成绩：_____

江 蘇 大 學

JIANGSU UNIVERSITY

2021-2022 学年第 1 学期

《操作系统》课程设计

题目：统计 Linux 系统缺页的次数

学院名称：_____

班级学号：_____

学生姓名：_____

教师姓名：薛安荣

教师职称：教　授

2022 年 1 月

目录

一、 设计目的与要求 1
1.1 设计目的 1
1.1.1 知识方面 1
1.1.2 能力与素质方面 1
1.2 设计要求 1
二、 设计内容 1
三、 设备与环境 1
四、 设计思想 2
4.1 下载解压内核 2
4.2 修改源码 2
4.2.1 切换当前路径到源码所在路径 2
4.2.2 定义缺页中断统计变量 2
4.2.3 修改内存管理代码 3
4.2.4 导出全局变量 pfcount 3
4.3 生成编译配置文件 3
4.3.1 安装编译所需工具 3
4.3.2 清理历史编译文件 4
4.3.3 生成配置文件 4
4.4 内核编译与安装 4
4.4.1 内核编译 4
4.4.2 模块编译 4
4.4.3 安装模块 4
4.4.4 安装内核 5
4.4.5 重启运行新内核 5
4.5 查看缺页中断次数 5
4.5.1 编写缺页中断内核模块 5
4.5.2 加载新模块到内核 7
4.5.3 用户态下查看缺页中断次数 7
五、 主要数据结构和流程 7
5.1 缺页中断内核模块 7
5.2 设计思路流程 8
六、 实验测试结果及结果分析 10
6.1 测试结果 10
6.2 结果分析 10
七、 课程设计总结 10
八、 附件 12
附件 1 课程设计答辩记录 12
8.1.1 答辩内容 12
8.1.2 答辩流程 12
附件 2 源程序清单 13

一、 设计目的与要求

1.1 设计目的

1.1.1 知识方面

（1）掌握操作系统功能模块的设计与实现方法。

1.1.2 能力与素质方面

（1） 能够在阅读和分析开源操作系统的基础上，对其进行功能模块划分；能够指出现有功能模块的不足，并能够通过文献的研究给出解决方案。
（2）能够完成操作系统功能模块的设计、实现与测试，同时在设计操作系统功能模块中，能体现优化和创新意识。
（3）能够制定合理的实验方案及对实验结果进行分析并得出结论，针对实验结果分析解决过程的影响因素，论证解决方案的合理性，以获得有效结论。
（4）能够根据设计任务和要求组成团队，分工协作，并能承担个体、团队成员以及负责人的角色。
（5）能够用口头和书面方式清晰表述设计原理及相关概念与原理，包括陈述发言，清晰表达和回应指令。
（6）能够撰写比较规范的课程设计报告。

1.2 设计要求

（1）能够借助文献研究，分析计算机领域复杂工程问题的解决过程的影响因素，论证解决方案的合理性，以获得有效结论。
（2）根据计算机专业技术知识，能够有效地实施单体设计，并具有创新意识。
（3）理解团队管理模式，能够承担承担个体、团队成员以及负责人的角色。
（4）能够就计算机领域复杂工程问题与社会公众和同行进行有效交流和沟通，包括陈述发言、清晰表达和回应指令。

二、 设计内容

通过在 Linux 内核中自建变量，并利用/proc 文件系统作为中介的方法，统计系统缺页的次数。
要求：
（1）在内核中实现缺页次数统计；
（2）编译并安装新内核；
（3）新建内核模块，并加载到新内核，通过/proc 实现用户态下查看缺页次数。

三、 设备与环境

表 1 实验环境

云计算系统	Alibaba Cloud Elastic Compute Service

操作系统	CentOS 7.6.1810
内核版本	Linux-4.20.4

四、 设计思想

4.1 下载解压内核

（1）下载 Linux-4.20.4 内核源码

```
# wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.20.4.tar.xz
```

（2）解压内核，并移动到/usr/src/kernels 目录下

```
# tar -xvf linux-4.20.4.tar.xz -C
# mv linux-4.20.4 /usr/src/kernels
```

4.2 修改源码

4.2.1 切换当前路径到源码所在路径

```
# cd /usr/src/kernels/linux-4.20.4
```

4.2.2 定义缺页中断统计变量

查阅资料，分析 Linux 内核文件目录组织结构定位到缺页中断处理是在 arch/x86/mm/fault.c 文件中。

用 vim 打开 fault.c 文件

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

进一步阅读 fault.c 文件源码，分析处理流程以及各个函数的作用，并结合相关注释发现 Linux 缺页中断判断函数是__do_page_fault，该函数通过一个 if-else 语句判断缺页中断是内核中断还是用户中断；如果当前执行流程在内核态，说明在内核态出了问题，进入内核态异常处理，由函数_do_kern_addr_fault 函数完成。

```
dotted_do_page_fault(struct pt_regs *regs, unsigned long hw_error_code,
                unsigned long address)
{
        prefetchw(&current->mm->mmap_sem);

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

        /* Was the fault on kernel-controlled part of the address space? */
        if (unlikely(fault_in_kernel_space(address)))
                do_kern_addr_fault(regs, hw_error_code, address);
        else
                do_user_addr_fault(regs, hw_error_code, address);
}
NOKPROBE_SYMBOL(__do_page_fault);
```

缺页中断判断函数

内核缺页中断异常处理

图 1 缺页中断类型判断

因此，我在 fault.c 文件中添加缺页中断统计变量 pfcount，然后在内核缺页中断异常处理函数中添加 pfcount 自增语句。每当 Linux 系统在内核中发生一次缺页中断，该语句便会自增，从而在内核中实现缺页中断次数统计。

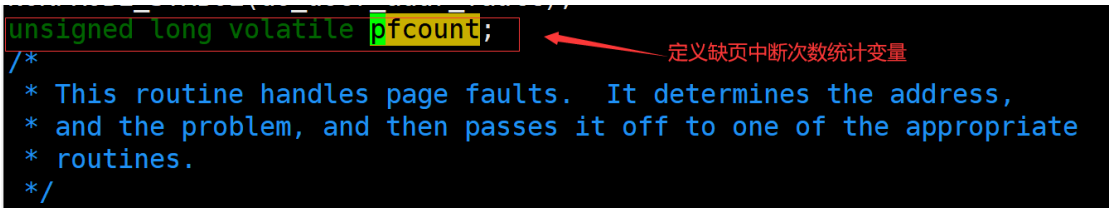

图 2 缺页中断变量声明

图 3 内核缺页中断异常处理

4.2.3 修改内存管理代码

首先，打开 include/linux/mm.h 头文件

```
# vim include/linux/mm.h
```

在 mm.h 中加入全局变量 pfcount 的声明 extern unsigned long volatile pfcount;代码加在 extern int page_cluster;语句之后,如图所示：

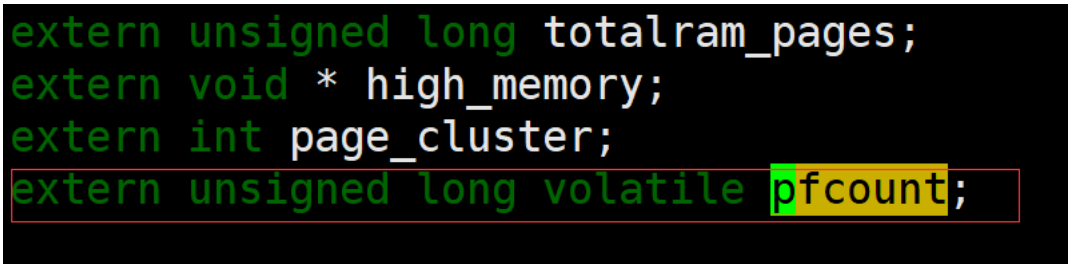

图 4 内存管理头文件加入 pfcount 变量声明

4.2.4 导出全局变量 pfcount

我们需要导出 pfcount 全局变量，以便于让系统所有模块访问；即在 kernel/kallsyms.c 文件的最后一行加入 EXPORT_SYMBOL(pfcount);可通过语句直接加入代码，无需打开文件。

```
# echo 'EXPORT_SYMBOL(pfcount);'>>kernel/kallsyms.c
```

为了确保其他模块可以访问，可通过下面语句进行验证

```
# cat kernel/kallsyms.c | grep pfcount
```

当出现下面语句输出时，证明前期操作正确

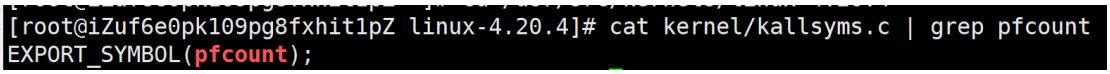

图 5 验证 pfcount 导出

4.3 生成编译配置文件

4.3.1 安装编译所需工具

首先安装 ncurses-devel elfutils-libelf-devel openssl-devel 这三个工具;

```
# yum install ncurses-devel elfutils-libelf-devel openssl-devel -y
```

4.3.2 清理历史编译文件

如果不是第一次编译，请先执行命令清理（包括后面编译出错，重新编译也要先执行下面的这条命令）

```
# make mrproper
```

4.3.3 生成配置文件

执行命令生成.config 配置文件

```
# make menuconfig
```

当执行此命令时，命令行提示缺少相关工具，我们只需按提示安装工具即可，然后再次执行 make menuconfig 命令，会进入如下界面，保存即可。

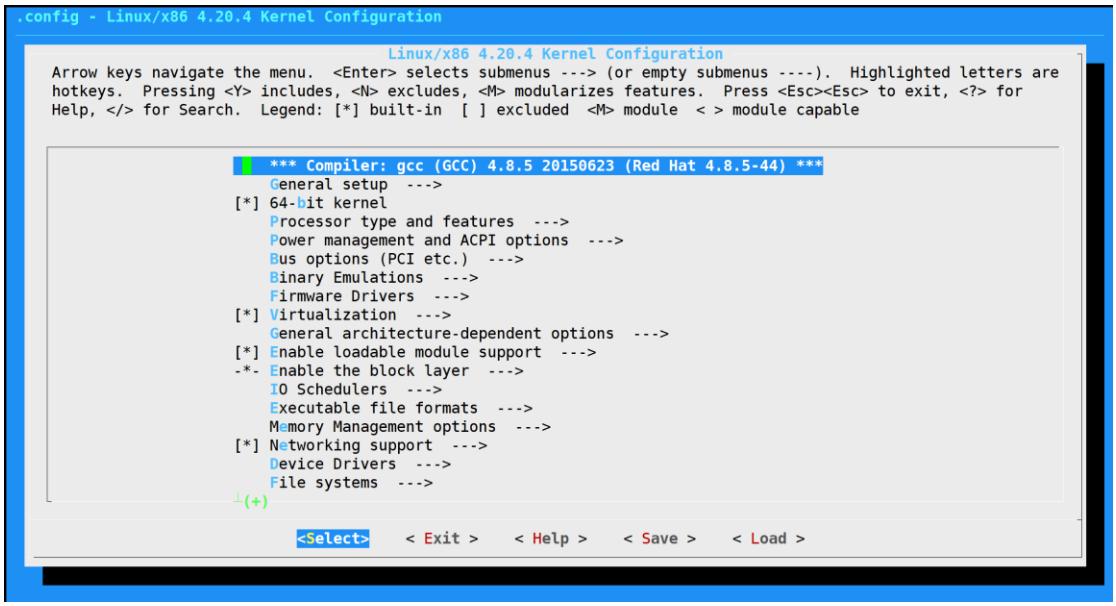

图 6 配置文件生成

4.4 内核编译与安装

4.4.1 内核编译

编译直接在主目录下使用 make 命令，此过程消耗时间较长，可以使用多线程编译，数字 8 为线程数，不能超过计算系统的核心数。

```
# make -j8
```

4.4.2 模块编译

```
# make modules
```

4.4.3 安装模块

由于自身编译源码，会有很多 debug 模块的存在，占用大量存储空间，我们需要在安装的时候排除它，添加参数 INSTALL_MOD_STRIP=1；

```
# make INSTALL_MOD_STRIP=1 modules_install
```

4.4.4 安装内核

```
# make INSTALL_MOD_STRIP=1 install
```

4.4.5 重启运行新内核

执行 reboot 命令（重启命令），并在重启时候键盘上下建选择我们自己安装的内核，回车进入；执行命令查看当前运行的内核版本确实是我们修改编译的内核：

```
[root@iZuf6e0pk109pg8fxhit1pZ linux-4.20.4]# uname -r
4.20.4
```

图 7 查看内核版本

4.5 查看缺页中断次数

在 home 目录新建一个 source 目录，并 cd 进去；

```
# mkdir source && cd source
```

4.5.1 编写缺页中断内核模块

（1）编写 pf.c 文件

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/string.h>
#include<linux/mm.h>
#include<linux/proc_fs.h>
#include<linux/uaccess.h>
#include<asm/uaccess.h>
#include<linux/slab.h>
#include<linux/sched.h>
#include<linux/seq_file.h>

extern unsigned long volatile pfcount;

//模块输出
static int my_proc_show(struct seq_file *m,void *v){
seq_printf(m,"The pfcount is %ld !\n",pfcount);
return 0;
}

//模块打开
static int my_proc_open(struct inode *inode,struct file *file){
return single_open(file,my_proc_show,NULL);
}
```

```
/*
*对于 Linux 字符设备驱动程序，file_operations 这个结构实际上是提供给虚拟文件系统
*（VFS）的文件接口，它的每一个成员函数一般都对应一个系统调用。用户进程利用系统
*调用对设备文件进行诸如读和写等操作时，系统调用通过设备文件的主设备号找到相应的
*设备驱动程序，并调用相应的驱动程序函数。
*/
struct file_operations fops={
.owner=THIS_MODULE,
.open=my_proc_open,
.release=single_release,
.read=seq_read,
.llseek=seq_lseek,
};

//模块初始化
static int pf_init(void){
struct proc_dir_entry *file;
struct proc_dir_entry *parent = proc_mkdir("pf",NULL);
file=proc_create("pfcount",0644,parent,&fops);
if(file==NULL)
printk("create_proc_entry failed.\n");
return 0;
}

// 模块退出
static void pf_exit(void){
remove_proc_entry("pf",NULL);
}

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

（2）编写 Makefile 文件

```
/*
*Makefile 的作用就是实现“自动编译”。当整个项目的 Makefile 都写好了以
*后，只需要 make 命令，就可以实现自动编译了。换句话说，就是 make 这个命
*令工具帮助我们实现了我们想要做的事，而 Makefile 就相当于是一个规则文
*件，make 程序会按照 Makefile 所指定的规则，去判断哪些文件需要先编译，
*哪些文件需要后编译，哪些文件需要重新编译，包括依赖的环境，文件输出位置等。
*/
ifneq ($(KERNELRELEASE),)
    obj-m:=pf.o
else
    KDIR:= /modules/$(shell uname -r)/build
```

```
    PWD:= $(shell pwd)
default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
endif
```

此时 source 目录包含两个文件：pf.c、Makefile

输入 make 编译

```
[root@iZuf6e0pk109pg8fxhit1pZ zlq]# make
make -C /usr/src/kernels/linux-4.20.4 M=/usr/src/ks/zl
q modules
make[1]: Entering directory `/usr/src/kernels/linux-4.
20.4'
  CC [M]  /usr/src/ks/zlq/pf.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /usr/src/ks/zlq/pf.mod.o
  LD [M]  /usr/src/ks/zlq/pf.ko
make[1]: Leaving directory `/usr/src/kernels/linux-4.2
0.4'
```

图 8 编译缺页中断模块

此时 source 包含八个文件，如图所示：Makefile、modules.order、Module。symvers、pf.c、pf.ko、pf.mod.c、pf.o

4.5.2 加载新模块到内核

```
# insmod pf.ko
```

4.5.3 用户态下查看缺页中断次数

/proc 文件系统是一种内核和内核模块用来向进程(process) 发送信息的机制(所以叫做/proc)。这个伪文件系统让你可以和内核内部数据结构进行交互，获取 有关进程的有用信息，在运行中(on the fly) 改变设置(通过改变内核参数)。 与其他文件系统不同，/proc 存在于内存之中而不是硬盘上。proc 文件系统以文件的形式向用户空间提供了访问接口，这些接口可以用于在运行时获取相关部件的信息或者修改部件的行为，因而它是非常方便的一个接口。

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

```
[root@iZuf6e0pk109pg8fxhit1pZ zlq]# cat /proc/pf/pfcount
The pfcount is 34795009 !
```

图 9 查看缺页中断次数

五、 主要数据结构和流程

5.1 缺页中断内核模块

（1）模块头文件

表 2 缺页中断模块所需头文件

#include<linux/init.h>	初始化
#include<linux/module.h>	最基本的文件，支持动态添加和卸载模块
#include<linux/kernel.h>	内核头文件，含有一些内核常用函数的原形定义
#include<linux/string.h>	字符串头文件，主要定义了一些有关字符串操作的嵌入函数
#include<linux/mm.h>	内存管理头文件，含有页面大小定义和一些页面释放函数原型
#include<linux/proc_fs.h>	包含了文件操作相关 struct 的定义
#include<linux/uaccess.h>	在 include/linux 下面寻找源文件
#include<asm/uaccess.h>	在 arch/arm/include/asm 下面寻找源文件
#include<linux/slab.h>	包含了 kcalloc、kzalloc 内存分配函数的定义
#include<linux/sched.h>	内核等待队列中要使用的 TASK_NORMAL、TASK_INTERRUPTIBLE 包含在这个头文件
#include<linux/seq_file.h>	在老版本的 Linux 内核中，proc 文件系统有一个缺陷：如果输出内容大于 1 个内存页（4096 Bytes），需要多次读，因此处理起来很难。另外，如果输出内容太大，速度会比较慢。在 2.6 内核中，由于大量使用了 seq_file 功能，使得内核输出大文件信息更容易

（1）文件操作

对于 Linux 字符设备驱动程序，file_operations 这个结构实际上是提供给虚拟文件系统（VFS）的文件接口，它的每一个成员函数一般都对应一个系统调用。用户进程利用系统调用对设备文件进行诸如读和写等操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，并调用相应的驱动程序函数。

```
struct file_operations fops={
.owner=THIS_MODULE,
.open=my_proc_open,
.release=single_release,
.read=seq_read,
.llseek=seq_lseek,
};
```

（2）模块主要功能函数

```
static int pf_init(void)//模块初始化
static int my_proc_open(struct inode *inode,struct file *file)//模块打开
static int my_proc_show(struct seq_file *m,void *v)//模块内容显示
static void pf_exit(void)//模块退出
```

5.2 设计思路流程

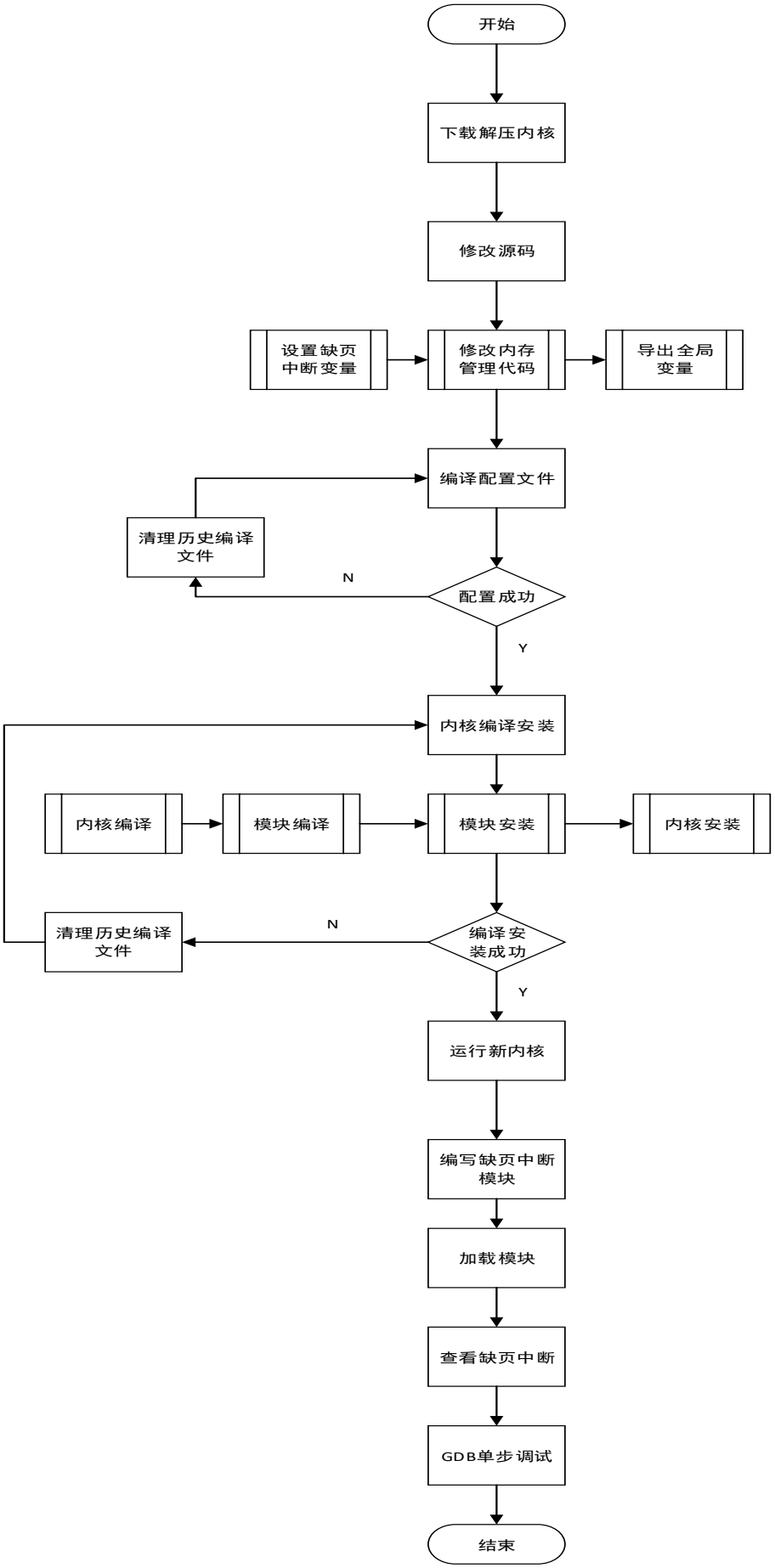
开始
下载解压内核
修改源码
设置缺页中断变量
修改内存管理代码
导出全局变量
编译配置文件
清理历史编译文件
N
配置成功
Y
内核编译安装
内核编译
模块编译
模块安装
内核安装
清理历史编译文件
N
编译安装成功
Y
运行新内核
编写缺页中断模块
加载模块
查看缺页中断
GDB单步调试
结束

图 10 设计思想流程

六、 实验测试结果及结果分析

6.1 测试结果

修改完内核，并编译安装运行新内核后，编写缺页中断内核模块，然后编译安装，即可在用户态下实现内核缺页中断次数统计，执行下面命令输出结果

```
[root@iZuf6e0pk109pg8fxhit1pZ ~]# cat /proc/pf/pfcount
The pfcount is 37638701 !
```

图 11 用户态查看缺页中断次数

6.2 结果分析

（1）可通过以下命令查看缺页中断信息

```
# ps -o majflt,minflt -C <program_name>
# ps -o majflt,minflt -p <pid>
```

其中，majflt 代表 major fault ，指大错误，minflt 代表 minor fault ，指小错误。这两个数值表示一个进程自启动以来所发生的缺页中断的次数。

其中 majflt 与 minflt 的不同是， majflt 表示需要读写磁盘，可能是内存对应页面在磁盘中需要 load 到物理内存中，也可能是此时物理内存不足，需要淘汰部分物理页面至磁盘中。

（2）调试分析

新建三个 terminal，分别用于 gdb 单步调试程序、minflt,maxflt 输出当前进程缺页中断次数、模块输出缺页中断次数；我们可以观察模块输出的缺页中断次数变化率是在调试程序的时候突然增大，因此结合设计思想中源码解读的理论正确与调试分析验证的准确性来判断缺页中断次数统计是否符合预期要求。

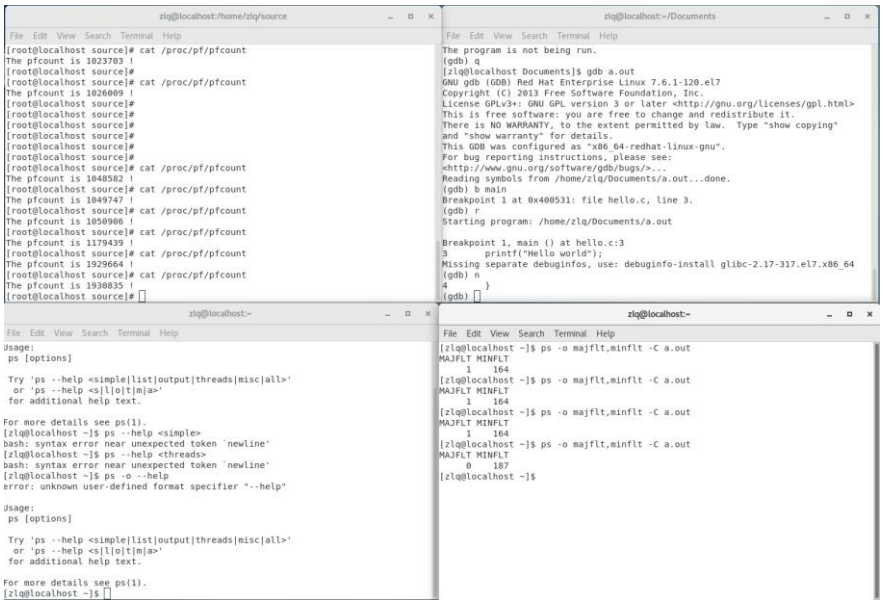

图 12 gdb 单步调试验证

七、 课程设计总结

操作系统课程主要学习掌握进程与线程、互斥与同步、处理机调度、内存管理、文件管理、I/O 管理、操作系统安全等知识，任何操作系统的架构设计都必须考虑到这些核心内容，平时的上机实验也基本覆盖了核心内容，因此通过前期理论学习和基础实验的设计为此次课程设计打好相关基础，明确设计需求与设计思路。

本次使用的是 Linux，全称 GNU/Linux，是一种免费使用和自由传播的类 UNIX 操作系统，它主要受到 Minix 和 Unix 思想的启发，是一个基于 POSIX 的多用户、多任务、支持多线程和多 CPU 的操作系统。它能运行主要的 Unix 工具软件、应用程序和网络协议。对于此次缺页中断次数统计题目，我首先明确题目要求，再通过查阅资料、图书、Google 学术、内核原有注释、Linux 内置命令帮助文档等方式掌握 Linux 系统内核结构与组织方式，明确各部分完成的操作系统功能。从而进一步定位到缺页中断的处理文件 fault.c，接着详细阅读该文件执行函数，并在内核缺页中断处理函数作出缺页中断统计。修改完内核源码后编译安装并运行新内核，接着编写缺页中断内核模块，这部分又了解掌握相关头文件的内置函数与功能，同时学习了 make 命令需要的 Makefile 文件编写规则、语法等知识，最终将模块加载到内核中，在用户态统计缺页中断次数。在调试验证环节，我通过编写程序进行 gdb 单步调试来创造缺页中断，并通过 minflt,maxflt 查看当前进程缺页中断次数，与自己得出的缺页中断次数相比较来分析验证。由于在分析题目要求时未考虑到调试环节，导致最终的调试部分做的不够完美，也是我本次课程设计最大的遗憾与改进方向。

总的来说，本次课程设计主题内容基本完成，实现效果良好。时间虽短，但在解决问题的过程中学习到很多知识，包括 Linux 系统内核组织方式、terminal 命令学习、源码解读、/proc 虚拟文件系统、Makefile 语法规则与编写、分析 Linux 头文件功能并进行调用编写模块、内核的修改多线程编译安装运行、gdb 工具的调试使用等非常重要的知识与技能，同时积累下很多宝贵经验。由于本次课程设计是小组合作展开，我们使用了阿里云计算系统来实现版本统一与多人协同工作，通过 Gitee 码云实现小组每人的文件上传便于版本控制与回滚，避免文件重名、丢失、忘记修改等问题，这些方式很大程度提升了小组合作和最终联调编译的速度与效率。

最后，感谢薛教授从理论教学到课程设计的一学期陪伴与解答，感谢小组伙伴的通力协作与支持，感谢自己解决问题过程中的坚持与资料的解惑，才有了最终良好的课程设计完成效果。

八、 附件

附件 1 课程设计答辩记录

8.1.1 答辩内容

1.简要概述题目要求与自己完成的实现效果
2.整个设计思路描述，演示代码修改、操作步骤，给出修改原因、关键步骤的作用，并给出通过 gdb 单步调试分析缺页中断次数是否正确的思路。
3.提问阶段
（1）什么是/proc?
/proc 文件系统是一种内核和内核模块用来向进程(process) 发送信息的机制(所以叫做/proc)。这个伪文件系统让你可以和内核内部数据结构进行交互，获取 有关进程的有用信息，在运行中(on the fly) 改变设置(通过改变内核参数)。与其他文件系统不同，/proc 存在于内存之中而不是硬盘上。proc 文件系统以文件的形式向用户空间提供了访问接口，这些接口可以用于在运行时获取相关部件的信息或者修改部件的行为，因而它是非常方便的一个接口。
(2)在整个题目解决过程中有什么完善的地方？

在阅读缺页中断源码、查阅资料、研究命令功能的帮助下，顺利的完成了缺页中断次数的统计输出，但是忽略了结果调试步骤的必要性，在第一天参考其他组检查的情况，对调试进行了研究分析。通过查阅资料，编写程序使用 gdb 设置断点，进行单步调试主动造成缺页中断，执行进程缺页中断次数统计，观察与未执行情况下缺页中断次数变化率是否突然增大。

但这是一个感性认知，我们应该将程序执行前后的缺页中断次数相减，并不断输出缺页中断次数，判断相减后的值是否落在进程缺页中断范围内，来得到更加理性的验证。

8.1.2 答辩流程

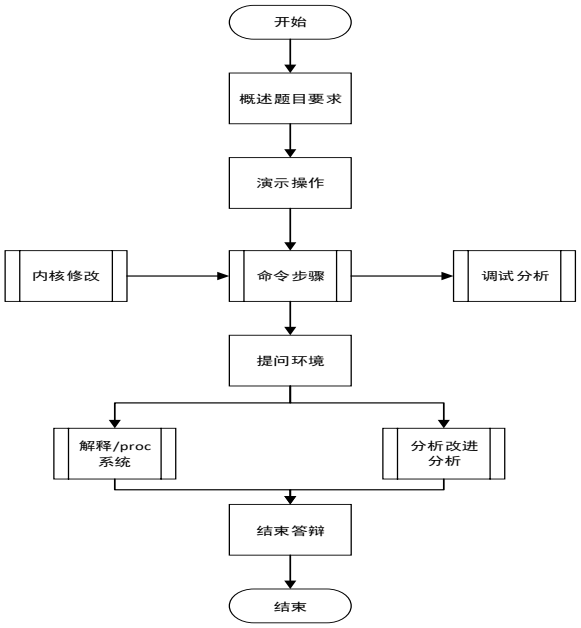

图 13 答辩流程

附件 2 源程序清单

注意：关于内核源码的修改在“四、设计思想”中进行源码解读并作出修改解释，内核模块文件（pf.c Makefile）的具体作用和函数构成已在“五、主要数据结构和流程”中详细描述，这里不再作以进一步注释、详解。

(1)pf.c 文件

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/string.h>
#include<linux/mm.h>
#include<linux/proc_fs.h>
#include<linux/uaccess.h>
#include<asm/uaccess.h>
#include<linux/slab.h>
#include<linux/sched.h>
#include<linux/seq_file.h>

extern unsigned long volatile pfcount;

static int my_proc_show(struct seq_file *m,void *v){
seq_printf(m,"The pfcount is %ld !\n",pfcount);
return 0;
}

static int my_proc_open(struct inode *inode,struct file *file){
return single_open(file,my_proc_show,NULL);
}

struct file_operations fops={
.owner=THIS_MODULE,
.open=my_proc_open,
.release=single_release,
.read=seq_read,
.llseek=seq_lseek,
};

static int pf_init(void){
struct proc_dir_entry *file;
struct proc_dir_entry *parent = proc_mkdir("pf",NULL);
file=proc_create("pfcount",0644,parent,&fops);
if(file==NULL)
printk("create_proc_entry failed.\n");
return 0;
```

```
}

static void pf_exit(void){
remove_proc_entry("pf",NULL);
}

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

（2）Makefile 文件

```
ifneq ($(KERNELRELEASE),)
	obj-m:=pf.o
else
	KDIR:= /modules/$(shell uname -r)/build
	PWD:= $(shell pwd)
default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
endif
```