Lab1实验报告

> 学号：21371220
>
> 姓名：杨硕

一、思考题

Thinking 1.1

指导书中向`objdump`传入参数`-DS`的含义：

- `-D` 反汇编目标文件的所有section
- `-S` 将源代码与反汇编混合

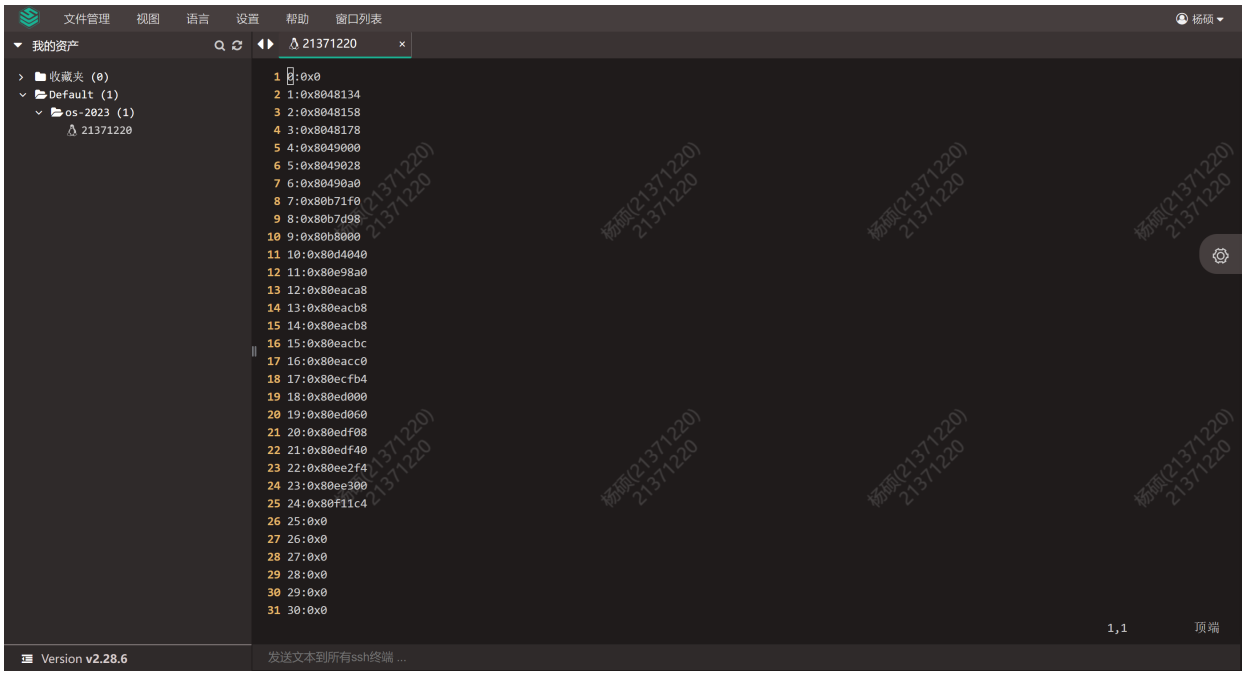
Thinking 1.2

使用编写的`readelf`程序解析之前在 target 目录下生成的内核 ELF 文

件`hello`，命令如下：

```
git@21371220:~ $ cd 21371220
git@21371220:~/21371220 (lab1)$ ls
1.txt  2.txt  copyme  err.txt  hello    include.mk  kern        lib       Makefile  mydir   ray     target  tests
2.ttx  bad.c  csc     gen      include  init        kernel.lds  machine.h  mk       myfile  readme  test.c  tools
git@21371220:~/21371220 (lab1)$ cd tools
git@21371220:~/21371220/tools (lab1)$ ls
include.mk  Makefile  readelf
git@21371220:~/21371220/tools (lab1)$ cd readelf
git@21371220:~/21371220/tools/readelf (lab1)$ ls
elf.h  hello.c  main.c  Makefile  readelf.c
git@21371220:~/21371220/tools/readelf (lab1)$ make
cc -c main.c
cc -c readelf.c
cc main.o readelf.o -o readelf
git@21371220:~/21371220/tools/readelf (lab1)$ ls
elf.h  hello.c  main.c  main.o  Makefile  readelf  readelf.c  readelf.o
git@21371220:~/21371220/tools/readelf (lab1)$ make hello
cc hello.c -o hello -m32 -static -g
git@21371220:~/21371220/tools/readelf (lab1)$ ls
elf.h  hello  hello.c  main.c  main.o  Makefile  readelf  readelf.c  readelf.o
git@21371220:~/21371220/tools/readelf (lab1)$ ./readelf hello > test.txt
git@21371220:~/21371220/tools/readelf (lab1)$ vim test.txt
```

解析结果重定向到`test.txt`中，部分内容如下：



我们编写的`readelf`程序是不能解析`readelf`文件本身的，而我们刚

才介绍的系统工具`readelf`则可以解析，原因如下：

执行`readelf -h hello`：

```
git@21371220:~/21371220/tools/readelf (lab1)$ readelf -h hello
ELF 头：
  Magic:   7f 45 4c 46 01 01 01 03 00 00 00 00 00 00 00 00
  类别:                              ELF32
  数据:                              2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          Intel 80386
  版本:                              0x1
  入口点地址:               0x8049600
  程序头起点:          52 (bytes into file)
  Start of section headers:          746252 (bytes into file)
  标志:             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         35
  Section header string table index: 34
```

执行`readelf -h readelf`：

```
git@21371220:~/21371220/tools/readelf (lab1)$ readelf -h readelf
ELF 头:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF64
  数据:                              2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              DYN (Position-Independent Executable file)
  系统架构:                          Advanced Micro Devices X86-64
  版本:                              0x1
  入口点地址:               0x1180
  程序头起点:          64 (bytes into file)
  Start of section headers:          14488 (bytes into file)
  标志:             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

通过解析后对比，两者的类别不一样，`hello`为`ELF32`，而`readelf`为`ELF64`，即两者位数不同，前者是32位，后者是64位。下面的系统架构也表明了位数不同，`hello`的系统架构是`Intel 80386`架构（实际上就是32位的`x86`架构），而`readelf`的系统架构为`Advanced Micro Devices X86-64`，也就是64位`x86`架构。而阅读我们编写的`readelf`程序，便可发现其只支持解析32位ELF格式的文件，故无法解析`readelf`文件

Thinking 1.3

因为我们的实验中，`GXemul`支持加载 ELF 格式内核，所以启动流程被简化为加载内核到内存，之后跳转到内核的入口，启动就完成了。也就是`GXemul`仿真器通过我们编写的`Linker Script`语句，将顶层`Makefile`链接生成的内核调整加载到正确位置，然后`kernel.lds`文件中的`ENTRY(_start)`将程序入口设置为`_start`，之后内核通过`start.S`的`_start`函数完成CPU初始化和栈指针`sp`初始化，跳转到`main`函数。

所以是因为是`GXemnul`仿真器简化了启动过程，才能保证内核入口被正确跳转到。

二、难点分析

Exercise 1.4

在编写`vprintfmt()`函数的过程中，第一步，寻找`%`和判断字符串结尾`\0`，开始代码如下：

```
if(*fmt != '%'){
    out(data, fmt, 1);
    fmt++;
    continue;
}
if(*fmt == '\0'){
    out(data, fmt ,1);
    break;
}
```

这样的写法导致在判断字符是否为`%`时，遇到字符串结尾`\0`也会输出跳过，继续判断输出，导致多输出（未考虑`\0`后仍有字符）调整顺序后解决：

```
if(*fmt == '\0'){
    out(data, fmt ,1);
    break;
}
if(*fmt != '%'){
    out(data, fmt, 1);
    fmt++;
    continue;
}
```

后面忘记在循环内重复初始化变量，导致在循环判断输出的过程中，这些变量的默认值会发生变化，导致输出错误，补充如下语句后解决：

```
ladjust = 0;
long_flag = 0;
padc = ' ';
width = 0;
```

三、实验体会

`lab1`实验难度比`lab0`明显增加，指导书内容也更为复杂繁多，在完成实验时，曾多次产生畏难的想法，但调整心态，认真研读指导书后，原本毫无头绪的问题，突然就有了解题的思路。事实证明，反复阅读指导书，尤其是自己不懂的部分，大有帮助。

此外，与同学的交流也十分重要，很多自己一直苦思却无法找到原因的问题，在与同学交流后，从他人的角度，换种思路，症结便可迅速找出。因此，在以后的实验中，也要注重与他人交流，或是多看看评论区大佬的答疑解惑，必获益良多。