实验 2b：命令解释器完全开发

实验目的：深入理解系统调用的实现方法，掌握内核程序开发的一般方法，深入掌握系统调用的使用方法，掌握可执行文件的结构，掌握高级语言源程序到可执行文件的转换过程。

实验内容：请基于版本 0 内核完成下列任务。

1) 创建版本 3 内核：打乱系统调用号，使所有的系统调用具有与原来不同的系统调用号，并启动自己的命令解释器。

2) 改写以前开发的命令解释器，使其不链接任何现有库，直接生成可在版本 3 内核上正常工作的可执行文件。

3) 改写以前开发的程序 mycat，同样要求不链接任何现有库，且功能不变。

4) 直接修改外部命令可执行文件 myls，使其可在版本 3 内核上正常工作。

5) 改写以前开发的程序 myls，同样要求不链接任何现有库，且功能不变。

要求提交实验报告和所有源码，实验报告应记录实验过程和主要画面。

实验步骤：可按如下步骤实施。

(1) 在版本 3 内核中打乱系统调用号并使用自己的命令解释器。

(2) 改写命令解释器，不使用库，直接生成可用的可执行文件。

a) 找出命令解释器 mysh 的可执行文件中调用的所有系统调用，方法：

- 生成带调试信息的命令解释器 mysh。在 bochs 虚拟机中编译 mysh.c，并给 gcc 加上”-g”选项，然后将生成的可执行文件 mysh 拷贝到 ubuntu 下。
- 反汇编可执行文件（mysh），使用如下命令：
  objdump -dlx mysh
  可以看到该可执行文件的所有汇编指令，分析其中的调用关系。可以使用重定向将上述命令的输出写入一个文件。
- 分析陷入指令，统计系统调用。在上面的反汇编指令查找陷入指令“int 0x80”，每一个陷入指令都会引发一个系统调用，系统调用号用 eax 寄存器传递，系统调用号与系统调用名之间的对应关系在硬盘镜像文件的“/usr/include/unistd.h”文件中有记录。

b) 对每一个系统调用，重新定义其接口函数[1]（如 execve），放到自己的源程序 mylib.c 中，具体方法类似于内核对接口函数 waitpid 的定义（在 wait.c 中）。其间需要使用 _syscall3 等宏，依函数参数个数的不同，需要使用不同的宏，可参考提供的 mylib.c 文件中的代码。参数个数可变的接口函数（如 ioctl）可参考内核代码中的 open 函数来定义，以处理...参数。

c) 如果存在命令解释器用到的其它 C 函数，尽量将其用系统调用替换（如用系统调用

[1] 系统调用接口函数的作用是将系统调用包装成普通的函数调用，一般放在标准库（如 libc.a）中。

read 和 write 分别替换 gets 和 printf）。如果无法替换，则从 glibc 源码库中提取出其源代码（可能会很复杂，可作简化），放入 mylib.c。

d) 将命令解释器源码和 mylib.c 放在一起重新编译（使用提供的 Makefile 参考文件），生成命令解释器的可执行文件。需要保证硬盘镜像文件中的文件“/usr/include/unistd.h”与内核源码中的对应文件一致。

e) 通过反汇编，静态分析命令解释器的可执行文件，确认所有的系统调用号都已正确生成。

f) 测试命令解释器，确保其正常运行。

(3) 改写应用程序 mycat。其过程类似于步骤(2)。

(4) 处理外部命令 myls。为了在命令解释器中运行外部命令 myls，需要对该命令对应的可执行文件进行修改。

a) 首先找出可执行文件 myls 中的所有系统调用，方法类似于步骤 2.a。

b) 修改外部命令 myls 的可执行文件，方法:

- 用 ultraedit 等二进制编辑器打开该可执行文件，找到对应的指令，直接修改系统调用号，使其与新内核中的编号相一致。
- 用 objdump 反汇编命令解释器的可执行文件，验证上述修改是否正确。
- 使用版本 3 内核启动系统执行 myls，进行测试。

(5) 改写应用程序 myls。其过程类似于步骤(2)。

常见问题

1. 反汇编时找不到 int 0x80 前面的系统调用号怎办？
一般是由于反汇编不准确造成的，可能是前面有多余的填充 0，场景如下：

```
3445:   5b                      pop    %ebx
3446:   c3                      ret
3447:   00 53 b8                add    %dl,-0x48(%ebx)
344a:   12 00                   adc    (%eax),%al
344c:   00 00                   add    %al,(%eax)
344e:   8b 5c 24 08             mov    0x8(%esp),%ebx
3452:   8b 4c 24 0c             mov    0xc(%esp),%ecx
3456:   cd 80                   int    $0x80
3458:   85 c0                   test   %eax,%eax
345a:   7d 0c                   jge    0x3468
```

此时看不到对 eax 的赋值，原因是上面地址 0x3447 处的 00 不是有效指令，应该跳过，从 0x3448 处开始反汇编，“b8 12 00 00 00”是一个有效机器码，此时的系统调用号应该是 0x12。正常情况如下：

```
3806:       00 00                   add    %al,(%eax)
3808:       53                      push   %ebx
3809:       b8 0d 00 00 00          mov    $0xd,%eax
380e:       8b 5c 24 08             mov    0x8(%esp),%ebx
3812:       cd 80                   int    $0x80
3814:       85 c0                   test   %eax,%eax
```

可用类似下面的指令解决：

objdump　-dlx　--start-address=0x3448　|　less

其中 0x3448 是忽略了 0 之后的有效代码的起始位置。

2. 反汇编时有些 int 0x80 找不到，怎么办？
   有一些陷入指令可能在 objdump 时看不到，原因是反汇编时机器码切分错误，可以直接在十六进制编辑器里先搜 cd 80，再往回搜 b8 ** 00 00 00

```
4946    3995:    00 00                   add    %al,(%eax)
4947    3997:    00 53 b8                add    %dl,-0x48(%ebx)
4948    399a:    06                      push   %es
4949    399b:    00 00                   add    %al,(%eax)
4950    399d:    00 8b 5c 24 08 cd       add    %cl,-0x32f7dba4(%ebx)
4951    39a3:    80 85 c0 7d 0c f7 d8    addb   $0xd8,-0x8f38240(%ebp)
4952    39aa:    a3 d4 6f 00 00          mov    %eax,0x6fd4
4953    39af:    b8 ff ff ff ff          mov    $0xffffffff,%eax
4954    39b4:    5b                      pop    %ebx
```

```
B 10 00 C7 05 D4 6F 00 00 0C 00 00 00 B8 FF ; 须..?
F FF 5B C3 00 00 00 53 B8 06 00 00 00 8B 5C ;
8 CD 80 85 C0 7D 0C F7 D8 A3 D4 6F 00 00 B8 ; $.蜈呷
F FF FF 5B C3 00 00 53 B8 1B 00 00 00 8B 5C ;
```