杜佳男 21210110

操作系统lab1实验报告

1、实验思考题

Thinking 1.1

请查阅并给出`objdump`中使用的参数的含义。使用其它体系结构的编译器（如课程平台的 MIPS 交叉编译器）重复上述各步编译过程，观察并在实验报告中提交相应的结果。

`objdump`参数：

-D --disassemble-all 与 -d 类似，但反汇编所有section.

-S --source 尽可能反汇编出源代码

```
Usage: objdump <option(s)> <file(s)>
 Display information from object <file(s)>.
 At least one of the following switches must be given:
  -a, --archive-headers    Display archive header information
  -f, --file-headers       Display the contents of the overall file header
  -p, --private-headers    Display object format specific file header contents
  -P, --private=OPT,OPT... Display object format specific contents
  -h, --[section-]headers  Display the contents of the section headers
  -x, --all-headers        Display the contents of all headers
  -d, --disassemble        Display assembler contents of executable sections
  -D, --disassemble-all    Display assembler contents of all sections
      --disassemble=<sym>  Display assembler contents from <sym>
  -S, --source             Intermix source code with disassembly
      --source-comment[=<txt>] Prefix lines of source code with <txt>
  -s, --full-contents      Display the full contents of all sections requested
  -g, --debugging          Display debug information in object file
  -e, --debugging-tags     Display debug information using ctags style
  -G, --stabs              Display (in raw form) any STABS info in the file
  -W[lLiaprmfFsoRtUuTgAckK] or
  --dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
```

x86编译器执行代码：

```
gcc -E hello.c > gcc_E_hello.txt
gcc -c hello.c -o hello1.o
objdump -DS hello1.o> objdump_hello.txt
```

mips交叉编译器执行代码：

```
/OSLAB/compiler/usr/bin/mips_4KC-gcc -E -I /usr/include hello.c >
mips_gcc_E_hello.txt
/OSLAB/compiler/usr/bin/mips_4KC-gcc -c -I /usr/include hello.c -o hello2.o
/OSLAB/compiler/usr/bin/mips_4KC-objdump -DS hello2.o > mips_dump_hello.txt
```

两者对比：

```
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 858 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 873 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2


# 3 "hello.c"
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

x86编译器预处理（部分）

```
extern int pclose (FILE *__stream);



extern char *ctermid (char *__s) __attribute__ ((__nothrow__ ));
# 840 "/usr/include/stdio.h"
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ ));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ )) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ ));
# 858 "/usr/include/stdio.h"
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 873 "/usr/include/stdio.h"

# 2 "hello.c" 2

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

mips编译器预处理（部分）

```
hello1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # f <main+0xf>
   f:   e8 00 00 00 00          callq  14 <main+0x14>
  14:   b8 00 00 00 00          mov    $0x0,%eax
  19:   5d                      pop    %rbp
  1a:   c3                      retq

Disassembly of section .rodata:

0000000000000000 <.rodata>:
   0:   48                      rex.W
   1:   65 6c                   gs insb (%dx),%es:(%rdi)
   3:   6c                      insb   (%dx),%es:(%rdi)
   4:   6f                      outsl  %ds:(%rsi),(%dx)
   5:   20 57 6f                and    %dl,0x6f(%rdi)
   8:   72 6c                   jb     76 <main+0x76>
   a:   64 21 00                and    %eax,%fs:(%rax)
```

x86编译器反汇编（部分）

```
Disassembly of section .text:

00000000 <main>:
   0:   3c1c0000    lui gp,0x0
   4:   279c0000    addiu   gp,gp,0
   8:   0399e021    addu    gp,gp,t9
   c:   27bdffe0    addiu   sp,sp,-32
  10:   afbf001c    sw  ra,28(sp)
  14:   afbe0018    sw  s8,24(sp)
  18:   03a0f021    move    s8,sp
  1c:   afbc0010    sw  gp,16(sp)
  20:   8f820000    lw  v0,0(gp)
  24:   24440000    addiu   a0,v0,0
  28:   8f990000    lw  t9,0(gp)
  2c:   0320f809    jalr    t9
  30:   00000000    nop
  34:   8fdc0010    lw  gp,16(s8)
  38:   00001021    move    v0,zero
  3c:   03c0e821    move    sp,s8
  40:   8fbf001c    lw  ra,28(sp)
  44:   8fbe0018    lw  s8,24(sp)
  48:   27bd0020    addiu   sp,sp,32
  4c:   03e00008    jr  ra
  50:   00000000    nop
    ...
```

mips编译器反汇编（部分）

可以看到在预处理阶段，二者的结果大致相同但也存在一定的差异，在对编译后未链接结果的反汇编中，因x86系统和mip系统的指令集存在一定的差异，二者的反汇编结果存在较大差异。

Thinking 1.2

使用自己编写的readelf和linux命令中自带的readelf分别编译testELF和vmlinux，发现二者都可以编译testELF，但自己编写的readelf无法解析vmlinux，而linux命令自带的可以，结果如下：

```
git@21210110:~/21210110/readelf$ ./readelf testELF
0xd9f765e8                              0:0x0
0xd9f76610                              1:0x8048154
0xd9f76638                              2:0x8048168
0xd9f76660                              3:0x8048188
0xd9f76688                              4:0x80481ac
0xd9f766b0                              5:0x80481cc
0xd9f766d8                              6:0x804828c
0xd9f76700                              7:0x804830e
0xd9f76728                              8:0x8048328
0xd9f76750                              9:0x8048358
0xd9f76778                              10:0x8048360
git@21210110:~/21210110/readelf$ readelf -S testELF
There are 30 section headers, starting at offset 0x1158:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000011 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 0000c0 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804828c 00028c 000081 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0804830e 00030e 000018 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048328 000328 000030 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048358 000358 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048360 000360 000050 08   A  5  12  4
```

测试testELF

```
git@21210110:~/21210110/readelf$ ./readelf vmlinux
Segmentation fault (core dumped)
git@21210110:~/21210110/readelf$ readelf -S vmlinux
There are 14 section headers, starting at offset 0x915c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        80010000 000080 00b30 00 WAX  0   0 16
  [ 2] .reginfo          MIPS_REGINFO    80010b30 000bb0 000018 18   A  0   0  4
  [ 3] .rodata.str1.4    PROGBITS        80010b48 000bc8 0000a2 01 AMS  0   0  4
  [ 4] .rodata           PROGBITS        80010bf0 000c70 000210 00   A  0   0 16
  [ 5] .data             PROGBITS        80010e00 000e80 000000 00  WA  0   0 16
  [ 6] .data.stk         PROGBITS        80010e00 000e80 008000 00  WA  0   0  1
  [ 7] .bss              NOBITS          80018e00 008e80 000000 00  WA  0   0 16
  [ 8] .pdr              PROGBITS        00000000 008e80 0001a0 00      0   0  4
  [ 9] .mdebug.abi32     PROGBITS        00000000 009020 000000 00      0   0  1
  [10] .comment          PROGBITS        00000000 009020 0000c8 00      0   0  1
  [11] .shstrtab         STRTAB          00000000 0090e8 000072 00      0   0  1
  [12] .symtab           SYMTAB          00000000 00938c 000250 10     13  24  4
  [13] .strtab           STRTAB          00000000 0095dc 0000c2 00      0   0  1
```

测试vmlinux

使用命令`readelf -h`分别查看testELF和vmlinux两个文件，结果如下：

```
git@21210110:~/21210110/readelf$ readelf -h testELF
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048490
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4440 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 27
git@21210110:~/21210110/readelf$ readelf -h vmlinux
```

查看testELF

```
git@21210110:~/21210110/readelf$ readelf -h vmlinux
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x80010000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          37212 (bytes into file)
  Flags:                             0x1001, noreorder, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         14
  Section header string table index: 11
```

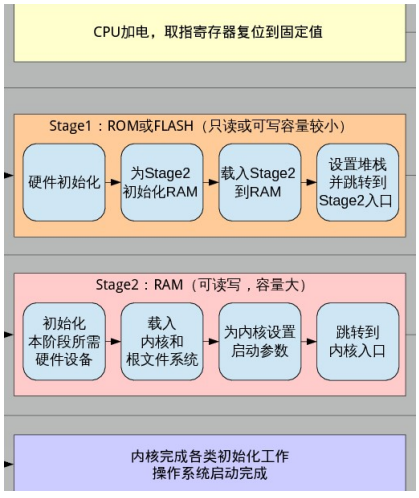
查看vmlinux

比较两个查看结果可以发现，testELF文件是小端存储模式（little endian），vmlinux文件是大端存储模式（big endian），而我们所编写的readelf文件只能解析小端存储的文件，而不能解析大端存储的文件，因此不能解析内核文件vmlinux。想要解析大端存储的ELF文件还需要额外添加将大端转化为小端的函数。

Thinking 1.3

在理论课上我们了解到，MIPS 体系结构上电时，启动入口地址为0xBFC00000（其实启动入口地址是根据具体型号而定的，由硬件逻辑确定，也有可能不是这个地址，但一定是一个确定的地址），但实验操作系统的内核入口并没有放在上电启动地址，而是按照内存布局图放置。思考为什么这样放置内核还能保证内核入口被正确跳转到？（提示：思考实验中启动过程的两阶段分别由谁执行。）

从实验指导书，我们可以发现操作系统的启动依赖于bootloader来实现，其作用为正确地调用内核来执行。观察下图（来自实验指导书）：



可以发现从CPU上电到操作系统内核被加载的整个启动的分为两大步骤：

stage1初始化硬件设备，此时RAM尚未初始化完成，stage1直接运行在存放bootloader的存储设备（比如FLASH）上， 并为stage2准备RAM空间，并将其代码复制到RAM空间，设置堆栈，跳转到stage2的入口函数。

stage2运行在RAM，初始化这一阶段需要使用的硬件设备以及其他功能，将内核镜像文件从存储器读到RAM中，并为内核设置启动参数，最后将CPU指令寄存器的内容设置为内核入口函数的地址，以此保证存放在内存中的内核入口可以被正确跳转到，再将控制权从bootloader转交给操作系统内核。

Thinking 1.4

sg_size 和 bin_size 的区别它的开始加载位置并非页对齐，同时 bin_size 的结束位置（va+i）也并非页对齐，最终整个段加载完毕的 sg_size 末尾的位置也 并非页对齐，请思考，为了保证页面不冲突（不重复为同一地址申请多个页，以及页上数据尽可能减少冲突），这样一个程序段应该怎样加载内存空间中。

`sg_size` 代表该段所需要的的总内存空间的大小，`bin_size` 则代表该段在ELF 文件中的大小。类似于课下实验，由于在内核加载的时候不一定是页对齐的，所以我们可以采用边加载边装入的方法，在装入之后，判断下一个加载的虚地址和前一个装入的尾地址是否在同一页内，若上一个程序的结束页为v，则下一个程序的起始页为v+1以避免冲突。

Thinking 1.5

内核入口在什么地方？main 函数在什么地方？我们是怎么让内核进入到想要的 main 函数的呢？又是怎么进行跨文件调用函数的呢？

由指导书，linker中程序入口的设置方法有：

- 使用ld命令时，通过参数“-e”设置
- 在linker scirpt中使用ENTRY()指令指定了程序入口
- 如果定义start，则start就是程序入口
- .text段的第一个字节
- 地址0处

观察课下实验中写过的 `scse0_3.lds` 文件即可知道**程序入口由** `ENTRY(_START)` **指定。**

对内核文件进行反汇编，指令如下：

```
/OSLAB/compiler/usr/bin/mips_4KC-objdump -D gxemul/vmlinux > objvmlinux.txt
```

可以得到内核文件的汇编文件，观察可以得到内核入口进而main函数的位置：

```
gxemul/vmlinux:     file format elf32-tradbigmips

Disassembly of section .text:

80010000 <_start>:
80010000:   40806000    mtc0    zero,$12
80010004:   00000000    nop
80010008:   40809000    mtc0    zero,$18
8001000c:   00000000    nop
80010010:   40809800    mtc0    zero,$19
80010014:   00000000    nop
80010018:   40088000    mfc0    t0,$16
8001001c:   2401fff8    li  at,-8
80010020:   01014024    and t0,t0,at
80010024:   35080002    ori t0,t0,0x2
80010028:   40888000    mtc0    t0,$16
8001002c:   0c004010    jal 80010040 <main>
80010030:   3c1d8040    lui sp,0x8040

80010034 <loop>:
80010034:   0800400d    j   80010034 <loop>
80010038:   00000000    nop
8001003c:   00000000    nop

80010040 <main>:
80010040:   27bdffe8    addiu   sp,sp,-24
80010044:   afbf0010    sw  ra,16(sp)
80010048:   3c048001    lui a0,0x8001
8001004c:   0c0042a2    jal 80010a88 <printf>
80010050:   24840b48    addiu   a0,a0,2888
80010054:   0c004020    jal 80010080 <mips_init>
80010058:   00000000    nop
```

可以看到，内核入口在`0x80010000`，main函数在.text段，地址为`0x80010040`。

`main`函数在汇编后其首地址会被翻译成一个标签main，通过在`start.S`中设置栈指针指向栈顶，为`main`的运行分配栈空间，然后`jal main`跨文件调用，即可跳转到`main`函数的地址处。

不同文件的函数首地址在汇编时都会被翻译成一个标签，而调用函数的位置处空缺等待填入，在链接过程中各个可重定向文件被集合到一个可执行文件中，实质分别将`.text`、`.data`、`.bss`数据段整合在一起，并整体划分地址，此时调用函数的地址处会被填入函数入口逻辑地址，就可以在执行过程中调用其他文件的函数了。

Thinking 1.6

查阅《See MIPS Run Linux》一书相关章节，解释 boot/start.S 中下 面几行对 CP0 协处理器寄存器进行读写的意义。具体而言，它们分别读/写了哪些寄存器的哪些特定位，从而达到什么目的?

每一步的作用已标注在代码区；

其中，指令`mtc0`的作用：将通用寄存器的内容移动到协处理器0寄存器。

Move to Coprocessor 0 **MTC0**

31 26	25 21	20 16	15 11	10 3	2 0
COP0	MT	rt	rd	0	sel
010000	00100			0000 000	
6	5	5	5	8	3

Format: `MTC0 rt, rd` **MIPS32**
`MTC0 rt, rd, sel` **MIPS32**

Purpose:

To move the contents of a general register to a coprocessor 0 register.

start.S内容：

```
/* Disable interrupts */
mtc0 zero, CP0_STATUS /*将CP0_STATUS置0*/
 ......
 /*禁止全局中断*/

/* disable kernel mode cache */
mfc0 t0, CP0_CONFIG  /*从CP0协处理器取出CONFIG数据*/
and t0, ~0x7  /*把$t0中数据的低三位清零*/
ori t0, 0x2  /* 把$t0中数据从低位起第二位置为1*/
mtc0 t0, CP0_CONFIG  /*将CONFIG赋值为t0中的数据*/
/*设置kseg0区不经过cache。cache需要先初始化才能使用。*/
```

那么，为什么将`CP0_CONFIG`后三位置为010就可以设置kseg0区不经过cache了呢?

首先看下面这张图（此思考题以下部分图片均来自《See MIPS Run Linux》）

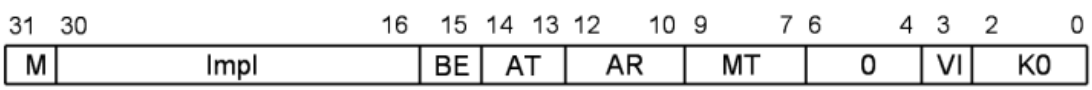


图 3.4: MIPS32/64 Config 寄存器的各个域

可以发现，协处理器0寄存器的Config寄存器的K0域对应后三位，而K0可写的域则用来决定固定的kseg0区是否经过高速缓存，如下所示：

> **K0** 可写的域，用来决定固定的 kseg0 区是否经过高速缓存。如果要经过高速缓存，其确切行为如何？该域的编码和在 **EntryLo0-1(C)** 中见到的 TLB 项的高速缓存选择域一样，在图 6.3 的注释中有说明。

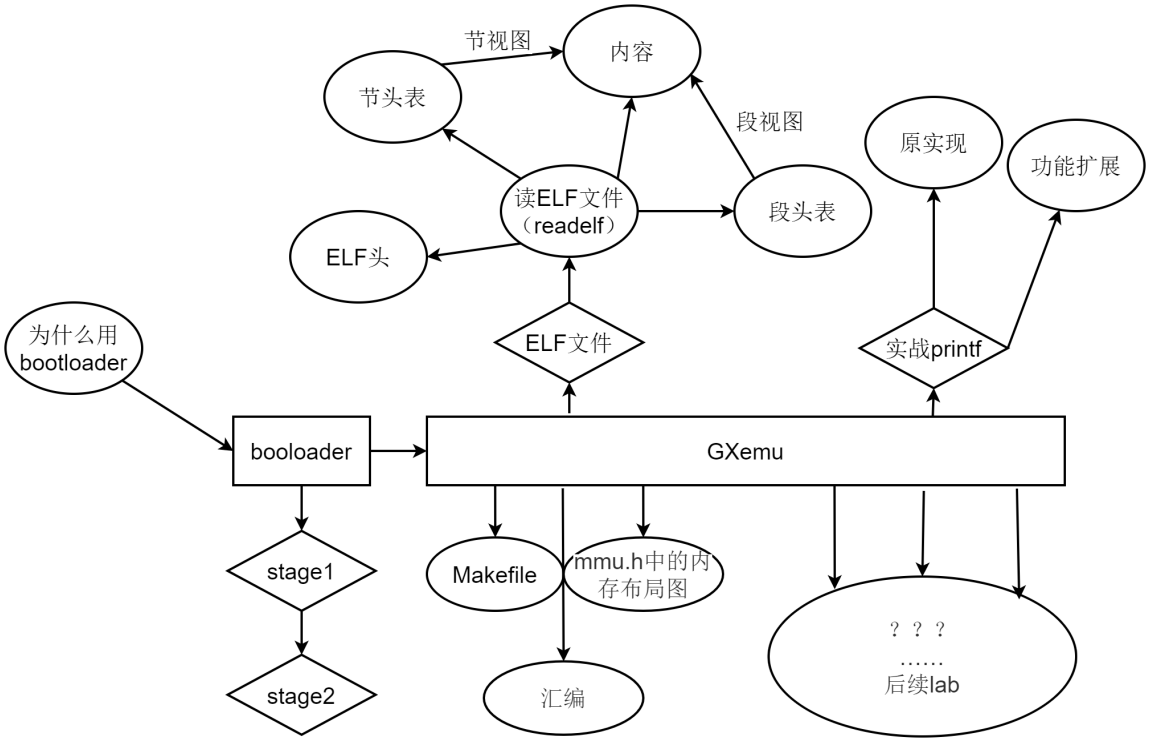
通过该部分介绍，我们可以知道K0域编码和TLB项高速缓存选择域一样，跟着索引我们继续来看：

> - C: 一个 3 位的域，本来是为高速缓存一致性的多处理器系统定义的，用来设置“高速缓存算法”（也叫做“高速缓存一致性属性(cache coherence attribute)”——有些手册把该域叫做“CCA”）。典型的操作系统都知道有些页面不需要在多个高速缓存之间自动跟踪变化——已知只有一个 CPU 使用的页面，或已知为只读的页面，这些不需要太多的关心。关闭对这些页面存取的高速缓存侦听和交互可以让系统更加有效率，这个域由操作系统用来记录相应页为，比如说，可以高速缓存但是不需要一致性管理（“cacheable noncoherent”）。
>
> 但是这个域也在瞄准嵌入式应用的 CPU 中使用，这时用来选择高速缓存工作方式——例如，标记某个具体的页为“透写”式管理（就是说，所有在那里进行的写操作都同时直接送到主存和高速缓存的映像）。
>
> 该域仅有的普遍支持的值包括，“不用高速缓存(uncached)”(2)和“可高速缓存但不要一致性(cacheable noncoherent)”(3)。

于是我们可以得知，该三位的域值为2，也即二进制的010代表着不用高速缓存。

2、实验难点图示

ELF部分，一旦熟悉了ELF结构就知道如何下手了；printf部分只要考虑全面也不会有太大的bug，关键部分还在于理论，特别是bootloader部分。



3、体会与感受

难度评价稍难，相比lab0难度有所拔高，实验花费时间每周写代码读文件约4~5h，debug约3h，有很多概念还是不明确，不询问别人某些部分的实现容易出现卡关的情况；因对一些概念不明确，且thinking的提示不太够用，实验报告用时约3h，书写效率偏低。整体感受较差，理解了printf函数的实现过程，理解了readelf的一些操作，但对于bootloader启动、内核、内存等知识还是一知半解。线上实验也有一点点坑，特别在于纯黑盒测试，因此不知道哪里出了问题，无从修改，两小时坐牢都在de同一个bug还de不出来，希望适当增加一些本地测试样例，比如%T实现时候把printf("%T%T",&a,&a)这样对同一个变量的输出加一点本地测试提示，完全没想到课下翻转没问题，课上正负翻转因为是地址操作直接影响结果，太痛苦了……这部分的内容本身就琐碎冗杂，希望能有好的方法梳理一下知识点吧！

4、指导书反馈

- 视频教程14:28左右，提到CROSS_COMPILE缺东西，实则不缺，只是一个路径宏；
- 在 printf 实验中可以适当提示一下已有的封装宏 `IsDigit(x)` 和 `Ctod(x)`；
- 在 thinking 中希望再适当增加一些引导；

5、残留难点

- 线上第一次实验找到bug在发现问题输出后忘了使用break结束，第二次实验实现 `%T` 输出，没注意到因为对地址修改，所以在多次输出同一个 `my_struct` 结构体时会反复修改，不知道修改以后是否正确；
- 对操作系统启动流程的具体细节还不是很清楚，一些步骤很模糊，还有bootload执行的操作内容没有理清；
- 对于汇编语言的记忆也有些模糊了；
- 对于ELF文件的格式还需要继续加深印象；
- 对于内存mmu.h的图也需要加深记忆。