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 2 添加系统调用

学生姓名：卢佳盈 专业：计算机科学与技术 学号：3180103570

电子邮件地址：ljy28501@163.com 手机：18868703211

实验日期：2020 年 12 月 5 日

一、实验目的

学习重建 Linux 内核。

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，当前进程的缺页次数，以及每个进程的“脏”页面数。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：

- 在 Linux 操作系统环境下重建内核
- 添加系统调用的名字
- 利用标准 C 库进行包装
- 添加系统调用号
- 在系统调用表中添加相应表项
- 修改统计缺页次数、“脏”页相关的内核结构和函数
- sys_mysyscall 的实现
- 编写用户态测试程序

三、主要仪器设备（必填）

host machine:

windows 10

处理器：Intel(R) Core(TM) i7-8565U CPU @1.80GHz

RAM：8.00GB

系统类型：64 位操作系统，基于 x64 的处理器

guest machine:

虚拟机：VMware
Ubuntu 16.04.10(32 位)
RAM：4GB

四、操作方法和实验步骤

1/2/3. 重装 4.6.0 内核

重复实验 1 中重装内核的步骤

```
root@miracle-virtual-machine:/usr/include/asm-generic# uname -r
4.6.0
```

4. 添加系统调用号

系统调用号在文件 unistd.h 里面定义。这个文件可能在你的 Linux 系统上会有两个版本：一个是 C 库文件版本，出现的地方是在 ubuntu 16.04 自带的/usr/include/asmgeneric/unistd.h；另外还有一个版本是内核自己的 unistd.h，出现的地方是在你解压出来的内核代码的对应位置（比如 include/uapi/asm-generic/unistd.h）。当然，也有可能这个 C 库文件只是一个对应到内核文件的链接。现在，你要做的就是在文件 unistd.h 中添加我们的系统调用号：__NR_mysyscall，x86 体系架构的系统调用号 223 没有使用，我们新的系统调用号定义为 223 号，如下所示：

ubuntu 16.04 为：/usr/include/asm-generic/unistd.h
kernel 4.6 为：include/uapi/asm-generic/unistd.h
在/usr/include/asm-generic/unistd.h 文件中的插入定义 223 号的行，作如下修改：

-- #define __NR3264_fadvise64 223
--__SC_COMP(__NR3264_fadvise64, sys_fadvise64_64, compat_sys_fadvise64_64)
++ #define __NR_mysyscall 223
++ __SYSCALL(__NR_mysyscall, sys_mysyscall)

```
/*#define __NR3264_fadvise64 223
__SC_COMP(__NR3264_fadvise64, sys_fadvise64_64, compat_sys_fadvise64_64)*/

#define __NR_mysyscall 223
__SYSCALL(__NR_mysyscall,sys_mysyscall)
```

在文件 include/uapi/asm-generic/unistd.h 中做同样的修改

```
/*#define __NR3264_fadvise64 223
__SC_COMP(__NR3264_fadvise64, sys_fadvise64_64, compat_sys_fadvise64_64)*/
#define __NR_mysyscall 223
__SYSCALL(__NR_mysyscall,sys_mysyscall)
```

5. 在系统调用表中添加或修改相应表项

我们前面讲过，系统调用处理程序（system_call）会根据 eax 中的索引到系统调用表（sys_call_table）中去寻找相应的表项。所以，我们必须在那里添加我们自己的一个值。

```
# 223 is unused
223     i386    mysyscall               sys_mysyscall

224     i386    gettid                  sys_gettid
225     i386    readahead               sys_readahead
sys32_readahead
226     i386    setxattr                sys_setxattr
```

到现在为止，系统已经能够正确地找到并且调用 sys_mysyscall。剩下的就只有一件事情，那就是 sys_mysyscall 的实现。

6. 修改统计系统缺页次数和进程缺页次数的内核代码

由于每发生一次缺页都要进入缺页中断服务函数 do_page_fault 一次，所以可以认为执行该函数的次数就是系统发生缺页的次数。可以定义一个全局变量 pfcount 作为计数变量，在执行 do_page_fault 时，该变量值加 1。在当前进程控制块中定义一个变量 pf 记录当前进程缺页次数，在执行 do_page_fault 时，这个变量值加 1。

先在 include/linux/mm.h 文件中声明变量 pfcount：

```
++ extern unsigned  long  pfcount;
```

```
struct mempolicy;
struct anon_vma;
struct anon_vma_chain;
struct file_ra_state;
struct user_struct;
struct writeback_control;
struct bdi_writeback;

extern unsigned long pfcount;
#ifndef CONFIG_NEED_MULTIPLE_NODES      /* Don't use mapnrs, do it properly */
extern unsigned long max_mapnr;
```

要记录进程产生的缺页次数，首先在进程 task_struct 中增加成员 pf，在 include/linux/sched.h 文件中（第 1394 行 jjj）的 task_struct 结构中添加 pf 字段：

```
++ unsigned  long  pf;
```

```
struct task_struct {
        unsigned long pf;
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;
```

统计当前进程缺页次数需要在创建进程是需要将进程控制块中的 pf 设置为 0，在进程创建过程中，子进程会把父进程的进程控制块复制一份，实现该复制过程的函数是 kernel/fork.c 文件中的 dup_task_struct()函数，修改该函数将子进程的 pf 设置成 0：

```
static struct task_struct *dup_task_struct(struct task_struct *orig)
{
    …..
    tsk = alloc_task_struct_node(node);
    if (!tsk)
        return NULL;
  ……
    ++ tsk->pf=0;
    ……
}
```

```
        tsk->splice_pipe = NULL;
        tsk->task_frag.page = NULL;
        tsk->wake_q.next = NULL;

        account_kernel_stack(ti, 1);

        kcov_task_init(tsk);

        tsk->pf=0;

        return tsk;
```

在 arch/x86/mm/fault.c 文件中定义变量 pfcount；并修改 arch/x86/mm/fault.c 中 do_page_fault()函数。每次产生缺页中断，do_page_fault()函数会被调用，pfcount 变量值递增 1,记录系统产生缺页次数，current->pf 值递增 1，记录当前进程产生缺页次数：

```
…
 ++ unsigned long pfcount;
#define CREATE_TRACE_POINTS
#include <asm/trace/exceptions.h>

unsigned long pfcount;

/*
 * Page fault error code bits:
 *
 __do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
        …
 ++ pfcount++;
 ++ current->pf++;
     …
}
```

```
__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, major = 0;
        unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE;

        tsk = current;
        mm = tsk->mm;

        pfcount++;
        current->pf++;
```

7. sys_mysyscall 的实现

我们把这一小段程序添加在 kernel/sys.c 里面。在这里，我们没有在 kernel 目录下另外添加自己的一个文件，这样做的目的是为了简单，而且不用修改 Makefile，省去不必要的麻烦。

mysyscall 系统调用实现输出系统缺页次数、当前进程缺页次数，及每个进程的“脏”页面数。

```
asmlinkage int sys_mysyscall(void){

        ……

//printk(“当前进程缺页次数：%lu,current->pf”)

//每个进程的“脏”页面数

        return 0;

}
```

```
asmlinkage int sys_mysyscall(void)
{
        printk(KERN_INFO"System Page Fault Number:%lu
\n",pfcount);
        printk(KERN_INFO"Current Process Page Fault
Number:%lu\n",current->pf);
        printk(KERN_INFO"Each Process:\n");
        struct task_struct *p;
        for_each_process(p){
                printk(KERN_INFO"%-20s %-6d %lu\n",p-
>comm,p->pid,p->nr_dirtied);
        }
        return 0
}
```

8. 编译内核和重启内核

用 make 工具编译内核：

sudo make bzImage -j2

编译内核需要较长的时间，具体与机器的硬件条件及内核的配置等因素有关。完成后产生的内核文件 bzImage 的位置在/usr/src/linux/arch/i386/boot 目录下，当然这里假设用户的 CPU 是 Intel x86 型的，并且你将内核源代码放在/usr/src/linux 目录下。

```
root@miracle-virtual-machine:/usr/src/linux/arch/i386/boot# ls
bzImage
```

如果编译过程中产生错误，你需要检查第 4、5、6、7 步修改的代码是否正确，修改后要再次使用 make 命令编译，直至编译成功。

```
  CC      arch/x86/boot/version.o
  CC      arch/x86/boot/compressed/aslr.o
  CC      arch/x86/boot/compressed/eboot.o
  GZIP    arch/x86/boot/compressed/vmlinux.bin.gz
  MKPIGGY arch/x86/boot/compressed/piggy.S
  AS      arch/x86/boot/compressed/piggy.o
  LD      arch/x86/boot/compressed/vmlinux
  OBJCOPY arch/x86/boot/vmlinux.bin
  ZOFFSET arch/x86/boot/zoffset.h
  AS      arch/x86/boot/header.o
  LD      arch/x86/boot/setup.elf
  OBJCOPY arch/x86/boot/setup.bin
  BUILD   arch/x86/boot/bzImage
Setup is 17308 bytes (padded to 17408 bytes).
System is 6498 kB
CRC dc11fcff
Kernel: arch/x86/boot/bzImage is ready  (#3)
```

如果选择了可加载模块，编译完内核后，要对选择的模块进行编译，然后安装。用下面的命令编译模块并安装到标准的模块目录中：

sudo make modules -j2

sudo make modules_install

通常，Linux 在系统引导后从/boot 目录下读取内核映像到内存中。因此我们如果想要使用自己编译的内核，就必须先将启动文件安装到/boot 目录下。安装内核命令：

sudo make install

```
Found linux image: /boot/vmlinuz-4.15.0-128-generic
Found initrd image: /boot/initrd.img-4.15.0-128-generic
Found linux image: /boot/vmlinuz-4.15.0-45-generic
Found initrd image: /boot/initrd.img-4.15.0-45-generic
Found linux image: /boot/vmlinuz-4.6.0-040600-generic
Found initrd image: /boot/initrd.img-4.6.0-040600-generic
Found linux image: /boot/vmlinuz-4.6.0
Found initrd image: /boot/initrd.img-4.6.0
Found linux image: /boot/vmlinuz-4.6.0.old
Found initrd image: /boot/initrd.img-4.6.0
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

grub 是管理 ubuntu 系统启动的一个程序。想运行刚刚编译好的内核，就要修改对应的 grub。

sudo mkinitramfs 4.6.0 -o /boot/initrd.img-4.6.0

```
root@miracle-virtual-machine:/usr/src/linux# sudo mkinitramfs 4.6.0 -o /boot/ini
trd.img-4.6.0
```

sudo update-grub2

```
root@miracle-virtual-machine:/usr/src/linux# sudo update-grub2
Generating grub configuration file ...
Warning: Setting GRUB_TIMEOUT to a non-zero value when GRUB_HIDDEN_TIMEOUT is se
t is no longer supported.
Found linux image: /boot/vmlinuz-4.15.0-128-generic
Found initrd image: /boot/initrd.img-4.15.0-128-generic
Found linux image: /boot/vmlinuz-4.15.0-45-generic
Found initrd image: /boot/initrd.img-4.15.0-45-generic
Found linux image: /boot/vmlinuz-4.6.0-040600-generic
Found initrd image: /boot/initrd.img-4.6.0-040600-generic
Found linux image: /boot/vmlinuz-4.6.0
Found initrd image: /boot/initrd.img-4.6.0
Found linux image: /boot/vmlinuz-4.6.0.old
Found initrd image: /boot/initrd.img-4.6.0
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

我们已经编译了内核 bzImage，放到了指定位置/boot。现在，请你重启主机系统，期待编译过的 Linux 操作系统内核正常运行！

sudo reboot

9. 编写用户态程序

要测试新添加的系统调用，需要编写一个用户态测试程序（test.c）调用 mysyscall 系统调用。mysyscall 系统调用中 printk 函数输出的信息在/var/log/messages 文件中(ubuntu 为/var/log/kern.log 文件)。

用户态程序

```
#include <linux/unistd.h>
# include <sys/syscall.h>
# include<stdio.h>
#define __NR_ mysyscall   223
int main()
{
        mysyscall(__NR_mysyscall);
        system("dmesg 1>log");
}
```

```
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#define __NR_mysyscall 223
int main(){
        syscall(__NR_mysyscall);
        system("dmesg 1>log");
}
```

- 用 gcc 编译源程序

  gcc –o test test.c

```
root@miracle-virtual-machine:/usr/src/linux# gcc test.c
```

- 运行程序

  ./test

```
root@miracle-virtual-machine:/usr/src/linux# ./a.out
```

五、实验结果和分析

mysyscall 的系统调用信息输出在/var/log/kern.log 文件：

```
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198826] System Page Fault
Number:1448901
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198829] Current Process
Page Fault Number:71
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198831] Each Process:
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198833]
systemd              1       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198835]
kthreadd             2       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198837]
ksoftirqd/0          3       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198838]
kworker/0:0          4       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198840]
kworker/0:0H         5       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198841] kworker/
u16:0        6       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198843]
rcu_sched            7       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198844]
rcu_bh               8       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198846]
migration/0          9       0
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198847]
```

六、问题解答

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？

```
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198826] System Page Fault
Number:1448901
Dec 15 16:14:41 miracle-virtual-machine kernel: [   341.198829] Current Process
Page Fault Number:71

Dec 15 16:15:42 miracle-virtual-machine kernel: [   402.642185] System Page Fault
Number:1477526
Dec 15 16:15:42 miracle-virtual-machine kernel: [   402.642187] Current Process
Page Fault Number:68

Dec 15 16:26:12 miracle-virtual-machine kernel: [ 1032.160156] System Page Fault
Number:1781498
Dec 15 16:26:12 miracle-virtual-machine kernel: [ 1032.160157] Current Process
Page Fault Number:64
```

   可以看到，操作系统原理上的却也次数随着 test 程序调用次数的增加而增加，而 test 程序打印的缺页次数却有可能在减少。因为 test 程序打印的缺页次数并不是操作系统原理上的缺页次数。操作系统原理上的缺页次数是页面置换次数乘以物理块数，而 test 程序打印的是__do_page_fault 函数执行的次数，即页访问出错的次数

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。

   存在。可以采用系统调用拦截，改变某一个系统调用号对应的服务程序为我们自己的编写的程序，从而相当于添加了我们自己的系统调用

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。

   我认为不安全。因为系统调用是应用程序主动进入操作系统内核的入口，如果可以修改系统调用，就可以通过应用程序非法访问内核，导致系统崩溃

七、讨论、心得

本次实验中我收获很多，除了关于操作系统添加系统调用这一块内容有了认识之外，还对内核的降级安装又有了新的认识。在实验一中，新安装的内核版本高于原来的版本，因此在重新开机后，系统会自动选择版本更高的内核。在这次试验中，原内核版本（4.15.0）比实验指导中的内核版本（4.6.0）更高，因此需要修改 cfg 文件，使系统进入 4.6.0 的内核，在一番操作后，我将 `/boot/grub/grub.cfg` 修改如下：

```
if [ "${next_entry}" ] ; then
   set default="${next_entry}"
   set next_entry=
   save_env next_entry
   set boot_once=true
else
   set default="gnulinux-advanced-
fa624024-707f-4271-8341-7de6fa6de7a5>gnulinux-4.6.0-advanced-
fa624024-707f-4271-8341-7de6fa6de7a5"
fi
```

才成功进入 4.6.0 的内核系统。

此外，在运行完用户态程序后，程序的输出并没有按照我的预期成功打印再终端，这让我非常迷惑，但根据实验指导找到 kern.log 文件后，发现缺页的信息都打印在日志文件中了。

APPENDIX 源码

mysyscall（）

```
asmlinkage int sys_mysyscall(void){
    printk(KERN_INFO"System Page Fault Number:%lu\n",pfcount);
    printk(KERN_INFO"Current Process Page Fault Number:%lu\n",current->
pf);
    printk(KERN_INFO"Each Process:\n");
    struct task_struct *p;
    for_each_process(P){
        printk(KERN_INFO"%-20s %-6d %lu\n",p->comm,p->pid,p->nr_dirtied
);
    }
    return 0;
}
```

test.c

```
#include<linux/unistd.h>
#include<sys/syscall.h>
#include<stdio.h>
#define __NR_mysyscall 223
int main(){
    syscall(__NR_mysyscall);
```

```
    system('dmesg 1>log');
}
```