林子杰_20373980_lab1

一、思考题

Thinking 1.1

首先，我们使用的命令格式为：

```
objdump -DS <targetfile> > <outputfile>
```

`-DS`是两个命令选项。`-D`意为反汇编所有section（节）的代码，这个比较好理解；`-S`经过查询资料，其作用是"尽可能反汇编出源代码，尤其是当编译的时候指定了`-g`这种调试参数时"。我觉得这句话很难理解，于是利用Mac本地中断写了一个简单的`helloworld.c`程序来进行命令测试。

```
objdump -D helloworld.c > dis1.txt    //命令1
objdump -DS helloworld.c > dis2.txt   //命令2
diff dis1.txt dis2.txt
```

执行上面的命令，发现反汇编的信息没有不同。但是在终端上，命令2会提示我们" **warning:** 'helloworld.o': failed to parse debug information for helloworld.o"。

于是我猜想，这一选项应当与显示调试信息有关。更深处的探究，目前还没有找到方法。

关于本题目的第二个部分，采用了课程平台的交叉编译器，过程如下。首先便写了一个简单的C文件（此编译器很多语句都不能正常编译）。

```
int main()
{
 int a=100;
 return 0;
}
```

首先使用OS平台的mips交叉编译器进行预处理，得到结果如下：

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "test.c"
int main()
{
 int a=100;
 return 0;
}
```

然后生成`.o`文件并进行反汇编，反汇编内容如下（内容过多，不全部展示）：

```
test.o:     file format elf32-tradbigmips

Disassembly of section .text:

00000000 <main>:
   0:   27bdffe8    addiu   sp,sp,-24
   4:   afbe0010    sw  s8,16(sp)
   8:   03a0f021    move    s8,sp
   c:   24020064    li  v0,100
  10:   afc20008    sw  v0,8(s8)
  14:   00001021    move    v0,zero
  18:   03c0e821    move    sp,s8
  1c:   8fbe0010    lw  s8,16(sp)
  20:   27bd0018    addiu   sp,sp,24
  24:   03e00008    jr  ra
  28:   00000000    nop
  2c:   00000000    nop
Disassembly of section .reginfo:

00000000 <.reginfo>:
   0:   e0000004    sc  zero,4(zero)
    ...
Disassembly of section .pdr:

00000000 <.pdr>:
   0:   00000000    nop
   4:   40000000    mfc0    zero,c0_index
   8:   fffffff8    sdc3    $31,-8(ra)
    ...
  14:   00000018    mult    zero,zero
  18:   0000001e    0x1e
  1c:   0000001f    0x1f
......
```

随后，在编译链接后进行反汇编，部分输出内容如下：

```
isassembly of section .pdr:

00000000 <.pdr>:
   0:   00000000    nop
   4:   40000000    mfc0    zero,c0_index
   8:   fffffff8    sdc3    $31,-8(ra)
    ...
  14:   00000018    mult    zero,zero
  18:   0000001e    0x1e
  1c:   0000001f    0x1f
Disassembly of section .comment:
```

```
00000000 <.comment>:
   0:   00474343    0x474343
   4:   3a202847    xori    zero,s1,0x2847
   8:   4e552920    c3  0x552920
   c:   342e302e    ori t6,at,0x302e
  10:   30202844    andi    zero,at,0x2844
  14:   454e5820    0x454e5820
  18:   454c444b    0x454c444b
  1c:   20342e31    addi    s4,at,11825
  20:   20342e30    addi    s4,at,11824
  24:   2e302900    sltiu   s0,s1,10496
```

Thinking 1.2

根据提示，使用`readelf -h`命令分别对vmlinux文件和用于测试的testELF文件，并且进行对比。

```
readelf -h ./gxemul/vmlinux
readelf -h ./readelf/testELF
```

可以发现，vmlinux的ELF头中记载的文件存储方式为`big endian`，而testELF记载的为`little endian`。我们手动编写的readelf文件只能解析`little endian`。

Thinking 1.3

为什么可以跳转到正确的位置？教程提到过，依赖于CPU的体系结构，我们可以使用bootloader来调用内核。bootloader分为两个阶段：stage1和stage2。内核并不参与这两个启动阶段，但是在这两个阶段我们已经做好了硬件及部分软件的初始化工作；同时，在stage2我们拥有了能够运行C语言的环境，故而我们可以使用C语言进行内核入口的跳转，保证跳转的正确性。

Thinking 1.4

由lab1-1的extra得到启发，我们在加载程序段的时候，可以设计一个C程序进行地址的判断。我们可以通过页的大小来计算前一个程序结束地址所处的页数M和要加载的程序的起始地址所处的页数N。倘若M小于N，则必然不存在冲突，可以直接加载程序；倘若M大于N，则必然冲突，可以为要加载的程序申请M+1页作为起始页；倘若M等于N，就进行地址大小的判断，根据结果决定是否要申请新的页作为加载起始地址。

Thinking 1.5

内核放在kseg0的内存段，查看`mmu.h`文件可知其入口起始地址为`0x80000000`；同样可知道main函数的入口微`0x80010000`。

在`start.S`文件中，我们使用jal指令跳转到main标签从而进入main函数。同样可通过地址跳转跨文件调用函数，使用约定寄存器和栈维护函数的参数与内部值。

Thinking 1.6

想要完成这一道思考题，我们需要查看操作数的具体值。查找`start.S`文件所include的头文件时，我发现了`#include <asm/cp0regdef.h>`这一文件，于是我在学号根目录下使用命令：

```
find -name cp0regdef.h
```

找到了`./include/asm/cp0regdef.h`这一隐藏文件。在里面，可以查询到mips指令使用的宏操作数——查看文件之后不难发现，这些宏都对应了寄存器。

根据上学期计组所学知识，现在可以逐步解析指令含义。

```
mtc0 zero, CP0_STATU
```

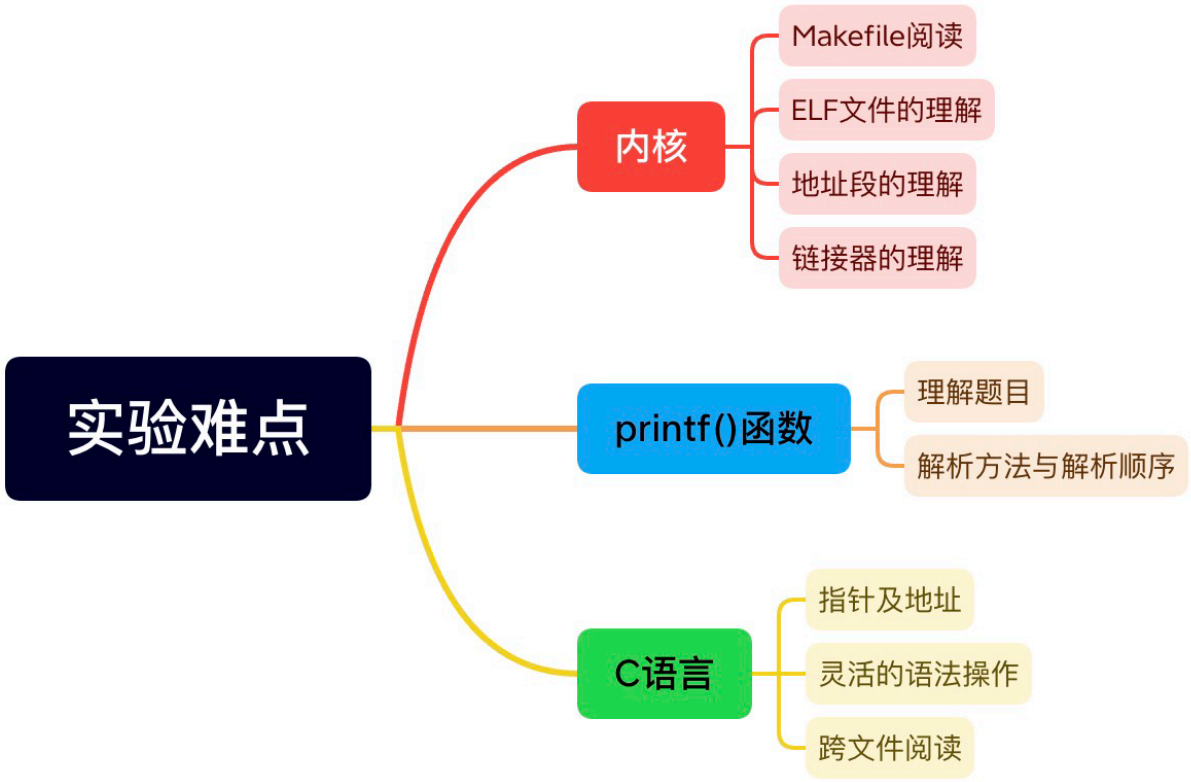
CP0_STATUS为$12寄存器，此指令可将$0寄存器的内容（即0）挪入CP0的$12号寄存器，将CP0置于屏蔽中断状态。

```
mfc0 t0, CP0_CONFIG   //拿取$16寄存器的内容
and t0, ~0x7          //~0x7为要清0位的反码，这里将低三位清零
ori t0, 0x2           //0x2为要置1的位，这里将第二位置1
mtc0 t0, CP0_CONFIG   //将修改后的值写回CP0
```

可见这一批操作将cp0中的config寄存器的0～2位置为010，分别对应config寄存器的G、V、D位。这三位在寄存器中被称为K0域，"用来决定固定的kseg0位是否经过高速缓存"，并决定其确切行为。

二、实验难点

使用思维导图归纳本次试验的难点。



1.内核

Makefile阅读：本次要求操作的Makefile是"顶层Makefile"（子目录下还有很多别的Makefile），对于刚接触Makefile的我来说，其语法的确比较复杂，于是我在完成课下内容前后分别阅读了一遍。其大致功能是让我们可以通过在根目录下执行`make`命令生成内核文件，执行`make clean`可以清除已有的内核文件。其中还进行了一些路径的定义以及跨文件命令的调用。

ELF文件的理解：一开始，我既不知道ELF文件是什么，也不知到exercise让我完成的`readelf.c`文件是用来做什么的。经过几天的查询、讨论和理解，我逐渐明白ELF文件实际上是一种"规范"，能够方便我们进行文件的存储与文件之间的链接。想要理解ELF文件以及完成exercise，仅仅阅读`readelf.c`文件是不够的，还应当去阅读`types.h`、`kerelf.h`等文件中的结构体、宏的定义，并且结合PPT以及ELF手册，才能明白结构体中每个数据的意义和用C语言解析该文件时每一步操作的原理。

地址段的理解：有关地址的`mmu.h`文件存放在`include`目录下，要多多阅读以明确存储空间各个区域的作用。

链接器的理解：Linker_Script是我们小操作系统的链接脚本。做完实验后，我目前只了解其部分功能，但对其作用、具体语法和文件格式还不甚明白。

2.printf()函数

本部分算是这次lab最困难的部分了。首先要做的就是理解题目。实现这一`printf()`函数并不像以往一样，新建一个C文件然后在其中写入熟悉的操作，而是需要我们跨越式地阅读多个文件、利用C语言的一些"高级特性"（指针等）、在陌生的环境中编写一个程序。首先要明确，我们要通过填补`print.c`文件来实现`printf()`函数。通过跨文件阅读、查阅指导书，辅以提示信息和往年代码，大致了解到这里需要我们实现类似于**字符串解析**的过程。需要注意解析的规范、方法和顺序。

3.C语言

理解lab1中所展示的C语言的强大、灵活的功能，是本lab中最具挑战性的部分。由于C语言基础不好，我在阅读代码和完成实验的过程中还是感觉比较吃力的；但操作系统的层次又决定了它地址操作的灵活性、文件调用的多样性和跨越性。对于这一部分，只能多读代码、慢慢适应了；如有余力，可以多多在自己的代码中模仿这些"优雅"的写法。

三、体会与感想

不得不说，lab1的难度相对于lab0提升了不少，在课下各种乱七八糟的学习时间加起来可能有二十多个小时（但其实跨度也比较长~~，主要是来自OO的压迫~~）完成实验后，感觉自己对这一部分知识的理解还是比较笼统，也没有能够很好地把握各个知识点之间的联系。希望后面自己能够多多回顾和探索，达到温故而知新的效果，为以后的实验打好基础。

另外，交叉编译器、gxemul的使用方法的给出都不明确（未告知要提前打一个`/OSLAB/`、交叉编译器的位置没有明确给出），希望以后可以说得明白一些。