

浙江大学

本科实验报告

课程名称：操作系统

姓 名：应承峻

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：软件工程

学 号：3170103456

指导教师：夏莹杰

2019 年 11 月 16 日

浙江大学实验报告

课程名称： 操作系统 实验类型： 综合

实验项目名称： 添加系统调用

学生姓名： 应承峻 专业： 软件工程 学号： 3170103456

电子邮件地址： 3170103456@zju.edu.cn 手机： 17326084929

实验地点： 玉泉曹光彪西 503 实验日期： 2019 年 11 月 16 日

一、实验目的和要求

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

笔记本电脑 1 台，相关配置如下：

处理器 英特尔 Core i7-8750H @ 2.20GHz 六核

内 存 16 GB （三星 DDR4 2667MHz）

主硬盘 PeM280240GP4C15B （240 GB/固态硬盘）

显 卡 Nvidia GeForce GTX 1060 （6 GB）

操作系统环境：Windows 10 64 位（DirectX 12）

Linux 版本： ubuntu-16.04

四、操作方法和实验步骤

1. 下载一份内核源代码

Linux 受 GNU 通用公共许可证（GPL）保护，其内核源代码是完全开放的。现在很多 Linux 的网站都提供内核代码的下载。推荐你使用 Linux 的官方网站：<http://www.kernel.org>。在这里你可以找到所有的内核版本。

在这里选择 4.6 内核版本下载

2. 部署内核源代码

此过程比较机械、枯燥，因而容易出错。请严格按下述步骤来操作。

首先，把 linux-4.6.tar.xz 包放在主目录下，解开 linux-4.6.tar.xz 包：

```
tar -xvf linux-4.6.tar.xz
```

解压出来的内核代码存放在/usr/src/linux-4.16 目录下。为了方便操作及一致性，可以通过路径/usr/src/linux 去访问它，这只要建一个符号链接：

```
ln -s /usr/src/linux-4.6/ /usr/src/linux
```



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:

```
sudo apt-get install libncurses5-dev
```

若上面这一步提示错误信息, 则输入下面的命令 `sudo apt-get -f install`, 建立库依赖关系。

查看 README 文件:

在你进行这项工作之前, 不妨先看一看 `/usr/src/linux` 目录下内核源代码自带的 README 文件。在这份文件中, 对怎样进行内核的解压, 配置, 安装都进行了详细的讲解。

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 /usr/src/linux
```

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
```

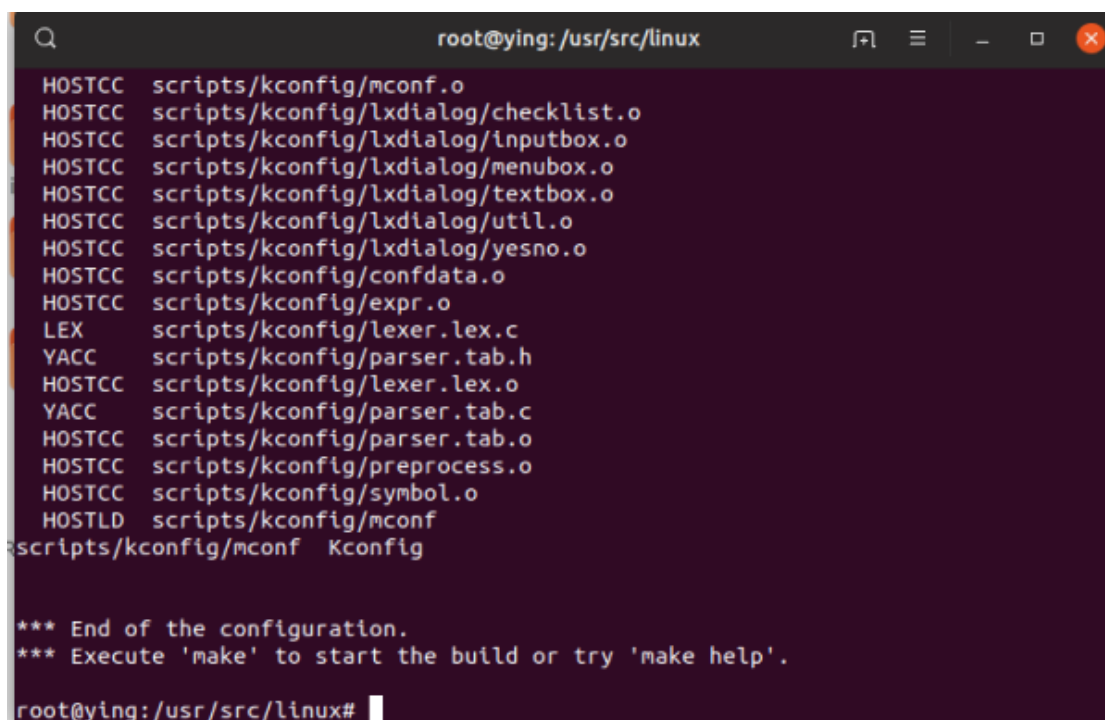
为了与正在运行的操作系统内核的运行环境匹配, 可以先把当前已配置好的文件复制到当前目录下, 新的文件名为 `.config` 文件:

```
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
```

```
root@ying:/usr/src# cd linux
root@ying:/usr/src/linux# make mrproper
root@ying:/usr/src/linux# cp /boot/config-`uname -r` .config
```



```
root@ying: /usr/src/linux
HOSTCC scripts/kconfig/mconf.o
HOSTCC scripts/kconfig/lxdialog/checklist.o
HOSTCC scripts/kconfig/lxdialog/inputbox.o
HOSTCC scripts/kconfig/lxdialog/menubox.o
HOSTCC scripts/kconfig/lxdialog/textbox.o
HOSTCC scripts/kconfig/lxdialog/utill.o
HOSTCC scripts/kconfig/lxdialog/yesno.o
HOSTCC scripts/kconfig/confdata.o
HOSTCC scripts/kconfig/expr.o
LEX scripts/kconfig/lexer.lex.c
YACC scripts/kconfig/parser.tab.h
HOSTCC scripts/kconfig/lexer.lex.o
YACC scripts/kconfig/parser.tab.c
HOSTCC scripts/kconfig/parser.tab.o
HOSTCC scripts/kconfig/preprocess.o
HOSTCC scripts/kconfig/symbol.o
HOSTLD scripts/kconfig/mconf
scripts/kconfig/mconf Kconfig

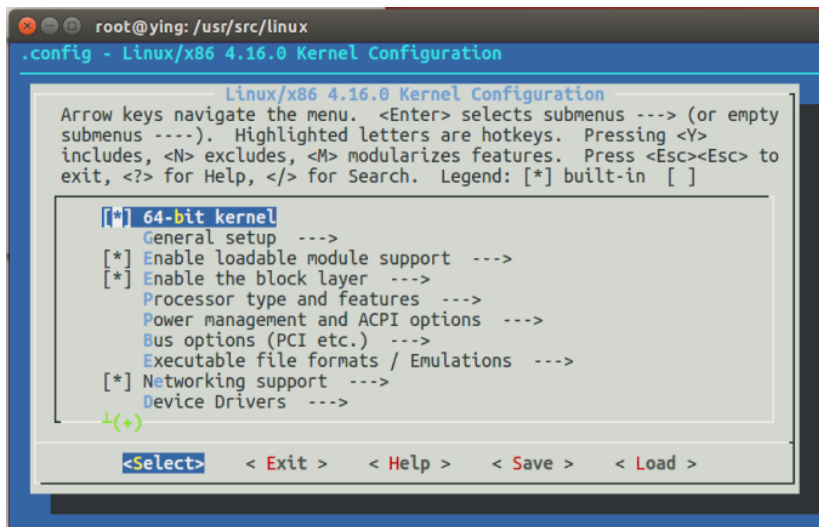
*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.
root@ying:/usr/src/linux#
```

进行配置时，大部分选项可以使用其缺省值，只有小部分需要根据用户不同的需要选择。例如，如果硬盘分区采用 ext2 文件系统（或 ext3 文件系统），则配置项应支持 ext2 文件系统（ext3 文件系统）。又例如，系统如果配有 SCSI 总线及设备，需要在配置中选择 SCSI 卡的支持。

对每一个配置选项，用户有三种选择，它们分别代表的含义如下：

- “<*>”或“[*]” — 将该功能编译进内核
- “[]” — 不将该功能编译进内核
- “[M]” — 将该功能编译成可以在需要时动态插入到内核中的模块

将与核心其它部分关系较远且不经常使用的部分功能代码编译成为可加载模块，有利于减小内核的长度，减小内核消耗的内存，简化该功能相应的环境改变时对内核的影响。许多功能都可以这样处理，例如像上面提到的对 SCSI 卡的支持，等等。



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/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` 文件中的查找定义 223 号的行，作如下修改：

```
-- #define __NR3264_fadvise64 223
-- __SC_COMP(__NR3264_fadvise64, sys_fadvise64_64, compat_sys_fadvise64_64)

609 /* mm/fadvise.c */
610 #define __NR3264_fadvise64 223
611 __SC_COMP(__NR3264_fadvise64, sys_fadvise64_64, compat_sys_fadvise64_64)
612
613 /* mm/. CONFIG MMU onlv */

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

608 __SC_3264(__NR3264_mmap, sys_mmap2, sys_mmap)
609 /* mm/fadvise.c */
610 #define __NR_mysyscall 223
611 __SYSCALL(__NR_mysyscall, sys_mysyscall)
612
```

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

```
#define __NR_mysyscall 223
```

`__SYSCALL(__NR_mysyscall, sys_mysyscall)`

注意：不同版本的内核，需要修改路径（子目录名和文件名）可能不一样，根据实际版本号查找这些.h 文件。系统调用号也可能不一样，可以根据内核版本不同对系统调用号进行修改。

添加系统调用号之后，系统才能根据这个号，作为索引，去找 `syscall_table` 中的相应表项。所以说，我们接下来的一步就是：

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。

[arch/x86/entry/syscalls/syscall_32.tbl](#)

222 is unused

223 i386 mysyscall sys_mysyscall

224 i386 gettid sys_gettid

225 i386 readahead sys_readahead

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。

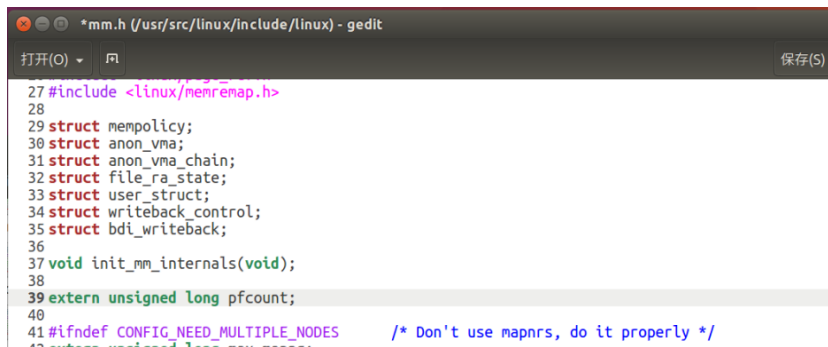
219	i386	madvise	sys_madvise	
220	i386	getdents64	sys_getdents64	
221	i386	fcntl64	sys_fcntl64	compat_sys_fcntl64
# 222 is unused				
# 223 is unused				
223	i386	mysyscall	sys_mysyscall	
224	i386	gettid	sys_gettid	
225	i386	readahead	sys_readahead	compat_sys_x86_readahead
226	i386	setxattr	sys_setxattr	
227	i386	lsetxattr	sys_lsetxattr	
228	i386	fsetxattr	sys_fsetxattr	
229	i386	getxattr	sys_getxattr	

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`：

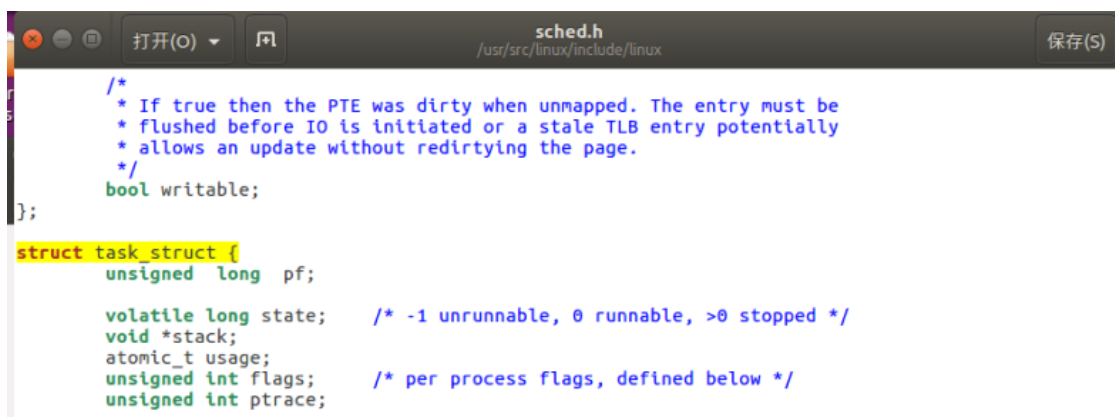
```
++ extern unsigned long pfcount;
```



```
27 #include <linux/memremap.h>
28
29 struct mempolicy;
30 struct anon_vma;
31 struct anon_vma_chain;
32 struct file_ra_state;
33 struct user_struct;
34 struct writeback_control;
35 struct bdi_writeback;
36
37 void init_mm_internals(void);
38
39 extern unsigned long pfcount;
40
41 #ifndef CONFIG_NEED_MULTIPLE_NODES    /* Don't use mapnr, do it properly */
42 extern unsigned long pfn_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;
```



```
/*
 * If true then the PTE was dirty when unmapped. The entry must be
 * flushed before IO is initiated or a stale TLB entry potentially
 * allows an update without redirtying the page.
 */
bool writable;
};

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;
    .....
}
```

```

#endif

/*
 * One for us, one for whoever does the "release_task()" (usually
 * parent)
 */
atomic_set(&tsk->usage, 2);
#ifdef CONFIG_BLK_DEV_IO_TRACE
tsk->btrace_seq = 0;
#endif

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;

free_ti:
free_thread_info(ti);
free_tsk:
free_task_struct(tsk);
return NULL;
}

```

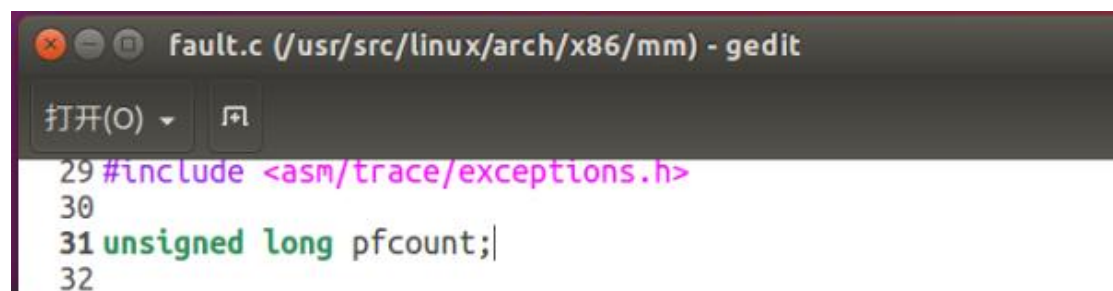
在 arch/x86/mm/fault.c 文件中定义变量 pfcount; 并修改 arch/x86/mm/fault.c 中 do_page_fault()函数。每次产生缺页中断,do_page_fault()函数会被调用,pfcount 变量值递增 1,记录系统产生缺页次数, current->pf 值递增 1, 记录当前进程产生缺页次数:

```

...
++ unsigned long pfcount;

__do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    ...
    ++ pfcount++;
    ++ current->pf++;
    ...
}

```



```

29 #include <asm/trace/exceptions.h>
30
31 unsigned long pfcount;
32

```

```

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

    tsk = current;
    mm = tsk->mm;

    pfcoun++;
    current->pf++;

    /*
     * Detect and handle instructions that would cause a page fault for
     * both a tracked kernel page and a userspace page.
     */
    if (kmemcheck_active(regs))
        kmemcheck_hide(regs);
    prefetchw(&mm->mmap_sem);
}

```

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;
}

```

注：在 `/kernel/sys.c` 中先添加 `#include <linux/linkage.h>` 的头文件，再添加系统调用函数

```

asmlinkage int sys_mysyscall(void) {
    printk(KERN_INFO "System Page Fault Number: %lu\n", pfcoun);
    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->pf);
    }
    return 0;
}

```

```

asmlinkage int sys_mysyscall(void) {

    printk(KERN_INFO "System Page Fault Number: %lu\n", pfcoun);
}

```

```

    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;
}

```

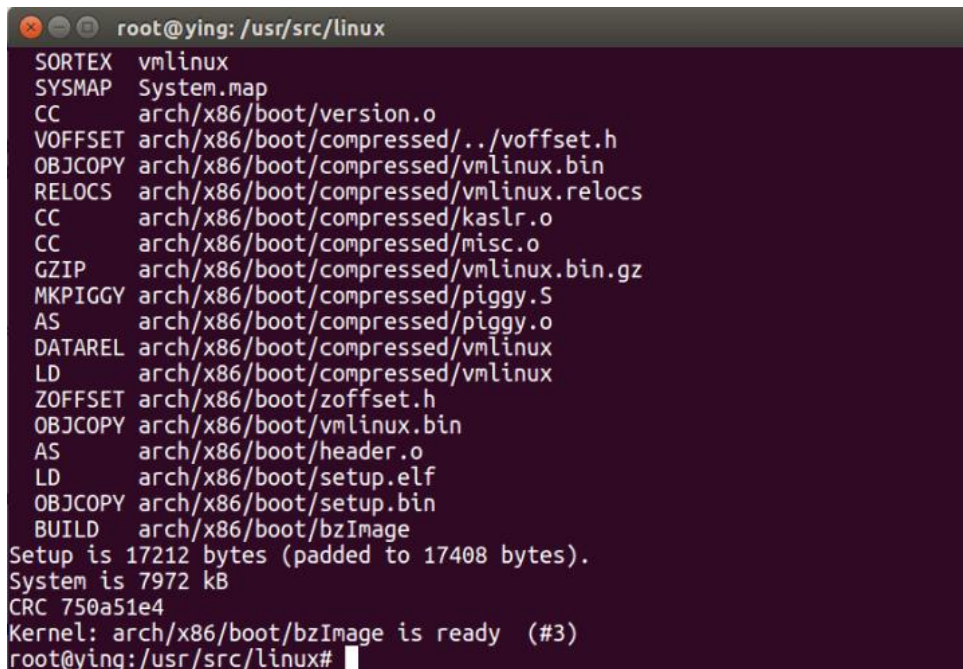
8. 编译内核和重启内核

用 make 工具编译内核：

```
sudo make bzImage -j12
```

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。

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。



```

root@ying: /usr/src/linux
SORTEX  vmlinux
SYSMAP  System.map
CC      arch/x86/boot/version.o
VOFFSET arch/x86/boot/compressed/./voffset.h
OBJCOPY arch/x86/boot/compressed/vmlinux.bin
RELOCS  arch/x86/boot/compressed/vmlinux.relocs
CC      arch/x86/boot/compressed/kaslr.o
CC      arch/x86/boot/compressed/misc.o
GZIP    arch/x86/boot/compressed/vmlinux.bin.gz
MKPIGGY arch/x86/boot/compressed/piggy.S
AS      arch/x86/boot/compressed/piggy.o
DATAREL arch/x86/boot/compressed/vmlinux
LD      arch/x86/boot/compressed/vmlinux
ZOFFSET arch/x86/boot/zoffset.h
OBJCOPY arch/x86/boot/vmlinux.bin
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 17212 bytes (padded to 17408 bytes).
System is 7972 kB
CRC 750a51e4
Kernel: arch/x86/boot/bzImage is ready (#3)
root@ying:/usr/src/linux#

```

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 -j12
```

```
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
```

```
root@ying: /usr/src/linux
System.map "/boot"
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 4.6.0 /boot/vmlinuz-4.6.0
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 4.6.0 /boot/vmlinuz-4.6.0
update-initramfs: Generating /boot/initrd.img-4.6.0
run-parts: executing /etc/kernel/postinst.d/pm-utils 4.6.0 /boot/vmlinuz-4.6.0
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 4.6.0 /boot/vmlinuz-4.6.0
run-parts: executing /etc/kernel/postinst.d/update-notifier 4.6.0 /boot/vmlinuz-4.6.0
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 4.6.0 /boot/vmlinuz-4.6.0
Generating grub configuration file ...
Warning: Setting GRUB_TIMEOUT to a non-zero value when GRUB_HIDDEN_TIMEOUT is set is no longer supported.
Found linux image: /boot/vmlinuz-4.6.0
Found initrd image: /boot/initrd.img-4.6.0
Found linux image: /boot/vmlinuz-4.4.0-21-generic
Found initrd image: /boot/initrd.img-4.4.0-21-generic
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 5.3.0 -o /boot/initrd.img-5.3.0
```

```
sudo update-grub2
```

```
done
root@ying: /usr/src/linux# mkinitramfs 4.6.0 -o /boot/initrd.img-4.6.0
root@ying: /usr/src/linux# sudo update-grub2
Generating grub configuration file ...
Warning: Setting GRUB_TIMEOUT to a non-zero value when GRUB_HIDDEN_TIMEOUT is set is no longer supported.
Found linux image: /boot/vmlinuz-4.6.0
Found initrd image: /boot/initrd.img-4.6.0
Found linux image: /boot/vmlinuz-4.4.0-21-generic
Found initrd image: /boot/initrd.img-4.4.0-21-generic
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 msyscall 系统调用。mysyscall 系统调用中 printf 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

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

- 用 gcc 编译源程序
gcc -o test test.c
- 运行程序
./test

代码如下：

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

#include <sys/syscall.h>

#include<stdio.h>

#define __NR_mysyscall 223

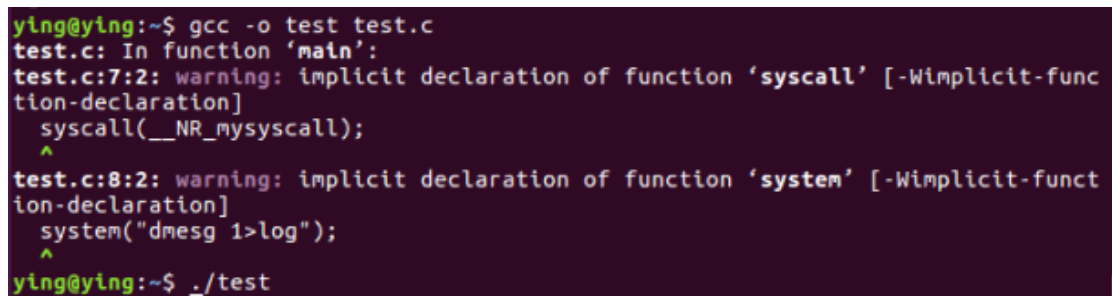
int main()

{

    syscall(__NR_mysyscall);

    system("dmesg 1>log");

}
```



```
ying@ying:~$ gcc -o test test.c
test.c: In function 'main':
test.c:7:2: warning: implicit declaration of function 'syscall' [-Wimplicit-func
tion-declaration]
  syscall(__NR_mysyscall);
  ^
test.c:8:2: warning: implicit declaration of function 'system' [-Wimplicit-funct
ion-declaration]
  system("dmesg 1>log");
  ^
ying@ying:~$ ./test
```

```

[ 703.171569] System Page Fault Number: 671571
[ 703.171571] Current Process Page Fault Number: 68
[ 703.171572] Each Process:
[ 703.171573] systemd          1      6644
[ 703.171574] kthreadd             2       0
[ 703.171575] ksoftirqd/0          3       0
[ 703.171575] kworker/0:0H         5       0
[ 703.171576] rcu_sched             7       0
[ 703.171577] rcu_bh               8       0
[ 703.171578] migration/0          9       0
[ 703.171579] watchdog/0          10      0
[ 703.171580] cpuhp/0             11      0
[ 703.171580] cpuhp/1             12      0
[ 703.171581] watchdog/1          13      0
[ 703.171582] migration/1          14      0
[ 703.171583] ksoftirqd/1          15      0
[ 703.171584] kworker/1:0H         17      0
[ 703.171584] cpuhp/2             18      0
[ 703.171585] watchdog/2          19      0
[ 703.171586] migration/2          20      0
[ 703.171587] ksoftirqd/2          21      0
[ 703.171587] kworker/2:0          22      0
[ 703.171588] kworker/2:0H         23      0
[ 703.171589] cpuhp/3             24      0
[ 703.171590] watchdog/3          25      0
[ 703.171591] migration/3          26      0
[ 703.171592] ksoftirqd/3          27      0
[ 703.171592] kworker/3:0H         29      0
[ 703.171593] cpuhp/4             30      0
[ 703.171594] watchdog/4          31      0
[ 703.171595] migration/4          32      0
[ 703.171596] ksoftirqd/4          33      0
[ 703.171596] kworker/4:0H         35      0
[ 703.171597] cpuhp/5             36      0
[ 703.171598] watchdog/5          37      0
[ 703.171599] migration/5          38      0

```

```

log (~/) - gedit
打开(O) 保存(S)

[ 766.900410] ext4-rsv-conver 350 0
[ 766.900411] systemd-journal 381 680
[ 766.900413] kauditd         384 0
[ 766.900414] systemd-udevd  392 1804
[ 766.900415] kworker/4:2     423 0
[ 766.900417] systemd-timesyn 480 197
[ 766.900418] kworker/1:1H   534 0
[ 766.900420] kworker/0:2    567 0
[ 766.900421] kworker/3:2    576 0
[ 766.900422] nfit           586 0
[ 766.900423] avahi-daemon   724 276
[ 766.900425] acpid          727 126
[ 766.900426] ModemManager   730 626
[ 766.900427] systemd-logind 734 240
[ 766.900429] avahi-daemon   738 40
[ 766.900430] rsyslogd       749 235
[ 766.900431] dbus-daemon    753 649
[ 766.900433] kworker/1:2    776 0
[ 766.900434] cups-browsed   784 514
[ 766.900435] NetworkManager 785 1295
[ 766.900436] cron           788 179
[ 766.900438] whoopsie       790 667
[ 766.900439] accounts-daemon 800 559
[ 766.900441] kworker/5:1H   818 0
[ 766.900442] kworker/6:1H   839 0
[ 766.900444] kworker/3:1H   887 0
[ 766.900445] irqbalance     896 68
[ 766.900447] polkitd        901 2242
[ 766.900448] lightdm        914 788
[ 766.900449] Xorg           930 204023
[ 766.900450] agetty         933 152
[ 766.900452] kworker/4:1H   999 0
[ 766.900453] vmware-vmblock- 1098 38
[ 766.900454] vmttoolsd      1119 1582
[ 766.900456] VGAuthService  1140 572
[ 766.900457] kworker/0:1H   1225 0

```

【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

数？有什么区别？

```
[ 703.171569] System Page Fault Number: 671571
[ 703.171571] Current Process Page Fault Number: 68

[ 1077.123113] System Page Fault Number: 855276
[ 1077.123115] Current Process Page Fault Number: 68
[ 1077.123116] Each Process:
[ 1077.123117]

[ 1125.844557] System Page Fault Number: 991480
[ 1125.844560] Current Process Page Fault Number: 65

[ 1145.979074] System Page Fault Number: 1119490
[ 1145.979076] Current Process Page Fault Number: 68
```

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 sys_call_table 中，因此可以通过修改表中的系统调用地址，使之成为我们自己实现的函数地址即可。

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。

五、讨论和心得

通过本次实验，我学习了如何修改内核，并熟悉了添加系统调用的过程，在实验过程中主要遇到了如下问题：

1. 当下载、解压完内核时，直接通过 GUI 界面进行剪切-粘贴式移动会爆出权限错误，此时需要通过终端切换到 root 模式，然后再使用 mv 命令移动。

2. 在执行 `make menuconfig` 时出现下图异常，查阅资料后，发现缺少组件，需要以此执行 `apt-get install bison -y` 和 `apt-get install flex`

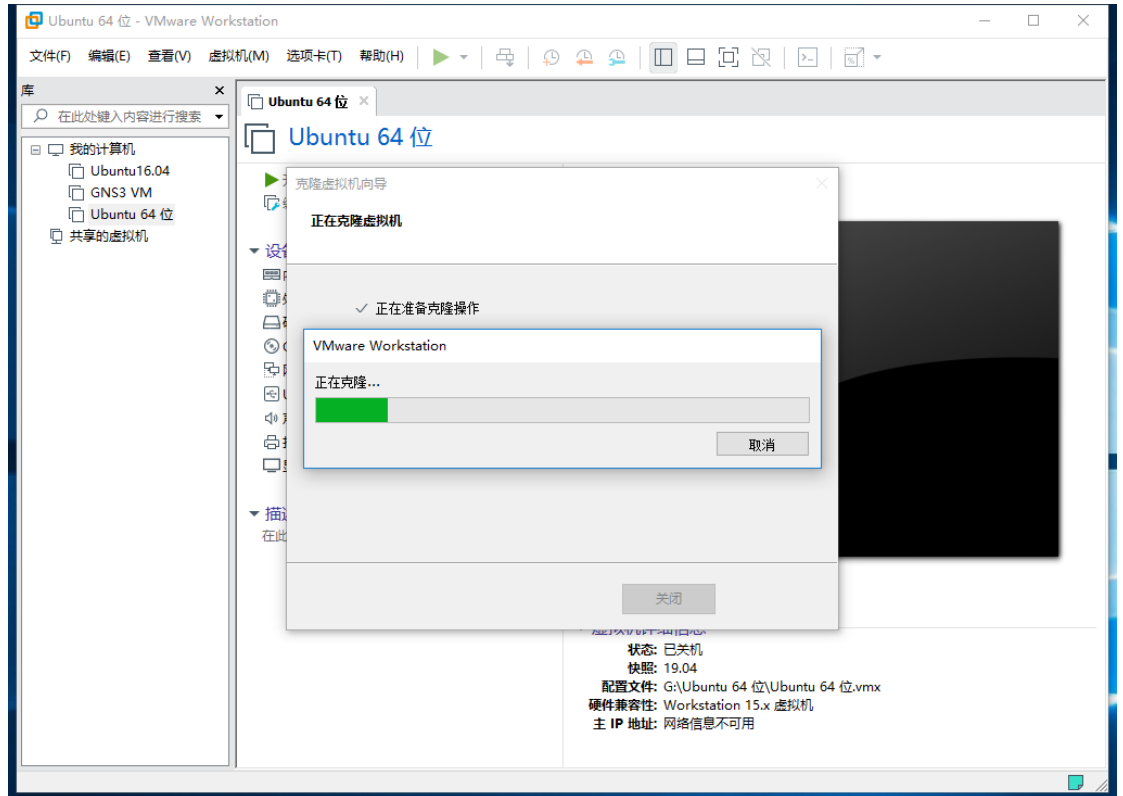
```
root@ying:/usr/src/linux# make menuconfig
YACC    scripts/kconfig/zconf.tab.c
/bin/sh: 1: bison: not found
scripts/Makefile.lib:217: recipe for target 'scripts/kconfig/zconf.tab.c' failed
make[1]: *** [scripts/kconfig/zconf.tab.c] Error 127
Makefile:514: recipe for target 'menuconfig' failed
make: *** [menuconfig] Error 2
```

```
root@ying:/usr/src/linux# make menuconfig
YACC    scripts/kconfig/zconf.tab.c
LEX     scripts/kconfig/zconf.lex.c
/bin/sh: 1: flex: not found
scripts/Makefile.lib:202: recipe for target 'scripts/kconfig/zconf.lex.c' failed
make[1]: *** [scripts/kconfig/zconf.lex.c] Error 127
Makefile:514: recipe for target 'menuconfig' failed
make: *** [menuconfig] Error 2
```

3. 在编译内核时，出现了下图错误，发现是上次实验中同样的问题需要先安装 `openssl`: `apt-get install libssl-dev`

```
HOSTCC  scripts/setarch/mkp
scripts/sign-file.c:25:30: fatal error: openssl/opensslv.h: 没有那个文件或目录
compilation terminated.
scripts/Makefile.host:90: recipe for target 'scripts/sign-file' failed
make[1]: *** [scripts/sign-file] Error 1
make[1]: *** 正在等待未完成任务....
CC      scripts/mod/devicetable-offsets.s
HOSTCC  arch/x86/tools/relocs_64.o
MKELF5  scripts/mod/elfconfig.h
```

4. 因为在编译内核的时候系统极其容易损坏，因此可以创建快照：



5. 在系统调用表中添加或修改相应表项时，64 位虚拟机应选择 `syscall_64.tbl`，而 32 位虚拟机选择 `syscall_32.tbl`

六、附录

[1] ubuntu 内核，在 <http://kernel.ubuntu.com/~kernel-ppa/mainline/>

[2] Linux 官方内核，在 <https://www.kernel.org/pub/linux/kernel/>

[3] Linux 公社 <http://www.linuxidc.com/Linux/2016-06/132707.htm>

[4] ubuntu 上安装 Linux Kernel 4.4，<http://www.linuxidc.com/Linux/2016-01/127383.htm>