lab1 实验报告

exercise 1

1.1

本例需要我们从 elf 文件中找到以下三个参数

1. 节头表的地址：不同节头表地址不一样，它是通过我们传进来的地址参数唯一确定的，因此存在 binary 地址中
2. 节头的表的数量：从 Elf32 结构体中的 e_shnum 中查找
3. 一个节头表的大小：从 Elf32 结构体的 e_shentsize 字段中查找

```
const void *sh_table;
Elf32_Half sh_entry_count;
Elf32_Half sh_entry_size;
/* Exercise 1.1: Your code here. (1/2) */
sh_table = binary + ehdr->e_shoff;//地址
sh_entry_count = ehdr->e_shnum;//数量
sh_entry_size = ehdr->e_shentsize;//大小
```

在找到这三个参数以后，需要我们依次输出每一个节头的地址，我们在上一问中已经得到了节头表的起始地址，那么我们只需要每次跳过**一个节头的大小**这么大的空间，因为节头在内存中连续存放，我们就可以依次得到所有的节头地址，这里我们用到指针以下的性质：

```
Struct st * p = (Struct st*)p + a// <=> p + a*sizeof(Struct)
```

实现代码如下：

```
for (int i = 0; i < sh_entry_count; i++)
{
  const Elf32_Shdr *shdr;
  unsigned int addr;
  shdr = (Elf32_Shdr *)sh_table + i;//jump one sh
  addr = shdr->sh_addr;
  printf("%d:0x%x\n", i, addr);
}
```

换言之，每一个节头的地址信息存放在节头表中，节头表的地址信息又存放在 Elf 头表中，因此我们需要两重访问先得到节头表中第一个节头的起始地址，进而通过累加，得到每一个节头的地址：

```
for (int i = 0; i < sh_entry_count; i++)
{
  const Elf32_Shdr *shdr;
  unsigned int addr;
  shdr = sh_table + i * sh_entry_size;
  addr = (unsigned int)(shdr->sh_addr);
  printf("%d:0x%x\n", i, addr);
}
```

易错点

1. 上述这些参数应该去 elf 头对应的结构体里面去找，而不是去节头表对应的结构体里去找，如果在后者中找到的参数，应该对应的是每个节的地址，节的数量，一个节的大小，需要对二者进行区分。
2. 我们需要通过指针相加的方式，获得节头表的地址，具体为：节头表实际地址 = elf头起始地址 + 节头表相对于 elf 头的偏移量，前者通过 binary 参数传入，后者通过在 elf 头中查询 e_shoff 可以得到，二者相加就是节头表的真实地址。
3. 指针相加可能有个错误如下：

```
//binary、ehdr为ELF头地址；e_shoff为节头表入口偏移
const void *sh_table = (Elf32_Shdr *)(binary + ehdr->e_shoff);//正确
const void *sh_table = (Elf32_Shdr *)(ehdr + ehdr->e_shoff);//错误
```

需要注意的是，e_shoff 这样的偏移量存储的单位都是以字节为单位的，也就是 void* 所占的长度，如果 e_shoff 是 3，那么说明节头表相对于 elf 往后的地址就是 3 个字节，如果用 ehdr 作为其实地址指针，那么加的偏移量就变成了 3 × ehdr 所指向的整个 elf 头的空间大小，就会导致错误

1.2

本例中需要我们完成 Linker_Script 文件，它负责把 Elf 文件中的各个节映射到对应的段，并且把各个段加载到对应的位置

在该文件中，有以下符号：

1. 其中的`.`号用作**定位计数器**，通过设置`.`的地址，声明接下来的节会被按序安放在该地址后。
2. 后面的代码如`.bss:{*(.bss)}`，表示将所有输入文件中的`.bss`节（右边的`.bss`）都放到输出的`.bss`节（左边的`.bss`）中。
3. 在不指明`.`为新的地址的时候，后面所分配的空间会接在前面分配的空间的后面，比如我们写`.text : { *(.text) } .data : { *(.data) }`这样的两条语句，data 节所分配的空间就会紧靠在 text 所分配的空间后面

具体实现代码如下：

```
        /* Step 1: Set the user program entry */
        . = 0x80020000;
        /* Step 2: Define the text section. */
        .text : {
                        *(.text)
                }
        /* Step 3: Define the data section. */
        .data : {
                        *(.data)
                }
        bss_start = .;
        /* Step 4: Define the bss section. */
        .bss : {
                *(.bss)
                }
        bss_end = .;
        . = 0x80400000;
        end = . ;
```

代码解析：其中 0x80020000 是内核代码段的入口位置，我们的内核代码需要从该位置开始加载，所以我们一上来需要把程序起始地址设置在这里，接下来我们依次填充 text，data，bss 这三部分的内容即可，在填好之后，我们需要记录每个段结束的地址为当前`.`中所存储的地址，最后我们需要把整个用户段程序的结束地址设置成为**内核栈**的栈顶位置，表示代码段不能越过内核栈顶进入更高地址的空间中执

行（换言之就是整个 0x80400000 到 0x80010000 最多就全给内核代码段用，它不能再占用此外更多的空间）

1.3

本例需要我们能在 start 文件中设置程序的入口，并且进行程序跳转

首先，我们查看 include 文件下的 mmu.h 可以找到内核区的栈顶位置是 0x80400000，因为在 OS 中，栈是从**高地址**向**地地址**扩展的，因此一开始栈是空的时候，栈指针自然就处在最高位置的内存，所以我们把这个位置的指针标记成 KSTACKTOP，名义上是 TOP，实际上就是栈底，当我们往栈中压入元素的时候，栈指针就会减少，从高地址向低地址扩展

注意：在系统内核区中还存在一个名为 KENRBASE 的位置，对应地址为 0x80020000，这个位置是内核代码段的起点，其中内核代码段和内核栈是两个完全不同的概念，代码段存储的是内核代码中 .text .data .bss 这些内容，且它的存储是从**低地址**向**高地址**扩展的，也就是说内核代码段和内核栈的栈顶是相向而行的，如果有一天二者相遇，就会发生内存冲突进而引起爆栈危险，这里内核代码段所存储的空间既不是堆也不是栈，而是就叫内核代码段的一段存储空间，相当于 C 语言中的代码区与全局区的总和，而内核栈中存储的东西则是在内核程序运行过程中所产生的局部变量，二者共用从 0x80020000 到 0x80400000 的这段地址空间

然后，我们通过 j 或者 jal 指令跳转进入程序主体即可，注意这里 mips_init 这个函数定义在 init.c 文件中，并没有返回值，因此我们只需要 j 而不需要 jal 即可完成

kusge：用户区，ksge1 ksge2：内核区（用户程序和变量无法访问），ksge3 ksge4：高级内核区（内核程序和变量也无法访问）

exercise 2

回调函数与函数指针

基本概念

回调函数（Callback Function） 是一种 **函数指针（Function Pointer）** 作为参数传递给另一个函数的机制，允许调用者在运行时动态决定调用哪个函数。

在 C++ 中，函数指针的典型定义如下：

```
int add(int a, int b)
{
    return a + b;
}

void work(int(*thing)(int, int), int a, int b)
{
    std::cout << thing(a, b) << std::endl;  // 通过函数指针调用函数
}

int main()
{
    work(add, 1, 2);  // 传递 add 作为回调函数
    return 0;
}
```

在 `work` 函数中，参数 `thing` 是一个指向 `int(int, int)` 类型的函数指针，我们可以通过 `thing(a, b)` 直接调用 `add` 函数。

使用 `typedef` 化简

在上面的 `work` 函数定义中，`int(*thing)(int, int)` 过于冗长，因此我们可以使用 `typedef` 来简化代码：

```
typedef int (*thing)(int, int);  // 定义函数指针类型别名

int add(int a, int b)
{
    return a + b;
}

void work(thing add, int a, int b)
{
    std::cout << add(a, b) << std::endl;  // 通过函数指针变量调用函数
}

int main()
{
    work(add, 1, 2);  // 传递 add 作为回调函数
    return 0;
}
```

规范化表述

`thing` 在两种代码中的区别

情况	描述
`int (*thing)(int, int)`	`thing` 是一个**函数指针变量**，它存储了函数 `add` 的地址，可以直接调用 `thing(a, b)`。
`typedef int (*thing)(int, int);`	`thing` 是一个**类型别名**，并不存储任何数据，它仅用于定义变量，例如 `thing fn = add;`，然后 `fn(a, b)` 调用函数。

第一种情况

```
void work(int(*thing)(int, int), int a, int b)
{
    std::cout << thing(a, b) << std::endl;
}
```

- 这里的 `thing` 是一个 **函数指针变量**，它 **可以存储** 某个函数的地址，比如 `add`。

第二种情况

```
typedef int (*thing)(int, int);  // 定义类型别名
void work(thing add, int a, int b)
{
    std::cout << add(a, b) << std::endl;
}
```

- 这里 `thing` **不是变量**，而是 **一个函数指针类型的别名**，相当于 `int(*)(int, int)`。
- `work` 的参数 `add` 是一个函数指针变量，它和 `int (*thing)(int, int)` 的作用相同。

- **区别在于**：`thing` 本身只是一个**类型名**，不能直接用于调用，而 `add` 是函数指针变量，存储了 `add` 函数的地址，可以用于调用。

概念	第一种情况	第二种情况
`thing`	**变量**，存储函数地址	**类型别名**，不存储数据
`add`	变量，存储函数地址	变量，存储函数地址
`thing(a, b)`	可以直接调用	错误，`thing` 只是类型名
`add(a, b)`	可以调用	可以调用

错误示例：

```
std::cout << thing(a, b) << std::endl;  // 错误的：thing 是类型，不能直接调用
```

正确写法：

```
std::cout << add(a, b) << std::endl;  // 正确的：add 是函数指针变量
```

out 函数的解析

在 include 文件下的 print.h 文件中，教程组对指针函数类型 `void (*)(void *data, const char *buf, size_t len)` 进行了简化定义，将其 typedef 成了 fmt_callback_t 这个类型，具体代码如下

```
typedef void (*fmt_callback_t)(void *data, const char *buf, size_t len);
```

而在 kern 文件下的 printk.c 文件中，教程组给出了我们所调用的 out 函数的本体函数，也就是上面这个函数指针所指向的函数，也就是 outputk 函数，具体代码如下

```
void outputk(void *data, const char *buf, size_t len) {
        for (int i = 0; i < len; i++) {
                printcharc(buf[i]);
        }
}
```

这个函数接受三个参数，将 buf 数组中，长度为 len 的内容，调用 printcharc 函数，进行标准输出，printcharc 函数应该是这里最底层的函数之一了，data 参数看似没用，后面可能会输出重定向的参数，用来接收 outputk 函数的输出

代码补全环节

在处理完 out 这个回调函数以后，我们正式进入 vprintfmt 这个函数的解析，解析过程和易错点已经在**函数体**中进行了注释：

具体来讲，我们要解析的就是 printf(“%-ld%s”, 100, “hi”); 这样的东西，fmt 就是 %-ld%s，ap 就是 100 “hi”

```
//入参含义：用来输出的函数 out，待位参数 data，待解析的输出格式 fmt，可变参数列表 ap
void vprintfmt(fmt_callback_t out, void* data, const char* fmt, va_list ap)
{
    char c;       //存字符
```

```
    const char* s;     //存字符串，这里 s 使用来往 out 里传参用的，也可以认为它是 fmt 的一个辈分
    long num;      //存数字

    int width;           //输出的最小宽度
    int long_flag;       //标记是否为 long 类型 %ld
    int neg_flag;        //标记是否是负数，仅针对十进制
    int ladjust;         //标记是否要左对齐 %-
    char padc;           //填充字符，默认为空格

    for (;;) {
        /* 寻找下一个 % 的位置 */
        /* Exercise 1.4: Your code here. (1/8) */
        int length = 0;
        s = fmt;
        for (; *fmt != '\0'; fmt++)
        {
            if (*fmt != '%')
            {
                length++;//计算连续的非 % 字符的数量
            }
            else
            {
                //遇到 % 的时候，先把前面收集好的没有 % 的部分全都输出
                out(data, s, length);
                length = 0;
                fmt++;//跳过 %
                break;
            }
        }
        /* 有可能整个字符串里都没有 %，那么需要一额外的输出，就在这里输出 */
        /* Exercise 1.4: Your code here. (2/8) */
        out(data, s, length);
        /* 检查是否到了字符串的末尾 */
        /* Exercise 1.4: Your code here. (3/8) */
        if (!*fmt)//等价于 *fmt == '\0'
        {
            break;
        }
        /* 找到了一个 % */
        /* Exercise 1.4: Your code here. (4/8) */
        ladjust = 0;         //默认右对齐，如果设成 1 就是左对齐
        padc = ' ';          //默认用空格填充，如果设成 '0' 就是拿 0 填充
        /* 检查输出格式，用来更新上面那两个变量 */
        /* Exercise 1.4: Your code here. (5/8) */
        if (*fmt == '-')          // %-d or %-ld 的格式，说明是左对齐
        {
            ladjust = 1;
            padc = ' ';
            fmt++;
        }
        else if (*fmt == '0')    // %0d or %0ld 的格式，说明右对齐，但是用 0 填充
        {
            ladjust = 0;
            padc = '0';
            fmt++;
        }
        /* 更新对齐宽度 */
```

```
        /* Exercise 1.4: Your code here. (6/8) */
        width = 0;              //对齐宽度
        while ((*fmt >= '0') && (*fmt <= '9'))
        {
            width = width * 10 + (*fmt) - '0';
            fmt++;
        }
        /* 检查是否是 %ld */
        /* Exercise 1.4: Your code here. (7/8) */
        long_flag = 0;          //是否需要输出 long 类型
        while (*fmt == 'l')
        {
            long_flag = 1;
            fmt++;
        }
        /* 确定输出的进制类型，以及十进制是否需要输出负号 */
        neg_flag = 0;           //负数记录
        switch (*fmt) {
        case 'b'://二进制
            if (long_flag) {
                num = va_arg(ap, long int);
            }
            else {
                num = va_arg(ap, int);
            }
            print_num(out, data, num, 2, 0, width, ladjust, padc, 0);
            break;

        case 'd':
        case 'D'://十进制有符号
            if (long_flag) {
                num = va_arg(ap, long int);
            }
            else {
                num = va_arg(ap, int);
            }

            /* 如果是 10 进制，需要考虑负号的情况，注意这里我们需要进行 long 类型和 int 类型的讨论
               因为如果我们不加以区分，那么在取反的时候就会因为类型不兼容而寄掉 */
            /* Exercise 1.4: Your code here. (8/8) */
            neg_flag = num < 0;
            num = neg_flag ? -num : num;//打印的时候，传进来的数字都是正数，正负号用一个额外的 neg_flag 进行标记
            print_num(out, data, num, 10, neg_flag, width, ladjust, padc, 0);
            break;

        case 'o':
        case 'O'://八进制
            if (long_flag) {
                num = va_arg(ap, long int);
            }
            else {
                num = va_arg(ap, int);
            }
            print_num(out, data, num, 8, 0, width, ladjust, padc, 0);
            break;
```

```
116
117         case 'u':
118         case 'U'://十进制无符号
119             if (long_flag) {
120                 num = va_arg(ap, long int);
121             }
122             else {
123                 num = va_arg(ap, int);
124             }
125             print_num(out, data, num, 10, 0, width, ladjust, padc, 0);
126             break;
127 
128         case 'x'://十六进制
129             if (long_flag) {
130                 num = va_arg(ap, long int);
131             }
132             else {
133                 num = va_arg(ap, int);
134             }
135             print_num(out, data, num, 16, 0, width, ladjust, padc, 0);
136             break;
137 
138         case 'X'://十六进制
139             if (long_flag) {
140                 num = va_arg(ap, long int);
141             }
142             else {
143                 num = va_arg(ap, int);
144             }
145             print_num(out, data, num, 16, 0, width, ladjust, padc, 1);
146             break;
147 
148         case 'R':// %R 类型，用来输出 "(<a>,<b>)"，其中 a b 是来自参数列表的两个整
    数，他们的输出格式一致
149             print_char(out, data, '(', 0, 0);// first char is (
150             //get the first num
151             if (long_flag)
152             {
153                 num = va_arg(ap, long int);
154             }
155             else
156             {
157                 num = va_arg(ap, int);
158             }
159             neg_flag = 0;
160             neg_flag = num < 0;
161             num = neg_flag ? -num : num;
162             print_num(out, data, num, 10, neg_flag, width, ladjust, padc,
    0);
163             print_char(out, data, ',', 0, 0);// second char is ,
164             //get the second num
165             if (long_flag)
166             {
167                 num = va_arg(ap, long int);
168             }
169             else
170             {
171                 num = va_arg(ap, int);
```

```
            }
            neg_flag = 0;
            neg_flag = num < 0;
            num = neg_flag ? -num : num;
            print_num(out, data, num, 10, neg_flag, width, ladjust, padc,
0);
            print_char(out, data, ')', 0, 0); // last char is ), remember
to break after it
            break;
        case 'c'://字符
            c = (char)va_arg(ap, int);// int 转 char 有讲究的
            print_char(out, data, c, width, ladjust);
            break;

        case 's'://字符串
            s = (char*)va_arg(ap, char*);
            print_str(out, data, s, width, ladjust);
            break;

        case '\0'://结束!
            fmt--;
            break;
        default:
            /* 输出一个普通字符，比如 ！@ 这种类型的单个字符 */
            out(data, fmt, 1);
        }
        fmt++;
    }
}
```

整型提升

C语言的整型提升：

输出单个字符的时候，对于参数列表取出来的东西，要先按照 int 取出来再强转 char，是因为 C 语言默认的整数提升的缘故，在 char short bool 这些长度小于 int 的类型传参的时候，他们会默认被改成 int 类型，所以我们只能先取 int 类型再强转成为 char 类型，而 char* long double 这些比 int 长的类型则不需要考虑整数提升。

可变参数列表

有关可变参数列表： va_list

`va_list` 相关的四个宏

在 `<stdarg.h>` 中，有四个与可变参数相关的宏：

宏	作用
`va_list`	定义一个用于存储可变参数信息的变量
`va_start(va_list ap, paramN)`	初始化 `ap`，指向可变参数的起始地址（`paramN` 是最后一个固定参数）
`va_arg(va_list ap, type)`	获取 `ap` 指向的参数值，并将 `ap` 指向下一个参数
`va_end(va_list ap)`	结束可变参数访问，清理 `ap`

`va_list` 工作原理

C 语言的参数传递遵循 **栈调用** 规则：

- **固定参数** 先入栈（如 `paramN`）
- **可变参数** 按顺序入栈
- `va_start` 获取 `paramN` 的地址，并推测可变参数的起始地址
- `va_arg` 读取参数后，指针 `ap` 递增到下一个参数

print_num

由于具体代码已经实现，这里只解析算法的实现思路：

1. 建立 buf 数组，把传进来的无符号整型按照 base 进制转化成字符串，如果 neg_flag 有标记则在末尾再添加一个负号
2. 计算这个字符串的实际长度 actualLength，和待输出长度 length 进行比较，如果后者小于前者的话则把后者强制置成前者
3. 因为我们的字符串是倒置的，我们考虑在末尾补上占位符，首先如果是左对齐，那么我们其实什么都不需要做，如果是右对齐，我们要考虑是 0 补位还是空格补位，是否有负号
4. 如果有负号且是 0 补位，则要输出成 -000xxxxx 的形式，如果有负号且是空格补位，则要输出成 -xxx 的形式
5. 最后根据是左对齐还是右对齐确定翻转区间，如果是左对齐则只按照真实长度翻转；如果是右对齐则要按照总输出长度翻转
6. 最后翻转输出即可

思考题

1.1

如果不使用交叉编译，使用`gcc -c`对文件进行编译，对编译而尚未链接的文件进行反汇编，则得到下面的代码

```
git@23371084:~/23371084/tools/readelf (lab1)$ gcc -c hello.c
git@23371084:~/23371084/tools/readelf (lab1)$ objdump -DS hello.o

hello.o:     文件格式 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 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf>
   f:   48 89 c7                mov    %rax,%rdi
  12:   e8 00 00 00 00          call   17 <main+0x17>
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   5d                      pop    %rbp
  1d:   c3                      ret
  ...
```

如果不使用交叉编译，对编译得到的可执行文件直接使用 objdump -DS 命令，则得到下面的代码

```
git@23371084:~/23371084/tools/readelf (lab1)$ gcc hello.c -o hello2
git@23371084:~/23371084/tools/readelf (lab1)$ objdump -DS hello2
```

```
hello2:     文件格式 elf64-x86-64


Disassembly of section .interp:

0000000000000318 <.interp>:
 318:   2f                      (bad)
 319:   6c                      insb   (%dx),%es:(%rdi)
 31a:   69 62 36 34 2f 6c 64    imul   $0x646c2f34,0x36(%rdx),%esp
 321:   2d 6c 69 6e 75          sub    $0x756e696c,%eax
 326:   78 2d                   js     355 <__abi_tag-0x37>
 328:   78 38                   js     362 <__abi_tag-0x2a>
 32a:   36 2d 36 34 2e 73       ss sub $0x732e3436,%eax
 330:   6f                      outsl  %ds:(%rsi),(%dx)
 331:   2e 32 00                cs xor (%rax),%al

Disassembly of section .note.gnu.property:

0000000000000338 <.note.gnu.property>:
 338:   04 00                   add    $0x0,%al
 33a:   00 00                   add    %al,(%rax)
 33c:   20 00                   and    %al,(%rax)
 ...
```

如果使用交叉编译 mips - linux - gnu - gcc hello.c 进行编译链接，并且直接使用 objdump -DS 进行反汇编，会发现系统直接报了下面的错误

```
objdump: can't disassemble for architecture UNKNOWN!
```

这是因为必须使用交叉编译链接对应的反汇编工具才能对上面的代码进行编译，教程平台机所支持的编译链是 mips - linux - gnu - ojjdump

```
git@23371084:~/23371084/tools/readelf (lab1)$ mips-linux-gnu-objdump -DS
hello.o

hello.o:     文件格式 elf32-tradbigmips


Disassembly of section .text:

00000000 <main>:
   0:   27bdffe0        addiu   sp,sp,-32
   4:   afbf001c        sw      ra,28(sp)
   8:   afbe0018        sw      s8,24(sp)
   c:   03a0f025        move    s8,sp
  10:   3c1c0000        lui     gp,0x0
  14:   279c0000        addiu   gp,gp,0
  18:   afbc0010        sw      gp,16(sp)
  1c:   3c020000        lui     v0,0x0
  20:   24440000        addiu   a0,v0,0
  24:   8f820000        lw      v0,0(gp)
  28:   0040c825        move    t9,v0
  2c:   0320f809        jalr    t9
  30:   00000000        nop
  34:   8fdc0010        lw      gp,16(s8)
```

```
  38:   00001025        move    v0,zero
  3c:   03c0e825        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
        ...
```

1.2

使用系统重的 readelf 指令对 readelf 和 hello 两个文件分别进行分析，可以发现两个文件类型的不同，后者是 ELF64 类型（64位），前者是 ELF32 类型（32位），同时两个文件在编译方式上也有所不同，前者的编译方式是`$(CC) $^ -o %@`而后者的编译方式是`$(CC) $^ -o $@ -m32 -static -g`，而参数 -m32 所代表的含义就是编译出来的是 32 位程序，这种程序既可以在 32 位操作系统上运行，也可以在 64 位操作系统上运行，这和我们对于两种文件类型的判定是不矛盾的

1.3

我们的虚拟机已经实现了 OS 加载过程中很复杂的准备工作，我们要做的就是**把内核加载到指定内存位置**，这也就是我们在 exercise1 中的任务，MIPS系统启动时首先接管的是 bootloader，随后 Linker Script 把各个节映射到对应的段上，内核文件也在这时被加载到合适的地址空间中整个 exercise1 就是告诉我们 OS 是怎么把一段内核程序加载到正确位置并且把它运行的，首先在 kernel.lds 文件中，我们把内核程序的 .text .data .bss 段分别加载到内核代码区的正确位置，接着在 start.S 中设置栈指针到栈底位置，并且跳入 C 语言代码的主函数，后面的部分即可交由 C 语言执行

重要函数解析

kernel.lds

```
/*
 设置语言类型为 mips
 */
OUTPUT_ARCH(mips)

/*
 把程序入口指定为 _start
 */
ENTRY(_start)

SECTIONS {
  /* Exercise 3.10: Your code here. */
  // 缺失 TLB 页表的时候，跳到 0x80000000 处理异常
  . = 0x80000000;
  .tlb_miss_entry : {
    *(.text.tlb_miss_entry)
  }
  // 0x80000180 用于存放一般的异常处理程序，如 syscall 等
  . = 0x80000180;
  .exc_gen_entry : {
    *(.text.exc_gen_entry)
  }

  /* Step 1: Set the loading address of the text section to the location
counter ".". */
  /* Exercise 1.2: Your code here. (1/4) */
```

```
    . = 0x80020000;

    /* Step 2: Define the text section. */
    /* Exercise 1.2: Your code here. (2/4) */

    .text : {
      *(.text)
    }

    /* Step 3: Define the data section. */
    /* Exercise 1.2: Your code here. (3/4) */
    .data : {
      *(.data)
    }

    bss_start = .;
    /* Step 4: Define the bss section. */
    /* Exercise 1.2: Your code here. (4/4) */
    .bss  : {
      *(.bss)
    }

    bss_end = .;
    . = 0x80400000;
    end = . ;
}
```

print 有关

实现 printf 函数的大致功能，用到下面三个文件

machine.c：它完成了最底层的输入和输出工作，printcharc 能够从控制台输出单个字符，scancharc 能够从控制台读入单个字符，halt 函数用来终止或重启系统

printk.c：它相当于是中间函数，实现了回调函数 outputk 和 outputbuf，它内部通过 printk 函数解析变长参数列表，进而调用 vprintfmt 函数，实现对输出格式的解析

print.c：实现 vprintfmt 函数，也就是我们要终点补全的内容

上机感想

本次上机 exam 部分速通，但 extra 还是没有做完，主要原因出在以下几点：

1. 对指针的使用不够熟练，比如修改一级指针指向的值可以直接用一级指针形参传参，但是修改一级指针所指向的地址就必须用二级指针形参传参，否则在函数体内修改的变量无法映射回原参，这里出现了很严重的问题，导致在指针修改的时候没有修改成正确的值
2. 对 gdb 调试的方式不够熟练，因为课下没有出现太大的 bug，所以就忽视了对 gdb 调试的练习，在上机面对复杂代码的同时只能靠 print 的方式打印中间输出信息 debug，效率比较低下，耽误了很多时间
3. C 语言好多知识点都出现了遗忘，包括指针，文件，系统栈等知识，需要补一下 C 语言的知识