

## 实验二：统计操作系统缺页次数

# 目录

一．实验目的	-----3
--------	--------

二．实验内容	-----3
--------	--------

三．实验步骤	-----3
--------	--------

# 统计操作系统缺页次数

## 一 实验目的

学习虚拟内存的基本原理和 Linux 虚拟内存管理技术；

深入理解、掌握 Linux 的按需调页过程；

掌握内核模块的概念和操作方法，和向 /proc 文件系统中增加文件的方法；

综合运用内存管理、系统调用、proc 文件系统、内核编译的知识。

## 二 实验内容

### 1. 原理

Linux 的虚拟内存技术采用按需调页，当 CPU 请求一个不在内存中的页面时，会发生缺页，缺页被定义为一种异常（缺页异常），会触发缺页中断处理流程。每种 CPU 结构都提供一个 `do_page_fault` 处理缺页中断。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。因此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。本实验通过动态加载模块的方法，利用 /proc 文件系统作为中介来获取该值。

### 2. 实验环境

操作系统：Ubuntu 12.04（内核版本为 3.2.0-23-generic-pae）

内核源码：linux-3.2.58

## 三 实验步骤

### 1. 下载一份内核源代码并解压

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。推荐使用 Linux 的官方网站：<http://www.kernel.org>。

在 terminal 下可以通过 `wget` 命令下载源代码：

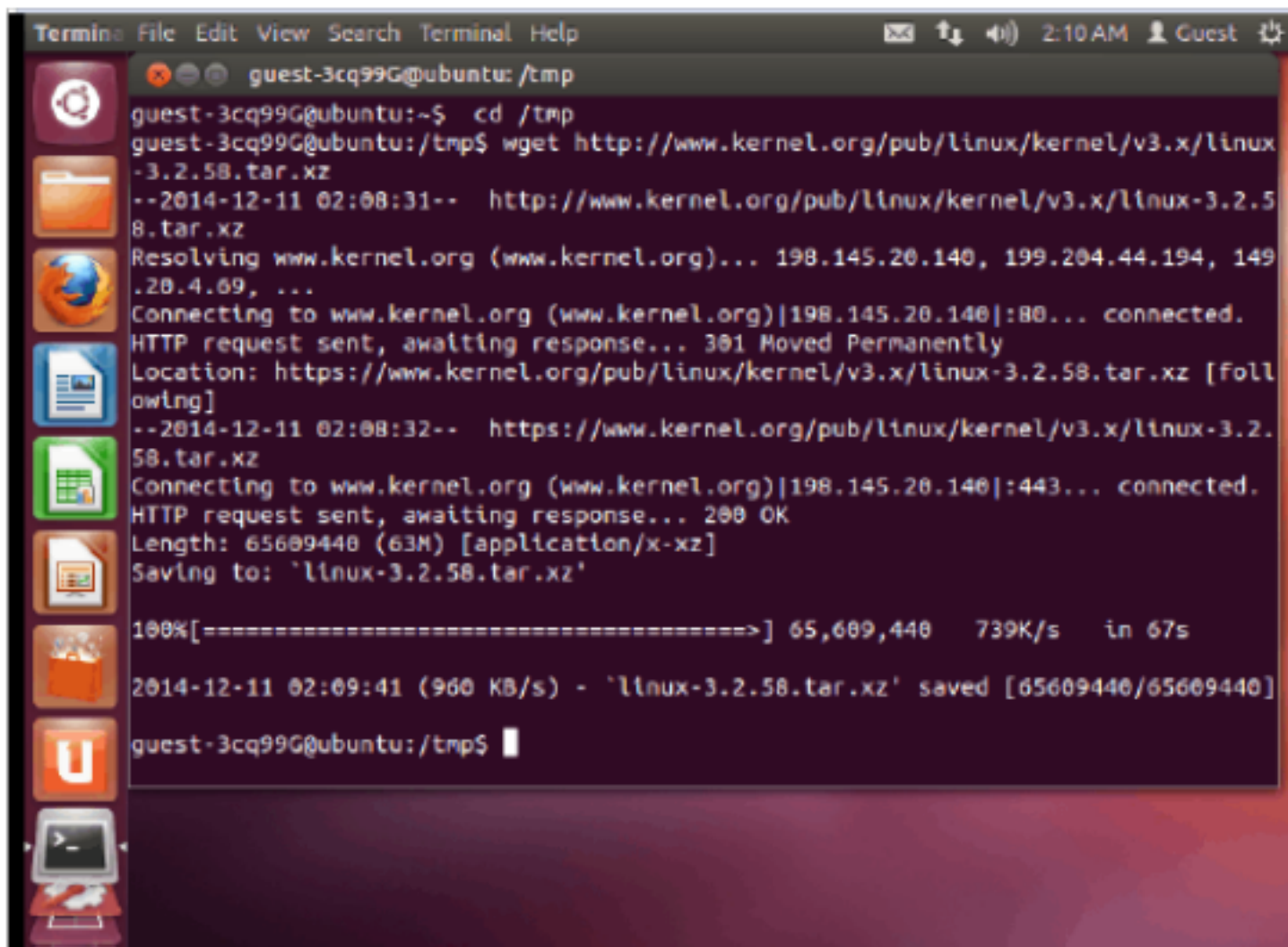
```
$ cd /tmp
```

```
$ wget http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.2.58.tar.xz
```

切换到 root 身份，解压源代码到 /usr/src 目录下：

```
# xz -d linux-3.2.58.tar.xz
```

```
# tar -xvf linux-3.2.58.tar -C /usr/src
```



## 2. 修改内核源代码，添加统计变量

1、切换到预编译内核目录

```
#cd /usr/src/linux-3.2.58
```

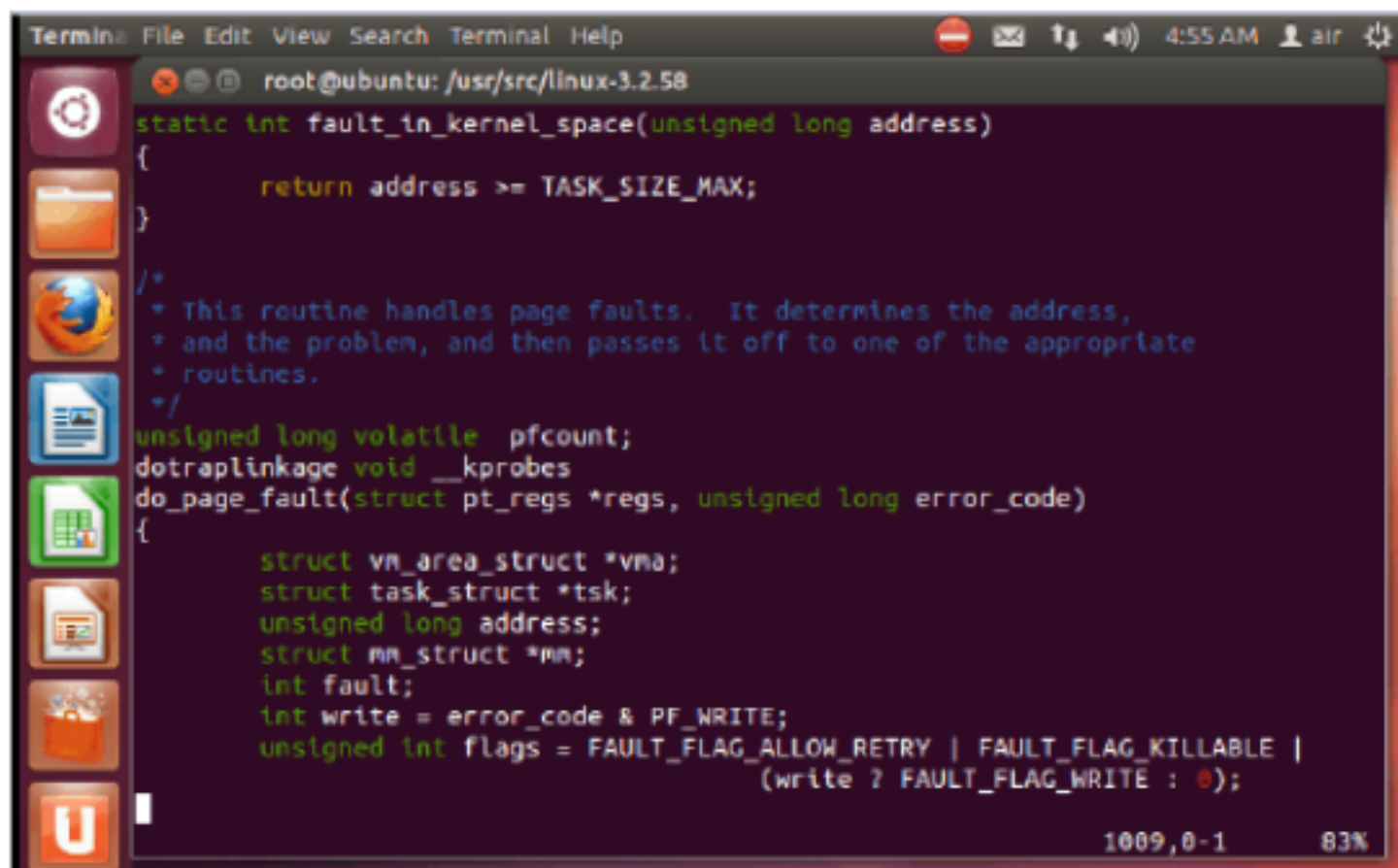
2、修改处理内存访问异常的代码

//用 vi 编辑器打开 fault.c ,一般使用 Intel x86 体系结构，则修改 arch/x86/ 目录下的文件

```
#vi arch/x86/mm/fault.c
```

//在 do\_page\_fault 函数的上一行定义统计缺页次数的全局变量 pfcoun

```
Unsigned long volatile pfcoun
```



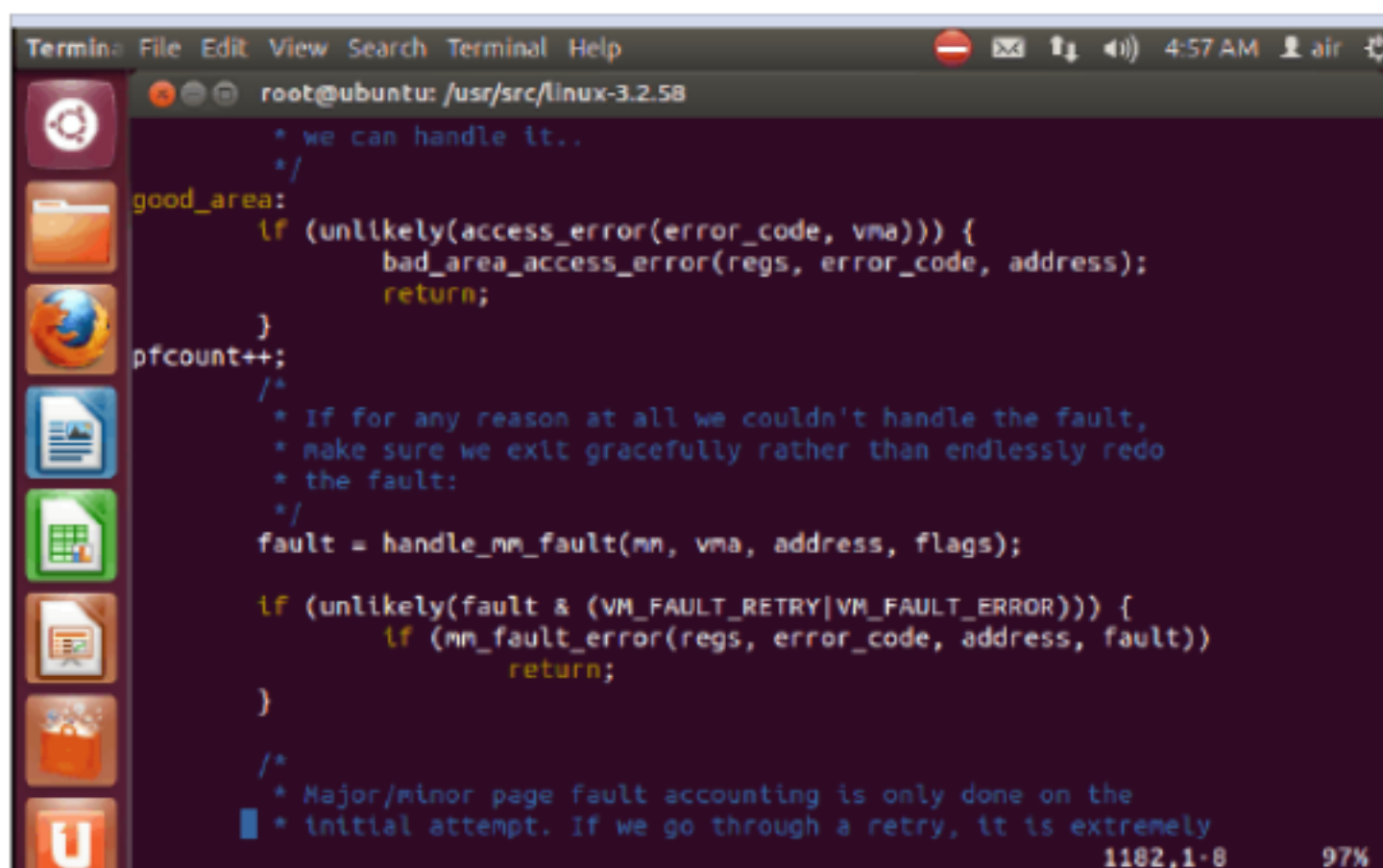
```
Termin File Edit View Search Terminal Help
root@ubuntu: /usr/src/linux-3.2.58

static int fault_in_kernel_space(unsigned long address)
{
    return address >= TASK_SIZE_MAX;
}

/*
 * This routine handles page faults.  It determines the address,
 * and the problem, and then passes it off to one of the appropriate
 * routines.
 */
unsigned long volatile pfcount;
do_traplinkage void __kprobes
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    unsigned long address;
    struct mm_struct *mm;
    int fault;
    int write = error_code & PF_WRITE;
    unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE |
        (write ? FAULT_FLAG_WRITE : 0);

    1009,0-1 83%
```

//将 pfcount 加入到 do\_page\_fault 中，用以统计缺页次数  
pfcount++;



```
Termin File Edit View Search Terminal Help
root@ubuntu: /usr/src/linux-3.2.58

    /* we can handle it..
    */
good_area:
    if (unlikely(access_error(error_code, vma))) {
        bad_area_access_error(regs, error_code, address);
        return;
    }
    pfcount++;
    /*
     * If for any reason at all we couldn't handle the fault,
     * make sure we exit gracefully rather than endlessly redo
     * the fault:
     */
    fault = handle_mm_fault(mm, vma, address, flags);

    if (unlikely(fault & (VM_FAULT_RETRY|VM_FAULT_ERROR))) {
        if (mm_fault_error(regs, error_code, address, fault))
            return;
    }

    /*
     * Major/minor page fault accounting is only done on the
     * initial attempt. If we go through a retry, it is extremely
     1182,1-8 97%
```

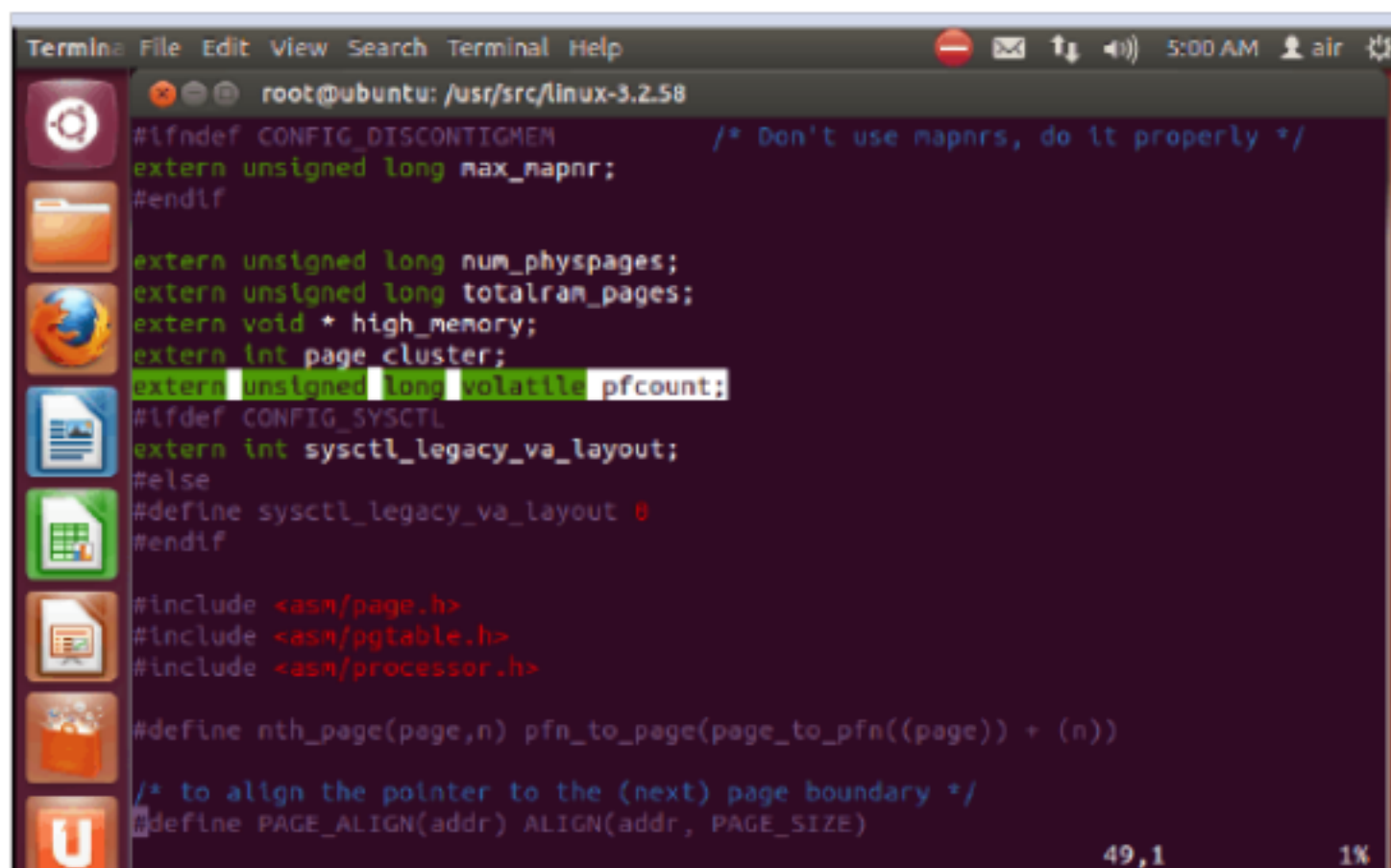
### 3、修改内存管理代码

//用 vi 编辑器打开头文件 mm.h

#vi include/linux/mm.h

//在 mm.h 中加入全局变量 pfcount 的声明，代码加在 extern int  
page\_cluster;语句之后

extern unsigned long volatile pfcount;



```
Termin File Edit View Search Terminal Help
root@ubuntu: /usr/src/linux-3.2.58

#ifndef CONFIG_DISCONTIGMEM /* Don't use mapnr, do it properly */
extern unsigned long max_mapnr;
#endif

extern unsigned long num_physpages;
extern unsigned long totalram_pages;
extern void * high_memory;
extern int page_cluster;
extern unsigned long volatile pfcount;
#ifdef CONFIG_SYSCTL
extern int sysctl_legacy_va_layout;
#else
#define sysctl_legacy_va_layout 0
#endif

#include <asm/page.h>
#include <asm/pgtable.h>
#include <asm/processor.h>

#define nth_page(page,n) pfn_to_page(page_to_pfn((page)) + (n))

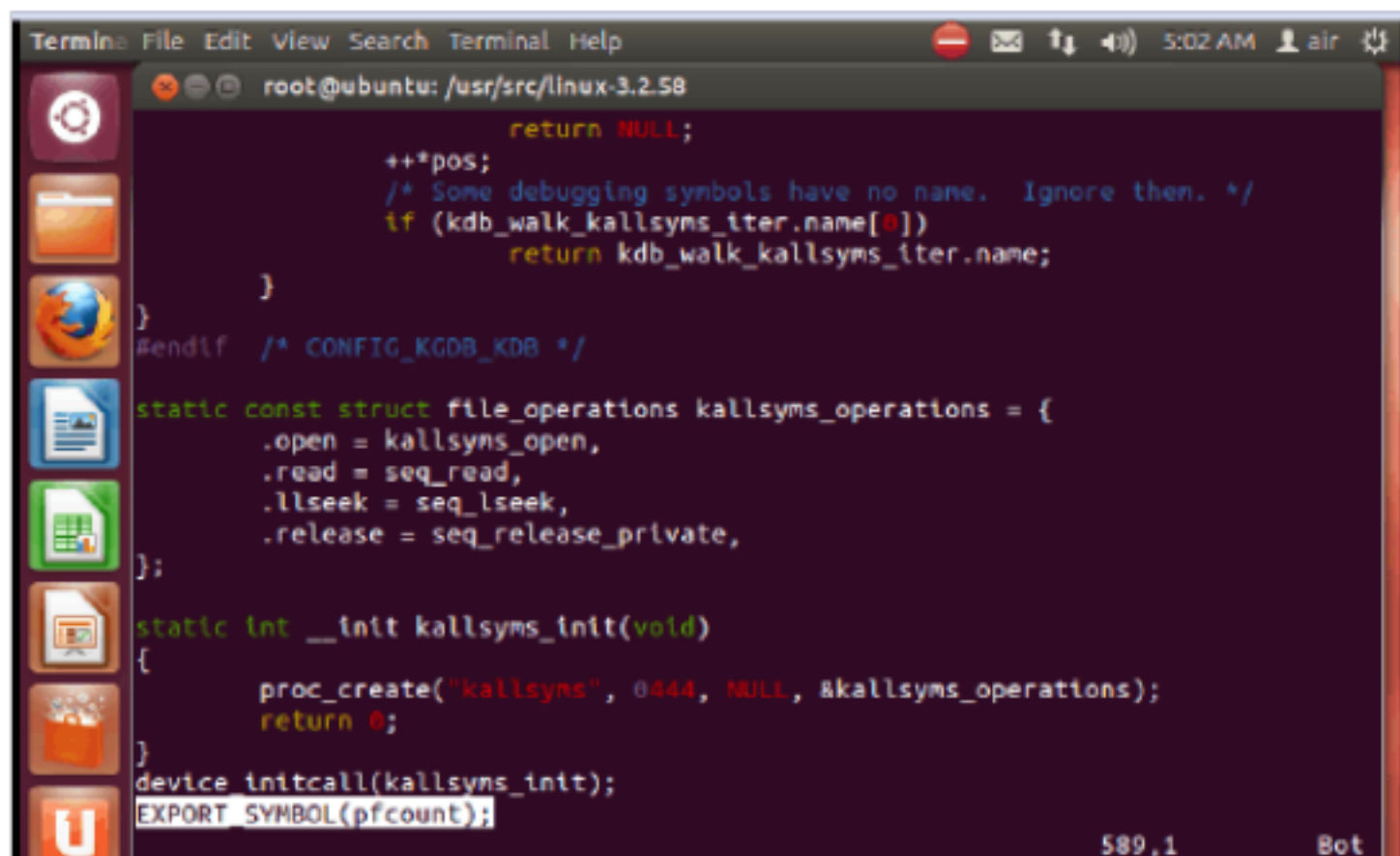
/* to align the pointer to the (next) page boundary */
#define PAGE_ALIGN(addr) ALIGN(addr, PAGE_SIZE)
```

4、导出 pfcount 全局变量，让整个内核（包括模块）都可以访问。方法是：

```
#vi kernel/kallsyms.c
```

```
//在文件最后加入一行代码
```

```
EXPORT_SYMBOL(pfcount);
```



```
Termin File Edit View Search Terminal Help
root@ubuntu: /usr/src/linux-3.2.58

        return NULL;
        ++*pos;
        /* Some debugging symbols have no name. Ignore them. */
        if (kdb_walk_kallsyms_iter.name[0])
            return kdb_walk_kallsyms_iter.name;
    }
}
#endif /* CONFIG_KGDB_KDB */

static const struct file_operations kallsyms_operations = {
    .open = kallsyms_open,
    .read = seq_read,
    .llseek = seq_lseek,
    .release = seq_release_private,
};

static int __init kallsyms_init(void)
{
    proc_create("kallsyms", 0444, NULL, &kallsyms_operations);
    return 0;
}
device_initcall(kallsyms_init);
EXPORT_SYMBOL(pfcount);
```

### 3. 配置编译新内核

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)。首先进入内核源代码目录:

```
# cd /usr/src/linux-3.2.58
```

如果不是第一次编译的话,有必要将内核源代码树置于一种完整和一致的状态(如果是第一次可跳过此步)。因此,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
```

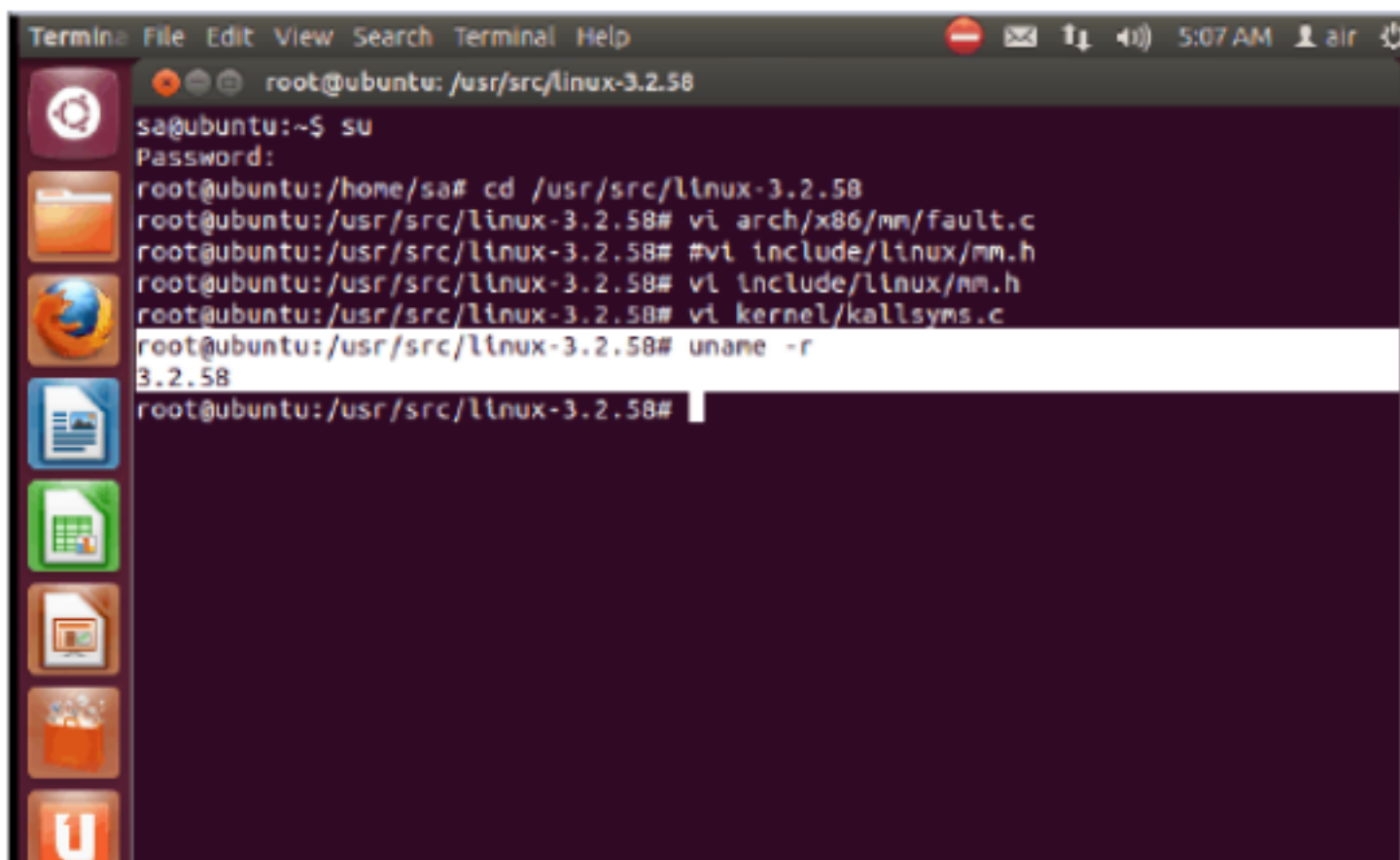
然后配置编译选项(此处使用原内核的配置文件,完整的配置命令看操作提示):

```
# cp /boot/config-3.2.0-23-generic-pae .config
```

该命令的作用是将原内核配置文件拷贝的当前目录下,并命名为 `.config`。若需要进一步修改配置请参照操作提示。

用编译 Linux 内核预备实验中的方法完成新内核的配置、编译、替换,重启后验证是否完成替换。

```
#uname -r //如果为 3.2.58 (与你采用的新内核版本一至)说明替换完成
```



#### 4.编写读取 pfcount 值的模块代码

系统重启后，执行如下操作：

```
cd /home/sa
    #mkdir source          //在当前用户目录下创建    source 文件夹，用于存放
编写用户程序
    #cd source             //切换到  source 目录
    #vi pf.c               //新建用于构建模块的代码
```

```
-----
/*pf.c*/
/*modules program*/
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <linux/proc_fs.h>
#include <linux/string.h>
#include <asm/uaccess.h>

struct proc_dir_entry *proc_pf;
struct proc_dir_entry *proc_pfcoun;
extern unsigned long      volatile pfcoun;

static inline struct proc_dir_entry *proc_pf_create(const char*
name, mode_t mode, read_proc_t * get_info)
{
    return
create_proc_read_entry(name,mode,proc_pf,get_info,NULL);
}

int get_pfcoun(char *buffer, char **start, off_t offset, int
length, int *peof, void *data)
{
    int len = 0;
    len = sprintf(buffer, "%ld \n", pfcoun);
    return len;
}

static int pf_init(void)
```



```
{
    proc_pf = proc_mkdir("pf", 0);
    proc_pf_create("pfcount", 0, get_pfcount);
    return 0;
}
```

```
static void pf_exit(void)
{
    remove_proc_entry("pfcount",proc_pf);
    remove_proc_entry("pf",0);
}
```

```
module_init(pf_init);
module_exit(pf_exit);
MODULE_LICENSE("GPL");
```

## 5. 编译、构建内核模块

```
cd /home/sa/source
```

```
#vi Makefile      //在 source 目录下建立 Makefile 文件
在 Makefile 中添加如下内容：
```

```
obj-m := pf.o
编译内核模块：
```

```
make -C /usr/src/linux-3.2.58 SUBDIRS=/home/sa/source modules
```

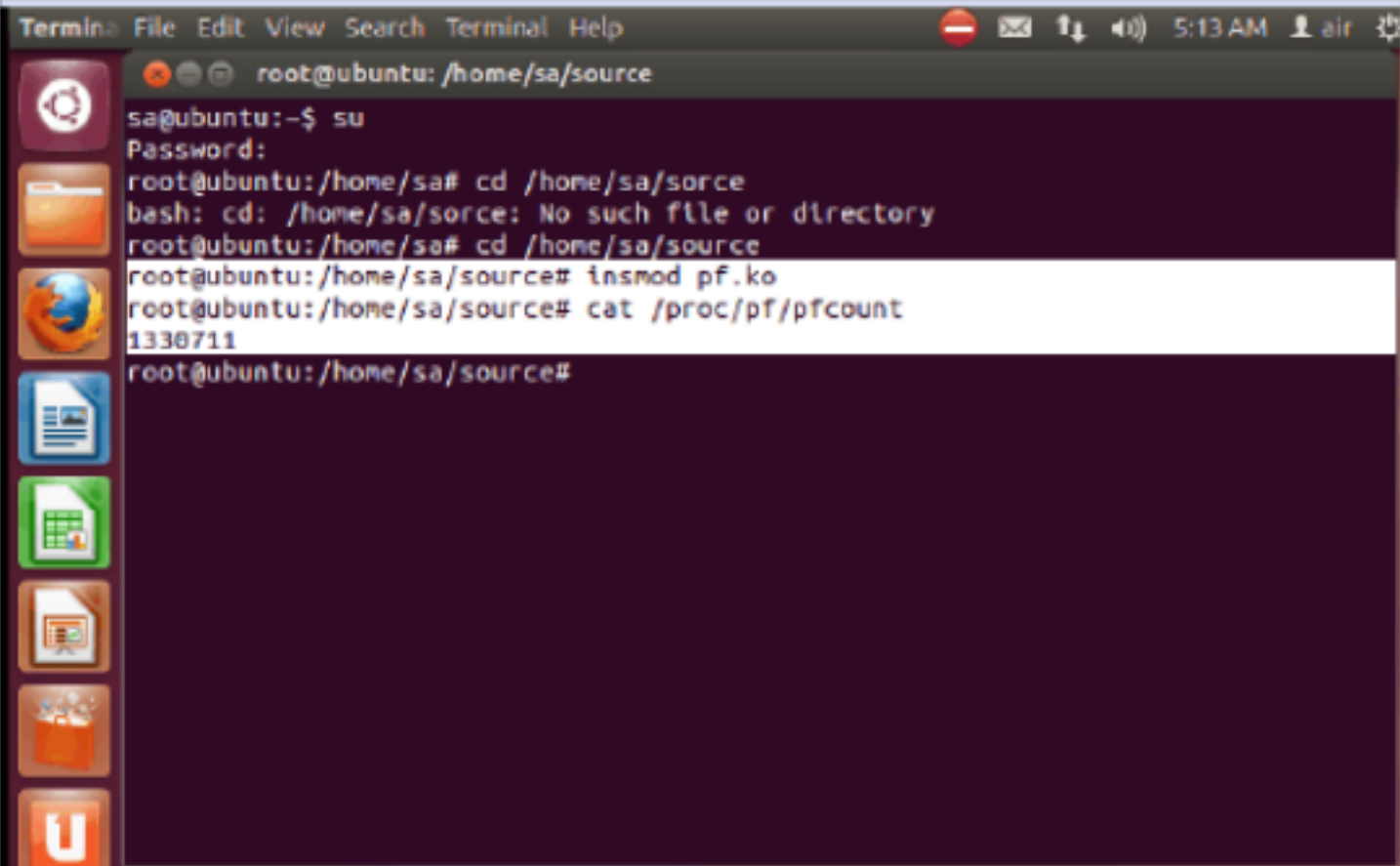
## 6. 加载模块到内核中

执行加载模块命令：

```
#insmod pf.ko
```

查看统计缺页次数：

```
#cat /proc/pf/pfcount
```



The image shows a terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Search", "Terminal", and "Help". The window has a dark theme and a sidebar on the left with icons for various applications. The terminal content is as follows:

```
root@ubuntu: /home/sa/source
sa@ubuntu:~$ su
Password:
root@ubuntu:/home/sa# cd /home/sa/sorce
bash: cd: /home/sa/sorce: No such file or directory
root@ubuntu:/home/sa# cd /home/sa/source
root@ubuntu:/home/sa/source# insmod pf.ko
root@ubuntu:/home/sa/source# cat /proc/pf/pfcount
1330711
root@ubuntu:/home/sa/source#
```