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：沈子衿　　学号：3160104734

电子邮件地址：zijinshen@zju.edu.cn

实验日期：2018 年 12 月 22 日

一、实验环境

主机配置：

- 主机型号：Lenovo ThinkPad T480
- 内存：16GB
- 处理器：i7-8550U
- 操作系统：Windows 10 家庭中文版

虚拟机配置：

- 虚拟机环境：Vmware Workstation Pro 14
- Ubuntu 版本：16.04
- Linux 内核版本：4.8

二、实验内容和结果及分析

1. 实验设计思路

本次实验的设计思路已经在实验参考中给出，总体还是较为清晰的，思路如下：

1） 下载并解压内核；

2） 为内核打补丁；

3） 配置内核；

4） 添加新的系统调用号（本次实验我们使用 223）；

5） 在系统调用表中添加或修改响应表项，以便系统调用处理程序检索；

6） 修改统计系统缺页次数和进程缺页次数的内核代码；

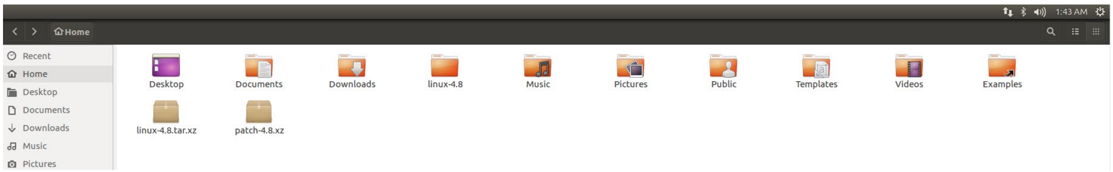
7） 实现简单的系统调用（sys_mysyscall）；

8） 编译内核和重启内核；

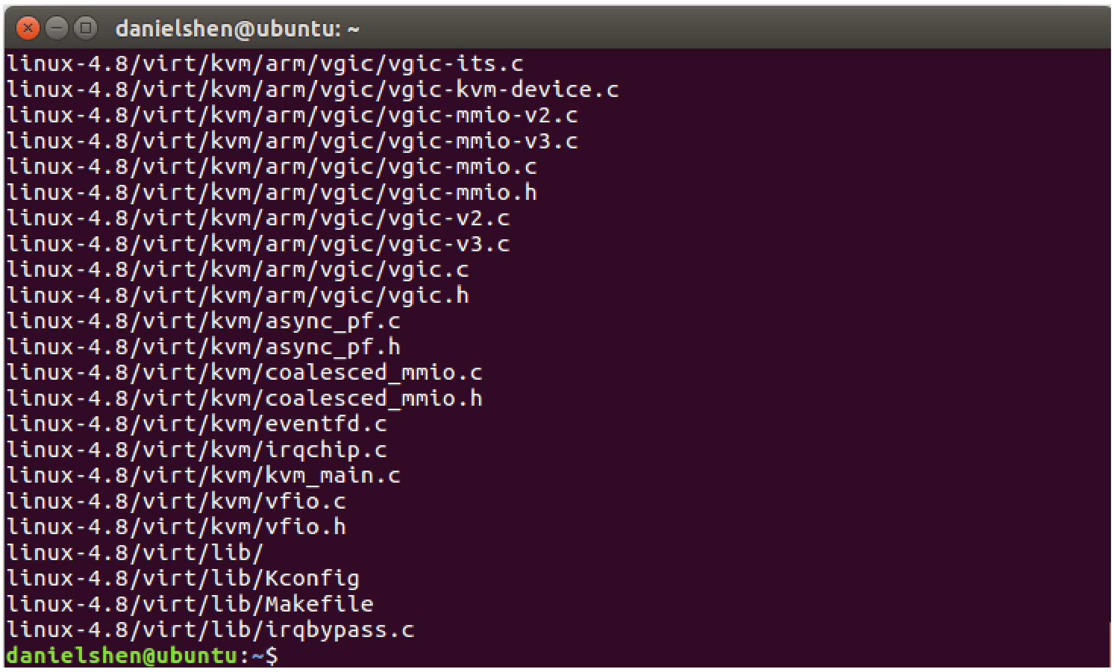
9） 撰写对应的用户态程序，输出结果。

2. 实验步骤及截图

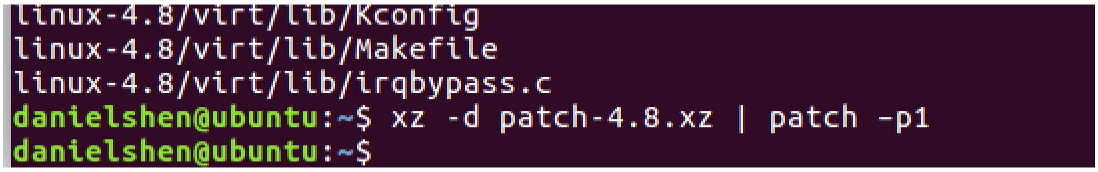
首先，从阿里云镜像下载 linux-4.8 版本内核和对应补丁，存放在主目录（~）下：



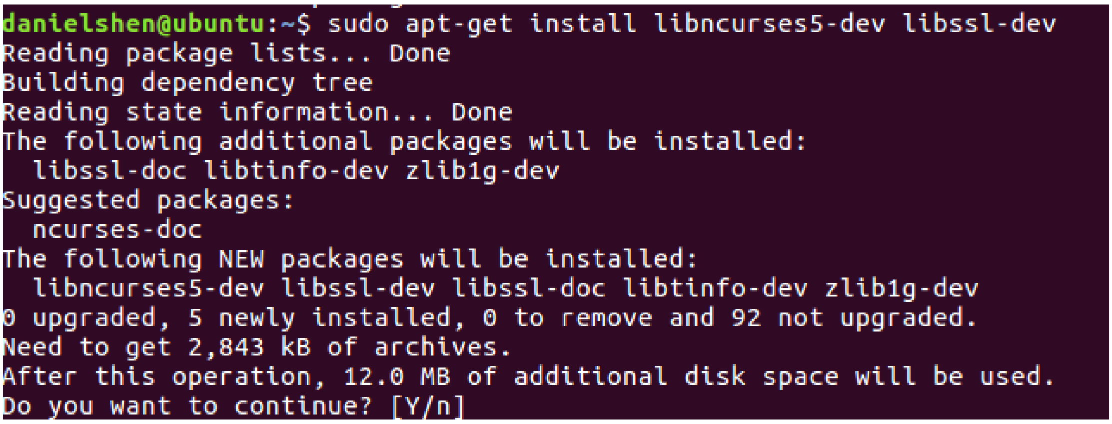
输入命令 tar -xvf linux-4.8.tar.xz 解压内核：



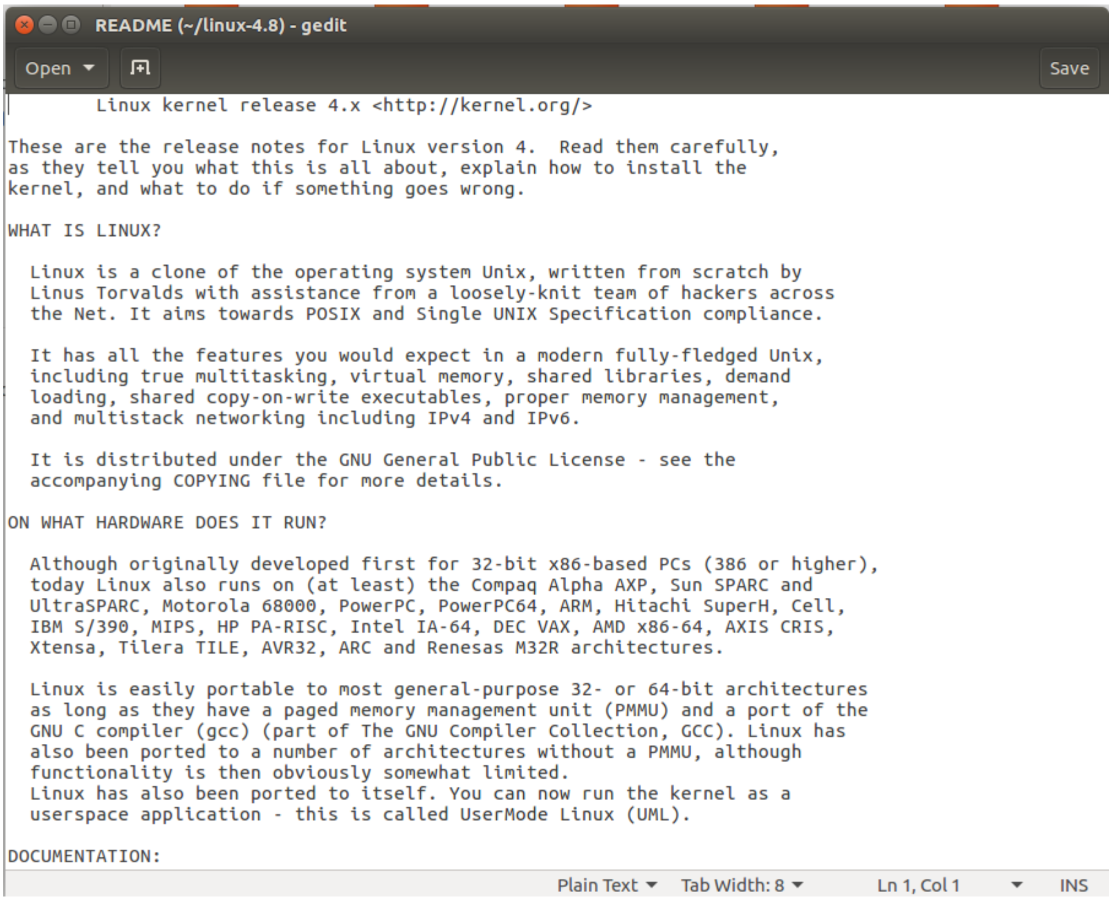
输入命令 xz -d patch-4.8.xz | patch -p1 l 为内核打补丁



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：

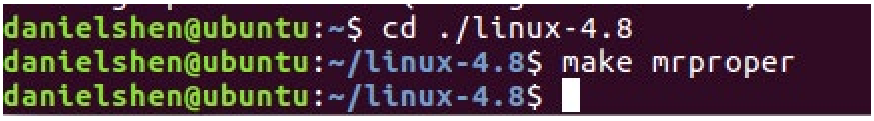


安装完成后，查看内核 README 文件：

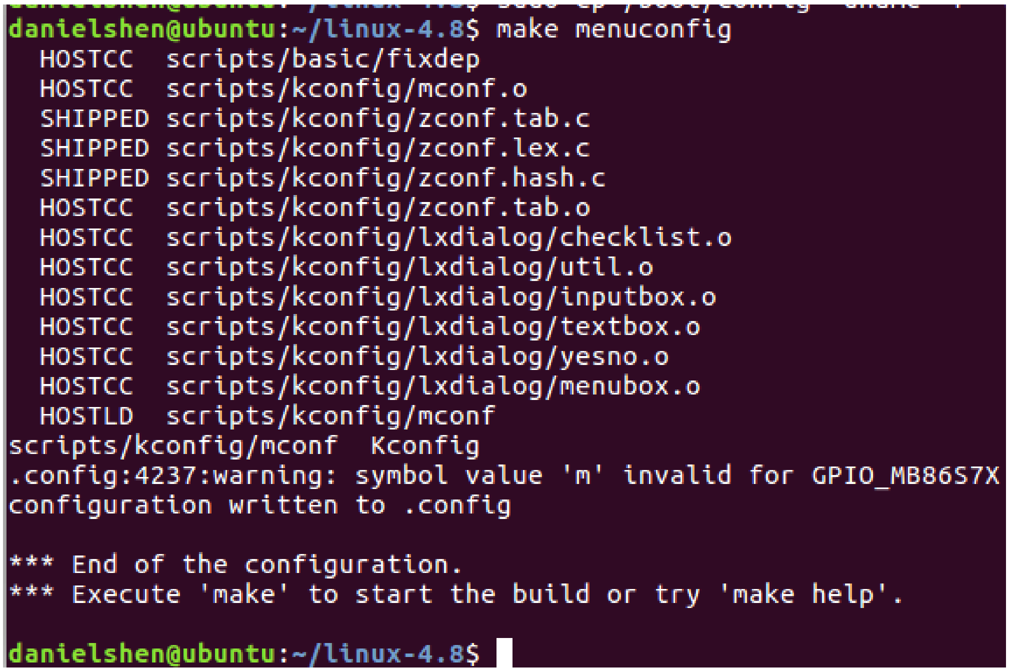



```
        Linux kernel release 4.x <http://kernel.org/>

These are the release notes for Linux version 4.  Read them carefully,
as they tell you what this is all about, explain how to install the
kernel, and what to do if something goes wrong.

WHAT IS LINUX?

  Linux is a clone of the operating system Unix, written from scratch by
  Linus Torvalds with assistance from a loosely-knit team of hackers across
  the Net. It aims towards POSIX and Single UNIX Specification compliance.

  It has all the features you would expect in a modern fully-fledged Unix,
  including true multitasking, virtual memory, shared libraries, demand
  loading, shared copy-on-write executables, proper memory management,
  and multistack networking including IPv4 and IPv6.

  It is distributed under the GNU General Public License - see the
  accompanying COPYING file for more details.

ON WHAT HARDWARE DOES IT RUN?

  Although originally developed first for 32-bit x86-based PCs (386 or higher),
  today Linux also runs on (at least) the Compaq Alpha AXP, Sun SPARC and
  UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH, Cell,
  IBM S/390, MIPS, HP PA-RISC, Intel IA-64, DEC VAX, AMD x86-64, AXIS CRIS,
  Xtensa, Tilera TILE, AVR32, ARC and Renesas M32R architectures.

  Linux is easily portable to most general-purpose 32- or 64-bit architectures
  as long as they have a paged memory management unit (PMMU) and a port of the
  GNU C compiler (gcc) (part of The GNU Compiler Collection, GCC). Linux has
  also been ported to a number of architectures without a PMMU, although
  functionality is then obviously somewhat limited.
  Linux has also been ported to itself. You can now run the kernel as a
  userspace application - this is called UserMode Linux (UML).

DOCUMENTATION:
```

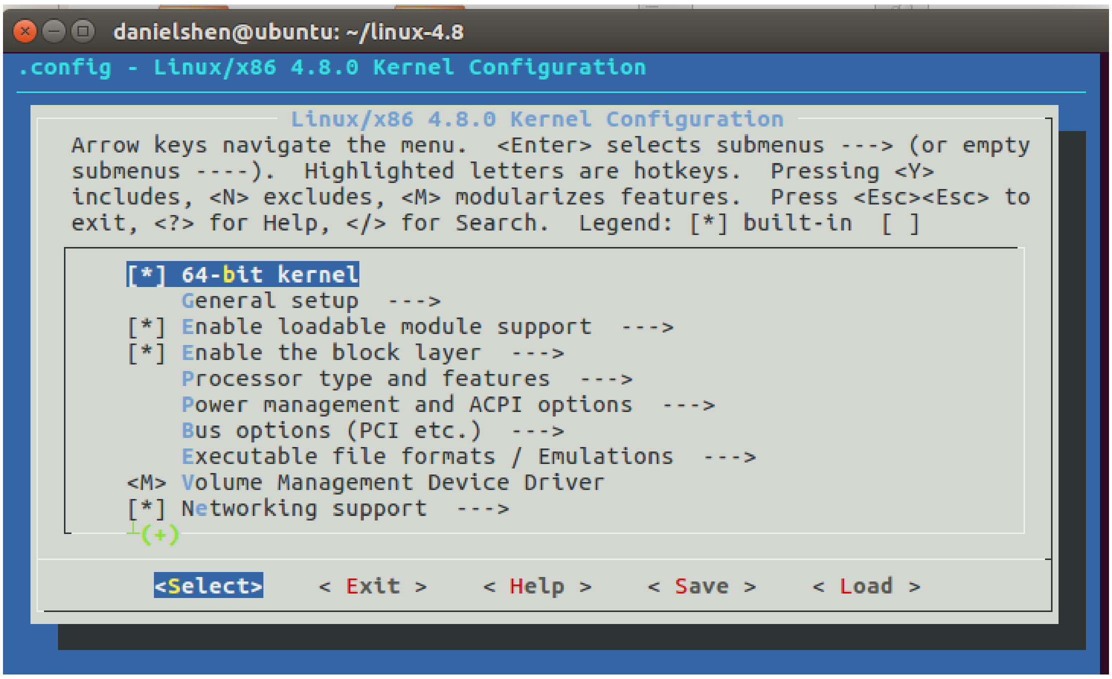
进入 linux-4.8 目录，输入 make mrproper 命令清空缓存：



输入 cp /boot/config-`uname -r` .config 命令复制配置文件到内核根目录，将新内核配置与与正在运行的操作系统内核的运行环境匹配，然后 make menuconfig：

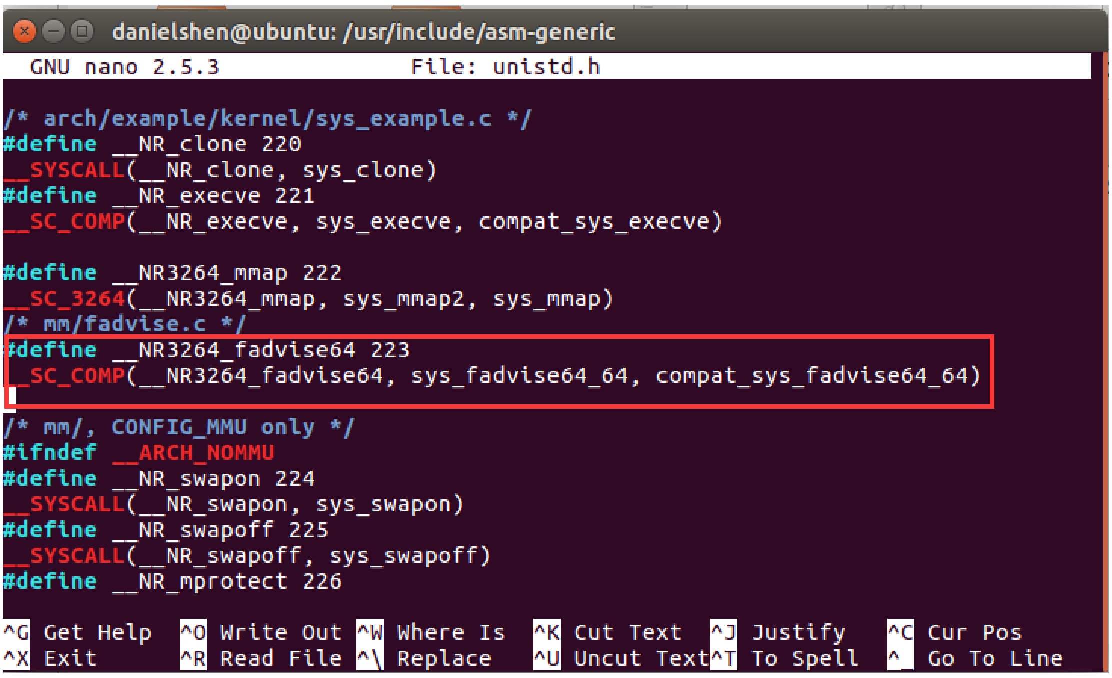


期间命令行会弹出一个配置界面，这里直接选择“exit”并“save”即可：

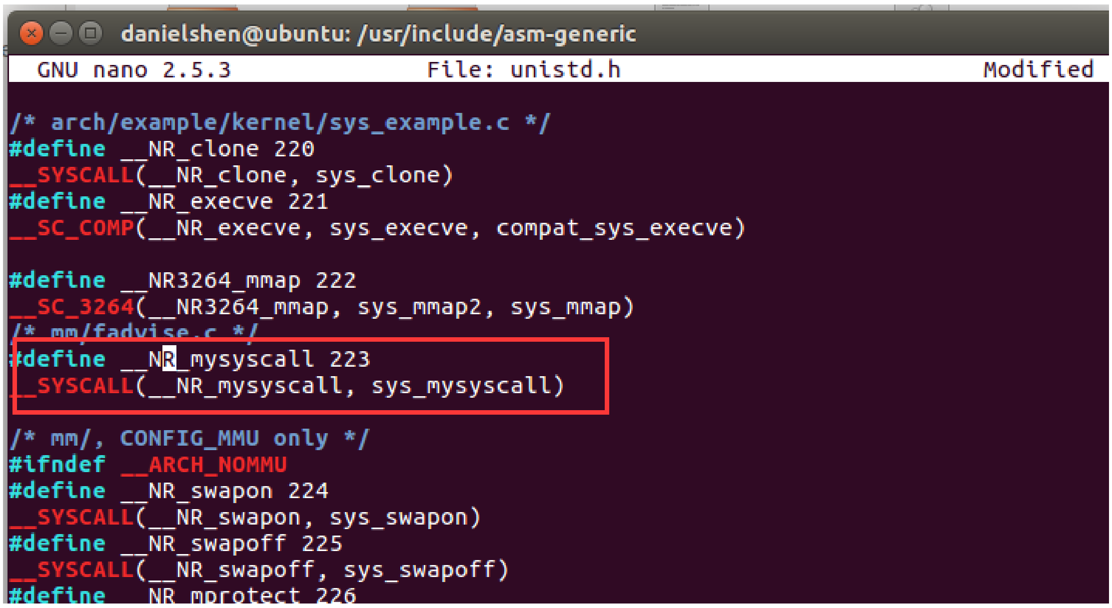


这之后，分别在本系统的 unistd.h 和新内核的 unistd.h 中添加系统调用号：

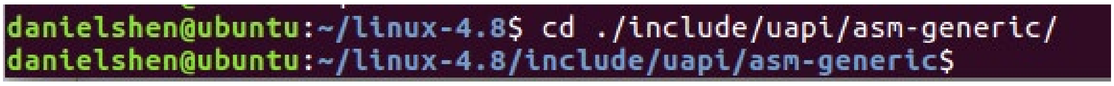
修改前：

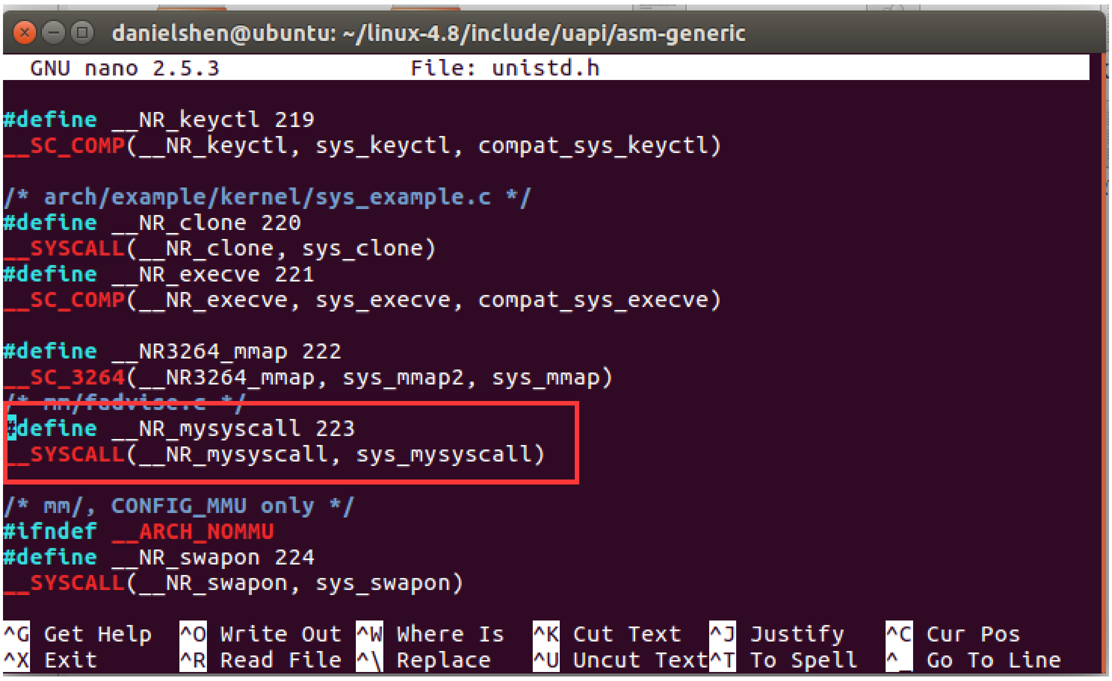


修改号：



对新内核中的 unistd.h 也作同样修改：





上述步骤完成后，修改系统调用表以便系统调用处理程序寻找相应表项：

danielshen@ubuntu: ~/linux-4.8/arch/x86/entry/syscalls

```
danielshen@ubuntu:/usr/include/asm-generic$ nano unistd.h
danielshen@ubuntu:/usr/include/asm-generic$ sudo nano unistd.h
[sudo] password for danielshen:
danielshen@ubuntu:/usr/include/asm-generic$ sudo nano unistd.h
danielshen@ubuntu:/usr/include/asm-generic$ cd ~
danielshen@ubuntu:~$ ls
Desktop     Downloads          linux-4.8          Music       Pictures   Templates
Documents   examples.desktop   linux-4.8.tar.xz   patch-4.8   Public     Videos
danielshen@ubuntu:~$ cd linux-4.8
danielshen@ubuntu:~/linux-4.8$ cd ./include/uapi/asm-generic/
danielshen@ubuntu:~/linux-4.8/include/uapi/asm-generic$ nano unistd.h
danielshen@ubuntu:~/linux-4.8/include/uapi/asm-generic$ nano unistd.h
danielshen@ubuntu:~/linux-4.8/include/uapi/asm-generic$ cd ~
danielshen@ubuntu:~$ cd linux-4.8/
danielshen@ubuntu:~/linux-4.8$ ls
arch     crypto         include  kernel       net             security
block    Documentation  init     lib          README          sound
certs    drivers        ipc      MAINTAINERS  REPORTING-BUGS  tools
COPYING  firmware       Kbuild   Makefile     samples         usr
CREDITS  fs             Kconfig  mm           scripts         virt
danielshen@ubuntu:~/linux-4.8$ cd arch/x86/entry/syscalls/syscall_64.tbl
bash: cd: arch/x86/entry/syscalls/syscall_64.tbl: Not a directory
danielshen@ubuntu:~/linux-4.8$ cd arch/x86/entry/syscalls/
danielshen@ubuntu:~/linux-4.8/arch/x86/entry/syscalls$ nano syscall_64.tbl
```

danielshen@ubuntu: ~/linux-4.8/arch/x86/entry/syscalls

```
  GNU nano 2.5.3            File: syscall_64.tbl                        Modifie

212     common  lookup_dcookie          sys_lookup_dcookie
213     common  epoll_create            sys_epoll_create
214     64      epoll_ctl_old
215     64      epoll_wait_old
216     common  remap_file_pages        sys_remap_file_pages
217     common  getdents64              sys_getdents64
218     common  set_tid_address         sys_set_tid_address
219     common  restart_syscall         sys_restart_syscall
220     common  semtimedop              sys_semtimedop
221     common  fadvise64               sys_fadvise64
222     64      timer_create            sys_timer_create
223     common  mysyscall               sys_mysyscall
224     common  timer_gettime           sys_timer_gettime
225     common  timer_getoverrun        sys_timer_getoverrun
226     common  timer_delete            sys_timer_delete
227     common  clock_settime           sys_clock_settime
228     common  clock_gettime           sys_clock_gettime
229     common  clock_getres            sys_clock_getres
230     common  clock_nanosleep         sys_clock_nanosleep
```

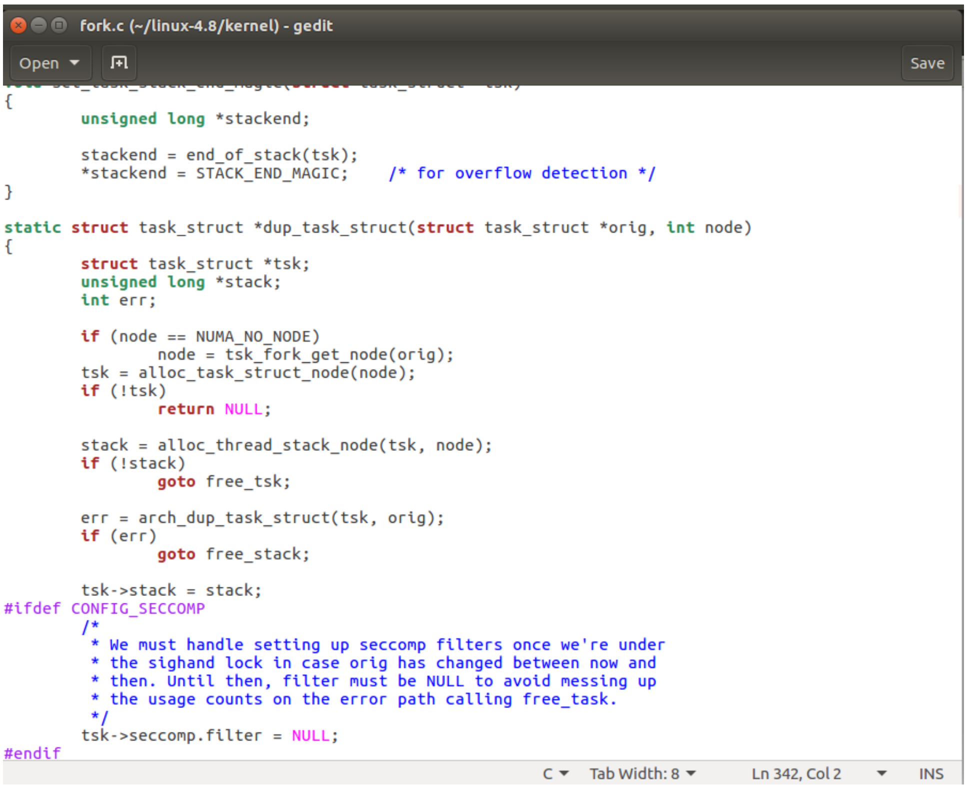
此后，在 include/linux/mm.h 中声明 pfcount 记录总缺页次数：

```
danielshen@ubuntu: ~/linux-4.8/include/linux
  GNU nano 2.5.3                 File: mm.h                               Modified

struct user_struct;
struct writeback_control;
struct bdi_writeback;

#ifndef CONFIG_NEED_MULTIPLE_NODES      /* Don't use mapnrs, do it properly */
extern unsigned long max_mapnr;
/*modified by zijin*/
extern unsigned long pfcount;
static inline void set_max_mapnr(unsigned long limit)
{
        max_mapnr = limit;
}
#else
static inline void set_max_mapnr(unsigned long limit) { }
#endif

extern unsigned long totalram_pages;
extern void * high_memory;
extern int page_cluster;
                              [ Unknown Command ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

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 字段以记录每个进程缺页的次数：

```
danielshen@ubuntu: ~/linux-4.8/include/linux
  GNU nano 2.5.3                File: sched.h                             Modified

         * flushed before IO is initiated or a stale TLB entry potentially
         * allows an update without redirtying the page.
         */
        bool writable;
};

struct task_struct {
        /*modified by zijin*/
        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
        struct llist_node wake_entry;
        int on_cpu;
        unsigned int wakee_flips;
                         [ XOFF ignored, mumble mumble ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

修改 dup_task_struct()函数将子进程的 pf 在复制之后置为 0。注意这一行代码需要放在比较靠后的位置：




```c
{
        unsigned long *stackend;

        stackend = end_of_stack(tsk);
        *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;
        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;

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

        tsk->stack = stack;
#ifdef CONFIG_SECCOMP
        /*
         * We must handle setting up seccomp filters once we're under
         * the sighand lock in case orig has changed between now and
         * then. Until then, filter must be NULL to avoid messing up
         * the usage counts on the error path calling free_task.
         */
        tsk->seccomp.filter = NULL;
#endif
```

```c
#ifdef CONFIG_CC_STACKPROTECTOR
        tsk->stack_canary = get_random_int();
#endif

        /*
         * One for us, one for whoever does the "release_tas
()" (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(stack, 1);

        kcov_task_init(tsk);
        tsk->pf = 0;
        return tsk;

free_stack:
        free_thread_stack(stack);
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 以记录当前进程产生缺页次数：

```
 */
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,
        PF_PK           =       1 << 5,
};

/* modified by zijin */
unsigned long pfcount;

/*
 * Returns 0 if mmiotrace is disabled, or if the fault is not
 * handled by mmiotrace:
```

```
{
        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;

        /*modifed by zijin*/
        pfcount++;
        current->pf++;

        tsk = current;
        mm = tsk->mm;

        /*
         * Detect and handle instructions that would cause a page fault for
         * both a tracked kernel page and a userspace page.
         */
        if (kmemcheck_active(regs))
```

最后，实现一个自己的 System_call：

```
  GNU nano 2.5.3          File: /linux-4.8/kernel/sys.c

#include <asm/io.h>
#include <asm/unistd.h>
extern unsigned long pfcount;
asmlinkage int sys_mysyscall(void)
{
printk("#$#\n");
printk("@all page fault: %lu\n", pfcount);
printk("@current page fault: %lu\n",current->pf);
struct task_struct *p = &init_task;
for(;(p=next_task(p))!=&init_task;)
{
printk("@The dirty page of process %d: %d\n",p->pid,p->nr_dirtied);
}
printk("$#$\n");
return 0;
}

#ifndef SET_UNALIGN_CTL
# define SET_UNALIGN_CTL(a, b)  (-EINVAL)

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

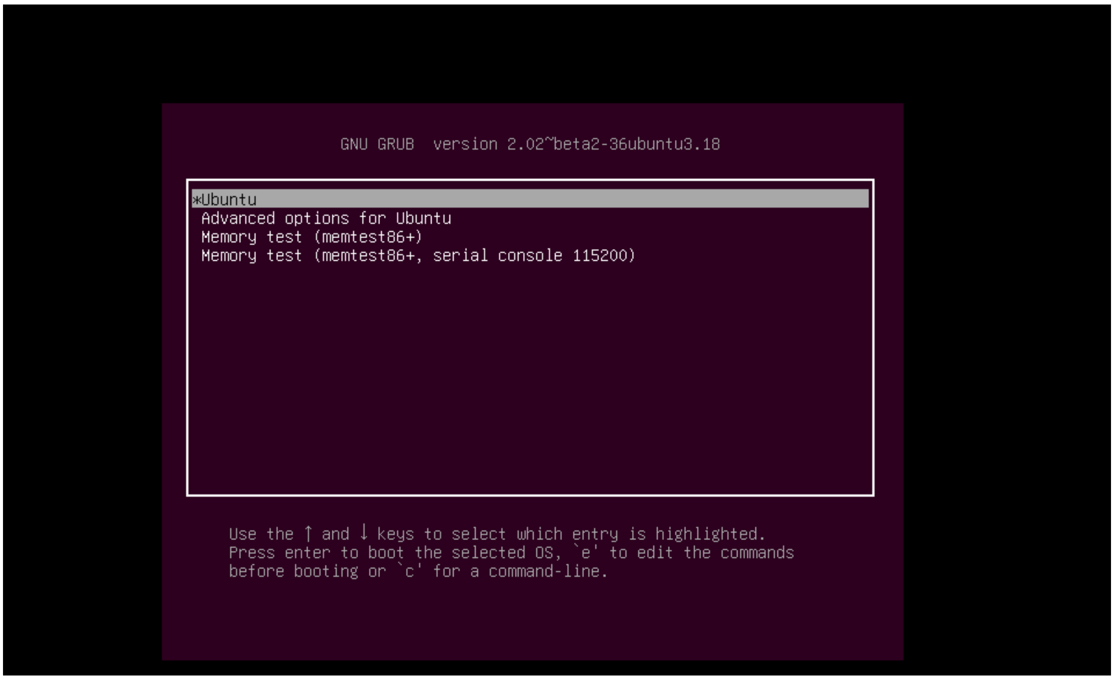
保存后执行 make。这个过程可能需要很长时间，故我使用 make -j 以释放所有 CPU 资源：

```
danielshen@ubuntu: ~/linux-4.8
  IHEX2FW firmware/emi62/midi.fw
  IHEX2FW firmware/emi62/spdif.fw
  IHEX    firmware/kaweth/new_code.bin
  IHEX    firmware/kaweth/trigger_code.bin
  IHEX    firmware/kaweth/new_code_fix.bin
  IHEX    firmware/kaweth/trigger_code_fix.bin
  IHEX    firmware/ti_3410.fw
  IHEX    firmware/ti_5052.fw
  IHEX    firmware/mts_cdma.fw
  IHEX    firmware/mts_gsm.fw
  IHEX    firmware/mts_edge.fw
  H16TOFW firmware/edgeport/boot.fw
  H16TOFW firmware/edgeport/boot2.fw
  H16TOFW firmware/edgeport/down2.fw
  IHEX    firmware/edgeport/down3.bin
  H16TOFW firmware/edgeport/down.fw
  IHEX2FW firmware/whiteheat_loader.fw
  IHEX2FW firmware/keyspan_pda/keyspan_pda.fw
  IHEX2FW firmware/whiteheat.fw
  IHEX2FW firmware/keyspan_pda/xircom_pgs.fw
  IHEX    firmware/cpia2/stv0672_vp4.bin
  IHEX    firmware/yam/9600.bin
  IHEX    firmware/yam/1200.bin
danielshen@ubuntu:~/linux-4.8$
```

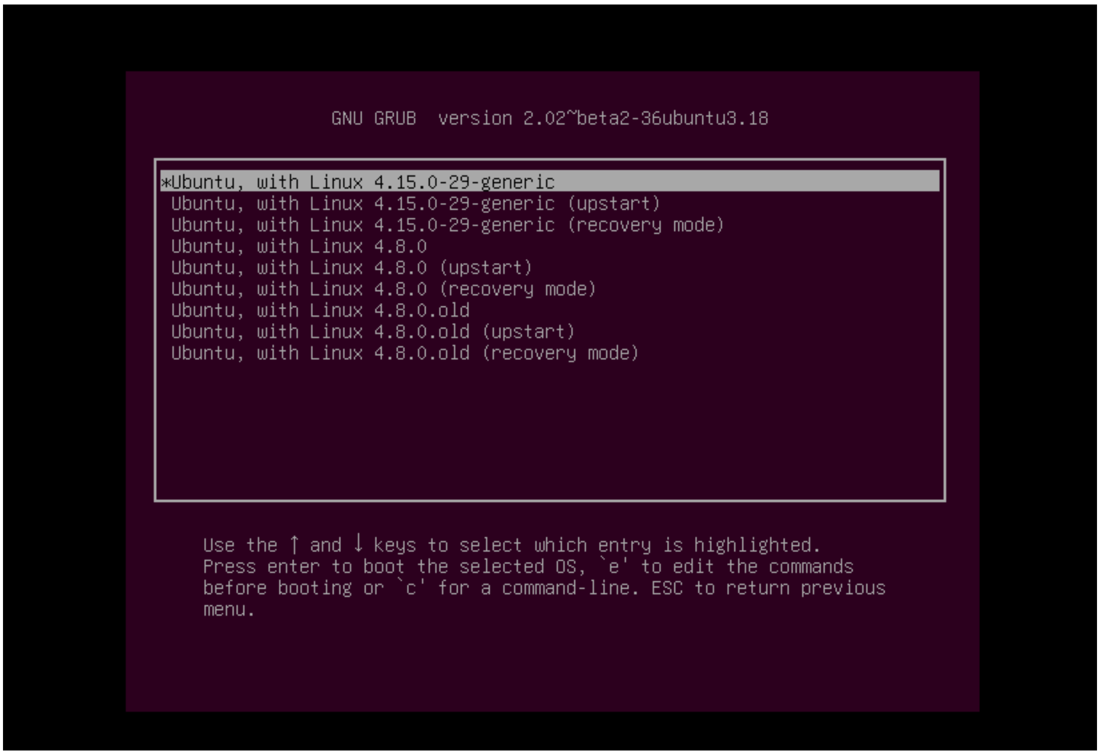
随后执行 make modules、make modules install 和 make install 命令进行最后配置：

```
danielshen@ubuntu: ~/linux-4.8
  AS [M]  arch/x86/crypto/sha512-mb/sha512_mb_mgr_submit_avx2.o
  AS [M]  arch/x86/crypto/sha512-mb/sha512_x4_avx2.o
  LD [M]  arch/x86/crypto/sha512-mb/sha512-mb.o
  CC [M]  arch/x86/events/intel/rapl.o
  LD [M]  arch/x86/events/intel/intel-rapl-perf.o
  CC [M]  arch/x86/events/intel/cstate.o
  LD [M]  arch/x86/events/intel/intel-cstate.o
  CC [M]  arch/x86/kernel/msr.o
  CC [M]  arch/x86/kernel/cpuid.o
  CC [M]  arch/x86/kernel/cpu/mcheck/mce-inject.o
  CC [M]  arch/x86/kvm/../../../virt/kvm/kvm_main.o
  CC [M]  arch/x86/kvm/../../../virt/kvm/coalesced_mmio.o
  CC [M]  arch/x86/kvm/../../../virt/kvm/eventfd.o
  CC [M]  arch/x86/kvm/../../../virt/kvm/irqchip.o
  CC [M]  arch/x86/kvm/../../../virt/kvm/vfio.o
  CC [M]  arch/x86/kvm/../../../virt/kvm/async_pf.o
  CC [M]  arch/x86/kvm/x86.o
^[^A  CC [M]  arch/x86/kvm/mmu.o
  CC [M]  arch/x86/kvm/emulate.o
  CC [M]  arch/x86/kvm/i8259.o
  CC [M]  arch/x86/kvm/irq.o
  CC [M]  arch/x86/kvm/lapic.o
  CC [M]  arch/x86/kvm/i8254.o
```

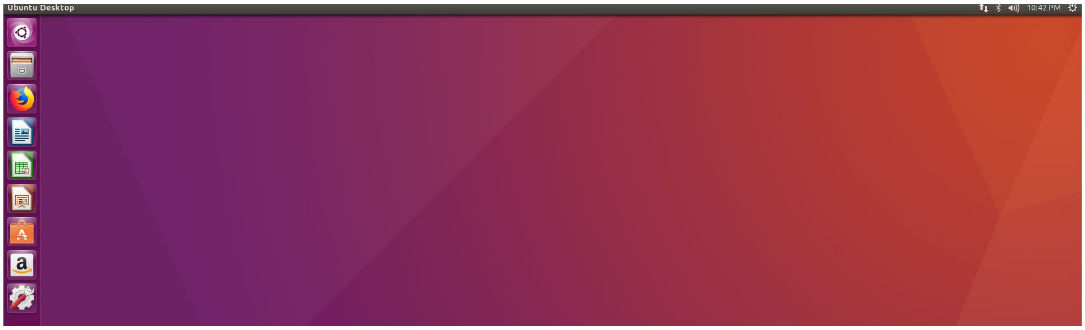
配置完毕，输入 sudo reboot 命令，回车按下的一瞬间按住 esc 键，直至进入内核切换界面



在这个界面选择 advanced options for ubuntu:



选择内核 Linux 4.8.0（无括号和.old 标注），回车启动，等待加载出图形界面：



3. 测试程序运行结果截图

编写读取结果的用户态程序。注意，本次使用的用户态程序和上一次实验使用的用户态程序是同一个：

```c
#include <stdio.h>
#include <string.h>
#include <linux/unistd.h>
#include <sys/syscall.h>
#define __NR_mysyscall 223


int main(){
        int ch;
        int output=0;
        FILE *fp;
        /*
        fp = fopen("/var/log/kern.log","w");
        if(fp==NULL)
        {//若打不开则输出错误信息
                printf("No Permission!\n");
                return 0;
        }
        fclose(fp);
        */
        //打开日志文件
        syscall(__NR_mysyscall);
        fp=fopen("/var/log/kern.log","r");
        if(fp==NULL)
        {//若打不开则输出错误信息
                printf("No Permission!\n");
                return 0;
        }

        //读取日志文件
        fseek(fp,0,SEEK_SET);
        //文件指针重定位到文件头
        ch=fgetc(fp);
        //找到内核模块输出记录开头
```

编译运行该程序，结果（部分）如下：

```
HEAD

all page fault: 547900
current page fault: 411
The dirty page of process 1: 0
The dirty page of process 2: 0
The dirty page of process 3: 0
The dirty page of process 4: 0
The dirty page of process 5: 0
The dirty page of process 6: 0
The dirty page of process 7: 0
The dirty page of process 8: 0
The dirty page of process 9: 0
The dirty page of process 10: 0
The dirty page of process 11: 0
The dirty page of process 12: 0
The dirty page of process 13: 0
The dirty page of process 14: 0
The dirty page of process 15: 0
The dirty page of process 16: 0
The dirty page of process 17: 0
The dirty page of process 18: 0
The dirty page of process 19: 0
The dirty page of process 20: 0
The dirty page of process 21: 0
The dirty page of process 22: 0
The dirty page of process 23: 0
The dirty page of process 24: 0
The dirty page of process 25: 0
The dirty page of process 26: 0
The dirty page of process 27: 0
The dirty page of process 28: 0
The dirty page of process 29: 0
The dirty page of process 30: 0
The dirty page of process 31: 0
The dirty page of process 32: 0
The dirty page of process 33: 0
The dirty page of process 34: 0
The dirty page of process 35: 0
The dirty page of process 36: 0
```

```
The dirty page of process 961: 0
The dirty page of process 963: 0
The dirty page of process 964: 0
The dirty page of process 965: 0
The dirty page of process 966: 0
The dirty page of process 967: 0
The dirty page of process 968: 0
The dirty page of process 969: 0
The dirty page of process 970: 0
The dirty page of process 971: 0
The dirty page of process 972: 0
The dirty page of process 973: 0
The dirty page of process 989: 0
The dirty page of process 990: 0
The dirty page of process 998: 1
The dirty page of process 1004: 0
The dirty page of process 1006: 88
The dirty page of process 1025: 11
The dirty page of process 1027: 1
The dirty page of process 1048: 0
The dirty page of process 1203: 0
The dirty page of process 1204: 0
The dirty page of process 1209: 0
The dirty page of process 1316: 0
The dirty page of process 1330: 0
The dirty page of process 1331: 0
The dirty page of process 1348: 0
The dirty page of process 1360: 1
The dirty page of process 1380: 1
The dirty page of process 1388: 0
The dirty page of process 1389: 0
The dirty page of process 1394: 1
The dirty page of process 1396: 12
The dirty page of process 1471: 0
The dirty page of process 1482: 0
The dirty page of process 1494: 0
The dirty page of process 1536: 1
The dirty page of process 1549: 0
The dirty page of process 1553: 0
```

```
The dirty page of process 1720: 0
The dirty page of process 1740: 0
The dirty page of process 1763: 2
The dirty page of process 1779: 7
The dirty page of process 1810: 0
The dirty page of process 1825: 0
The dirty page of process 1826: 0
The dirty page of process 1828: 5
The dirty page of process 1829: 0
The dirty page of process 1833: 0
The dirty page of process 1835: 0
The dirty page of process 1854: 0
The dirty page of process 1856: 0
The dirty page of process 1871: 0
The dirty page of process 1883: 0
The dirty page of process 1888: 0
The dirty page of process 1893: 0
The dirty page of process 1899: 0
The dirty page of process 1914: 0
The dirty page of process 1915: 0
The dirty page of process 1926: 7
The dirty page of process 1944: 0
The dirty page of process 2007: 0
The dirty page of process 2033: 0
The dirty page of process 2039: 0
The dirty page of process 2049: 0
The dirty page of process 2056: 0
The dirty page of process 2060: 9
The dirty page of process 2067: 0
The dirty page of process 2086: 0
The dirty page of process 2109: 0
The dirty page of process 2118: 0
The dirty page of process 2151: 0
EOF
```

4. 结果分析

结合上述截图分析，大部分进程的脏页数都为 0，少部分进程拥有 0-100 的脏页数，极少部分进程拥有略多于 100 的脏页数。数据基本合理，满足客观事实和实验要求，可以认为实验取得了成功。下面回答实验刚开始提出的问题：

- 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，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？
  答：有一定区别。Test 程序中的“缺页次数”指的是调用 do_page_fault 的次数，真实的缺页次数应当不大于程序中的“缺页次数”，因为一次缺页可能调用多次 do_page_fault 函数，也可能有其他进程调用这一函数。
- 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。
  答：还可以通过内核模块的方式添加简单的系统调用。其原因是：编译内核的方式费时间，一般的 PC 机都要两三个小时，而且不方便调试，一旦出现问题前面的工作都前功尽弃。其基本步骤是：1. 找系统调用表在内存中的位置；2. 找一个空闲的系统调用号；3. 修改寄存器写保护位；4. 实现系统调用函数 5. 执行 make 后将编译好的模块插入内核。
- 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。
  答：无论是采用插入模块的方式还是修改代码的方式编辑系统调用，严格来说都不是绝对安全的，甚至存在严重风险。以我自己的实验经历为例，在实验一中，如果内核模块代码的输出存在格式上的问题，可能破坏系统日志的结构；在实验二中，修改内核代码的行为更是危险重重，除了编译时间漫长，无法应对可能存在的系统宕机情况外，一个小小的 runtime error 就可能导致黑屏、卡死、GUI 无法加载等致命错误。在添加系统调用号的操作中，我为了使用 223 号调用而删除了一个已经存在的调用（获取系统时间的调用），这种修改如果不能得到专业人士的把关，势必会影响一些依赖此调用运行的应用程序，甚至操作系统模块的功能。

5. 源程序

系统调用函数：

```
extern unsigned long pfcount;
asmlinkage int sys_mysyscall(void)
{
    printk("#$#\n");
    printk("@all page fault: %lu\n", pfcount);
    printk("@current page fault: %lu\n",current->pf);
    struct task_struct *p = &init_task;
    while((p=next_task(p))!=&init_task)
    {
        printk("@The dirty page of process %d: %d\n",p->pid,p->nr_dirtied);
    }
    rintk("$#$\n");
    return 0;
}
```

用户态程序（和实验一使用的是同一个）：

```
#include <stdio.h>
#include <string.h>
```

```
#include <linux/unistd.h>
#include <sys/syscall.h>
#define __NR_mysyscall 223


int main(){
    int ch;
    int output=0;
    FILE *fp;
    //打开日志文件
    syscall(__NR_mysyscall);
    fp=fopen("/var/log/kern.log","r");
    if(fp==NULL)
        {//若打不开则输出错误信息
        printf("No Permission!\n");
        return 0;
    }

    //读取日志文件
    fseek(fp,0,SEEK_SET);
        //文件指针重定位到文件头
    unsigned long offset = 0;
    ch=fgetc(fp);
    //找到内核模块输出记录开头
        FILE *begin = NULL;
    while(ch!=EOF)
    {
        while(ch!=EOF)
        {
            //输出记录开头的特殊标记为"#$#"
            if(ch=='#')
            {
                ch=fgetc(fp);
                if(ch=='$')
                {
                    ch=fgetc(fp);
                    if(ch=='#')
                    {

                        printf("HEAD\n");
                        break;
                    }
                }
            }
```

```
            ch=fgetc(fp);
        }
        //打印出内核模块的输出记录，@是用于识别每一行的特殊标记
        while(ch!=EOF)
        {
            if(ch == '@')
            {
                output = 1;
            }
            else if(ch == '\n')
            {
                printf("\n");
                output = 0;
            }
            else if(ch =='$')
            {//输出记录结尾的特殊标记为"$#$"
            ch=fgetc(fp);
            if(ch=='#')
                        {
                ch=fgetc(fp);
                if(ch=='$')
                {
                    printf("END\n");
                    ch=fgetc(fp);
                    break;
                }
            }
            }if(output == 1 && ch != '@')
            {
                printf("%c",ch);
            }
            ch=fgetc(fp);
        }
    }
    //关闭日志文件
    fclose(fp);
    return 0;
}
```

三、讨论、心得（20 分）

本次实验的操作步骤较为清晰，很多提示也都在实验指导中给出，看似要比实验一简单。但在完成该实验的过程中，我依然遇到了各种没有预料到的困难，有的至今没有找到最优的解决方案：

1. Ubuntu 的版本问题。本次实验要求的 Ubuntu 版本是 16.04，而我常用的版本是 18.x。天真的我认为发行版的更新不会影响实验的推进，但残酷的现实给我上了一课——切换内核后，虚拟机无论如何也无法启动了。最终我选择重新安装 16.04 版本的 Ubuntu，将原来的 18.x 在移动硬盘中备份后从本机中删除了。整个过程耗费了我将近 3 个小时的时间。
2. 即使是正确的版本，问题依然存在。我使用新安装的 16.04LTS 和在阿里云镜像下载的内核和补丁严格按照步骤推进实验，幻想这次可以一遍成功。但残酷的现实又给我上了一课——切换内核后，虚拟机启动了，但提示：

```
    Gave up waiting for root device. Common problems:
    -boot args (cat.proc/cmdline)
        -check root delay=(did the system wait long enough?)
        -check root=(did teh system wait for the right device?)
    -miss modules (cat/proc/modules;ls/dev)
ALERT! /dev/disk/by-uuid/acc3414d-926c-453c-b458-cf47088d77d2 does not exist.dropping to a shell!
```

   期待中的图形界面并没有显示。检查所有文件修改操作无误后，上网查阅资料得知，可能是 kernel 中的某个 uuid 字段出了问题，尝试了一些方法，无果，遂怀疑是 vmware 配置的问题，但拷贝室友的虚拟机文件在本机的 vmware 上运行，并没有问题……最终我再一次重装了 Ubuntu，这次使用的全部是官网资源，然后成功启动了……所以是阿里云的问题？并不是，因为室友就是从阿里云上下载的内核，一点问题也没有……可能是之前打补丁或者输某些命令时出了问题吧。
3. 图形界面算是打开了，但还是有问题——脏页数太多了，有的进程有上万的脏页数，这很明显是不可能的。查阅资料得知，是 tsk->pf=0 的位置错了……这是因为 arch_dup_task_struct(tsk,orig)函数的关系，如果看一下这个函数怎么实现的话就会知道：它直接把 orig 赋给了 tsk，所以如果 pf 在上面初始化，子进程的 pf 就还是父进程的，不满足要求。
4. 还有一个坑是在 1 之前踩的：reboot 之后忘了长按 esc 切换内核……

总之这个实验可以说是处处皆坑。不管坑是老师故意让我们踩的还是确实没有说清楚，这次实验的收获还是蛮大的，可以说是“痛并快乐着”。以及，这是我上大学之后写的最长的实验心得……因为真的很有“心得”。