Lab 0: Linux内核重建及添加系统调用

姓名：韩耕诗

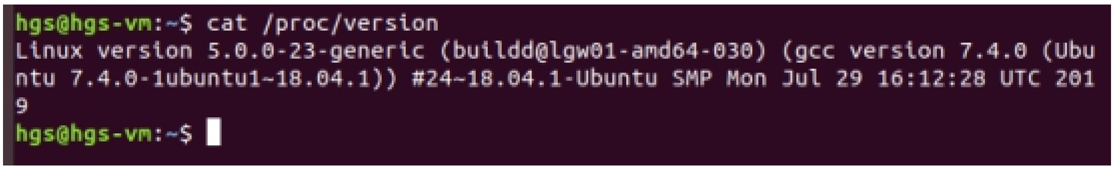
学号：3170103236

Part 1 Linux内核重建

1.1 Steps

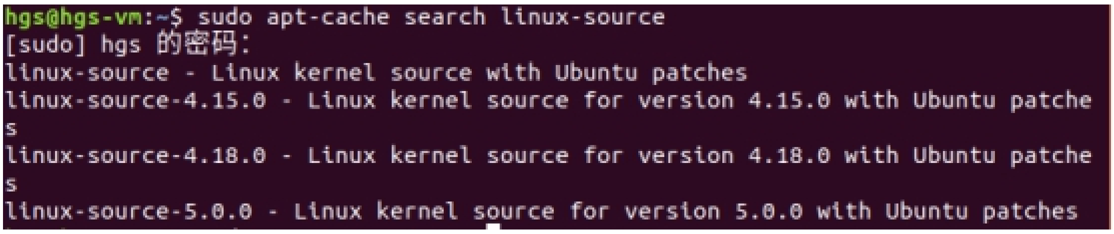
查询当前版本 cat /proc/version，当前内核为5.0.0版本.

```
cat /proc/version
```



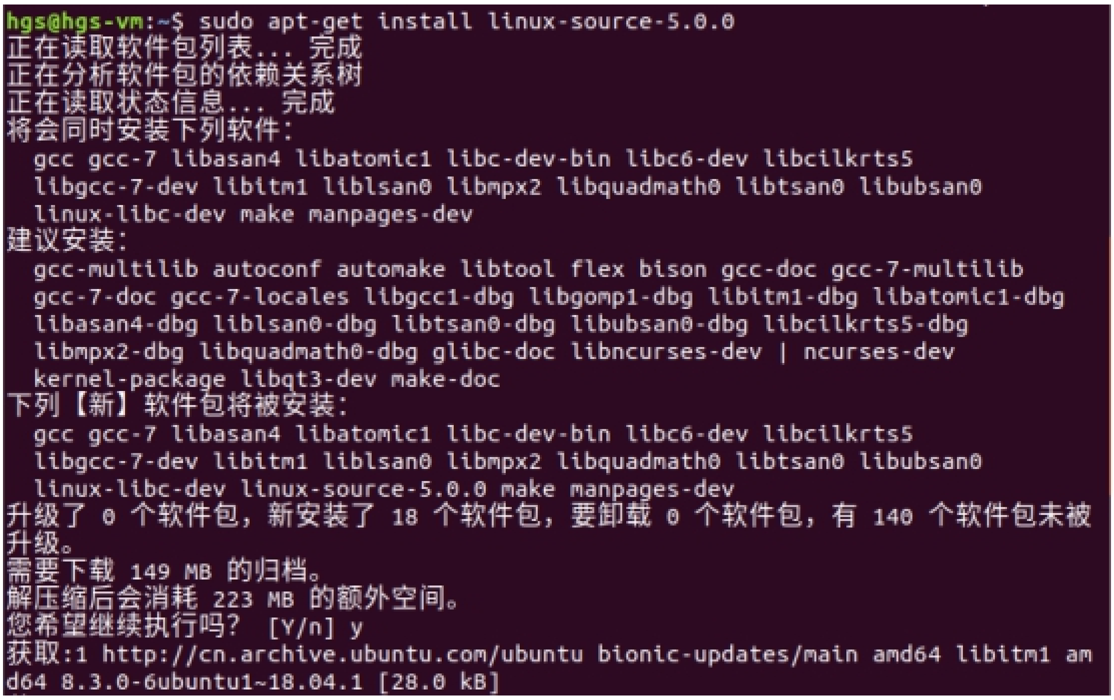
查找源代码

```
sudo apt-cache search linux-source
```



下载5.0.0版本的内核

```
sudo apt-get install linux-source-5.0.0
```



```
hgs@hgs-vm:~$ sudo apt-get install linux-source-5.0.0
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会同时安装下列软件：
  gcc gcc-7 libasan4 libatomic1 libc-dev-bin libc6-dev libcilkrts5
  libgcc-7-dev libitm1 liblsan0 libmpx2 libquadmath0 libtsan0 libubsan0
  linux-libc-dev make manpages-dev
建议安装：
  gcc-multilib autoconf automake libtool flex bison gcc-doc gcc-7-multilib
  gcc-7-doc gcc-7-locales libgcc1-dbg libgomp1-dbg libitm1-dbg libatomic1-dbg
  libasan4-dbg liblsan0-dbg libtsan0-dbg libubsan0-dbg libcilkrts5-dbg
  libmpx2-dbg libquadmath0-dbg glibc-doc libncurses-dev | ncurses-dev
  kernel-package libqt3-dev make-doc
下列【新】软件包将被安装：
  gcc gcc-7 libasan4 libatomic1 libc-dev-bin libc6-dev libcilkrts5
  libgcc-7-dev libitm1 liblsan0 libmpx2 libquadmath0 libtsan0 libubsan0
  linux-libc-dev linux-source-5.0.0 make manpages-dev
升级了 0 个软件包，新安装了 18 个软件包，要卸载 0 个软件包，有 140 个软件包未被
升级。
需要下载 149 MB 的归档。
解压缩后会消耗 223 MB 的额外空间。
您希望继续执行吗？ [Y/n] y
获取:1 http://cn.archive.ubuntu.com/ubuntu bionic-updates/main amd64 libitm1 am
d64 8.3.0-6ubuntu1~18.04.1 [28.0 kB]
```

部署内核源代码 将源代码解压到build_kernel/linux-source-5.0.0

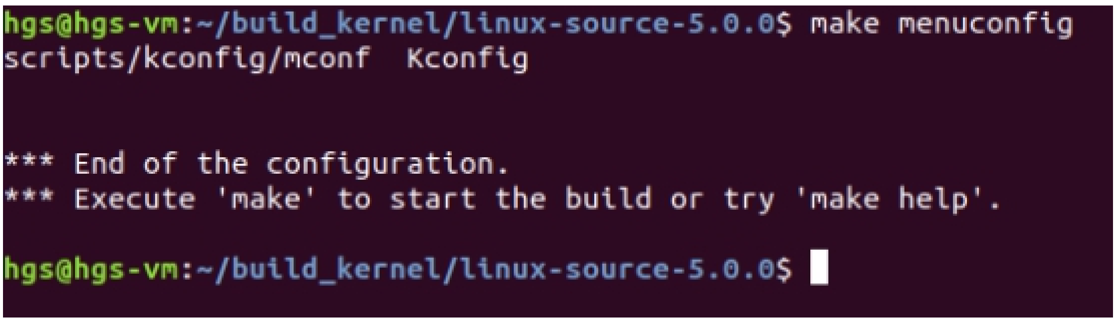
```
cd
mkdir build_kernel
cd build_kernel
cp /usr/src/linux-source-4.13.0.tar.bz2 .
tar jxvf linux-source-4.13.0.tar.bz2
```

配置内核

```
cp /usr/src/linux-headers-5.0.0-23-generic/.config build_kernel/linux-source-
5.0.0
cd build_kernel/linux-source-4.13.0
make menuconfig
```

make menuconfig菜单出现后依次选择load，OK，save，OK，exit，exit

```
hgs@hgs-vm:~$ cp /usr/src/linux-headers-5.0.0-23-generic/.config build_kernel/linux-sou
rce-5.0.0
hgs@hgs-vm:~$ cd build_kernel/linux-source-5.0.0
```



```
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$ make menuconfig
scripts/kconfig/mconf  Kconfig


*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.

hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$
```

编译内核和模块，安装，生成启动文件

```
make -j4
sudo make modules_install -j 4
sudo make install -j 4
sudo update-initramfs -c -k 5.0.21
sudo update-grub
```

```
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$ make -j4
scripts/kconfig/conf  --syncconfig Kconfig
  DESCEND  objtool
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
  CC [M]  net/bridge/netfilter/ebt_snat.o
  CC [M]  net/bridge/netfilter/ebt_log.o
  CC [M]  net/bridge/netfilter/ebt_nflog.o
```

```
  DEPMOD  5.0.21
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$ sudo make install -j 4
sh ./arch/x86/boot/install.sh 5.0.21 arch/x86/boot/bzImage \
        System.map "/boot"
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 5.0.21 /boot/vmlin
uz-5.0.21
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 5.0.21 /boot/vmlinu
z-5.0.21
update-initramfs: Generating /boot/initrd.img-5.0.21
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 5.0.21 /boot/vm
linuz-5.0.21
run-parts: executing /etc/kernel/postinst.d/update-notifier 5.0.21 /boot/vmlinu
z-5.0.21
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 5.0.21 /boot/vmlinuz
-5.0.21
Sourcing file `/etc/default/grub'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-5.0.21
Found initrd image: /boot/initrd.img-5.0.21
Found linux image: /boot/vmlinuz-5.0.0-23-generic
Found initrd image: /boot/initrd.img-5.0.0-23-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$
```

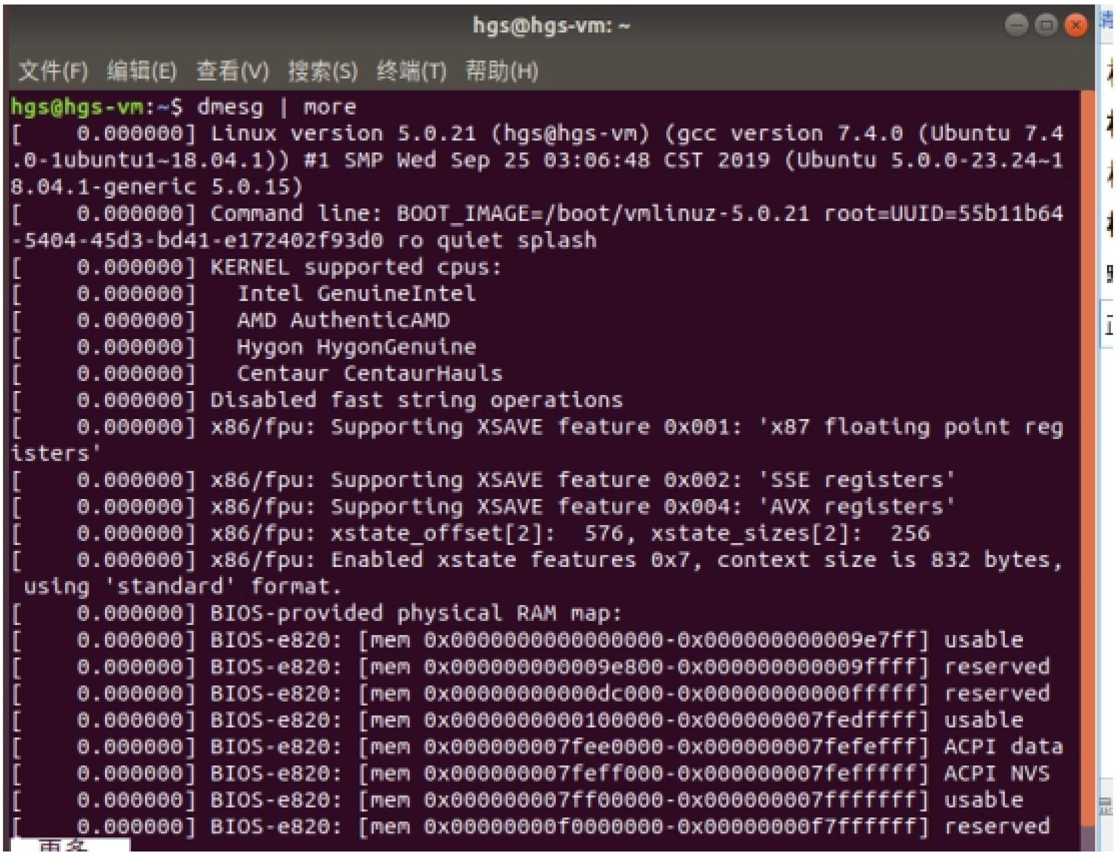
```
done
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$ ls -l /lib/modules/
总用量 8
drwxr-xr-x 5 root root 4096 8月   6 03:05 5.0.0-23-generic
drwxr-xr-x 3 root root 4096 9月  25 11:59 5.0.21
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$ sudo update-initramfs -c -k 5.0.2
1
update-initramfs: Generating /boot/initrd.img-5.0.21
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$
```

```
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$ sudo update-grub
Sourcing file `/etc/default/grub'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-5.0.21
Found initrd image: /boot/initrd.img-5.0.21
Found linux image: /boot/vmlinuz-5.0.0-23-generic
Found initrd image: /boot/initrd.img-5.0.0-23-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
hgs@hgs-vm:~/build_kernel/linux-source-5.0.0$
```

重启系统后验证

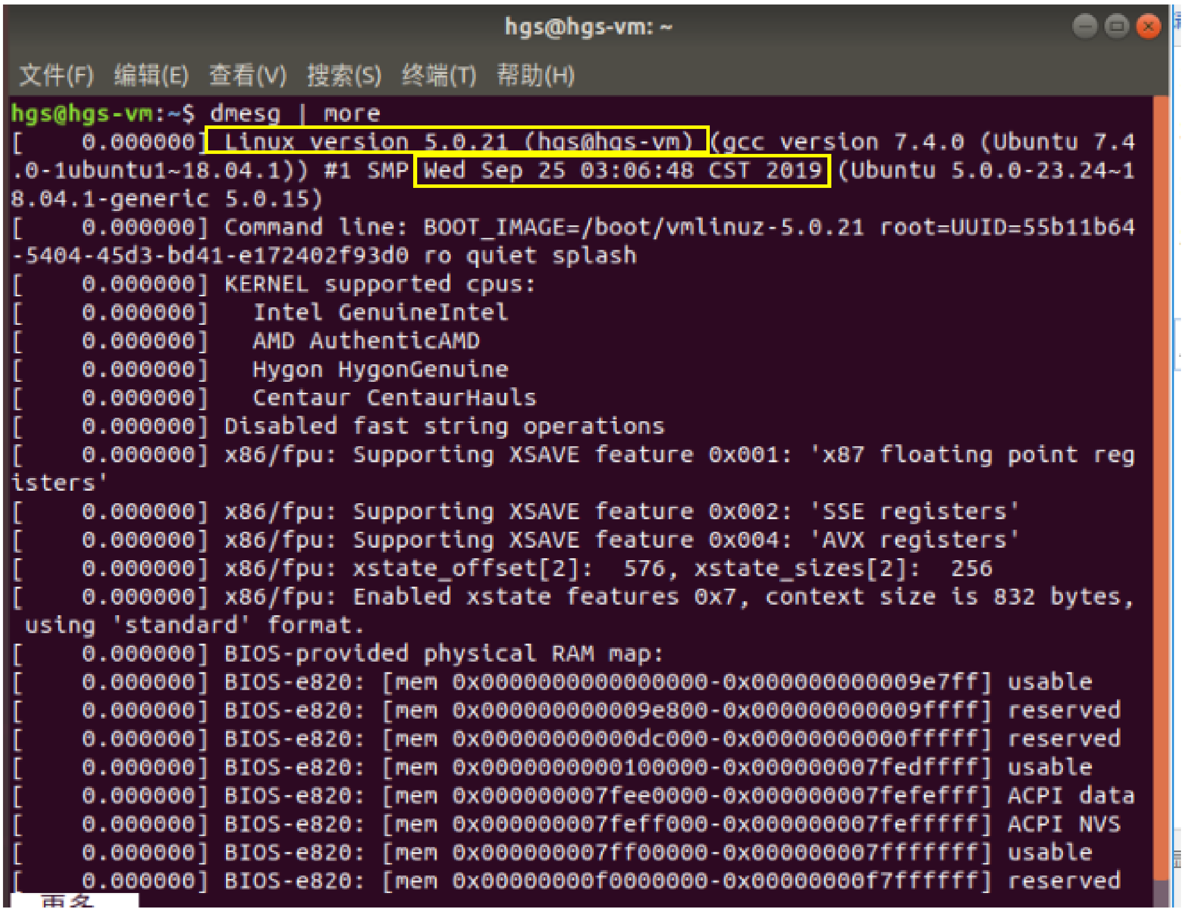
```
uname -a
dmesg | more
```

```
[    0.000000] BIOS-e820: [mem 0x00000000fee00000-0x00000000fee00fff] reserved
hgs@hgs-vm:~$ uname -a
Linux hgs-vm 5.0.21 #1 SMP Wed Sep 25 03:06:48 CST 2019 x86_64 x86_64 x86_64 GN
U/Linux
hgs@hgs-vm:~$
```



1.2 Questions

1.提交你编译后的内核 dmesg | more 命令运行结果的截图，需要能明确显示出来你编译内核的时间和机器名，用户名（图片加框）



2.Linux 内核目录下有一个.config 文件，请说明这个文件的作用?

.config文件是内核配置文件，顶层Makefile会使用这个.config文件来构建内核。大多数内核软件模块也通过.config文件间接地读取配置内容。

3.在Linux内核代码树中，很多子目录Makefile 文件和 Kconfig文件，请分别解释这两个文件的作用?

每个Kconfig分别描述了所属目录源文档相关的内核配置菜单，在执行内核配置make menuconfig时，会从Kconfig中读出菜单，用户选择后保存到.config的内核配置文档中。

在内核编译时，主Makefile调用这个基于Kconfig生成的.config，就知道了用户的选择。

要想添加新的驱动到内核的源码中，就要修改Kconfig，这样就能够选择这个驱动。

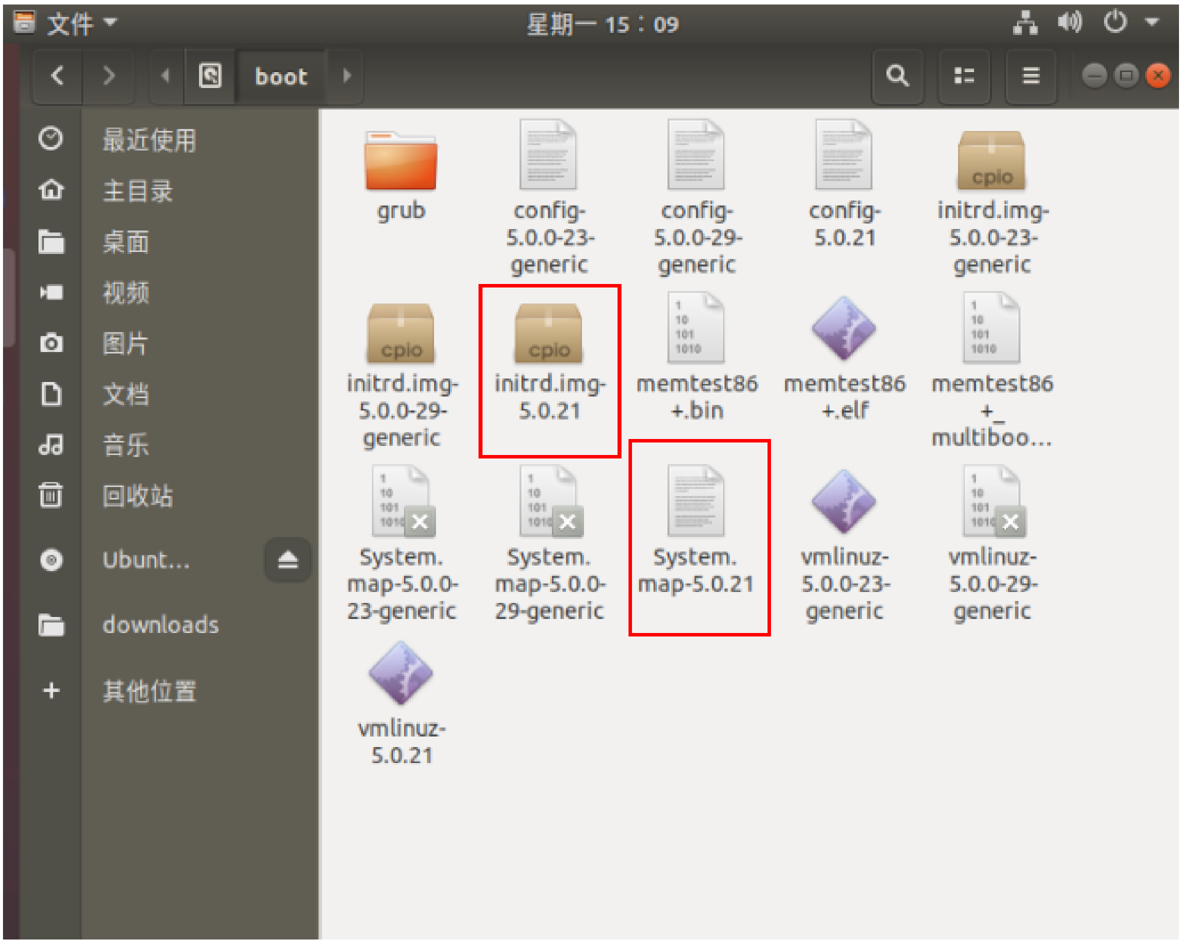
如果想使新的驱动被编译，则要修改Makefile文件。

4.浏览/boot 目录，你一定发现了 System.map-4.13.16 文件，以及 initrd.img-4.13.0 文件。这两个文件分别起什么作用?你能否设计一个实验来验证你的判断?

initrd 的含义是initial RAM disk，就是启动系统所需加载的虚拟磁盘。在系统启动过程中，kernel、initrd和system module是依次加载的。initrd包含一部分内核模块，主要是一些关键的外部硬件，如SATA、SCSI和USB等外设。如果intrid不存在或者失败会影响系统启动。

System.map文件是对应内核的符号映射表，顾名思义就是将内核中的符号（包括变量和函数）和它的地址联系起来的一个列表，是所有符号及其对应地址的一个列表。

设计一个实验：将这两个文件复制到其他文件夹后，在boot文件夹下删除这两个文件，重启系统。



删除这两个文件后重启系统出现错误。

```
BusyBox v1.27.2 (Ubuntu 1:1.27.2-2ubuntu3.2) built-in shell (ash)
Enter 'help' for a list of built-in commands.

(initramfs)
```

开机高级选项进入之前的内核版本5.0.23-generic，将这两个文件恢复，重启成功。

Part 2 添加系统调用

2.1 Steps

2.1.1 在系统调用表中添加相应表项

找到arch/x86/entry/syscalls/syscall_64.tbl，在common中添加一条

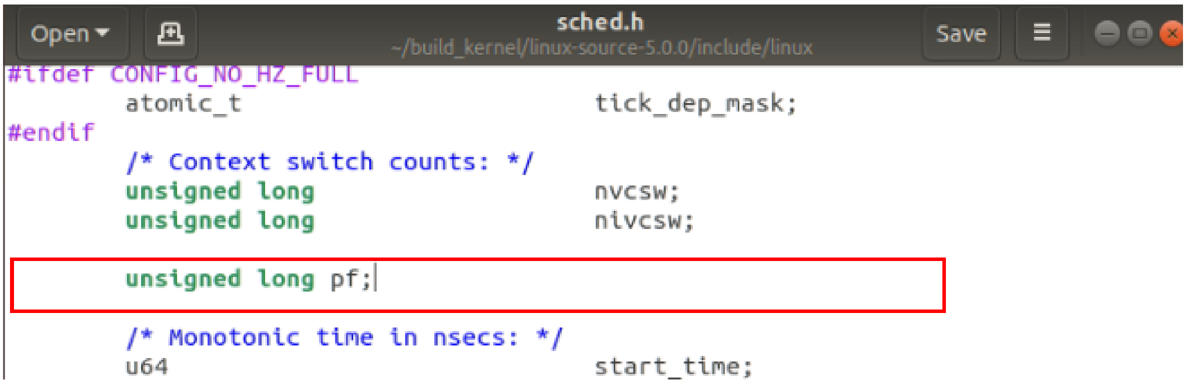
```
syscall_64.tbl
~/build_kernel/linux-source-5.0.0/arch/x86/entry/syscalls

320     common  kexec_file_load         __x64_sys_kexec_file_load
321     common  bpf                     __x64_sys_bpf
322     64      execveat                __x64_sys_execveat/ptregs
323     common  userfaultfd             __x64_sys_userfaultfd
324     common  membarrier              __x64_sys_membarrier
325     common  mlock2                  __x64_sys_mlock2
326     common  copy_file_range         __x64_sys_copy_file_range
327     64      preadv2                 __x64_sys_preadv2
328     64      pwritev2                __x64_sys_pwritev2
329     common  pkey_mprotect           __x64_sys_pkey_mprotect
330     common  pkey_alloc              __x64_sys_pkey_alloc
331     common  pkey_free               __x64_sys_pkey_free
332     common  statx                   __x64_sys_statx
333     common  io_pgetevents           __x64_sys_io_pgetevents
334     common  rseq                    __x64_sys_rseq
335     common  mysyscall               __x64_sys_mysyscall

#
```

2.1.2 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 文件中声明变量 pfcount

```
mm.h
~/build_kernel/linux-source-5.0.0/include/linux

/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _LINUX_MM_H
#define _LINUX_MM_H
extern unsigned long pfcount;
#include <linux/errno.h>
```

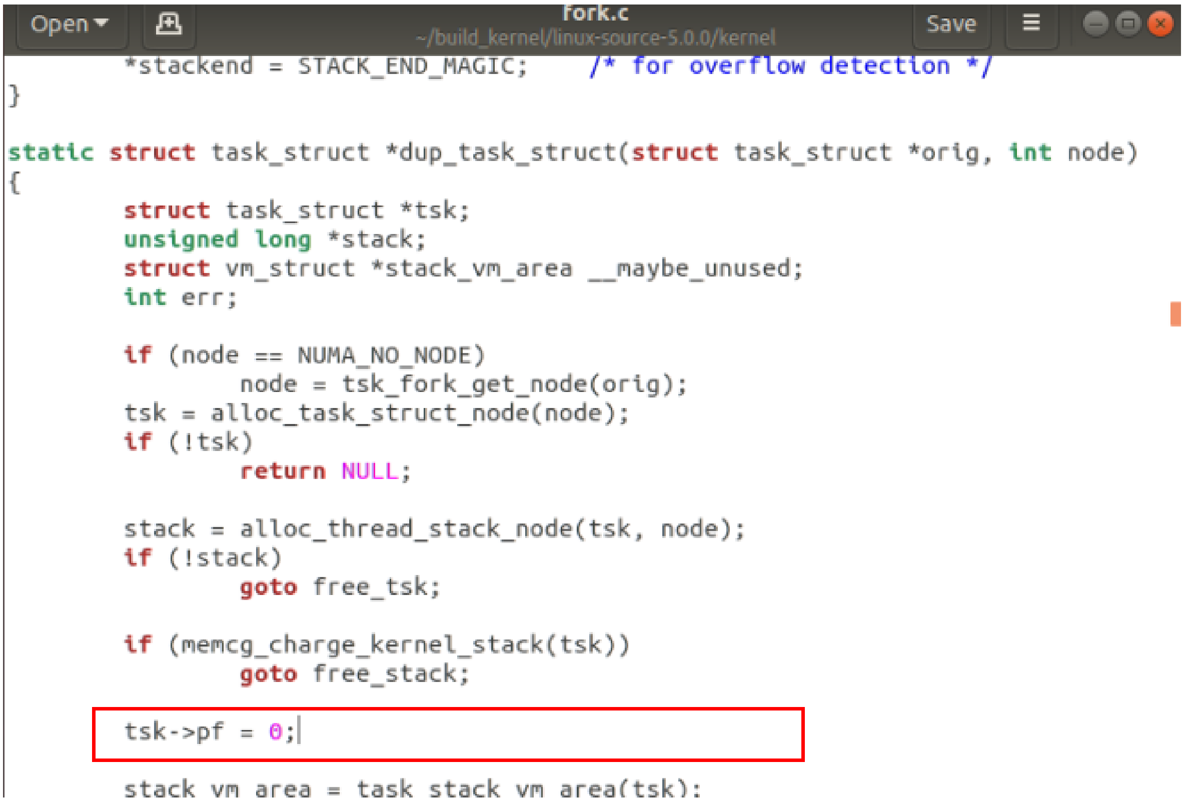
在进程 task_struct 中增加成员 pf ， 在 include/linux/sched.h 文件中的 task_struct 结构中添加 pf 字段



sched.h
~/build_kernel/linux-source-5.0.0/include/linux

```c
#ifdef CONFIG_NO_HZ_FULL
	atomic_t			tick_dep_mask;
#endif
	/* Context switch counts: */
	unsigned long			nvcsw;
	unsigned long			nivcsw;

	unsigned long pf;

	/* Monotonic time in nsecs: */
	u64				start_time;
```

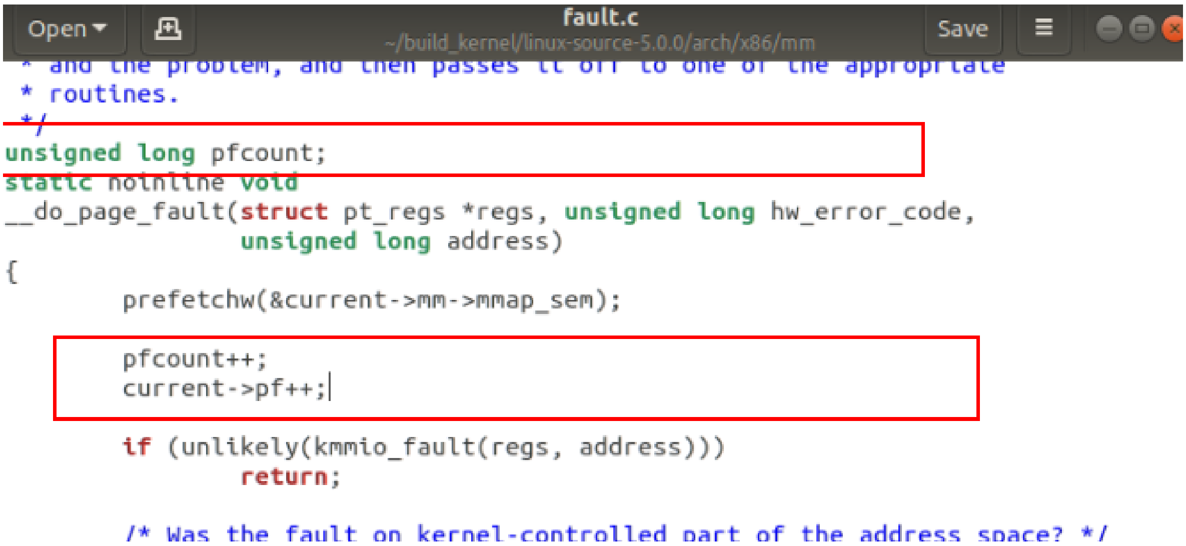
统计当前进程缺页次数需要在创建进程是需要将进程控制块中的pf设置为0，在进程创建过程中，子进程会把父进程的进程控制块复制一份，实现该复制过程的函数是kernel/fork.c文件中的dup_task_struct()函数，修改该函数将子进程的pf设置成0。



fork.c
~/build_kernel/linux-source-5.0.0/kernel

```c
	*stackend = STACK_END_MAGIC;	/* for overflow detection */
}

static struct task_struct *dup_task_struct(struct task_struct *orig, int node)
{
	struct task_struct *tsk;
	unsigned long *stack;
	struct vm_struct *stack_vm_area __maybe_unused;
	int err;

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

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

	if (memcg_charge_kernel_stack(tsk))
		goto free_stack;

	tsk->pf = 0;

	stack_vm_area = task_stack_vm_area(tsk);
```

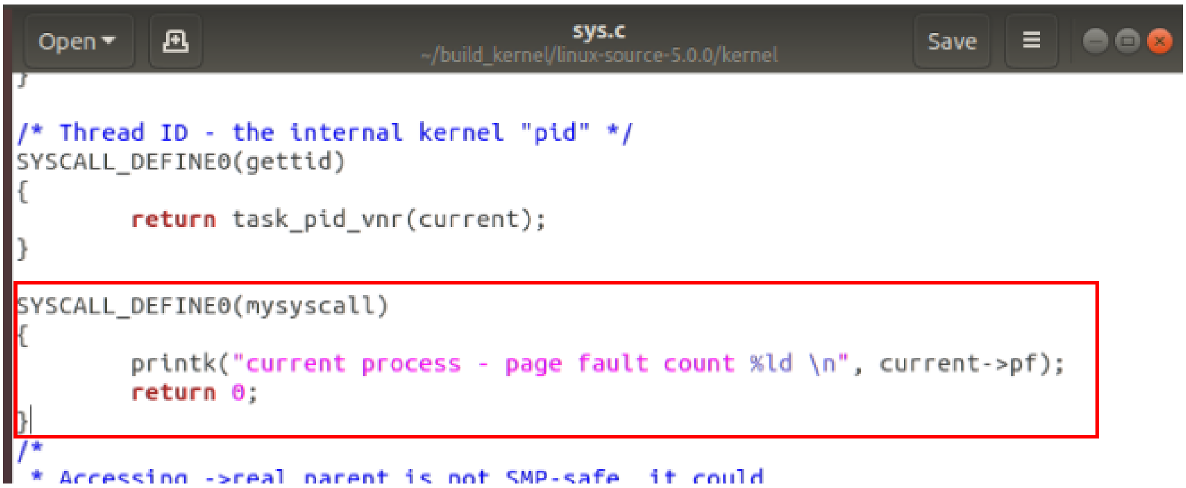
在arch/x86/mm/fault.c文件中定义变量pfcount，并修改arch/x86/mm/fault.c中do_page_fault()函数。每次产生缺页中断，do_page_fault()函数会被调用，pfcount变量值递增1，记录系统产生缺页次数，current->pf值递增1，记录当前进程产生缺页次数。



```
fault.c
~/build_kernel/linux-source-5.0.0/arch/x86/mm
 * and the problem, and then passes it off to one of the appropriate
 * routines.
 */
unsigned long pfcount;
static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long hw_error_code,
                unsigned long address)
{
        prefetchw(&current->mm->mmap_sem);

        pfcount++;
        current->pf++;

        if (unlikely(kmmio_fault(regs, address)))
                return;

        /* Was the fault on kernel-controlled part of the address space? */
```

2.1.3 sys_mysyscall的实现

把以下一小段程序添加在kernel/sys.c里面。

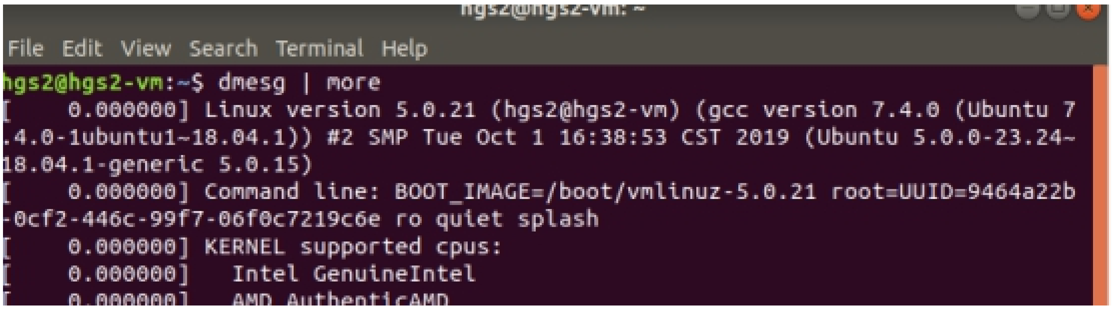


```
sys.c
~/build_kernel/linux-source-5.0.0/kernel
}

/* Thread ID - the internal kernel "pid" */
SYSCALL_DEFINE0(gettid)
{
        return task_pid_vnr(current);
}

SYSCALL_DEFINE0(mysyscall)
{
        printk("current process - page fault count %ld \n", current->pf);
        return 0;
}
/*
 * Accessing ->real_parent is not SMP-safe, it could
```

2.1.4 重新编译内核和重启

```
make -j4

Sudo make modules-install -j 4

Sudo make install -j 4

Sudo update-initramfs -c -k 5.0.21

Sudo update-grub
```

重启后用 dmesg | more 验证内核编译成功



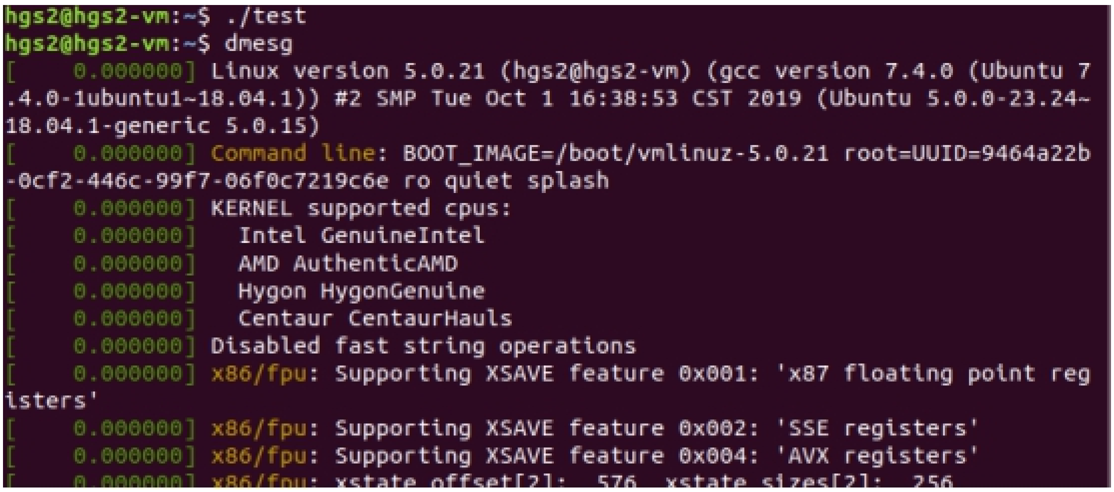
2.1.5 编写用户态程序

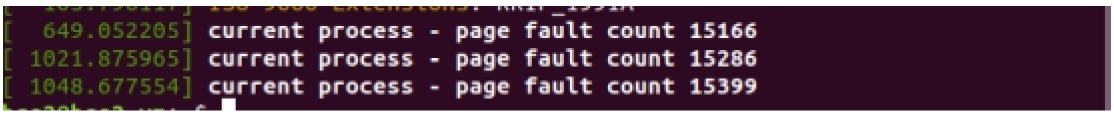
编写test.c

```
#include <linux/unistd.h>
#include <sys/syscall.h>
#define __NR_mysyscall 335
int main()
{
    syscall(__NR_mysyscall);
}
```

编译运行test，并且用dmesg验证

```
gcc -o test test.c
./test
dmesg
```

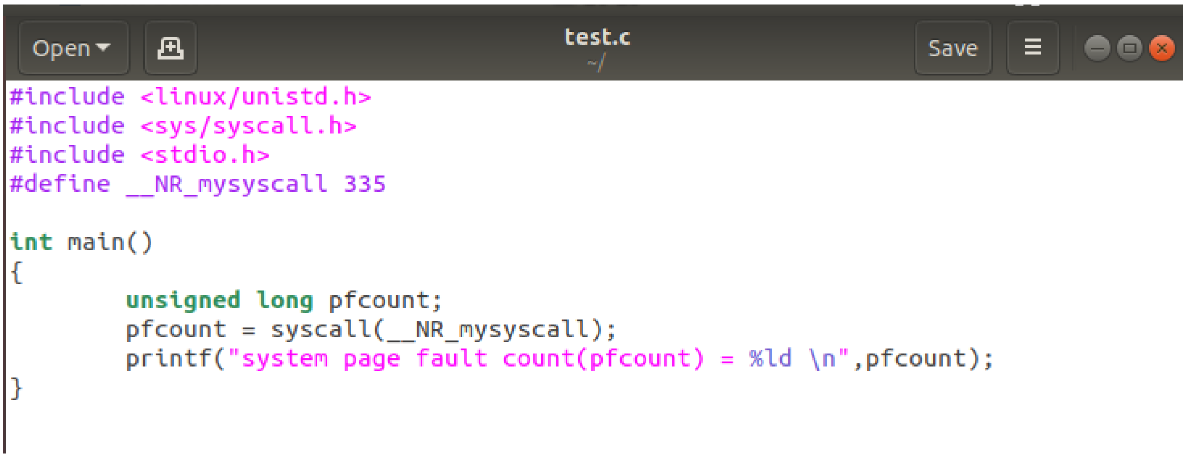


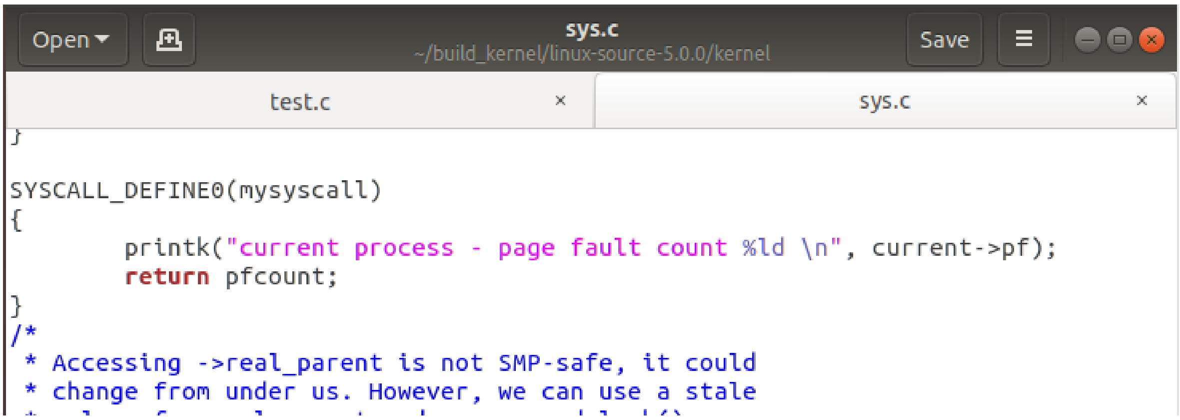


2.2 Questions

1.在 test.c 中添加打印整个系统 page fault 的变量值（也就是 pfcount 的值）。上传你的 test.c 代码。

修改test.c以及sys.c





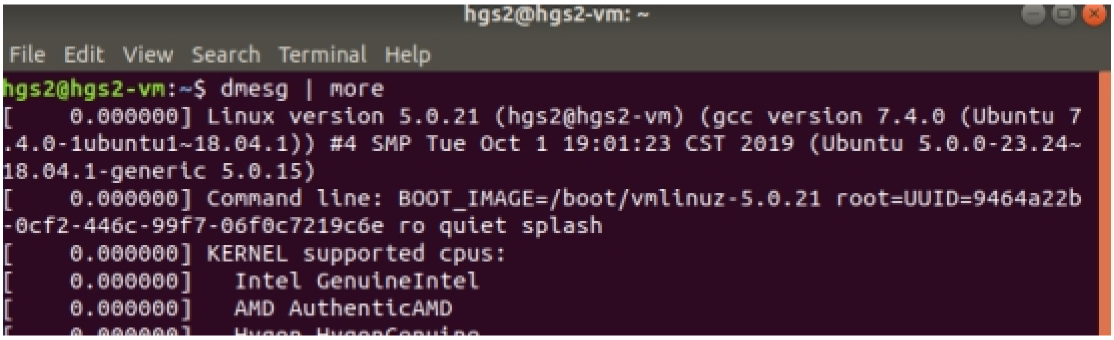
test.c源代码修改如下：

```
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#define __NR_mysyscall 335
int main()
{
    unsigned long pfcount;
    pfcount = syscall(__NR_mysyscall);
    printf("system page fault count(pfcount) = %ld \n",pfcount);
}
```

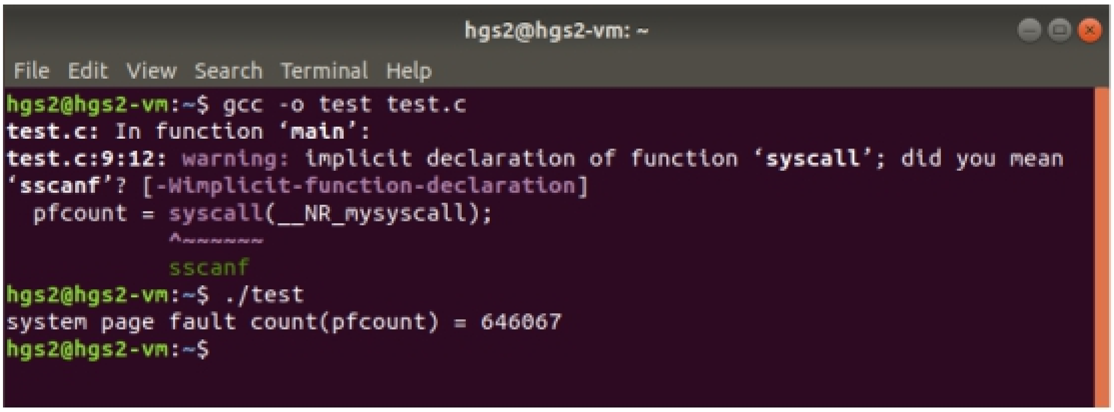
重新编译内核

```
make -j4
sudo make modules_install -j 4
sudo make install -j 4
sudo update-initramfs -c -k 5.0.21
sudo update-grub
```

重启系统，用 dmesg | more 验证内核编译成功



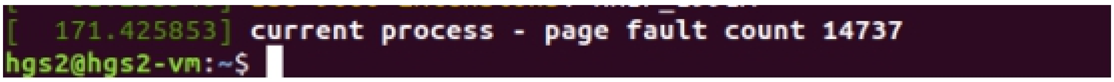
重新用gcc编译test程序



2.运行 test 程序后，dmesg 的截图证明你的系统调用添加成功，并且能在用户态被调用。

```
./test
dmesg
```

如下图成功运行test程序，dmesg末尾有调用日志。



3.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。

（后面第7题改正实验指导中存在的错误后，重新做了一遍第3题）

多次运行test以及dmesg

```
./test
dmesg
```

第一次运行test，系统缺页次数646067，当前进程缺页次数14737

第二次运行test，系统缺页次数651051，当前进程缺页次数14838

```
hgs2@hgs2-vm:~$ ./test
system page fault count(pfcount) = 651051
hgs2@hgs2-vm:~$ dmesg
```

```
[  171.425853] current process - page fault count 14737
[  353.914321] current process - page fault count 14838
hgs2@hgs2-vm:~$
```

第三次运行test，系统缺页次数651809，当前进程缺页次数14928

```
hgs2@hgs2-vm:~$ ./test
system page fault count(pfcount) = 651809
hgs2@hgs2-vm:~$ dmesg
```

```
[  171.425853] current process - page fault count 14737
[  353.914321] current process - page fault count 14838
[  438.484637] current process - page fault count 14928
hgs2@hgs2-vm:~$
```

第四次运行test，系统缺页次数652233，当前进程缺页次数15026

```
hgs2@hgs2-vm:~$ ./test
system page fault count(pfcount) = 652233
hgs2@hgs2-vm:~$ dmesg
```

```
[  171.425853] current process - page fault count 14737
[  353.914321] current process - page fault count 14838
[  438.484637] current process - page fault count 14928
[  511.112083] current process - page fault count 15026
hgs2@hgs2-vm:~$
```

第五次运行test，系统缺页次数652703，当前进程缺页次数15118

```
hgs2@hgs2-vm:~$ ./test
system page fault count(pfcount) = 652703
hgs2@hgs2-vm:~$ dmesg
```

```
[  171.425853] current process - page fault count 14737
[  353.914321] current process - page fault count 14838
[  438.484637] current process - page fault count 14928
[  511.112083] current process - page fault count 15026
[  567.691433] current process - page fault count 15118
hgs2@hgs2-vm:~$
```

4.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。

除了内核编译法来添加系统调用之外，也可以通过module进行内核添加来添加系统调用。这种方法是采用系统调用拦截的一种方式，改变某一个系统调用号对应的服务程序，变为自己编写的程序，从而相当于添加了系统调用。

5.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。

我认为对于一个操作系统而言，修改系统调用并不够安全。因为当修改系统调用时，有可能会对原来系统调用表中的其他调用进行修改，从而使得其名称发生变化或者是缺少对应编号的系统调用，可能会给调用正常系统调用接口的准确性和安全性造成威胁。

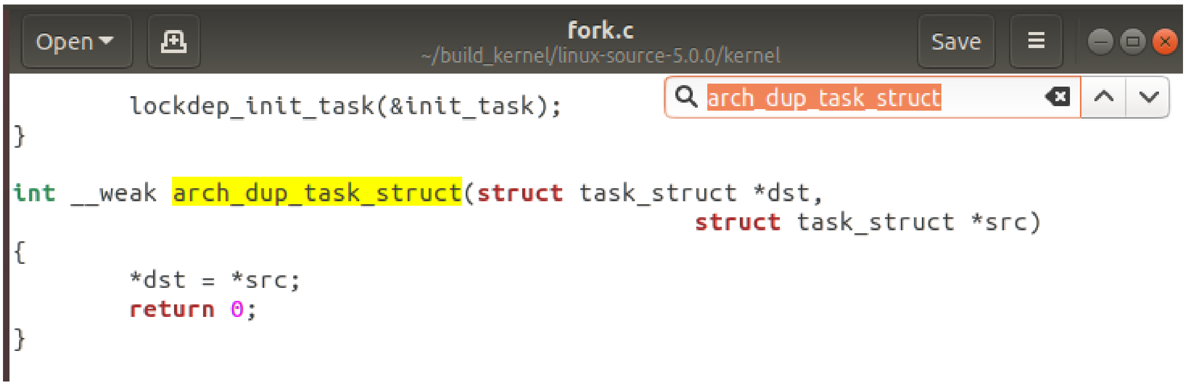
6.在实验过程中遇到了什么问题，你是如何解决的。

在实验过程中，我在添加完系统调用，重新编译内核后，重启系统失败，系统无法开机。因为在课程群里看到有同学出现了相同的问题是重装系统解决的，所以我也重新安装了Ubuntu，再次编译内核后再添加系统调用以及重新编译，这次就成功了。

7.在实验指导2_添加系统调用里有一个错误，希望大家能发现并写在作业中，作为第7个问题。（hint：子进程的pf如何保证从0开始计数？）

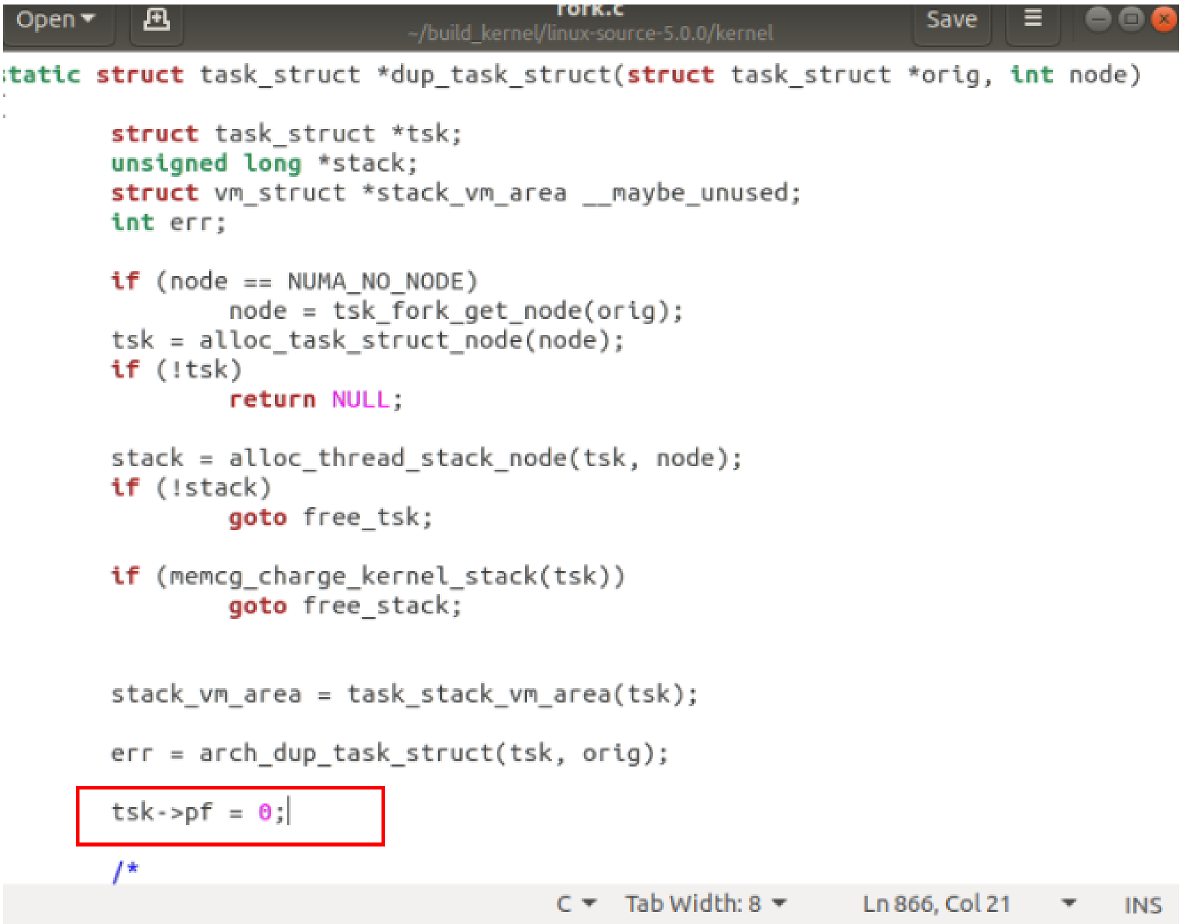
“tsk->pf = 0;”的位置不对。在kernel/fork.c文件中，“tsk->pf = 0” 应该放在 “err = arch_dup_task_struct（tsk, orig); ” 函数调用语句之后，因为这个函数里面有对tsk->pf造成改变。

如下图是arch_dup_task_struct（）函数




```c
        lockdep_init_task(&init_task);
}

int __weak arch_dup_task_struct(struct task_struct *dst,
                                               struct task_struct *src)
{
        *dst = *src;
        return 0;
}
```

经过改正后的kernel/fork.c中的dup_task_struct()函数如下

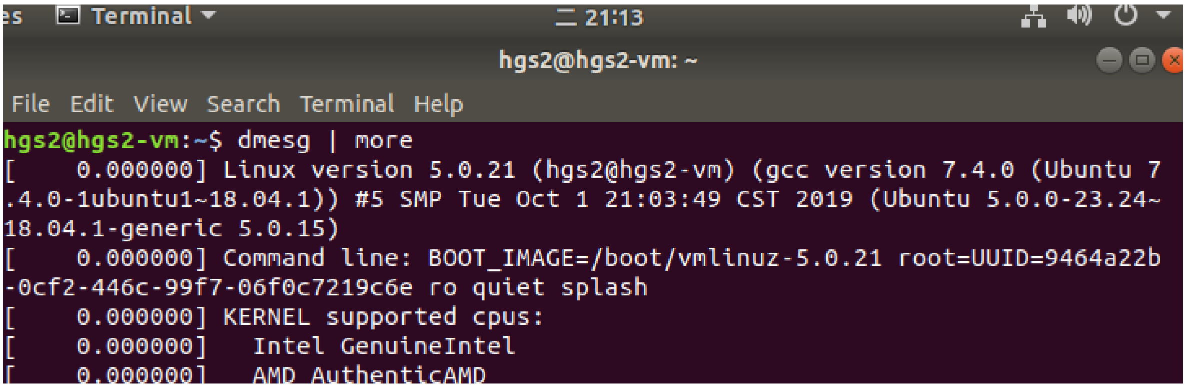



```c
static struct task_struct *dup_task_struct(struct task_struct *orig, int node)
{
        struct task_struct *tsk;
        unsigned long *stack;
        struct vm_struct *stack_vm_area __maybe_unused;
        int err;

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

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

        if (memcg_charge_kernel_stack(tsk))
                goto free_stack;


        stack_vm_area = task_stack_vm_area(tsk);

        err = arch_dup_task_struct(tsk, orig);

        tsk->pf = 0;

        /*
```

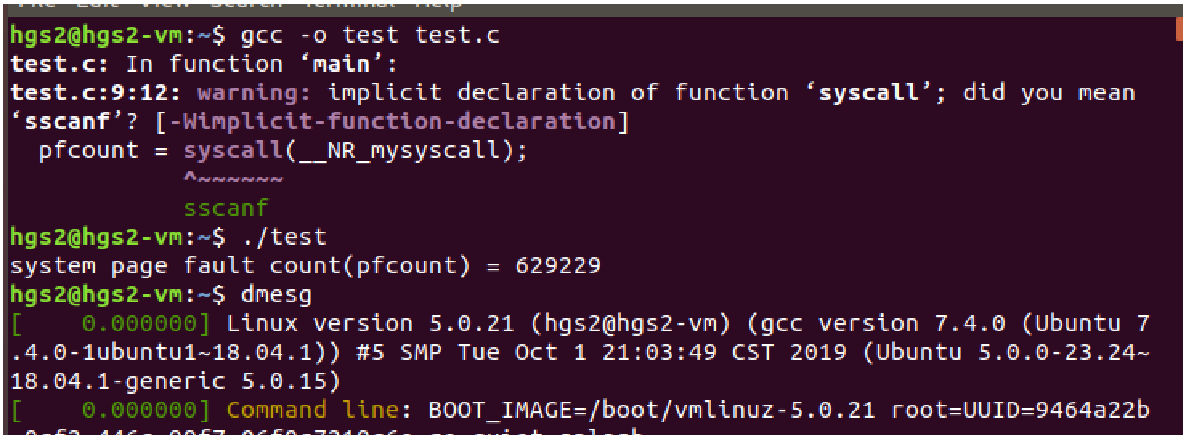
改正后重新编译内核

```
make -j4
sudo make modules_install -j 4
sudo make install -j 4
sudo update-initramfs -c -k 5.0.21
sudo update-grub
```

重启系统，用 dmesg | more 验证内核编译成功



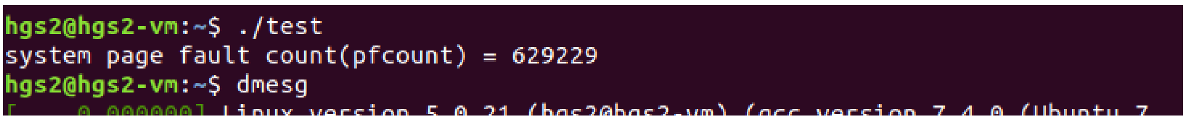
重新编译运行test程序以及dmesg



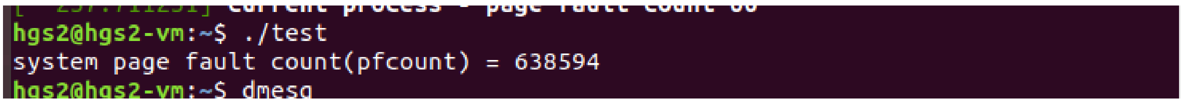
从当前进程的page fault count 来看，改正后数值是比较合理的

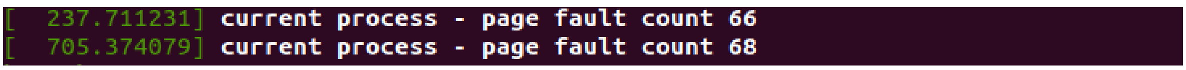
那么重新做第2题：多次运行test程序，每次运行test后记录下系统缺页次数和当前进程缺页次数

第1次运行test



第2次运行test





第3次运行test

```
hgs2@hgs2-vm:~$ ./test
system page fault count(pfcount) = 639550
hgs2@hgs2-vm:~$ dmesg
```

```
[  237.711231] current process - page fault count 66
[  705.374079] current process - page fault count 68
[  786.670646] current process - page fault count 66
hgs2@hgs2-vm:~$
```

第4次运行test

```
hgs2@hgs2-vm:~$ ./test
system page fault count(pfcount) = 642137
hgs2@hgs2-vm:~$ dmesg
```

```
[  237.711231] current process - page fault count 66
[  705.374079] current process - page fault count 68
[  786.670646] current process - page fault count 66
[  937.591141] current process - page fault count 67
hgs2@hgs2-vm:~$
```

第5次运行test

```
hgs2@hgs2-vm:~$ ./test
system page fault count(pfcount) = 642577
hgs2@hgs2-vm:~$ dmesg
```

```
[  237.711231] current process - page fault count 66
[  705.374079] current process - page fault count 68
[  786.670646] current process - page fault count 66
[  937.591141] current process - page fault count 67
[  981.694806] current process - page fault count 66
hgs2@hgs2-vm:~$
```

总共运行了5次test程序，统计如下表。

系统缺页次数(pfcount)	当前进程缺页次数(pf)
629229	66
638594	68
639550	66
642137	67
642577	66

可以看出每次运行，系统缺页次数都会明显增长，而当前进程的缺页次数则基本不变。