浙江大学实验报告

课程名称：操作系统 实验类型：综合型

实验项目名称：添加系统调用

学生姓名：刘轩铭 学号 3180106071

电子邮件地址：519102931@qq.com

实验日期：2020 年 12 月 4 日

一、 实验环境

阿里云服务器：1G 内存；

Ubuntu16.04；

Linux-4.8 及其补丁；

二、 实验内容和结果及分析

1. 实验设计思路

- 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
- 修改统计缺页次数相关的内核结构和函数
- sys_mysyscall 的实现
- 编写用户态测试程序

2. 实验步骤及截图

1. 下载内核源代码

在 http://mirrors.aliyun.com/linux-kernel/下载 4.8 版本:

linux-4.8.tar.xz 和 **patch-4.8.xz（补丁）**文件。

```
wget http://mirrors.aliyun.com/linux-kernel/v4.x/linux-4.8.tar.xz

wget http://mirrors.aliyun.com/linux-kernel/v4.x/patch-4.8.xz
```

2. 部署内核源代码

把内核代码文件 linux-4.8.tar.xz 存放在根目录下。解压内核包，生成的内核源代码放在 linux.4.8 目录中：

```
tar -xvf linux-4.8.tar.xz
```

然后在 linux.4.8 目录中打入内核补丁：

```
xz -d patch-4.8.xz | patch -p1
```

3. 配置内核

第 1 次编译内核的准备：

在 ubuntu 环境下，用命令 make menuconfig 对内核进行配置时，需要用终端模式下的字符菜单支持软件包 libncurses5-dev，因此你是第一次重建内核，需要下载并安装该软件包，下载并安装命令如下：

```
apt-get install libncurses5-dev libssl-dev

apt-get install bc
```

配置内核：

在编译内核前，一般来说都需要对内核进行相应的配置。配置是精确控制新内核功能的机会。配置过程也控制哪些需编译到内核的二进制映像中(在启动时被载入)，哪些是需要时才装入的内核模块（module）。

```
cd /linux-4.8
```

第一次编译的话，有必要将内核源代码树置于一种完整和一致的状态。因此，我们推荐执行命令 make mrproper。它将清除目录下所有配置文件和先前生成核心时产生的.o 文件：

```
make mrproper
```

为了与正在运行的操作系统内核的运行环境匹配，可以先把当前已配置好的文件复制

到当前目录下，新的文件名为.config 文件：

```
cp /boot/config-`uname -r` .config
```

这里，命令`uname  -r`得到当前内核版本号。然后：

```
make menuconfig
```

在出现的窗口中依次选择择 load→OK→Save→OK→EXIT→EXIT，完成配置。

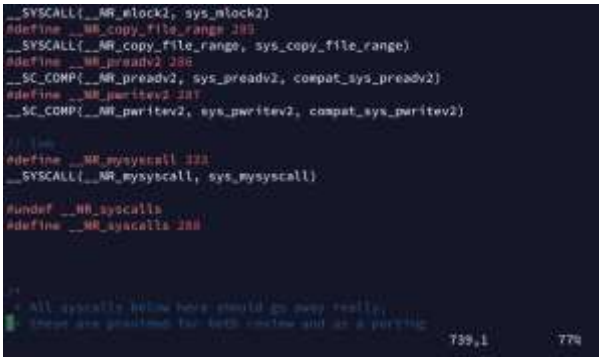
4. 添加系统调用号

系统调用号在文件 unistd.h 里面定义。这个文件可能在 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/usr/include/asm-generic/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 库文件只是一个到对应内核文件的连接。现在，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 333 没有使用,我们新的系统调用号定义为 333 号，如下所示：

ubuntu 16.04 为：/usr/include/asm-generic/unistd.h

kernel 4.8 为：/linux-4.8/include/uapi/asm-generic/unistd.h

在**/usr/include/asm-generic/unistd.h** 文件中的查找定义 333 号的行，添加或修改 333 号系统调用 mysyscall：

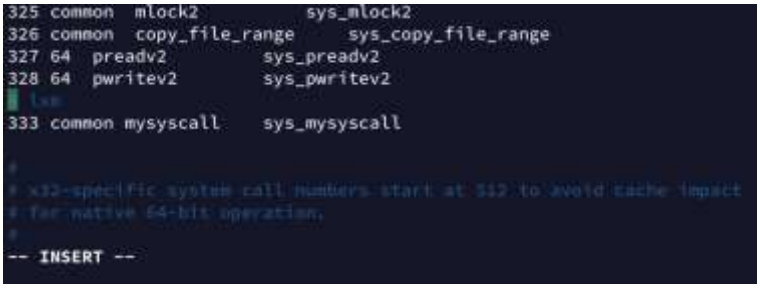


之后再在文件 **linux-4.8/include/uapi/asm-generic/unistd.h** 中做同样的修改.

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。

在 **arch/x86/entry/syscalls/syscall_64.tbl** 中添加如下：



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：

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 文件中的 task_struct 结构中添加 pf 字段：

```
struct task_struct {
    // lxm
    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;

#ifdef CONFIG_SMP
```

统计当前进程缺页次数需要在创建进程时将进程控制块中的 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, int node)
{
    struct task_struct *tsk;
    unsigned long *stack;
    int err;

    if (node == NUMA_NO_NODE)
        node = tsk_fork_get_node(orig);
    tsk = alloc_task_struct_node(node);
    if (!tsk)
        return NULL;

    // lxm
    tsk->pf = 0;

    stack = alloc_thread_stack_node(tsk, node);
    if (!stack)
        goto free_tsk;

    err = arch_dup_task_struct(tsk, orig);
    if (err)
        goto free_stack;

    tsk->stack = stack;
```

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：

```
// lxm
unsigned long pfcount = 0;

enum x86_pf_error_code {

    PF_PROT     =       1 << 0,
    PF_WRITE    =       1 << 1,
    PF_USER     =       1 << 2,
    PF_RSVD     =       1 << 3,
    PF_INSTR    =       1 << 4,
```

：

```
__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;

    // lxm
    pfcount++;
    current->pf++;

    tsk = current;
    mm = tsk->mm;
```

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。

```
// lxm
extern unsigned long pfcount;
asmlinkage int sys_mysyscall(void)
{
    printk("page fault of system:%lu\n",pfcount);
    printk("page fault of current process: %lu\n",current->pf);
    printk("dirty page of all process:\n");
    struct task_struct *p=NULL;
    for (p = &init_task; (p = next_task(p)) != &init_task;)
    {
        // output process information.
        printk("pid:%ld  dirty page: %ld\n", 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：

```
root@iZbp1bwwc1xkjp6@yxsr52Z:~/linux-4.0# make
  HOSTCC  scripts/kconfig/conf.o
  HOSTLD  scripts/kconfig/conf
scripts/kconfig/conf  --silentoldconfig Kconfig
  SYSTBL  arch/x86/entry/syscalls/../../include/generated/asm/syscalls_32.h
  SYSHDR  arch/x86/entry/syscalls/../../include/generated/asm/unistd_32_ia32.h
  SYSHDR  arch/x86/entry/syscalls/../../include/generated/asm/unistd_64_x32.h
  SYSTBL  arch/x86/entry/syscalls/../../include/generated/asm/syscalls_64.h
  HYPERCALLS arch/x86/entry/syscalls/../../includo/generated/asm/xen-hypercalls
  SYSHDR  arch/x86/entry/syscalls/../../include/generated/uapi/asm/unistd_32.h
  SYSHDR  arch/x86/entry/syscalls/../../include/generated/uapi/asm/unistd_64.h
  SYSHDR  arch/x86/entry/syscalls/../../include/generated/uapi/asm/unistd_x32.h
  HOSTCC  scripts/basic/bin2c
  HOSTCC  arch/x86/tools/relocs_32.o
  HOSTCC  arch/x86/tools/relocs_64.o
  HOSTCC  arch/x86/tools/relocs_common.o
  HOSTLD  arch/x86/tools/relocs
  CHK     include/config/kernel.release
  UPD     include/config/kernel.release
  WRAP    arch/x86/include/generated/asm/clkdev.h
  WRAP    arch/x86/include/generated/asm/cputime.h
  WRAP    arch/x86/include/generated/asm/dma-contiguous.h
```

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~/linux/arch/i386/boot 目录下，当然这里假设用户的 CPU 是 Intel x86 型的，并且内核源代码放在~/linux 目录下。如果编译过程中产生错误，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。

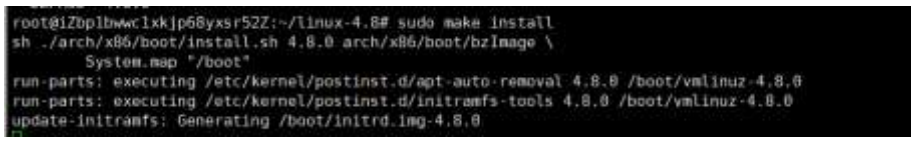
如果选择了可加载模块，编译完内核后，要对选择的模块进行编译。用下面的命令编译模块并安装到标准的模块目录中：

```
sudo make modules

sudo make modules_install
```

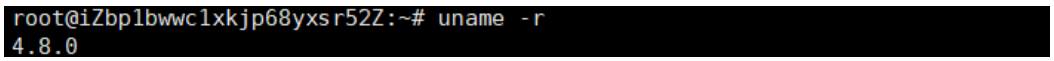
通常，Linux 在系统引导后从/boot 目录下读取内核映像到内存中。因此我们如果想要使用自己编译的内核，就必须先将启动文件安装到/boot 目录下。安装内核命令：

```
sudo make install
```



我们已经编译了内核 bzImage，放到了指定位置/boot。现在，重启主机系统。

```
sudo reboot
```



重启后查看内核版本，发现现在已经更改为 4.8.0

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)。/var/log/messages (ubuntu 为/var/log/kern.log 文件)文件中的内容也可以在 shell 下用 dmesg 命令查看到。

用户态程序 lab2_test.c:

```
#include <liunx/unistd.h>

#include <sys/syscall.h>

#define __NR_ mysyscall  333

int main()

{

    syscall(__NR_mysyscall);

}
```

- 用 gcc 编译源程序

  gcc -o test test.c

- 运行程序

  ./test

3. 测试程序运行结果截图

./test:

```
root@iZbp1bwwc1xkjp68yxsr52Z:~/os_course# ./test
```

查看日志中的记录：

```
root@iZbp1bwwc1xkjp68yxsr52Z:~/os_course# dmesg
[  416.145739] page fault of system:257353
[  416.145741] page fault of current process: 8168
[  416.145741] dirty page of all process:
[  416.145742] pid:1--dirty page: 0
[  416.145743] pid:2--dirty page: 0
[  416.145744] pid:3--dirty page: 0
[  416.145744] pid:5--dirty page: 0
[  416.145745] pid:6--dirty page: 0
[  416.145746] pid:7--dirty page: 0
[  416.145746] pid:8--dirty page: 0
[  416.145747] pid:9--dirty page: 0
[  416.145748] pid:10--dirty page: 0
[  416.145748] pid:11--dirty page: 0
[  416.145749] pid:12--dirty page: 0
[  416.145750] pid:13--dirty page: 0
[  416.145750] pid:14--dirty page: 0
```

4. 结果分析

通过此次实验，打印出了当前内核中的总的页中断数目为 257353，当前进程的页中断

数目为 8168，每个进程的脏页数目也逐一进行了输出。完成了实验。

5. 源程序

lab2_test.c

```c
#include <linux/unistd.h>
#include <sys/syscall.h>
#define __NR_mysyscall 333
int main()
{
    syscall(__NR_mysyscall);
}
```

mysyscall():

```c
// lxm
extern unsigned long pfcount;
asmlinkage int sys_mysyscall(void)
{
    printk("page fault of system:%lu\n",pfcount);
    printk("page fault of current process: %lu\n",current->pf);
    printk("dirty page of all process:\n");
    struct task_struct *p=NULL;
    for (p = &init_task; (p = next_task(p)) != &init_task;)
    {
        // output process information.
        printk("pid:%ld--dirty page: %d\n", p->pid, p->nr_dirtied);
    }
    return 0;
}
```

三、 讨论与心得

在此次实验中，一开始使用的是 ubuntu 虚拟机，编译一次内核需要四个小时的时间，而且电脑不能关机，会很麻烦。后来我在阿里云服务器上面进行编译，编译的速度快了，而且不用一直开机等着。而且在此过程中，我也学会了 screen 让进程在后台运行的方法，大大加深了我对连接服务器的理解。

这一次实验让我收获最大的莫过于知道了如何完成对于内核的编译。其实很迷惑为什么 make install 之前要先 make modules_install,后来上网查找资料，我明白了这是要为安装添加一些驱动。如果没有这些驱动，就无法完成对于内核的安装。

最后看到自己的内核终于安装成功，uname -r 的结果显示正确，还是非常有成就感的。

完成实验后回答问题：

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？**

第二次：

```
[  416.145739] page fault of system:257353
[  416.145741] page fault of current process: 8168
[  416.145741] dirty page of all process:
```

第三次：

```
[  932.444711] page fault of system:273961
[  932.444713] page fault of current process: 8475
[  932.444714] dirty page of all process:
```

第四次：

```
[  977.233010] page fault of system:274791
[  977.233020] page fault of current process: 8723
[  977.233020] dirty page of all process:
```

第五次：

```
[ 1035.040471] page fault of system:280201
[ 1035.040473] page fault of current process: 8831
[ 1035.040473] dirty page of all process:
```

第六次：

```
[ 1061.831373] page fault of system:282228
[ 1061.831375] page fault of current process: 8942
[ 1061.831375] dirty page of all process:
```

从上图可以看出，系统缺页次数在变化，而当前进程缺页次数维持在 8000-9000 的左右。

系统缺页次数和当前进程缺页次数不是操作系统原理上的缺页次数。修改系统调用统计的“缺页次数”实际上是页错误次数，即调用 do_page_fault 函数的次数，而操作系统原理上的缺页次数是进程的物理块数×页面置换次数。

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。

通过内核模块实现添加系统调用。这种方法其实是系统调用拦截的实现。系统调用服务程序的地址是放在 sys_call_table 中通过系统调用号定位到具体的系统调用地址，那么我们通过编写内核模块来修改 sys_call_table 中的系统调用的地址为我们自己定义的函数的地址，就可以实现系统调用的拦截。

通过模块加载时，将系统调用表里面的那个系统调用号的那个系统调用号对应的系统调用服务例程改为我们自己实现的系统历程函数地址。首先要获取 sys_call_table 的地址，然后插入模块，编写用户态程序。

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。

是不安全的。用户进程需要获得系统服务（调用系统程序），这时就必须利用系统提供给用户的“特殊接口”——系统调用了.它的特殊性主要在于规定了用户进程进入内核的具体位置；换句话说，用户访问内核的路径是事先规定好的，只能从规定位置进入内核，而不准许肆意跳入内核。有了这样的陷入内核的统一访问路径限制才能保证内核安全无虞。

如果用户能随意修改系统调用，就有可能破坏这种安全性。导致系统出现故障。