实验二：

添加Linux系统调用及熟悉常见系统调用

实验目的

- 了解系统调用的基本过程
- 学习如何添加Linux系统调用
- 熟悉Linux下常见的系统调用

实验环境

- OS：Ubuntu 18.04
- Linux内核版本：Kernel 0.11
- **注意**：本次实验是以实验一为基础，如有不熟悉的步骤，请重新浏览实验一文档。
- **请大家留意加粗的内容**

提交要求

1、本次实验要求提交**实验录屏、实验报告和代码**

- **按下面描述的方式组织相关文件（具体的实验报告和录屏的要求见实验内容部分）**

```
- 顶层目录（可自行命令，如EXP2）
  - EXP2.1 子目录1
    - linux (Linux 0.11 修改源码文件夹，名字为linux)
      - kernal 文件夹
      - include 文件夹
    - 测试程序(test.c)
    - HDC中usr/include/unistd.h 文件
    - 2.1实验录屏(2分钟之内)
  - EXP2.2 子目录2
    - lab2_shell.c
    - 2.2实验录屏(4分钟之内)
  - 实验报告(学号+姓名+实验二命名，提交PDF版本)
```

- **将上述文件压缩**
  - 格式为 .7z/.rar/.zip
  - 命名格式为 学号_姓名_实验2，如果上传后需要修改，由于ftp服务器关闭了覆盖写入功能，需要将文件重新命名为学号_姓名_实验2_修改n (n为修改版本)，以最后修改版本为准。
    - 如PB10001000_张三_实验2.zip ，PB10001000_张三_实验2_修改3.zip

2、提交到ftp服务器

- **服务器地址：** ftp://OS2020:OperatingSystem2020@nas.colins110.cn:2001/
- **上传至文件夹: 第二次实验**

- **实验截止日期：2020-04-26 23:59**

实验内容

一、添加Linux系统调用

目标：了解Linux是如何实现系统调用功能，并在Linux 0.11中添加两个系统调用

!!!注意: 请大家先学习完 "背景知识学习" 这一小节(在本文档的后面)

!!!如果有问题请先参阅本部分第六点

1、分配系统调用号，修改系统调用表

- 我们需要增加两个系统调用，分别称为print_val和str2num。对应的函数与如下形式 .

```
  void print_val(int a);    //通过printk在控制台打印如下信息（假设a是1234）：
  // in sys_print_val: 1234
  int str2num(char *str, int str_len, long *ret);    //将一个有str_len个数字的
字符串str转换成十进制数字，然后将结果写到ret指向的地址中，其中数字大小要合适，应当小于
100000000(1*e^8).
```

需要根据背景知识学习中的内容, 完成两个系统调用函数的添加.

添加的时候需要注意

- 修改system_call.s 文件中的系统调用数量(nr_system_call参数)
- 增加系统调用编号时, 需要在下面同步增加系统调用原型函数.
- 修改函数指针表时(sys_xxx), 需要按上面的格式增加函数原型, 如:

```
  extern int sys_setreuid();
  extern int sys_setregid();
  // 这里添加函数原型

  fn_ptr sys_call_table[] = { ... }
```

- 需要**进入Linux 0.11文件系统**(方法见实验一), 修改其中usr/include/unistd.h文件, 其格式和Linux源码中的unistd.h文件相同, 修改方式也类似(这里建议不要直接拷贝linux源码中的unistd.h文件).

2、实现系统调用函数

需要新建一个文件kernel/xxx.c, 在其中完成两个系统调用的具体实现.

注意第二个系统调用需要进行内核和用户空间的信息交换, 具体的解释已经在文后的背景知识学习的部分给出. 这里我们写回的数据结构是long, 需要用到put_fs_long宏.

修改Makefile

make是一个自动构建程序的程序. 一个工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作. make则读取Makefile中的内容去完成编译工作. 大家如果对makefile感兴趣可以参阅[1] [2], 或者以Makefile为关键词自行搜索.

我们这次需要修改kernel目录下的Makefile文件. 需要修改两处
第一处为(xxx为文件名, 下同)

```
OBJS  = sched.o system_call.o traps.o asm.o fork.o \
    panic.o printk.o vsprintf.o sys.o exit.o \
    signal.o mktime.o

改为

OBJS  = sched.o system_call.o traps.o asm.o fork.o \
    panic.o printk.o vsprintf.o sys.o exit.o \
    signal.o mktime.o xxx.o
```

第二处为文件最后, 需要新增

```
xxx.s xxx.o: xxx.c  ../include/asm/segment.h
```

这里做一个解释, 后面跟着的那个路径(../include/asm/segment.h), 是我们在编写xxx.c时需要include的文件, **如果我们修改了include的文件(如增加), 需要在makefile文件中做对应的修改**.

3、编译内核

- 执行如下命令编译内核

  ```
  make clean
  make
  ```

  其中make clean 是清除编译的中间结果, make是执行编译的命令.
  然后按实验一的方式运行.

4、编写测试程序

- 创建一个简单的用户程序（文件名为test.c）验证已实现的系统调用正确性，要求能够从终端读取一串数字字符串，通过str2num系统调用将其转换成数字，然后通过print_val系统调用打印该数字。为避免混乱，执行用户程序后需要有如下输出：

  ```
          Give me a string:
          78234
          in sys_print_val: 78234
  ```

  一个标准测试程序应该为(下面只是一个参考).

```
/* 有它，_syscallx等才有效。详见unistd.h */
#define __LIBRARY__

#include <unistd.h> /* 有它，编译器才能获知自定义的系统调用的编号 */

_syscall1(int, print_val,int, a); /* print_val()在用户空间的接口函数, 下同 */
_syscall3(int, str2num,char*,str,int,str_len, long*, ret);

int main() {
    ...
    return 0;
}
```

5、运行测试程序

进入Linux 0.11系统内, 使用GCC静态编译用户程序

```
gcc test.c -o test
```

然后执行程序:

```
./test
```

6、常见问题

- 如果编译出错, 首先执行 make clean, 然后执行make操作编译
- 如果出现系统执行错误, 很可能是**hdc磁盘文件损坏**, 请重新下载hdc image文件

7、提交要求

- 本部分实验报告要求:
  - 大概描述实验过程
  - 展示实验结果
  - **回答问题**
    - 简要描述如何在Linux 0.11添加一个系统调用
    - 系统是如何通过系统调用号索引到具体的调用函数的?
    - 在Linux 0.11中, 系统调用最多支持几个参数? 有什么方法可以超过这个限制吗?
- 本部分代码要求
  - 代码要求提交**include, kernel** 这两个文件夹, hdc中的usr/include/unistd.h, 测试代码(test.c), 格式见上.
- 本部分录屏要求
  - 需要一边**展示修改部分**一边**口述大概的添加流程**. 请尽量控制在2分钟内.

二、熟悉Linux下常见的系统调用函数

目标：利用Linux提供的系统调用，实现一个简单shell程序

1、熟悉系统调用函数的用法

```
//本实验中可用到的系统调用:
int read(int fildes, char * buf, off_t count); //从fildes对应的文件中读取count个字符到buf内（fildes为文件描述符）
int write(int fildes, const char * buf, off_t count); //从buf写count个字符到fildes对应的文件中
pid_t fork();   //创建进程
pid_t waitpid(pid_t pid,int* status,int options);   //等待指定pid的子进程结束
int execl(const char *path, const char *arg, ...);  //根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新程序的内容替换了

//Linux 0.11中未提供，本次实验中需要我们实现的三个函数
int system(const char* command);    //调用fork()产生子进程，在子进程执行参数command字符串所代表的命令，此命令**执行完后**随即返回原调用的进程
FILE* popen(const char* command, const char* mode); //popen函数先执行fork，然后调用exec以执行command，并且根据mode的值（"r"或"w"）返回一个指向子进程的stdout或指向stdin的文件指针
int pclose(FILE* stream);   //关闭标准I/O流，等待命令执行结束，与popen对应
```

2、利用上面的函数实现一个简单的shell程序

- **实现如下功能的shell：**

  （1）**支持子命令：**每行命令可能由若干个子命令组成，如"ls -l; cat 1.txt; ps -a"，由“;”分隔每个子命令，需要按照顺序依次执行这些子命令。

  （2）**支持管道：**每个子命令可能包含一个管道符号“|”，如"cat 1.txt | grep abcd"（这个命令的作用是打印出1.txt中包含abcd字符串的行），作用是先执行前一个命令，并将其标准输出传递给下一个命令，作为它的标准输入。（不熟悉管道符作用的同学，可以先在Ubuntu自带的终端中测试该命令，也可搜索“管道符”的相关资料帮助理解）

  （3）最终的shell程序应该有类似如下的输出：

```
OSLab2->echo abcd;date;uname -r
abcd
Mon Apr 8 09:35:57 CST 2019
2.6.26
OSLab2->ls | grep 1
gcclib140
OSLab2->
```

- 为了将实验重点放在对于系统机制和系统调用的理解与使用，程序的部分代码已在文件附件lab2_shell.c中给出，其中包括：
  - 对输入按";"做划分的parseCmd()
  - 对输入按"|"做划分的main函数部分
  - popen、system两个函数的大致框架(这里命名为os_popen和os_system，os_pclose已给出完整代码)
    - 注：为了简化实验难度，实验中实现的os_popen返回值为int类型的文件描述符(file descriptor)，可直接供系统调用read、write使用
  - shell程序的大致框架
  - 注：在给出的代码框架中，只**在标明序号的地方添加内容**就可以完成实验。其他部分代码的修改也是可以的，但没必要；全部代码自己写也可以，只要能够**满足上述功能要求和下述实验内容。**
- 所以，在已给出代码的基础上，本部分要做的实验内容为
  - **实现一般的单条命令执行**，可选实现os_system()并在main函数中引用，或者直接在main函数中实现执行单条命令的过程
  - **理解pipe系统调用和管道通信**，将执行指令的子进程的标准I/O通过建立管道传输到父进程中并返回给调用者，实现os_popen函数
    - 提示：STDOUT_FILENO/STDIN_FILENO为标准I/O的文件描述符，pipe[0]和pipe[1]也是文件描述符，都可作为系统调用read、write的参数使用
  - 通过调用os_popen，实现shell的管道功能，即先执行"|"前的命令，获取其标准输出，并在执行"|"后的命令时作为标准输入
  - **在Linux0.11下编译通过**并可以满足前述要求

3、Linux0.11下的编译与运行

- 建议在主机（如Ubuntu虚拟机）中编译运行测试后再将源代码复制到qemu Linux0.11虚拟机中（使用Lab1第三部分给出的方法）进行编译和运行。

  注：在主机上测试通过可以得到最高80%的分数，Linux0.11下测试通过可拿到最高全部分数。

- Linux 0.11下使用gcc版本为1.4，在使用0.11中gcc时应注意以下几点

1. 不支持" //comments "格式的注释， 如有注释请使用" /* comments */ "
2. 对于变量的定义尽可能放在每个代码段的开头，代码中间加入变量定义会出现undefined问题
3. 编译命令相同，使用

   ```
   $ gcc shell.c -o shell
   ```

   即可对"shell.c"文件编译链接， 在当前目录下生成可执行文件"shell"，

   ```
   $ ./shell
   ```

   运行此可执行文件
4. gcc编译过程中的warning可以暂时忽略，若出现error信息则需要留意(代码中存在错误)

4、提交要求

- 本部分实验报告要求:
  - 大概描述代码添加过程、思路
  - 展示实验结果
- 本部分代码要求
  - 提交可以实现要求的源代码lab2_shell.c
- 本部分录屏要求
  - 需要一边**展示添加和修改的代码**一边**口述大概的修改思路**，然后**在Linux 0.11下展示**测试结果。请尽量控制在4分钟内。

背景知识学习

写在前面: 本部分内容主要参考了赵炯老师的<Linux内核完全注释> 和 哈尔滨工业大学的操作系统实验.

大家有兴趣也可以去看一看这些资料.

首先我们回顾一下操作系统是如何实现系统调用的:

1. 应用程序调用库函数（API）；
2. API将系统调用号存入EAX，然后通过中断调用使系统进入内核态；
3. 内核中的中断处理函数根据系统调用号，调用对应的内核函数（系统调用）；
4. 系统调用完成相应功能，将返回值存入EAX，返回到中断处理函数；
5. 中断处理函数返回到API中；
6. API将EAX返回给应用程序。

应用程序如何调用系统调用

在通常情况下，调用系统调用和调用一个普通的自定义函数在代码上并没有什么区别，但调用后发生的事情有很大不同。调用自定义函数是通过 ***call*** 指令直接跳转到该函数的地址，继续运行。而调用系统调用，是调用系统库中为该系统调用编写的一个接口函数，叫API（Application Programming Interface）。API并不能完成系统调用的真正功能，它要做的是去调用真正的系统调用，过程是:

1. 把系统调用的编号存入EAX
2. 把函数参数存入其它通用寄存器
3. 触发0x80号中断（int 0x80）

0.11的lib目录下有一些已经实现的API。Linus编写它们的原因是在内核加载完毕后，会切换到用户模式下，做一些初始化工作，然后启动shell。而用户模式下的很多工作需要依赖一些系统调用才能完成，因此在内核中实现了这些系统调用的API。我们不妨看看最常用的write系统调用.

```
lib/write.c
/*
 *  linux/lib/write.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>

_syscall3(int,write,int,fd,const char *,buf,off_t,count)
```

*syscall3是一个宏，定义在include/unistd.h中，可以发现其中还有多个相似的*syscallx宏，x表示可传入的参数个数

```
include/unistd.h
#define _syscall3(type,name,atype,a,btype,b,ctype,c) \
type name(atype a,btype b,ctype c) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
if (__res>=0) \
    return (type) __res; \
errno=-__res; \
return -1; \
}
```

展开后可以看到是将传入宏的参数替换成了一个函数定义, 它先将宏**NR_write存入EAX，将参数fd存入EBX，然后进行0x80中断调用。调用返回后，从EAX取出返回值，存入**res，再通过对**res的判断决定传给API的调用者什么样的返回值。其中**NR_write就是系统调用的编号，在include/unistd.h中定义, **在编号后面还定义了系统调用的函数原型**:

```
include/unistd.h
#ifdef __LIBRARY__

...
define __NR_write   4 // 系统调用编号
...
int write(int fildes, const char * buf, off_t count); // 系统调用原型
...
```

"int80" 做了什么?

在内核初始化时，主函数（在init/main.c中)调用了sched_init()初始化函数:

```
init/main
void main(void)
{
    ……
    time_init();
    sched_init();
    buffer_init(buffer_memory_end);
    ……
}
```

而sched_init()的定义为:

```
kernel/sched.c
void sched_init(void)
{
    ……
    set_system_gate(0x80,&system_call);
}
```

set_system_gate是个宏，在include/asm/system.h中定义为:

```
include/asm/system.h
#define set_system_gate(n,addr) \
    _set_gate(&idt[n],15,3,addr)
```

_set_gate的定义是:

```
include/asm/system.h
#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ ("movw %%dx,%%ax\n\t" \
    "movw %0,%%dx\n\t" \
    "movl %%eax,%1\n\t" \
    "movl %%edx,%2" \
    : \
    : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
    "o" (*((char *) (gate_addr))), \
    "o" (*(4+(char *) (gate_addr))), \
    "d" ((char *) (addr)),"a" (0x00080000))
```

这段的主要方法就是填写IDT（中断描述符表），将system_call函数地址写到0x80对应的中断描述符中，也就是在中断0x80发生后，自动调用函数system_call(具体过程感兴趣可以参考<Linux内核0.11完全注释>, 第四章相关内容)

我们接着看system_call函数, 这是一个纯汇编函数:

```
kernel/system_calls.s
……
nr_system_calls = 72        # 这是系统调用总数。如果增删了系统调用，必须做相应修改(请特别注意这里)
……
.globl system_call
.align 2
system_call:
    cmpl $nr_system_calls-1,%eax # 检查系统调用编号是否在合法范围内
    ja bad_sys_call
```

```
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx
    pushl %ebx        # push %ebx,%ecx,%edx，是传递给系统调用的参数
    movl $0x10,%edx         # 让ds,es指向GDT，内核地址空间
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx         # 让fs指向LDT，用户地址空间
    mov %dx,%fs
    call sys_call_table(,%eax,4) # 这句是关键
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)
    jne reschedule
    cmpl $0,counter(%eax)
    je reschedule
```

还记得我们之前把系统调用号保存在EAX寄存器中吗? 这里我们看call sys_call_table(,%eax,4) 这一句. 这句的大概意思是在sys_call_table中去找%eax寄存器对应的系统调用函数. sys_call_table定义如下:

```
include/linux/sys.h

fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, ...}
```

我们可以看到sys_write就是这个列表的第五个(下标为之前保存在EAX寄存器中的write系统调用的系统调用号). 我们可以得知, write系统调用实际的执行者是sys_write函数, 这个函数在fs/read_write.c中, 感兴趣大家可以自己去看看. **我们在添加系统调用的时候, 也需要在这里给出系统调用函数指针, 同时在上面参照格式写出原型**

我们最后对write系统调用做一个总结:

1. 调用内核函数write()
2. 获取系统调用索引号, 并调用int 80触发中断(include/unistd.h)
3. 中断处理函数system_call把参数入栈(kernel/system_call.s)
4. 根据索引在sys_call_table中查找函数地址(include/linux/sys.h)
5. 执行真正的系统调用函数sys_write() (fs/read_write.c)

用printk()调试内核

实验环境提供了基于C语言和汇编语言的两种调试手段。除此之外，适当地向屏幕输出一些程序运行状态的信息，也是一种很高效、便捷的调试方法，有时甚至是唯一的方法，被称为“printf法”。

要知道到，printf()是一个只能在用户模式下执行的函数，而系统调用是在内核模式中运行，所以printf()不可用，要用printk()。它和printf的接口和功能基本相同，只是代码上有一点点不同。

本次实验我们可以将其当作内核中的printf实验, 其和printf的使用方法基本相同.

在用户态和核心态之间传递数据

指针参数传递的是应用程序所在地址空间的逻辑地址，在内核中如果直接访问这个地址，访问到的是内核空间中的数据，不会是用户空间的。所以这里还需要一点儿特殊工作，才能在内核中从用户空间得到数据。

我们以open()系统调用为例, 看看如何在内核空间读取用户空间的数据。

```
int open(const char * filename, int flag, ...)
{
    ……
    __asm__("int $0x80"
            :"=a" (res)
            :"0" (__NR_open),"b" (filename),"c" (flag),
            "d" (va_arg(arg,int)));
    ……
}
```

可以看出，系统调用是用eax、ebx、ecx、edx寄存器来传递参数的。其中eax传递了系统调用号，而ebx、ecx、edx是用来传递函数的参数的，其中ebx对应第一个参数，ecx对应第二个参数，依此类推。如open所传递的文件名指针是由ebx传递的，也即进入内核后，通过ebx取出文件名字符串。open的ebx指向的数据在用户空间，而当前执行的是内核空间的代码，如何在用户态和核心态之间传递数据？接下来我们继续看看open的处理

```
system_call: //所有的系统调用都从system_call开始
    ……
    pushl %edx
    pushl %ecx
    pushl %ebx                  # push %ebx,%ecx,%edx，这是传递给系统调用的参数
    movl $0x10,%edx              # 让ds,es指向GDT，指向核心地址空间
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx              # 让fs指向的是LDT，指向用户地址空间
    mov %dx,%fs
    call sys_call_table(,%eax,4)    # 即call sys_open
```

由上面的代码可以看出，获取用户地址空间（用户数据段）中的数据依靠的就是段寄存器fs，下面该转到sys_open执行了，在fs/open.c文件中：

```
int sys_open(const char * filename,int flag,int mode)  //filename这些参数从哪里来？
/*是否记得上面的pushl %edx,    pushl %ecx,    pushl %ebx？
  实际上一个C语言函数调用另一个C语言函数时，编译时就是将要
  传递的参数压入栈中（第一个参数最后压，…），然后call …，
  所以汇编程序调用C函数时，需要自己编写这些参数压栈的代码…*/
{
    ……
    if ((i=open_namei(filename,flag,mode,&inode))<0) {
        ……
    }
    ……
}
```

它将参数传给了open_namei()。再沿着open_namei()继续查找，文件名先后又被传给dir_namei()、get_dir()。在get_dir()中可以看到：

```
static struct m_inode * get_dir(const char * pathname)
{
    ……
    if ((c=get_fs_byte(pathname))=='/') {
        ……
    }
    ……
}
```

处理方法就很显然了：**用get_fs_byte()获得一个字节的用户空间中的数据。**那如何实现从核心态拷贝数据到用户态内存空间中呢？我们看一看include/asm/segment.h：

```
extern inline unsigned char get_fs_byte(const char * addr)
{
    unsigned register char _v;
    __asm__ ("movb %%fs:%1,%0":"=r" (_v):"m" (*addr));
    return _v;
}

extern inline void put_fs_byte(char val,char *addr)
{
    __asm__ ("movb %0,%%fs:%1"::"r" (val),"m" (*addr));
}
```

很显然, 通过**put_fs_byte**就可以实现.

我们可以看到除了get(put)_fs_byte. 还有其他函数, 如 **get(put)_fs_word, get(put)_fs_long**. 其含义看名字就能大概明白, 我们需要**针对不同的数据类型使用不同的调用方法**.

参考资料

- Linux 内核0.11 完全注释
- 哈尔滨工业大学操作系统实验手册