第2节 系统调用

前言：

一定要对系统调用的整个流程熟悉，里面涉及到的知识非常综合，前期如果没学到后面的东西，你可能对某些代码非常懵逼，比如一些段寄存器的修改，以及寻址，用户态的数据从哪里搬到内核态，怎么搬的，从用户态切换到内核态，背后发生了哪些事情，等等一些列问题，都不是单独学本章内容就能搞懂的，整个执行流程我在os部分笔记里面用画图的方式概要写了出来，具体还要结合哈工大os课程反复观看，不会的地方需要到处查资料，然后才能逐步把核心的细节搞明白。

系统调用这个lab，我们实现一个简化版的输出字符串的功能，比如用户态api叫zyw(char *),然后内核中使用sys_zyw(char *)来具体实现输出字符串的功能，当这些都设计好了之后，我们再写一个c程序调用这个用户api，并启动我们修改后的linux-0.11系统，在这个系统中运行这个c程序，看看修改后的这个linux-0.11是否正常运行，我们设计的系统调用是否能正常工作。

参考课程和lab教程：

https://www.bilibili.com/video/BV19r4y1b7Aw/?p=5

操作系统原理与实践 - 系统调用 - 蓝桥云课 (lanqiao.cn)

HIT-Linux-0.11/2-syscall/2-syscall.md at master · Wangzhike/HIT-Linux-0.11 (github.com)

哈工大操作系统之基础（Lab2 添加系统调用实验全过程+跳坑解析） testlab2-CSDN博客

Linux操作系统（哈工大李治军老师)实验楼实验2-系统调用(1)*哔哩哔哩*bilibili

https://www.bilibili.com/video/BV16L411n7pa/?p=19

用户API——zyw（）

参考：linux-0.11/lib/open.c

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

#define __LIBRARY__
#include <unistd.h>
#include <stdarg.h>

int open(const char * filename, int flag, ...)
{
    register int res;
    va_list arg;

    va_start(arg,flag);
    __asm__("int $0x80"
        :"=a" (res)
        :"0" (__NR_open),"b" (filename),"c" (flag),
```

```
        "d" (va_arg(arg,int)));
    if (res>=0)
        return res;
    errno = -res;
    return -1;
}
```

直接打开上面这个，参考它的写法即可

使用 touch linux-0.11/lib/zyw.c 创建文件，并使用gedit linux-0.11/lib/zyw.c 把下面的内容拷贝进去保存即可。

```
#define __LIBRARY__
#include <unistd.h>
#include <stdarg.h>

int zyw(const char * str)
{
    register int res;
    __asm__("int $0x80"
        :"=a" (res)
        :"0" (__NR_zyw),"b" (str));
    if (res>=0)
        return res;
    return -1;
}
```

注意，在system_call中，在call sys_call_table之前，有一个压栈动作

```
pushl %edx
pushl %ecx      # push %ebx,%ecx,%edx as parameters
pushl %ebx      # to the system call
```

因此sys_zyw()函数前三个参数就是%ebx，%ecx，%edx，不过我们用不上这么多参数，

只需要第一个参数，也就是%ebx。假如只需要%edx，那么前两个参数必须占位。

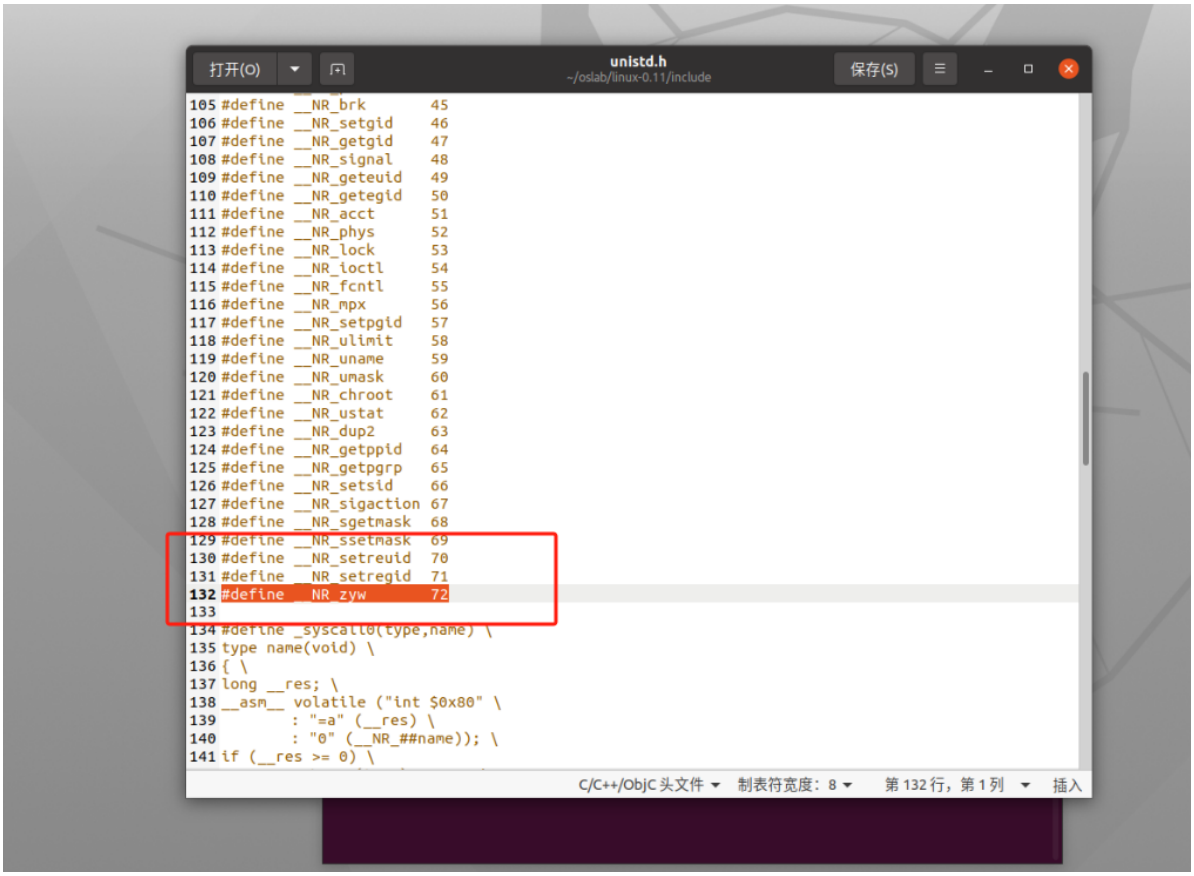
关于参数传递，这里就不多说了。

内核系统调用函数——sys_zyw()

系统调用号__NR_zyw

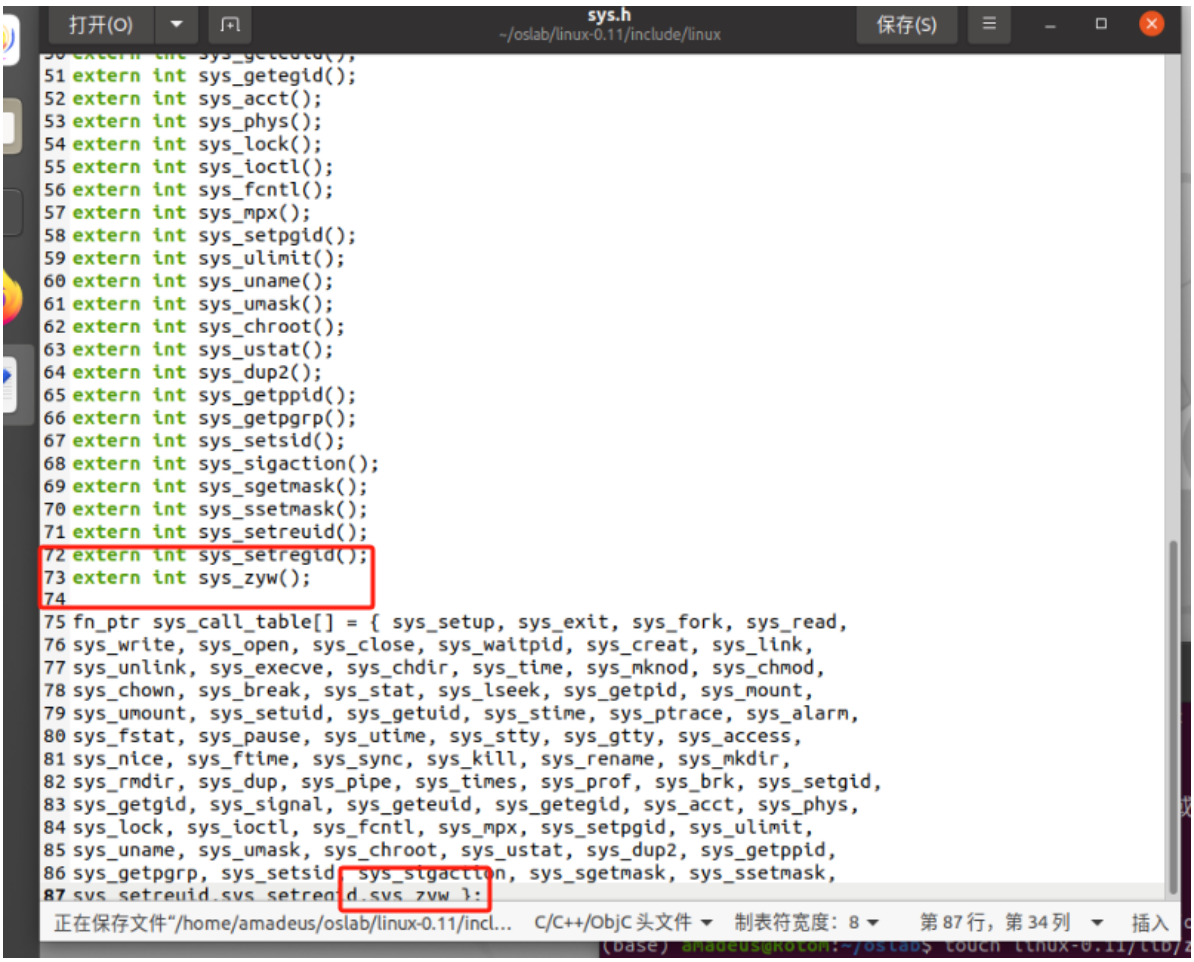
修改位置：linux-0.11/include/unistd.h

直接gedit打开添加定义即可



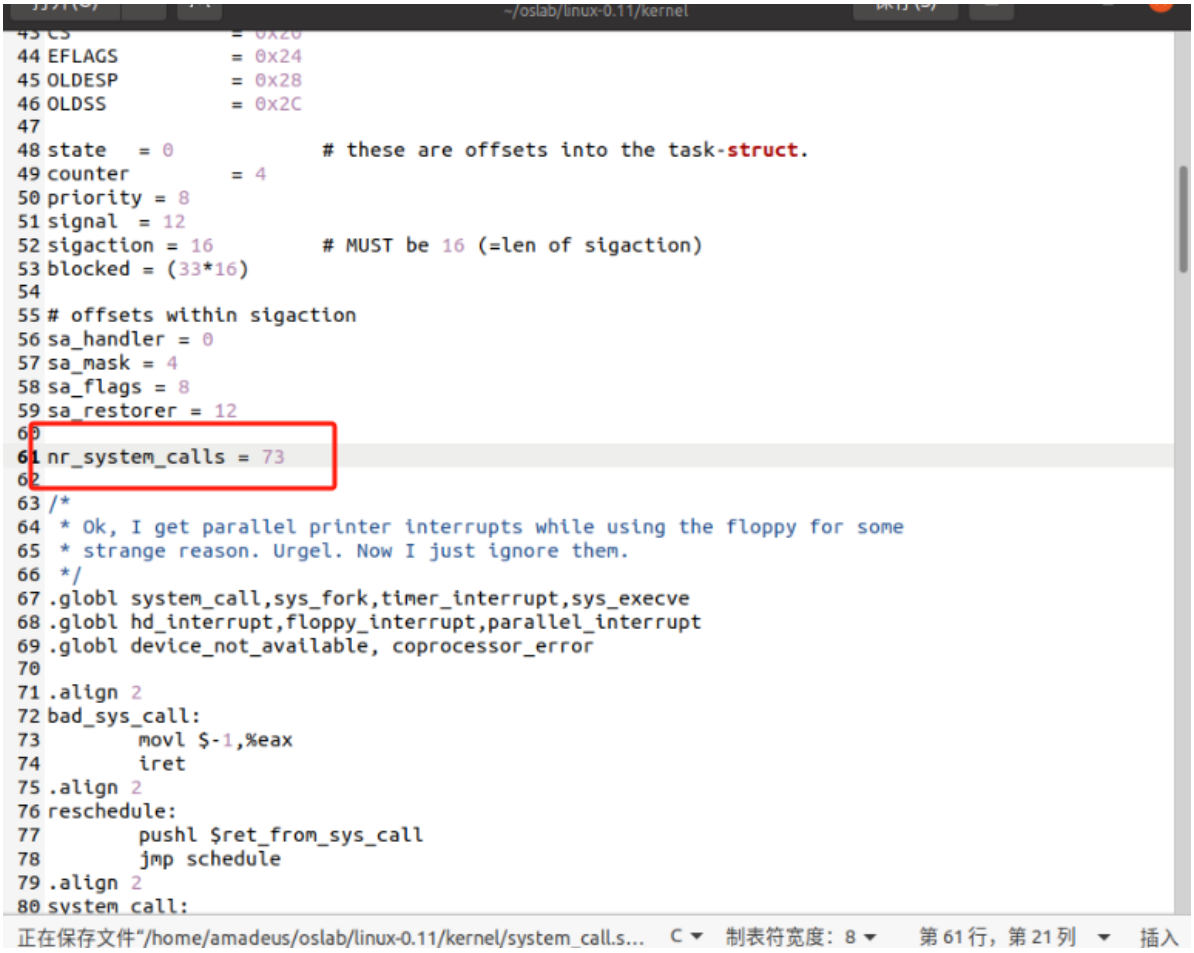
函数调用表，sys_call_table[]

位置：linux-0.11/include/linux/sys.h



函数调用表大小，nr_system_calls

位置：linux-0.11/kernel/system_call.s



函数本体，sys_zyw

位置：linux-0.11/kernel/zyw.c

```
#include <errno.h>
#include <asm/segment.h>
#include <stdarg.h>
#include <linux/kernel.h>

int sys_zyw(const char *str) {
    int i;
    char tmp[100];
    for(i = 0; i < 100; i++) {
        // 将用户态数据搬到内核态
        tmp[i] = get_fs_byte(str + i);
        if(tmp[i] == '\0')
            break;
    }
    printk("%s",tmp);
    return i;  // 简单返回显示的字符串数量，不作各种判断了
}
```

可以看到用`get_fs_byte()`获得一个字节的用户空间中的数据。

`get_fs_byte()`在*/include/asm/segment.h*中实现：

```
static inline unsigned char get_fs_byte(const char * addr)
{
    unsigned register char _v;

    __asm__ ("movb %%fs:%1,%0":"=r" (_v):"m" (*addr));
    return _v;
}
```

`movb %%fs:%1,%0` 就是把1号寄存器中的内容，也就是`*addr`，其实就是用户态的数据，搬到0号寄存器
`"=r" (_v)` 就是把0号寄存器的内容，搬到内核态的内存变量`_v`中
经过上面就实现了将用户态的一个字符搬到内核态中去。

注意，寻址的时候，`fs=0x17`，二进制就是`10111`，`TI=1`，会查`ldt`段表，而不是`gdt`段表，这是能找到用户态的关键，
而`addr`就是一开始从用户态传来的偏移地址，就是我们的`zyw()`里面的参数。

理解这个代码，需要你理解`mmu`的工作原理。

c程序

前面都是操作系统层面的，最后提供给应用程序一个zyw()接口，现在我们编写应用程序，直接使用这个接口。

另外需要注意，linux-0.11操作系统部分前面已经全部写完了，后面这个c程序写在linux-0.11挂载的的hdc文件系统中，当然，我们在ubuntu下也能挂载，为了编写程序方便，我们先用Ubuntu挂载，把c程序写进去，然后卸载这个文件系统，最后编译运行linux-0.11系统，在linux-0.11中来编译我们的c程序，并运行，看看是否work。

1. 挂载命令

```
sudo ./mount-hdc
```

2. 然后在hdc/usr/root目录下创建test.c文件，这就是我们的c语言主程序

```
touch hdc/usr/root/test.c
```

"test.c"文件内容如下

```
#define __LIBRARY__
#include <unistd.h>
#include <stdio.h>
#include <stdarg.h>
#define __NR_zyw 72
int zyw(const char * str)
{
    register int res;
    __asm__("int $0x80"
        :"=a" (res)
        :"0" (__NR_zyw),"b" (str));
    if (res>=0)
        return res;
```

```
    return -1;
}

int main() {
    const char *message = "Hello, ZYW system call!";
    zyw(message);
    return 0;
}
```

注意，用户API这里重写一遍了，因为前面写的在这里没有#include到，到底怎么才能#include进来，后面有空再研究，现在直接抄过来

3. 然后修改hdc/usr/include/unistd.h , 添加一行。

这个其实前面在linux-0.11系统里面已经写了，但是在主程序中却看不到，不清楚为啥

```
#define __NR_zyw 72
```

4. 最后卸载hdc

```
sudo umount hdc
```

假如没有卸载hdc就run了，会导致文件系统破坏，删除oslab，重新来一遍。

Makefile文件修改

（1）第一处

```
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 zyw.o
```

添加了 `zyw.o`。

（2）第二处

```
### Dependencies:
exit.s exit.o: exit.c ../include/errno.h ../include/signal.h \
  ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
  ../include/asm/segment.h
```

改为：

```
### Dependencies:
zyw.s zyw.o: zyw.c ../include/linux/kernel.h ../include/unistd.h
exit.s exit.o: exit.c ../include/errno.h ../include/signal.h \
  ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
  ../include/asm/segment.h
```

添加了 `who.s who.o: who.c ../include/linux/kernel.h ../include/unistd.h`。

Makefile 修改后，和往常一样 `make all` 就能自动把 `zyw.c` 加入到内核中了。

开始编译运行

首先进入linux-0.11目录下进行编译

```
cd linux-0.11
make clean
make all
```

然后启动linux-0.11

```
cd ~/oslab
./run
```

在linux-0.11下，ls查看文件

```
ls
```

你会发现hdc里面的test.c就在里面

然后，我们编译test.c，注意，我们还是在linux-0.11系统下，不是在ubuntu下

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

-Wall是可选项，可提供编译错误提示

然后就正常编译成功，最后我们执行这个程序

```
./test
```

你会在屏幕上看见我们在test.c主程序中定义的message字符串，这个字符串就是通过zyw()系统调用，复制到内核中，然后由printk进行输出的。