实　　验　　报　　告

学　　号	21030031009	姓　　名	惠欣宇	专业班级	计算机 4 班
课程名称	**操作系统**			学期	**2023 年秋季学期**
任课教师	窦金凤	完成日期	2023. 12. 15	上机课时间	周五 3-6
实 验 名 称	**内核、Boot 和 printf**				

Exercise 部分：

Exercise 1.1

> Exercise 1.1 请修改 include.mk 文件，使交叉编译器的路径正确。之后执行 make 指令，如果配置一切正确，则会在 gxemul 目录下生成 vmlinux 的内核文件。 ■

CROSS_COMPILE: 此变量设置为 MIPS 架构的交叉编译器的路径。交叉编译器用于为不同于编译器运行所在的目标架构构建程序。

CC: 此变量设置为编译器命令。它使用 CROSS_COMPILE 变量指定 MIPS 架构的交叉编译器（在这种情况下是 mips_4KC-gcc）。

CFLAGS: 这些是在编译过程中使用的编译器标志。它们控制优化级别（-O）、ABI 调用（-mno-abicalls）、使用位置无关代码（-fPIC）等选项。

LD: 此变量设置为链接器命令，使用 MIPS 架构的交叉编译器（mips_4KC-ld）。

这些变量通常稍后在 Makefile 中使用，用于构建和链接项目。特别是 CFLAGS 变量包含了编译器的选项。

修改结果如图 1 所示。

```
1  # Common includes in Makefile
2  #
3  # Copyright (C) 2007 Beihang University
4  # Written By Zhu Like ( zlike@cse.buaa.edu.cn )
5
6
7  CROSS_COMPILE :=  /OSLAB/compiler/usr/bin/mips_4KC-
8  CC            := $(CROSS_COMPILE)gcc
9  CFLAGS        := -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot -Wall -fPIC
10 LD            := $(CROSS_COMPILE)ld
```

图 1　include.mk 修改结果

Exercise 1.2

> Exercise 1.2 阅读./readelf 文件夹中 kerelf.h、readelf.c 以及 main.c 三个文件中的代码，并完成 readelf.c 中缺少的代码，readelf 函数需要输出 elf 文件的所有 section header 的序号和地址信息，对每个 section header，输出格式为:"%d:0x%x\n"，两个标识符分别代表序号和地址。 ■

用于读取和打印 ELF（可执行与可链接格式）二进制文件的节头部信息。函数首先检查文件大小和 ELF 格式是否符合标准。然后，它通过 ELF 文件头部信息定位节头表的起始地址。接着，它循环遍历节头表，并打印每个节的地址信息。最后，函数返回 0。值得注意的是，代码中引用了一个名为 is_elf_format 的函数，用于验证 ELF 文件的格式是否正确。

完成的代码如图 2 所示。

```c
int readelf(u_char *binary, int size)
{
        Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
        int Nr;
        Elf32_Shdr *shdr = NULL;
        u_char *ptr_sh_table = NULL;
        Elf32_Half sh_entry_count;
        Elf32_Half sh_entry_size;
        if (size < 4 || !is_elf_format(binary)) {
                printf("not a standard elf format\n");
                return -1;
        }
        ptr_sh_table =(u_char*)( binary + ehdr -> e_shoff);
        sh_entry_count  = ehdr -> e_shnum;
        sh_entry_size = ehdr -> e_shentsize;

        for( Nr = 0; Nr < sh_entry_count; Nr++ ){
            shdr = (Elf32_Shdr*)(ptr_sh_table + sh_entry_size * Nr);
            printf("%d:0x%x\n",Nr,shdr->sh_addr);
        }
        return 0;
}
```

图 2　readelf.c 修改结果

Exercise 1.3

Exercise 1.3 填写 tools/scse0_3.lds 中空缺的部分，将内核调整到正确的位置上。■

OUTPUT_ARCH(mips): 指定目标体系结构为 MIPS。
ENTRY(_start): 指定程序的入口点为 _start。
SECTIONS: 此部分定义了各个节（sections），并指定它们在可执行文件中的地址。
. = 0x80000080;: 设置当前位置（address）为 0x80000080。
.except_vec3: 定义一个名为 except_vec3 的节，其中包含 .text.exc_vec3 部分的内容。
. = 0x80010000;: 设置当前位置为 0x80010000。
.text: 定义一个名为 text 的节，包含所有 .text 部分的内容。
.bss: 定义一个名为 bss 的节，包含所有 .bss 部分的内容。
.data: 定义一个名为 data 的节，包含所有 .data 部分的内容。
.sdata: 定义一个名为 sdata 的节，包含所有 .sdata 部分的内容。
. = 0x80400000;: 设置当前位置为 0x80400000。
end = .;: 定义了一个名为 end 的标签，其值为当前位置，用于表示可执行文件的结束。
这个链接脚本的目的是为 MIPS 架构的可执行文件定义各个节的布局和位置。
修改结果如图 3 所示。

```
OUTPUT_ARCH(mips)
ENTRY(_start)

SECTIONS
{
    . = 0x80000080;
    .except_vec3 : {
        *(.text.exc_vec3)
    }

    . = 0x80010000;

  /*** exercise 3.13 ***/

  .text : {
        *(.text)
    }

  .bss  : {
   *(.bss)
   }

  .data : {
    *(.data)
    }

  .sdata : {
    *(.sdata)
  }

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

图 3　scse0_3.lds 修改结果

Exercise 1.4

> **Exercise 1.4** 完成 boot/start.S 中空缺的部分。设置栈指针，跳转到 main 函数。使用/OSLAB/gxemul -E testmips -C R3000 -M 64 elf-file 运行 (其中 elf-file 是你编译生成的 vmlinux 文件的路径)。 ■

主要用于处理异常，包含了一些初始化操作，例如禁用中断、禁用监视异常以及设置栈等。程序的入口点是 _start，它调用 main 函数并进入一个无限循环。
修改结果如图 4 所示。

```
74      /* jump to main */
75      jal     main
76
77  loop:
78      j       loop
79      nop
80  END(_start)
```

图 4　start.S 修改结果

Exercise 1.5

> **Exercise 1.5** 阅读相关代码和下面对于函数规格的说明，补全 lib/print.c 中 lp_Print() 函数中缺失的部分来实现字符输出。 ■

用于格式化输出字符串，类似于 C 标准库中的 printf 函数。接受一个输出函数指针 output、一个输出函数的参数 arg、一个格式字符串 fmt 和一个 va_list 类型的可变参数列表 ap。
函数主要通过解析格式字符串 fmt 来格式化输出不同类型的数据。其中包含了一个宏定义 OUTPUT，用于输出字符序列，同时进行错误检查。以下是代码的主要逻辑：
定义了一些局部变量，包括字符缓冲区 buf 和一些用于处理格式化的标志和参数。
进入一个无限循环（for(;;)），在循环中解析格式字符串。
在 Part1 中，通过查找 % 字符，将非格式化部分输出到目标函数中。
在 Part2 中，处理格式化字符串中的不同格式，如 %b、%d、%o、%u、%x、%X、%c、%s 等。
根据格式类型，调用相应的处理函数，如 PrintNum、PrintChar 和 PrintString。
输出结果到目标函数中。解析格式字符串直至遇到字符串结束符 \0。
具体代码如下：

```
void lp_Print(void (*output)(void *, char *, int), void * arg, char *fmt,    va_list ap) {
#define    OUTPUT(arg, s, l)    \
   { if (((l) < 0) || ((l) > LP_MAX_BUF)) { \
           (*output)(arg, (char*)theFatalMsg, sizeof(theFatalMsg)-1); for(;;); \
      } else { \
         (*output)(arg, s, l); \
      } \
   }
      char buf[LP_MAX_BUF];
      char c;
      char *s;
      long int num;
      int longFlag;
      int negFlag;
      int width;
      int prec;
      int ladjust;
      char padc;
      int length;
      for(;;) {
```

```
        /* Part1: your code here */
{
        char * curFmt = fmt;
        while(1){
                if(*curFmt == '\0'){
                        break;
                }else if (*curFmt == '%'){
                        break;
                }
                curFmt ++;
        }
        OUTPUT(arg,fmt,curFmt-fmt);
        fmt = curFmt;
        if(*fmt == '\0'){
                break;
        }
}
fmt++;
ladjust = 0;
if(*fmt == '-'){
        ladjust = 1;
        fmt++;
}
padc = ' ';
if(*fmt == '0'){
        padc = '0';
        fmt++;
}
width = 0;
while ((*fmt >= '0' && *fmt <= '9')){
        width = width * 10 + (*fmt - '0');
        fmt++;
}
if(*fmt == '.'){
        prec = 0;
        fmt++;
        while(*fmt >= '0' && *fmt <= '9'){
                prec = prec * 10 + (*fmt - '0');
                fmt ++;
        }
}else{
        prec = 6;
}
longFlag = 0;
```

```
if(*fmt == 'l'){
        longFlag = 1;
        fmt ++;
}
negFlag = 0;
switch (*fmt) {
case 'b':
        if (longFlag) {
        num = va_arg(ap, long int);
        } else {
        num = va_arg(ap, int);
        }
        length = PrintNum(buf, num, 2, 0, width, ladjust, padc, 0);
        OUTPUT(arg, buf, length);
        break;
 case 'd':
 case 'D':
        if (longFlag) {
        num = va_arg(ap, long int);
        } else {
        num = va_arg(ap, int);
        }
        /*    Part2:
                your code here.
                Refer to other part (case 'b',case 'o' etc.) and func PrintNum to complete this part.
                Think the difference between case 'd' and others. (hint: negFlag).
        */
        if(num < 0){
                num = num * -1;
                negFlag = 1;
        }
        length = PrintNum(buf, num, 10, negFlag, width, ladjust, padc,0);
        OUTPUT(arg, buf, length);
        break;
 case 'o':
 case 'O':
        if (longFlag) {
        num = va_arg(ap, long int);
        } else {
        num = va_arg(ap, int);
        }
        length = PrintNum(buf, num, 8, 0, width, ladjust, padc, 0);
        OUTPUT(arg, buf, length);
        break;
```

```
    case 'u':
    case 'U':
        if (longFlag) {
        num = va_arg(ap, long int);
        } else {
        num = va_arg(ap, int);
        }
        length = PrintNum(buf, num, 10, 0, width, ladjust, padc, 0);
        OUTPUT(arg, buf, length);
        break;
    case 'x':
        if (longFlag) {
        num = va_arg(ap, long int);
        } else {
        num = va_arg(ap, int);
        }
        length = PrintNum(buf, num, 16, 0, width, ladjust, padc, 0);
        OUTPUT(arg, buf, length);
        break;
    case 'X':
        if (longFlag) {
        num = va_arg(ap, long int);
        } else {
        num = va_arg(ap, int);
        }
        length = PrintNum(buf, num, 16, 0, width, ladjust, padc, 1);
        OUTPUT(arg, buf, length);
        break;
    case 'c':
        c = (char)va_arg(ap, int);
        length = PrintChar(buf, c, width, ladjust);
        OUTPUT(arg, buf, length);
        break;
    case 's':
        s = (char*)va_arg(ap, char *);
        length = PrintString(buf, s, width, ladjust);
        OUTPUT(arg, buf, length);
        break;
    case '\0':
        fmt --;
        break;
    default:
        OUTPUT(arg, fmt, 1);
}
```

```
        fmt ++;
        }
        OUTPUT(arg, "\0", 1);
}
```

Thinking 部分：

Thinking 1.1

> **Thinking 1.1** 也许你会发现我们的 readelf 程序是不能解析之前生成的内核文件 (内核文件是可执行文件) 的，而我们之后将要介绍的工具 readelf 则可以解析，这是为什么呢？(提示：尝试使用 readelf -h，观察不同) ■

回答：

readelf 是一个用于解析 ELF（可执行与可链接格式）文件的工具，它能够提供详细的 ELF 文件信息，包括程序头、节头、符号表等。对于内核文件而言，其中包含了操作系统内核的机器码以及其他相关信息。

readelf 能够解析 ELF 文件的不同部分，而你提供的 readelf 函数则是专门用于解析 ELF 文件的节头部信息的一部分。可能的原因如下：

缺少对其他部分的解析: readelf 函数中只对 ELF 文件的节头部信息进行了解析，而 readelf 工具通常能够解析更多的部分，包括程序头、符号表等。内核文件中可能包含其他部分的信息，因此只解析节头部信息可能无法完全理解整个文件的结构。

特定标志或属性未考虑: readelf 工具通常会处理各种特殊情况、标志和属性，以确保能够正确解析各种类型的 ELF 文件。readelf 函数可能未涵盖所有可能的情况，导致无法正确解析某些内核文件。

缺少一些必要的信息: 内核文件可能使用一些特定的标志、属性或特性，而 readelf 函数可能未实现对这些特性的解析，导致解析不完整。

Thinking 1.2

> **Thinking 1.2** main 函数在什么地方？我们又是怎么跨文件调用函数的呢？ ■

内核入口在 0x00000000 处，main 函数在 0x80010000 处。我们在 start.S 文件中编写 mips 汇编代码，设置堆栈后直接跳到 main 函数中。在跨文件调用函数时，每个函数会有一个固定的地址，调用过程为将需要存储的值进行进栈等保护，再用 jal 跳转到相应函数的地址。

本次实验用时 8 小时