header 中获取到段应该被加载到内存中的位置，以及段的大小

利用 readseg 函数将每一个段加载到内存中，至此完成了将 OS 加载到内存中的操作

ELF文件0x1000位置后面的0xd1ec比特被载入内存0x00100000

ELF文件0xf000位置后面的0x1d20比特被载入内存0x0010e000

bootmain 函数 -- part 3:

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

这段的意思就是根据ELF头部储存的入口信息，找到内核的入口，然后使用函数调用的方式跳转到该地址上去

至此，完整地分析了 bootloader 加载 ELF 格式的 OS kernel 的过程

练习5：实现函数调用堆栈跟踪函数

```
void
print_stackframe(void) {
    uint32_t ebp = read_ebp();
    uint32_t eip = read_eip();
    for (int i = 0; i < STACKFRAME_DEPTH && ebp != 0; ++i) {
        cprintf("ebp:0x%08x eip:0x%08x ", ebp, eip);
        uint32_t* ptr = (uint32_t *) (ebp + 8);
        cprintf("args:0x%08x 0x%08x 0x%08x 0x%08x\n", ptr[0], ptr[1],
ptr[2], ptr[3]);
        print_debuginfo(eip - 1);
        eip = *((uint32_t *) (ebp + 4));
        ebp = *((uint32_t *) ebp);
    }
}
```

这么写即可，先是读出来 ebp 和 eip，然后进入 for 循环

在循环里，先打印出 ebp 和 eip 的值，然后获取 ebp + 8 的地址作为 ptr 的值

这个值是 ret + 4 的位置（也就是传参的初始位置）

然后打印出 ptr 的值所对应的地址以及它的后 3 位地址上的值，即打印参数

调用 print_debuginfo 函数，输出调代码内调用函数的信息

调整 ebp 和 eip 为下一轮的值，继续进行循环

直到 i 等于 STACKFRAME_DEPTH 或者是 ebp 的值为 0 时，退出循环

练习6：完善中断初始化和处理

Exercise 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？

Solution:

1. IDT 中的每一个表项均占 8 个字节

```
struct gatedesc {
unsigned gd_off_15_0 : 16;  // 低16位为段内偏移
unsigned gd_ss : 16;        // 段选择子占16位
unsigned gd_args : 5;       // # args, 0 for interrupt/trap gates ,also 0
unsigned gd_rsv1 : 3;       // reserved(should be zero I guess)
unsigned gd_type : 4;       // type(STS_{TG,IG32,TG32}) interrupt or trap
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; // 作为高16位的段内偏移。
};
```

2. (gd_off_31_16 << 16) + gd_off_15_0

   0-15 位为偏移的低 16 位，高 16 位位于中断描述符的最高 16 位(48 - 64 位)

   第 16 - 31 位定义了处理代码入口地址的段选择子

   使用其在 GDT 中查找到相应段的 base address，加上 offset 就是中断处理代码的入口

Exercise 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。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。

Solution:

先看别的函数

SETGATE 函数：

```
/* *
 * 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;        \
}
```

翻译一下注释里的内容：

```
设置一个正常的中断/陷阱门描述符
  - istrap: 1 为陷阱(= 异常)门，0 为中断门
  - sel:代码段选择中断/陷阱处理程序
  - off:中断/陷阱处理程序代码段中的偏移量，即 __vectors[] 中的内容
  - dpl: 描述符权限级别 - 软件使用int指令显式调用中断/陷阱门所需的权限级别。
```

然后分析原答案：

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); ++i) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    // set for switch from user to kernel
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK],
DPL_USER);
    // load the IDT
    lidt(&idt_pd);
}
```

idt_init 函数的功能是初始化 IDT 表，IDT 表中每个元素均为门描述符，记录一个中断向量的属性

包括中断向量对应的中断处理函数的 段选择子/偏移量、门类型（是中断门还是陷阱门）、DPL等

因此初始化 IDT 表实际上是初始化每个中断向量的这些属性

`extern uintptr_t __vectors[];`

调用外部定义变量 __vectors[]

将其放在函数内部，使其仅在本函数内可见，结构上更合理

`for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); ++i) ...`

这里一般就是 i < 0x100 了，接下来调用 SETGATE 函数

由 vector.S 文件开头可知，中断处理函数属于 .text 的内容

因此中断处理函数的段选择子即 .text 的段选择子 -- GD_KTEXT

在中断表中有两个中断， T_SWITCH_TOU 和 T_SWITCH_TOK ，一个是切换到用户态，另一个是切换回内核态

跟 pkfxxxx 师傅交流后了解到了这些东西：

T_SWITCH_TOU 和 T_SWITCH_TOK 是专门用来在 lab 中做一个模拟测试的，在之后的 lab 里这两者会被弃用

且 T_SWITCH_TOK 调用是用 T_SYSCALL 实现的，所以这里写 T_SWITCH_TOK 也行，写 T_SYSCALL 也行

`lidt(&idt_pd);` 加载 IDT 表，使其完成初始化

Exercise 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”。

Solution:

trap 函数：

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

可以看出 trap 函数功能的实现都在 **trap_dispatch 函数**里，那么就要修改 **trap_dispatch 函数**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");
        }
    }
}
```

可以看到第一个 case 语句判断的就是是否发生了时钟中断，注释告诉我们可以使用 clock.c 里面的 ticks 全局变量

所以我们可以这么写代码来完成目标：

```
    case IRQ_OFFSET + IRQ_TIMER:
        if(++ticks % 100 == 0){
            print_ticks();
            ticks = 0;
        }
        break;
```

做完 Exercise 6.2 后，该代码就能够生效并且打印 100 ticks 了

扩展练习 Challenge 1：

稍微分析跟踪一下 ISR 的流程，首先在中断表中注册的 vectors 数组中存放着准备参数和跳转到 `__alltraps` 函数的几个指令，在 `__alltraps` （在 kern/trap/trapentry.S 中定义）函数中，将原来的段寄存器压栈后作为参数 `struct trapframe *tf` 传递给 `trap_dispatch` ，并在其中分别处理。

中断处理函数在退出的时候会把这些参数全部 `pop` 回寄存器中，于是我们可以趁它还在栈上的时候修改其值，在退出中断处理的时候相应的段寄存器就会被更新

又到了分析源代码的时间：

kern/init/init.c

```
static void lab1_switch_to_user(void) {
------------------------------------------------------------
    "sub $0x8, %%esp \n"
    让 SS 和 ESP 这两个寄存器有机会 POP 出时更新 SS 和 ESP
    因为从内核态进入中断，它的特权级没有改变，是不会 push 进 SS 和 ESP 的
```

```
    但是我们又需要通过 POP SS 和 ESP 去修改它们
    进入 T_SWITCH_TOU(120) 中断，将原来的栈顶指针还给 ESP 栈底指针
-------------------------------------------------------------
    asm volatile (
        "sub $0x8, %%esp \n"
        "int %0 \n"
        "movl %%ebp, %%esp"
        :
        : "i"(T_SWITCH_TOU)
    );
}

static void lab1_switch_to_kernel(void) {
-------------------------------------------------------------
    进入 T_SWITCH_TOK(121) 中断
    因为我们强行改为内核态，会让cpu认为没有发生特权级转换。于是esp的值就不对了
    这时我们需要将原来的栈顶指针还给esp栈底指针
-------------------------------------------------------------
    asm volatile (
        "int %0 \n"
        "movl %%ebp, %%esp \n"
        :
        : "i"(T_SWITCH_TOK)
    );
}
```

kern/trap/trap.c

```
通过"改造"一个中断 来进入我们想进入的用户态或者内核态
    case T_SWITCH_TOU:
        if (tf->tf_cs != USER_CS) {
            switchk2u = *tf;
            switchk2u.tf_cs = USER_CS;
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
            switchk2u.tf_eflags |= FL_IOPL_MASK; // IOPL 改为 0
            // tf->esp的位置
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;

            // 设置临时栈，指向switchk2u，这样iret返回时
            // CPU会从switchk2u恢复数据，而不是从现有栈恢复数据
            *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
        }
        break;
    case T_SWITCH_TOK:
        if (tf->tf_cs != KERNEL_CS) {
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
            switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct
trapframe) - 8));
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
        }
        break;
```

为了能正常地输出，需要修改 IO 权限位，这里说的是

在 EFLAGS 寄存器中的第 12/13 位控制着 IO 权限

这个域只有在 GDT 中的权限位为 0 （最高权限）时，通过 `iret` 或 `popf` 指令修改

只有在 IO 权限位大于等于 GDT 中的权限位才能正常使用 `in` `out` 指令

我们可以在 `trap_dispatch` 中通过 `trap_frame` 中对应位修改 EFLAGS 。

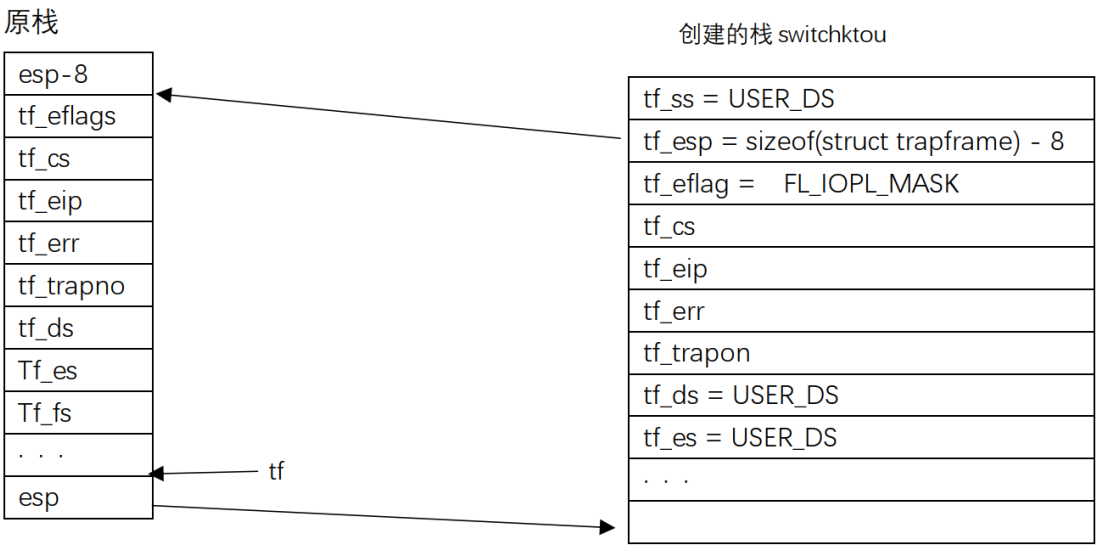
测试前要记得先开启测试环境并定义变量：

1. 测试环境：在 kern/init/init.c 文件中 kern_init 函数里，取消对 lab1_switch_test 函数的注释
2. 定义变量：在 kern/trap/trap.c 文件中写入如下代码以定义变量

```
struct trapframe switchk2u, *switchu2k;
```

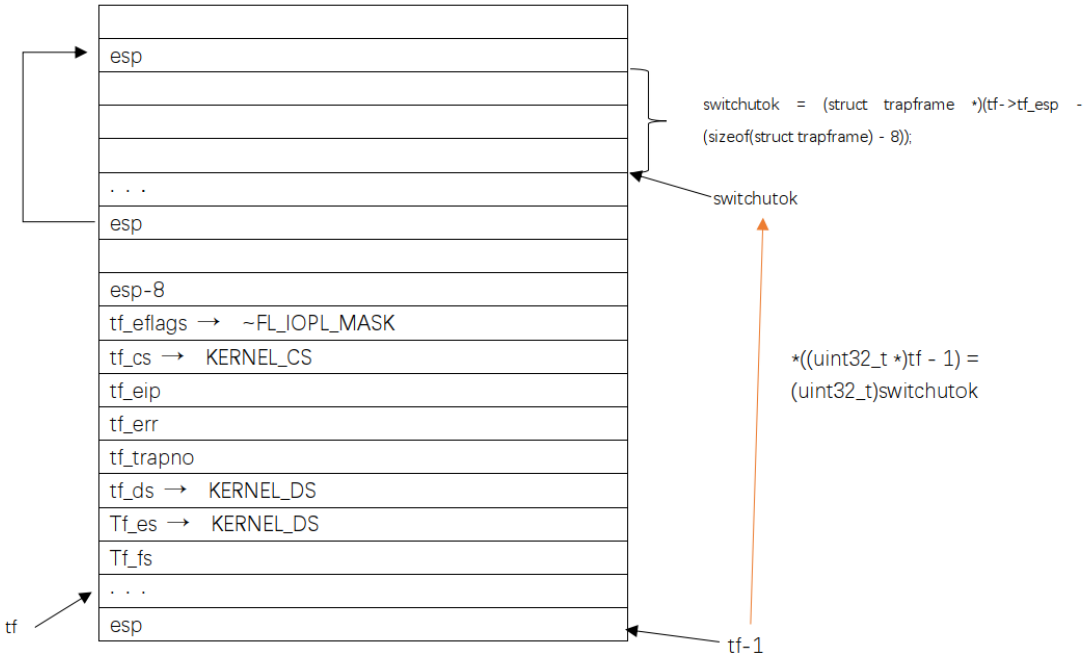
接下来用图理解，果然图示能帮助我这个智障理解答案内容。。。

lab1_switch_to_user：



这个大致能看懂，不赘述了，反正就是倒腾 esp 指向伪造的地址，最底下那个 esp 原本是 ebp 来着

lab1_switch_to_kernel：



首先是留下 SS 和 ESP 的位置

然后调用中断修改中断栈里面的内容

最后退出中断的时候，跳到内核态中，将 ebp 赋给 esp，修复 esp 的位置

注：经测试发现 `case T_SWITCH_TOK:` 里不写后三行也能得满分

应该是没留 ss 和 esp 的位置，留了更好

扩展练习 Challenge 2:

击键有对应的中断号，在 `case T_SWITCH_TOU:` 上面，原始代码如下：

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

早知道就把用户到内核和内核到用户封装成函数了，又把 if 语句照抄了一遍

答案如下：

```
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        switch (c) {
        case '0':
            if (tf->tf_cs != KERNEL_CS) {
                tf->tf_cs = KERNEL_CS;
                tf->tf_ds = tf->tf_es = KERNEL_DS;
                tf->tf_eflags &= ~FL_IOPL_MASK;
                switchu2k = (struct trapframe *)(tf->tf_esp -
(sizeof(struct trapframe) - 8));
                memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
                *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
```

```
            }
            print_trapframe(tf);
            break;
        case '3':
            if (tf->tf_cs != USER_CS) {
                switchk2u = *tf;
                switchk2u.tf_cs = USER_CS;
                switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss =
USER_DS;
                switchk2u.tf_eflags |= FL_IOPL_MASK;
                switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe)
- 8;
                *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
            }
            print_trapframe(tf);
            break;
        }
        cprintf("kbd [%03d] %c\n", c, c);
        break;
```

参考链接

https://www.jianshu.com/p/2f95d38afa1d

https://blog.csdn.net/linuxweiyh/article/details/90301424

https://www.bookstack.cn/read/how-to-write-makefile/b473e56b6c52d350.md

https://blog.csdn.net/huohongpeng/article/details/72624910

http://shanzy.bokee.com/834368.html

https://wenku.baidu.com/view/2baeca4fff00bed5b9f31dcf.html

https://xr1s.me/2018/05/15/ucore-lab1-report/

https://yuerer.com/操作系统-uCore-Lab-1/

https://blog.csdn.net/sinat_30955745/article/details/80976997