

---

# Lab2 - 添加系统调用

## 1 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

## 2 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：

- 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

### 3 实验环境

- Linux 版本: Ubuntu-14.04-64

多次尝试在 VMware Workstation 上安装 Ubuntu-13.04-64 以及 Ubuntu-13.04, 然而没有一次成功, 于是选择了比较接近的 Ubuntu-14.04-64。

- Linux 内核版本: Linux-3.11.4

- 虚拟机配置

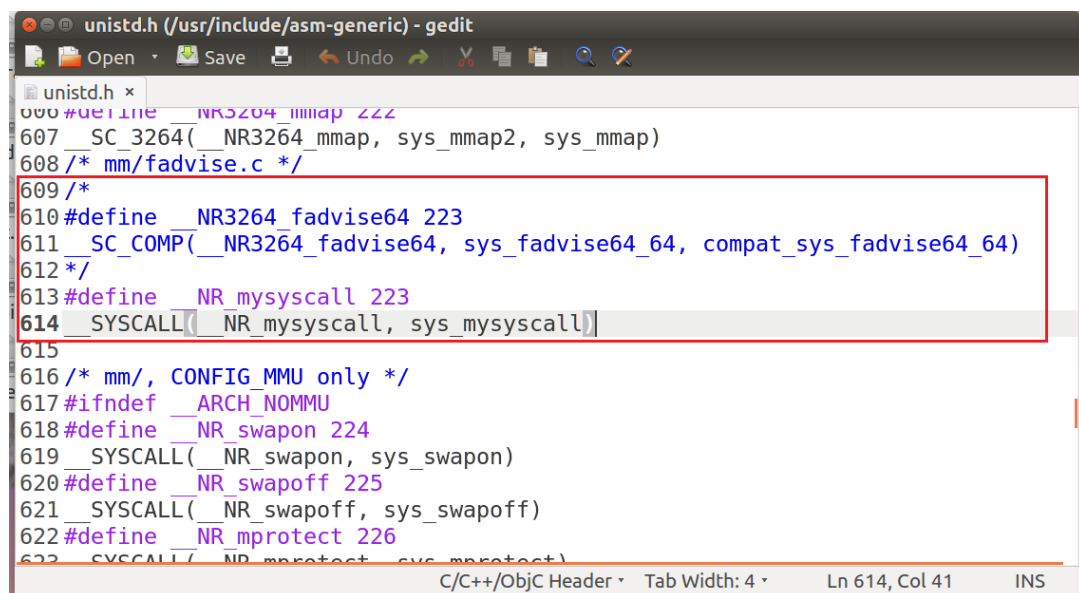
内存: 2G

硬盘: 20G

处理器: 1 个, 2 核

### 4 实验过程与结果

- 1) 下载内核源码 `linux-3.11.4.tar.xz`, 解压到文件夹 `linux-3.11.4`
- 2) 修改系统文件 `/usr/include/asm-generic/unistd.h`, 找到 223 号定义的位置, 注释掉原来的内容, 将 223 号定义到要添加的系统调用



```
unistd.h (/usr/include/asm-generic) - gedit
607 __SC_3264(__NR3264_mmap, sys_mmap2, sys_mmap)
608 /* mm/fadvise.c */
609 /*
610 #define __NR3264_fadvise64 223
611 __SC_COMP(__NR3264_fadvise64, sys_fadvise64_64, compat_sys_fadvise64_64)
612 */
613 #define __NR_mysyscall 223
614 __SYSCALL(__NR_mysyscall, sys_mysyscall)
615
616 /* mm/, CONFIG_MMU only */
617 #ifndef __ARCH_NOMMU
618 #define __NR_swapon 224
619 __SYSCALL(__NR_swapon, sys_swapon)
620 #define __NR_swapoff 225
621 __SYSCALL(__NR_swapoff, sys_swapoff)
622 #define __NR_mprotect 226
623 __SYSCALL(__NR_mprotect, sys_mprotect)
```

修改内核文件 `include/uapi/asm-generic/unistd.h`, 作与系统文件 `unistd.h` 相同的操作

```
unistd.h (~/Desktop/lab2/linux-3.11.4/include/uapi/asm-generic) - gedit
unistd.h x
000 #define __NR3264 mmap zzz
607 __SC_3264(__NR3264_mmap, sys_mmap2, sys_mmap)
608 /* mm/fadvise.c */
609 /*
610 #define __NR3264_fadvise64 223
611 __SC_COMP(__NR3264_fadvise64, sys_fadvise64_64, compat_sys_fadvise64_64)
612 */
613 #define __NR_mysyscall 223
614 __SYSCALL(__NR_mysyscall, sys_mysyscall)
615
616 /* mm/, CONFIG MMU only */
617 #ifndef __ARCH_NOMMU
618 #define __NR_swapon 224
619 __SYSCALL(__NR_swapon, sys_swapon)
620 #define __NR_swapoff 225
621 __SYSCALL(__NR_swapoff, sys_swapoff)
622 #define __NR_mprotect 226
623 __SYSCALL(__NR_mprotect, sys_mprotect)
C/C++/ObjC Header Tab Width: 4 Ln 614, Col 41 INS
```

- 3) 在系统调用表中添加 223 号调用，在内核源码中找到文件 `arch/x86/syscalls/syscall_32.tbl`，将 223 号调用映射到要添加的调用（该行原本是注释“223 is unused”）

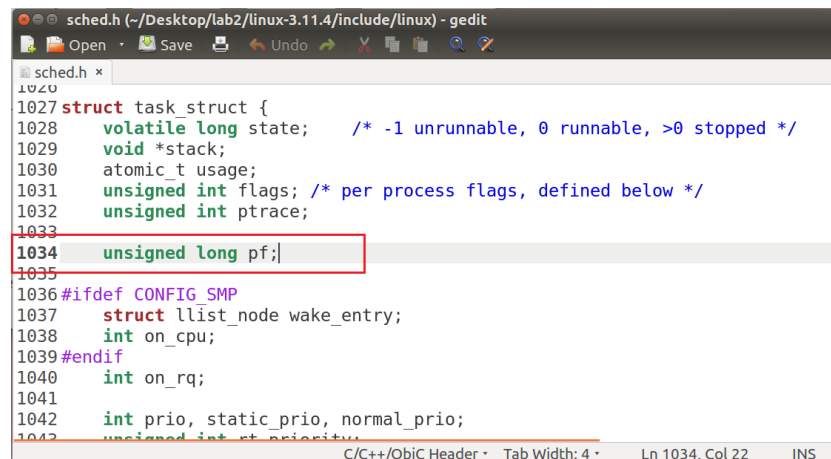
```
syscall_32.tbl (~/Desktop/lab2/linux-3.11.4/arch/x86/syscalls) - gedit
syscall_32.tbl x
224 215 i386 setfsuid32 sys_setfsuid
225 216 i386 setfsgid32 sys_setfsgid
226 217 i386 pivot_root sys_pivot_root
227 218 i386 mincore sys_mincore
228 219 i386 madvise sys_madvise
229 220 i386 getdents64 sys_getdents64 compat_sys_getdents64
230 221 i386 fcntl64 sys_fcntl64 compat_sys_fcntl64
231 # 222 is unused
232 223 i386 mysyscall sys_mysyscall
233 224 i386 gettid sys_gettid
234 225 i386 readahead sys_readahead sys32_readahead
235 226 i386 setxattr sys_setxattr
236 227 i386 lsetxattr sys_lsetxattr
237 228 i386 fsetxattr sys_fsetxattr
238 229 i386 getxattr sys_getxattr
239 230 i386 lgetxattr sys_lgetxattr
240 231 i386 fgetxattr sys_fgetxattr
241 232 i386 lgetxattr sys_lgetxattr
Plain Text Tab Width: 4 Ln 231, Col 16 INS
```

- 4) 修改统计系统缺页次数和进程缺页次数的内核代码

先在 `include/linux/mm.h` 文件中声明外部变量 `pfcount`

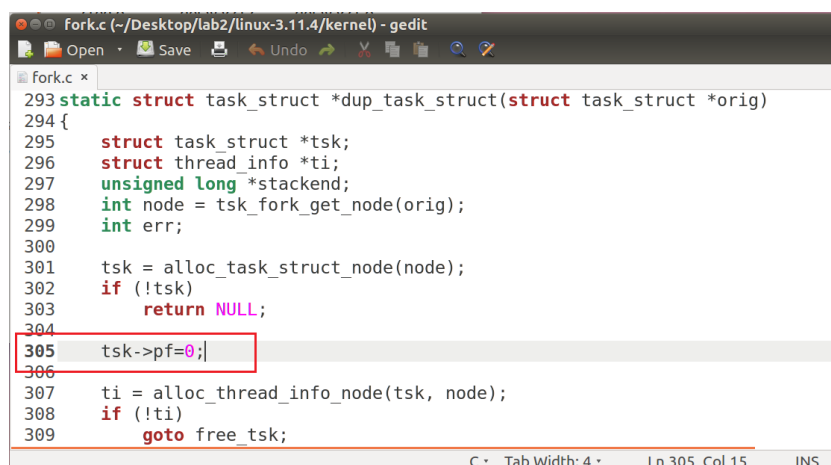
```
mm.h (~/Desktop/lab2/linux-3.11.4/include/linux) - gedit
mm.h x
20
21 struct mempolicy;
22 struct anon_vma;
23 struct anon_vma_chain;
24 struct file_ra_state;
25 struct user_struct;
26 struct writeback_control;
27
28 extern unsigned long pfcount;
29
30 #ifndef CONFIG_NEED_MULTIPLE_NODES /* Don't use mapnrs, do it properly */
31 extern unsigned long max_mapnr;
32
33 static inline void set_max_mapnr(unsigned long limit)
34 {
35     max_mapnr = limit;
36 }
37 #else
C/C++/ObjC Header Tab Width: 4 Ln 28, Col 30 INS
```

在进程结构的原型 `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



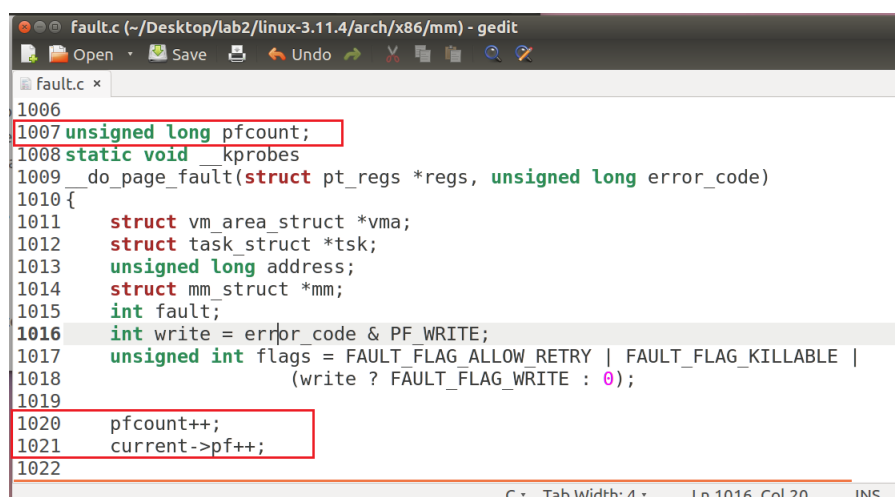
```
1027 struct task_struct {
1028     volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
1029     void *stack;
1030     atomic_t usage;
1031     unsigned int flags; /* per process flags, defined below */
1032     unsigned int ptrace;
1033
1034     unsigned long pf;
1035
1036 #ifdef CONFIG_SMP
1037     struct llist_node wake_entry;
1038     int on_cpu;
1039 #endif
1040     int on_rq;
1041
1042     int prio, static_prio, normal_prio;
1043     unsigned int rt_priority;
```

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, 修改该函数使得子进程被创建的时候 `pf` 被设置成 0



```
293 static struct task_struct *dup_task_struct(struct task_struct *orig)
294 {
295     struct task_struct *tsk;
296     struct thread_info *ti;
297     unsigned long *stackend;
298     int node = tsk_fork_get_node(orig);
299     int err;
300
301     tsk = alloc_task_struct_node(node);
302     if (!tsk)
303         return NULL;
304
305     tsk->pf=0;
306
307     ti = alloc_thread_info_node(tsk, node);
308     if (!ti)
309         goto free_tsk;
```

修改 `arch/x86/mm/fault.c` 文件, 定义变量 `pfcount`, 修改 `_do_page_fault()` 函数使得每次产生缺页中断, `pfcount` 递增 1, `current->pf` 递增 1



```
1006
1007 unsigned long pfcount;
1008 static void __kprobes
1009 _do_page_fault(struct pt_regs *regs, unsigned long error_code)
1010 {
1011     struct vm_area_struct *vma;
1012     struct task_struct *tsk;
1013     unsigned long address;
1014     struct mm_struct *mm;
1015     int fault;
1016     int write = error_code & PF_WRITE;
1017     unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE |
1018         (write ? FAULT_FLAG_WRITE : 0);
1019
1020     pfcount++;
1021     current->pf++;
1022 }
```

5) 实现 `sys_mysyscall`, 在 `kernel/sys.c` 文件中添加相应代码

```
sys.c (~/Desktop/lab2/linux-3.11.4/kernel) - gedit
2155     __put_user(s.procs, &info->procs) ||
2156     __put_user(s.totalhigh, &info->totalhigh) ||
2157     __put_user(s.freehigh, &info->freehigh) ||
2158     __put_user(s.mem_unit, &info->mem_unit))
2159     return -EFAULT;
2160
2161     return 0;
2162 }
2163
2164 extern unsigned long pfcount;
2165 asmlinkage int sys_mysyscall(void)
2166 {
2167     printk("page fault count for total: %lu\n", pfcount);
2168     printk("page fault count for current process: %lu\n", current->pf);
2169     return 0;
2170 }
2171 #endif /* CONFIG_COMPAT */
```

6) 重新编译内核

先生成配置文件, 执行 `cp /boot/config-`uname -r` .config`

将命令行窗口调到全屏, 然后执行 `make menuconfig`

使用默认配置, 直接退出

执行 `make`, 等待

执行 `sudo make modules_install`, 安装生成的内核模块

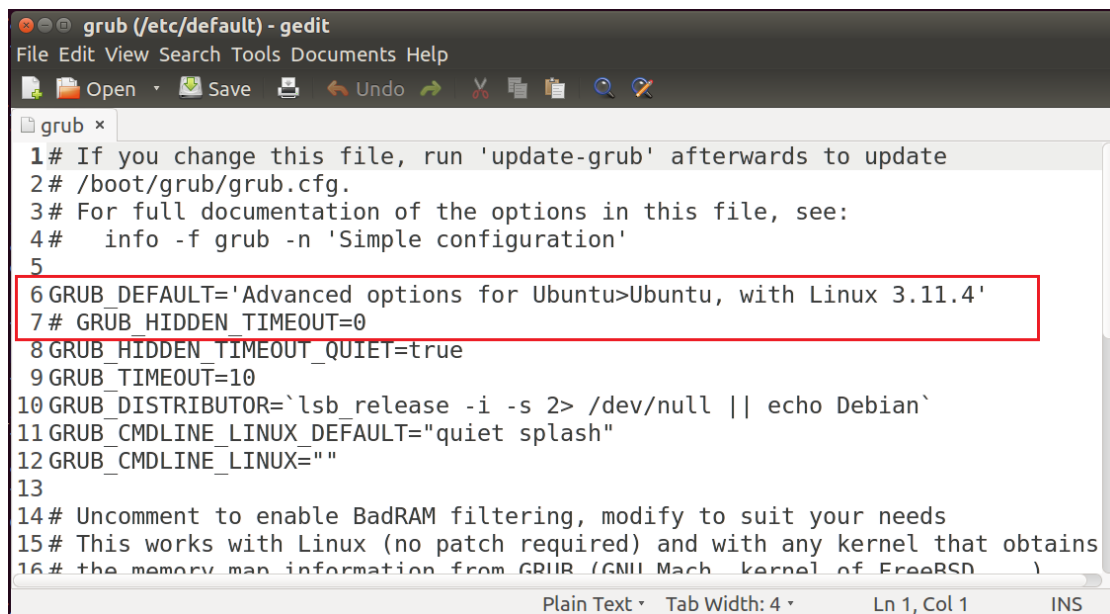
执行 `sudo make install`, 安装内核

执行 `grep menuentry /boot/grub/grub.cfg`, 查看可用的内核, 发现已经有了刚编译好的 Linux 3.11.4 版本的内核

```
oslab@oslab-ubuntu-vm:~/Desktop/lab2/linux-3.11.4$ grep menuentry /boot/grub/grub.cfg
if [ x"${feature_menuentry_id}" = xy ]; then
    menuentry_id_option="--id"
    menuentry_id_option=""
export menuentry_id_option
menuentry 'Ubuntu' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-simple-5b287000-6dc2-43d6-98c4-fbc9c064d999' {
    submenu 'Advanced options for Ubuntu' $menuentry_id_option 'gnulinux-advanced-5b287000-6dc2-43d6-98c4-fbc9c064d999' {
        menuentry 'Ubuntu, with Linux 3.13.0-24-generic' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-3.13.0-24-generic-advanced-5b287000-6dc2-43d6-98c4-fbc9c064d999' {
            menuentry 'Ubuntu, with Linux 3.13.0-24-generic (recovery mode)' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-3.13.0-24-generic-recovery-5b287000-6dc2-43d6-98c4-fbc9c064d999' {
                menuentry 'Ubuntu, with Linux 3.11.4' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-3.11.4-advanced-5b287000-6dc2-43d6-98c4-fbc9c064d999' {
                    menuentry 'Ubuntu, with Linux 3.11.4 (recovery mode)' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-3.11.4-recovery-5b287000-6dc2-43d6-98c4-fbc9c064d999' {
                        menuentry 'Memory test (memtest86+)' {
                        menuentry 'Memory test (memtest86+, serial console 115200)' {
oslab@oslab-ubuntu-vm:~/Desktop/lab2/linux-3.11.4$
```

7) 更换系统内核

执行 `sudo gedit /etc/default/grub`, 修改启动信息



```
grub (/etc/default) - gedit
File Edit View Search Tools Documents Help
Open Save Undo Cut Copy Paste Find
grub x
1# If you change this file, run 'update-grub' afterwards to update
2# /boot/grub/grub.cfg.
3# For full documentation of the options in this file, see:
4# info -f grub -n 'Simple configuration'
5
6 GRUB_DEFAULT='Advanced options for Ubuntu>Ubuntu, with Linux 3.11.4'
7 # GRUB_HIDDEN_TIMEOUT=0
8 GRUB_HIDDEN_TIMEOUT_QUIET=true
9 GRUB_TIMEOUT=10
10 GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
11 GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
12 GRUB_CMDLINE_LINUX=""
13
14# Uncomment to enable BadRAM filtering, modify to suit your needs
15# This works with Linux (no patch required) and with any kernel that obtains
16# the memory map information from GRUB (GNU Mach, kernel of FreeBSD ...)

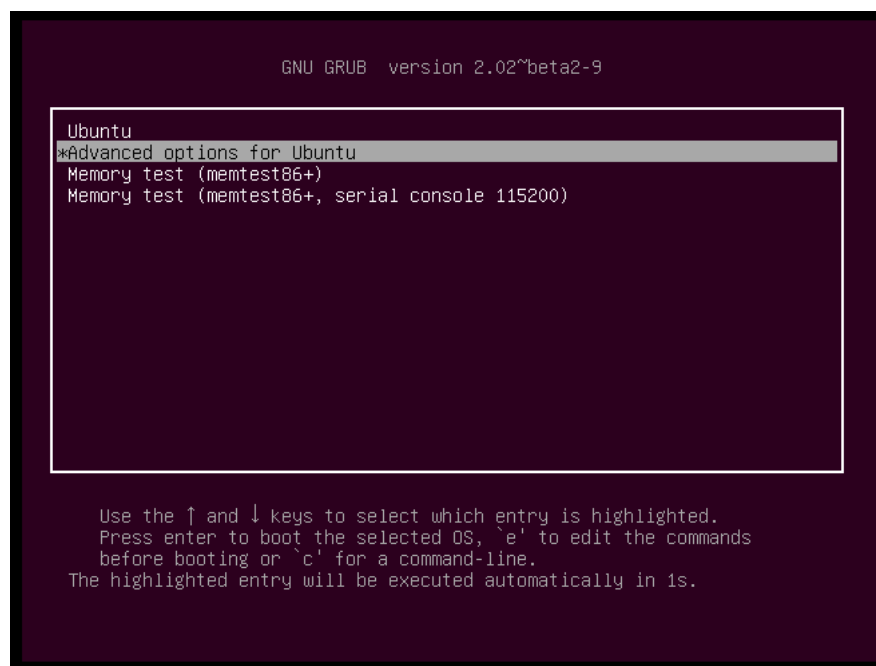
Plain Text Tab Width: 4 Ln 1, Col 1 INS
```

执行 `sudo update-grub`，更新启动信息

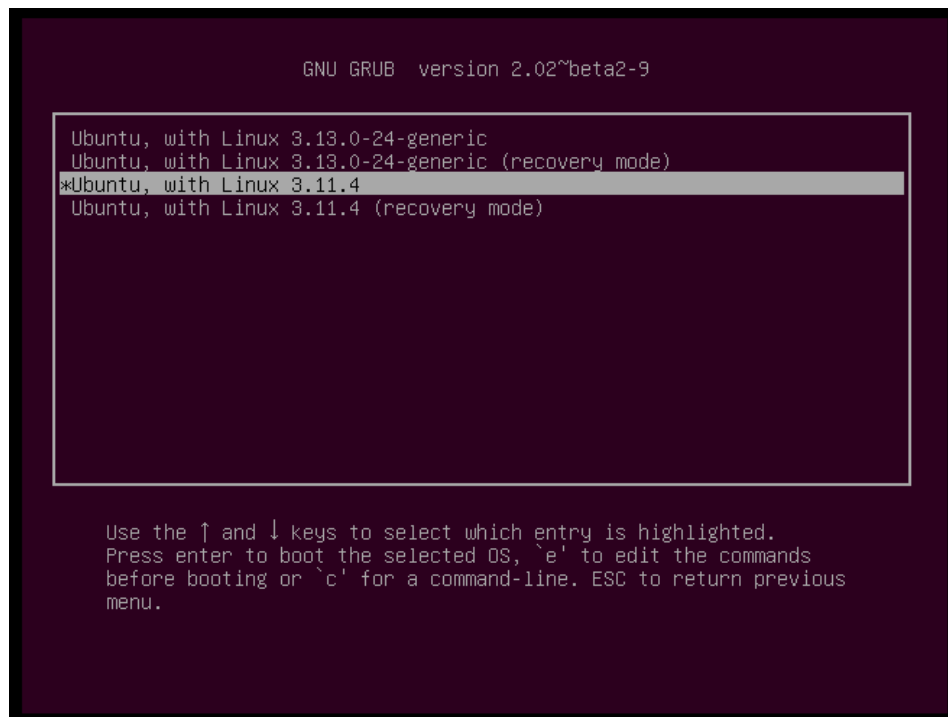
```
oslab@oslab-ubuntu-vm:~/Desktop/lab2/linux-3.11.4$ sudo update-grub
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-3.13.0-24-generic
Found initrd image: /boot/initrd.img-3.13.0-24-generic
Found linux image: /boot/vmlinuz-3.11.4
Found initrd image: /boot/initrd.img-3.11.4
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
oslab@oslab-ubuntu-vm:~/Desktop/lab2/linux-3.11.4$
```

执行 `sudo reboot`，重启

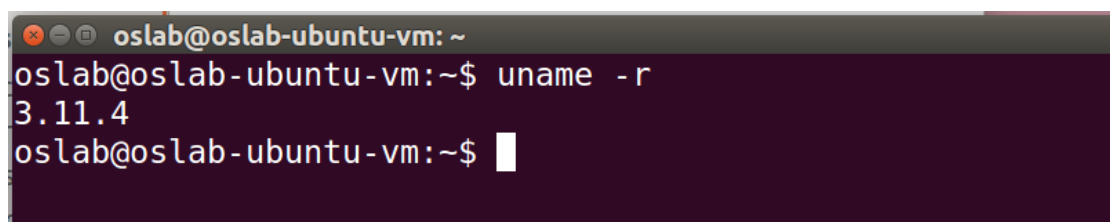
进入内核选择界面



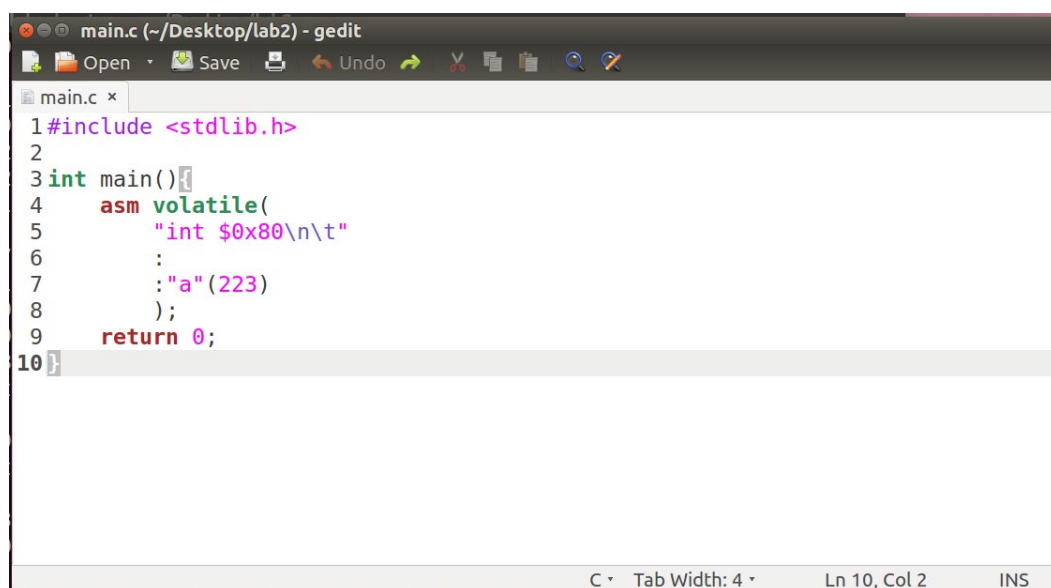
选择刚编译好的 Linux-3.11.4 内核



开机登陆后，执行 `uname -r` 查看内核版本号，此时内核已切换成功



## 8) 编写用户程序调用添加的系统调用



编译，运行，查看结果

```
oslab@oslab-ubuntu-vm: ~/Desktop/lab2
oslab@oslab-ubuntu-vm:~$ cd Desktop/lab2/
oslab@oslab-ubuntu-vm:~/Desktop/lab2$ gcc main.c
oslab@oslab-ubuntu-vm:~/Desktop/lab2$ ./a.out
oslab@oslab-ubuntu-vm:~/Desktop/lab2$ dmesg
```

可以看到系统调用被成功调用，系统和当前进程的缺页统计被输出到了日志文件

```
hostname!
[ 3035.727638] e1000: eth0 NIC Link is Down
[ 3041.742305] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: N
one
[ 3075.526327] page fault count for total: 1687649
[ 3075.526329] page fault count for current process: 214561
oslab@oslab-ubuntu-vm:~/Desktop/lab2$
```

## 5 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？

程序打印的缺页次数并不是操作系统原理上的缺页次数，修改内核后系统调用统计的是 `__do_page_fault()` 函数执行的次数，即页访问出错的次数。而操作系统上的缺页次数应该是页面置换次数乘以物理块数。

- 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。

可以通过对系统调用进行拦截而实现。由于系统调用程序的地址存储于系统调用表中，因此可以通过修改表中的系统调用地址，使之成为自己实现的函数地址。

- 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。

不安全。

因为系统调用是属于操作系统的特殊接口，用于进程获取系统服务的功能。正确的系统调用能够保证用户程序按照规定的逻辑访问内核，如果系统调用出现异常，用户程序可能不会以正确的方式访问内核，从而出现问题，导致系统崩溃。



## 6 遇到的问题与解决方案

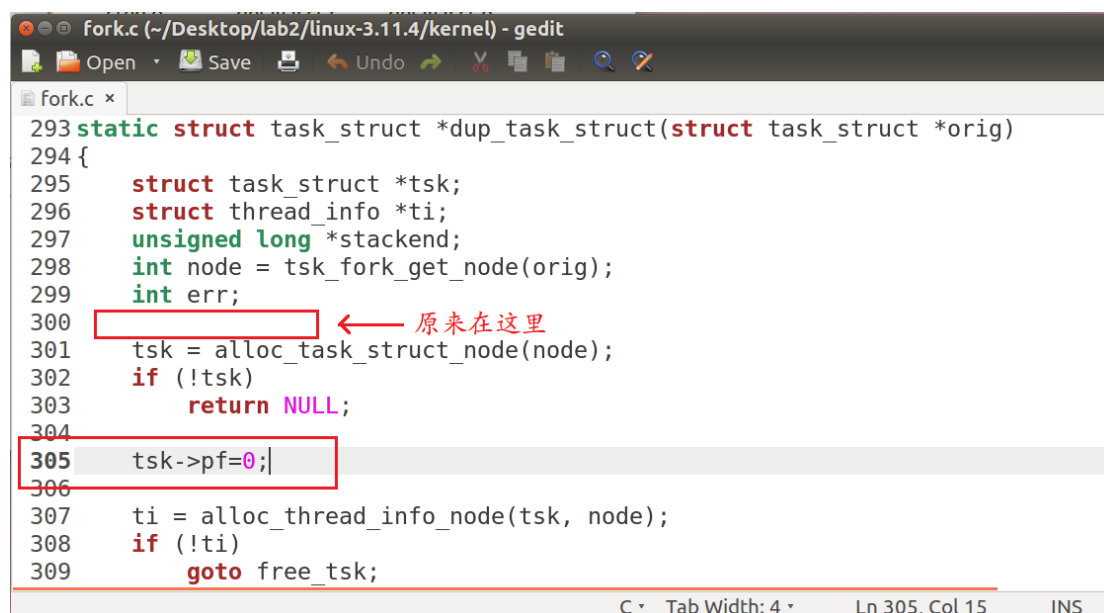
### 1) Ubuntu-13.04 无法安装在 VMware Workstation 上

先多次尝试了安装 Ubuntu-13.04-64，又多次尝试了安装 Ubuntu-13.04，均失败，然后换了 Ubuntu-14.04-64，一次成功。

### 2) 切换内核失败

第一次是编译内核失败：在 `kernel/sys.c` 中，实现 `sys_mysyscall` 的细节的时候，未声明外部变量 `pfcount` 就在函数中引用，然后报出了 **undefined reference** 的错误，于是在函数上方添加 `extern unsigned long pfcount;`。

第二次是启用新内核失败：编译安装都已经完成了，但是发现不能正常重启，发现在修改 `kernel/fork.c` 文件中的 `dup_task_struct()` 函数，使新建的子进程的 `pf` 值被设置为 0 的时候，把 `tsk->pf=0;` 语句放在了 `tsk` 的 `alloc` 前，也就是操作了野指针，于是将该语句移到了 `alloc` 和判断 `tsk` 是否为空的下面。



```
fork.c (~/Desktop/lab2/linux-3.11.4/kernel) - gedit
Open Save Undo Redo
fork.c x
293 static struct task_struct *dup_task_struct(struct task_struct *orig)
294 {
295     struct task_struct *tsk;
296     struct thread_info *ti;
297     unsigned long *stackend;
298     int node = tsk_fork_get_node(orig);
299     int err;
300     tsk = alloc_task_struct_node(node);
301     if (!tsk)
302         return NULL;
303
304     tsk->pf=0;
305
306     ti = alloc_thread_info_node(tsk, node);
307     if (!ti)
308         goto free_tsk;
309 }
```

### 3) 重启后发现还是原来的内核

重启时没有经历重新选择内核的过程，重启后查看内核版本发现还是原来的内核版本，发现是因为没有修改启动信息，于是修改 `/etc/default/grub` 文件，并更新启动信息，在启动的时候选择新的内核。

### 4) 用户程序调用新添加的系统调用无输出

用 `syscall()` 的方式调用新添加的 `mysyscall`，但是用 `dmesg` 查看消息日志时却发现没有预期的输出。

查阅资料后想起修改系统调用表的时候只修改了 32 位的系统调用表，但是 `syscall()` 应该调用的是 64 位的系统调用，于是改用 `int 0x80` 直接调用 223 号系统调用，得到了预期的输出。

---

## 7 心得与体会

本次实验历时 2 天，主要时间都花在了编译内核上，一共重来了 4 次，感觉还是比较耗时间的一次实验，当然在实验中也学到了很多，包括 Linux 内核的编译与安装，Linux 系统调用的实现框架、用户界面、参数传递、进入/返回过程，以及对 Linux 内核源代码的初步了解。

实验中遇到了很多问题，也查阅了很多资料，学到了很多关于虚拟机的使用技巧，比如可以通过创建快照来保存虚拟机的状态，在出现重大错误的时候就可以及时回退。

经过这次实验，对 Linux 系统的基本操作熟悉了很多，对 Ubuntu 的用户界面也基本了解了。实验中也发现自己还有很多不足，应该自勉，争取在之后的实验和课程学习中做的更好。