

# 浙江大学实验报告

课程名称： 操作系统分析及实验 实验类型： 综合型/设计性

实验项目名称： 实验3 向内核添加系统调用

学生姓名： 胡亮泽 专业： 计算机科学与技术 学号： 3120102116

电子邮件地址（必须）： huliangze@yeah.net 手机： 18868103152

实验日期： 2014 年 12 月 26 日

## 一、实验目的

学习 Linux 内核的系统调用，理解、掌握 Linux 系统调用的实现框架、用户界面、参数传递、进入/返回过程。阅读 Linux 内核源代码，通过添加一个简单的系统调用实验，进一步理解 Linux 操作系统处理系统调用的统一流程。了解 Linux 操作系统缺页处理，进一步掌握 task\_struct 结构的作用。

## 二、实验内容

在现有的系统中添加一个不用传递参数的系统调用。这个系统调用的功能是实现统计操作系统缺页总次数和当前进程的缺页次数，严格来说这里讲的“缺页次数”实际上是页错误次数，即调用 do\_page\_fault 函数的次数。实验主要内容：

1. 添加系统调用的名字
2. 利用标准 C 库进行包装
3. 加系统调用号
4. 在系统调用表中添加相应表项
5. 修改统计缺页次数相关的内核结构和函数
6. sys\_mysyscall 的实现
7. 编写用户态测试程序

## 三、主要仪器设备（必填）

VAIO E 系列，Ubuntu 14.04，

3.13.0-40-generic #69-Ubuntu SMP Thu Nov 13 17:56:26 UTC 2014 i686 i686 i686

GNU/Linux

## 四、操作方法和实验步骤

1. 在解压后的内核目录下，找到 **include/uapi/asm-generic/unistd.h** 文件，在其中找到关于 223 号系统调用的定义。删除原定义，并添加如下图所示的定义内容。

```
/* mm/fadvise.c */
#define __NR_mysyscall 223
__SYSCALL(__NR_mysyscall, sys_mysyscall)
```

如图，定义了一个名为 sys\_mysyscall 的系统调用

2. 找到目录 `/usr/include/asm-generic/` 下的 `unistd.h` 头文件, 做同样的修改, 如下图所示:

```
/* mm/fadvise.c */
#define __NR_mysyscall 223
__SYSCALL(__NR_mysyscall, sys_mysyscall)
```

3. 在系统调用表中添加和 223 号系统调用相关的索引。

由于在虚拟机中安装的 ubuntu 操作系统是 32 位的, 因此找到 `arch/x86/syscalls/` 目录下的 `syscall_32.tbl` 文件 (如果是 64 位, 需要修改 `syscall_64.tbl`), 将 223 号系统调用的索引加入列表。如下图所示:

```
220 i386 getdents64 sys_getdents64 compat_sys_getdents64
221 i386 fcntl64 sys_fcntl64 compat_sys_fcntl64
# 222 is unused
223 i386 mysyscall sys_mysyscall
224 i386 gettid sys_gettid
225 i386 readahead sys_readahead
```

这样一来, 入口函数名字就是 `sys_mysyscall` 了, 系统调用已经可以根据调用号找到相应的函数了。一行中的 4 列所代表的分别是系统调用号, ABI 接口名称, 名字和入口函数名

4. 在解压的内核目录下找到 `include/linux` 目录下的 `mm.h` 头文件, 在其中定义 `pfcount` 变量, 如下图所示:

```
#include <linux/mm_types.h>
#include <linux/range.h>
#include <linux/pfn.h>
#include <linux/bit_spinlock.h>
#include <linux/shrinker.h>

extern unsigned long pfcount;
```

该变量主要用以记录所有进程的缺页总次数

5. 在 `include/linux` 目录下找到 `sched.h`, 在其中的进程信息数据结构 `task_struct` 中定义一个新的变量 `pf`. 如下图所示:

```

struct task_struct {
    unsigned long pf;
    volatile long state;    /* -1
    void *stack;

```

该变量的作用主要是记录进程自身产生缺页的次数。

6.接下来在 kernel 目录下找到 fork.c 中的 dup\_task\_struct 函数，该函数负责在创建新进程时进行数据的复制。但是记录一个新进程的缺页次数的变量 pf 在复制后必须清零，保证新的进程从 0 开始计数。因此在下图中返回 tsk 之前添加了一行代码。

```

tsk->splice_pipe = NULL;
tsk->task_frag.page = NULL;

account_kernel_stack(ti, 1);

tsk->pf = 0;
return tsk;

```

如图所示，将新进程的 pf 清零，可以重新开始计数。注意，必须在最后在进行 pf 的清零，否则过程中会有对 pf 的处理，返回的值不一定是 0。

6.在定义完记录缺页次数的变量后，需要重新计算缺页。由于每次缺页都会调用 \_\_do\_page\_fault 函数，因此调用函数的次数就是总共缺页的次数，只需要每次调用 \_\_do\_page\_fault 函数时修改 pfcount 和 pf 变量就可以完成缺页计算了。

因此，在 arch/x86/mm/ 目录中，找到 fault.c 文件，做出如下图修改：

```

unsigned long pfcount;

static ninline void
__do_page_fault(struct pt_regs *regs, unsigned long error_code,
                unsigned long address)
{
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct mm_struct *mm;
    int fault;
    unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_

    tsk = current;
    mm = tsk->mm;

    pfcount ++;
    current->pf++;
}

```

在图片的末尾，加上了 pfcount++ 和 current->pf++ 的代码，分别表示总缺页次数和当前进程缺页次数加 1。

最后，在 kernel/ 目录中的定义系统调用的 sys.c 文件中定义新的系统调用的函数，将其具体的执行代码写出，如下图所示：

```

asmlinkage int sys_mysyscall(void){

    printk("当前进程缺页次数: %lu \n",current->pf);
    return 0;

}

```

该系统调用的作用是在系统日志文件中打印出当前进程缺页次数，以及总缺页次数。

7.最后在 syscalls.h 头文件中对新的系统调用函数做出申明，否则可能会在编译内核的最后出现该系统调用函数未定义的错误，导致 2 个小时的内核编译功亏一篑！

```

asmlinkage int sys_mysyscall(void);

```

修改完源代码后，就是重新编译内核和内核模块，以及安装内核模块和内核的过程了，实验二中已经完成了完整的操作，这里不多做赘述。

8.成功安装完系统内核后重新启动，通过一个在 C 程序中执行 syscall(223)，并利用 dmesg 命令，可以在系统日志中找到相关的信息。

## 五、实验中遇到的问题

实验中遇到的最大问题就是在编译内核结束后出现错误提示，提示内容主要是自定义的系统调用 `sys_mysyscall` 未定义。由于该错误在编译内核的最后才会出现，因此为了尝试解决这个问题编译了 4 次内核，花费了不少时间。

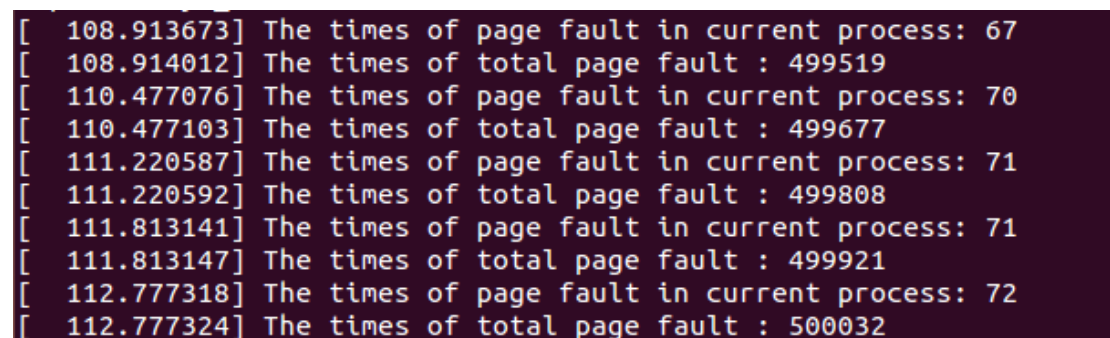
经过查阅资料，解决方案主要是：

1. 现在 `syscalls.h` 中对新的系统调用 `sys_mysyscall` 函数做出声明
2. 在编译内核时，使用 `make mrproper` 命令和 `make clean` 命令清除前一次编译留下的所有内容。

通过以上步骤，成功解决了问题。

## 六、实验结果和分析

在执行多次系统调用后，利用 `dmesg` 命令查看系统日志中的信息，得到如图结果：



```
[ 108.913673] The times of page fault in current process: 67
[ 108.914012] The times of total page fault : 499519
[ 110.477076] The times of page fault in current process: 70
[ 110.477103] The times of total page fault : 499677
[ 111.220587] The times of page fault in current process: 71
[ 111.220592] The times of total page fault : 499808
[ 111.813141] The times of page fault in current process: 71
[ 111.813147] The times of total page fault : 499921
[ 112.777318] The times of page fault in current process: 72
[ 112.777324] The times of total page fault : 500032
```

由结果可知，新的内核中已经添加了新的系统调用，并且很好的得到了执行。

每行都成功显示出了系统调用执行时当前进程的缺页次数和总缺页次数。

## 七、讨论、心得

1. 本次的实验让我掌握了如何简单地向内核中添加系统调用。系统调用是操作系统向用户提供内核服务的唯一方法，学会这个方法将使我对操作系统的使用更加灵活，也可以利用操作系统完成更多我希望的自定义操作。
2. 本次实验让我更加深入的了解操作系统内核的本质，对于一些系统调用的实现，也不再显得那么神秘，相反，只要愿意，并且不出错的情况下，完全可以对内核的系统调用函数做出修改，添加，删除的操作。
3. 对于编译内核中可能出现的一些错误，比如未定义的错误，学会了应对方案。

## 八、思考题

1. 多次运行 test 程序，每次运行 test 后记录下系统缺页次数和当前进程缺页次数，给出这些数据。test 程序打印的缺页次数是否就是操作系统原理上的缺页次数？有什么区别？

两者并不相同。

经过资料的查阅，不只是在缺页的时候调用 `__do_page_fault` 函数，还包括程序出错，虚拟地址被写保护等情况，这些情况也会调用该函数。这些情况下缺页次数也会增加。

2. 除了通过修改内核来添加一个系统调用外，还有其他的添加或修改一个系统调用的方法吗？如果有，请论述。

还可以通过添加内核模块或者修改内核模块来添加或者修改系统调用

3. 对于一个操作系统而言，你认为修改系统调用的方法安全吗？请发表你的观点。

内核的系统调用几乎都是精心设计的，在内核运行过程中存在各种系统的调用，这些调用很可能就是操作系统正常工作的关键。因此，如果不是对操作系统内核工作机制和某个系统调用的工作机理，调用场合等十分了解，修改系统调用是十分危险的，因为我们并不清楚这个系统调用会在哪些关键的场合被用到，而我们的修改又会造成什么致命的影响，严重的情况甚至会造成操作系统的崩溃。

当然，如果对操作系统的内核十分了解，并且确定我们的修改不会造成致命或者不好的影响，修改系统内核还是可以的。还可以用这种方法完成很多灵活的操作，而操作系统并不完美，我们可以通过这种方法打造最适合自己的系统内核