苏 州 大 学 实 验 报 告

院、系	计算机学院	年级专业	20 计科	姓名	柯骅	学号	2027405033
课程名称	操作系统课程实践					成绩	
指导教师	李培峰	同组实验者	无		实验日期	2023.04.27	

实 验 名 称 实验 7 虚拟内存管理

一、 实验目的

1. 理解操作系统中缺页中断的工作原理。
2. 学会通过修改内核实现统计系统缺页次数的方法。
3. 进一步理解虚拟内存管理的原理。
4. 学会观察 /proc 中有关虚拟内存的内容。
5. 学会使用相关工具统计一段时间内的缺页次数。

二、 实验内容

1. （统计系统缺页次数）

通过修改 Linux 内核中的相关代码，统计系统缺页次数。

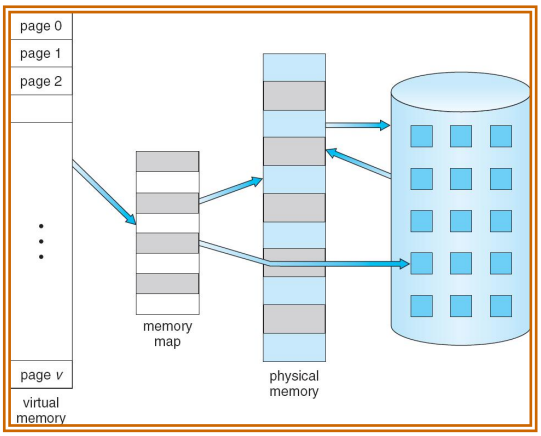
2. （统计一段时间内的缺页次数）

通过查看 /proc/vmstat 的变化来统计一段时间内的缺页次数。

三、 实验步骤

虚拟内存技术：

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。



当进程运行时，先将其一部分装入内存，另一部分暂留在磁盘，当要执行的指令或访问的数据不在内存时，由操作系统自动完成将它们从磁盘调入内存执行。

系统缺页：

缺页是引入了虚拟内存后的一个概念。操作系统启动后，在内存中维护着一个虚拟地址表，进程需要的虚拟地址在虚拟地址表中记录。一个程序被加载运行时，只是加载了很少的一部分到内存，另外一部分在需要时再从磁盘载入。被加载到内存的部分标识为“驻留”，而未被加载到内存的部分标为“未驻留”。操作系统根据需要读取虚拟地址表，如果读到虚拟地址表中记录的地址被标为“未驻留”，表示这部分地址记录的程序代码未被加载到内存，需要从磁盘读入，则这种情况就表示"缺页"。这个时候，操作系统触发一个“缺页”的硬件陷阱，系统从磁盘换入这部分未“驻留”的代码。

引入了分页机制(也就有了缺页机制)，则系统只需要加载程序的部分代码到内存，就可以创建进程运行， 需要程序的另一部分时再从磁盘载入并运行，从而允许比内存大很多的程序同时在内存运行。

教务处制

1. （统计系统缺页次数）

本实验采用内核源代码的方式来统计系统缺页次数，因此涉及相关内核源代码的修改、内核的重新编译、缺页次数的输出等内容。具体步骤如下

（1）在内核源代码中找到 include/linux/mm.h 文件，使用如下命令声明变量 pfcount 用于统计缺页次数，如下图所示：

```
# cd usr/src/linux-4.16.10/include/linux
# vim mm.h
# /extern int page_cluster + 回车 //查找声明变量的位置，随后使用 i 进入编辑模式添加下行代码
# extern unsigned long volatile pfcount;
```

```
extern unsigned long totalram_pages;
extern void * high_memory;
extern int page_cluster;
extern unsigned long volatile pfcount;
```

（2）同样，在/arch/x86/mm/fault.c 文件中定义变量 pfcount，代码与结果如下：

```
# cd usr/src/linux-4.16.10/arch/x86/mm
# vim fault.c
# /static nokprobe_inline + 回车  //查找声明变量的位置，随后使用 i 进入编辑模式添加下行代码
# unsigned long volatile pfcount;
```

```
unsigned long volatile pfcount;
static nokprobe_inline int
```

在 do_page_fault()函数中找到并修改 good_area，以使变量 pfcount 递增 1，，代码与结果如下：

```
# /good_area + 回车 +n  //查找 good_area 的位置，随后使用 i 进入编辑模式添加下行代码
# pfcount++;
```

```
good_area:
        if (unlikely(access_error(error_code, vma))) {
                bad_area_access_error(regs, error_code, address, vma);
                return;
        }
        pfcount++;
```

（3）修改 kernel/kallsymc.c 文件，即在这个文件的最后插入“EXPORT_SYMBOL(pfcount);”。该步骤的作用是使得 EXPORT_SYMBOL 标签内定义的函数或变量对全部内核代码公开。既可以使用文本编辑器修改这个文件，也可以使用 echo 命令来完成修改，命令如下：

```
# gedit kallsymc.c //使用文本编辑器进行修改
# echo ''EXPORT_SYMBOL(pfcount);'>>kernel/kallsyms.c //echo 命令添加内容
```

结果如下：

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

```
static int __init kallsyms_init(void)
{
        proc_create("kallsyms", 0444, NULL, &kallsyms_operations);
        return 0;
}
device_initcall(kallsyms_init);
EXPORT_SYMBOL(pfcount);
                        C ▾  制表符宽度：8 ▾   第 688 行，第 24 列 ▾   插入
```

教务处制

（4）重新配置、编译内核并安装

清除旧目标文件并重新配置内核，依次执行以下语句（与实验 5 类似）：

```
# cd /usr/src/linux-4.16.10
# make mrproper
# make clean
# make menuconfig
```

选择<Save>保存为.config 后选择<Exit>退出 menuconfig 界面，然后输入如下指令：

```
# sudo gedit .config
```

使用 Ctrl+F,搜索 CONFIG_SYSTEM_TRUSTED_KEYS 和 CONFIG_SYSTEM_REVOCATION_KEYS，删除这两条引号中的内容(而不包括引号)，随后保存，以防编译失败，随后依次输入以下几条指令对内核与模块进行编译与安装：

```
# sudo make -j16
# sudo make modules_install
# sudo make install
```

编译安装成功如下图：

```
LD [M]  sound/usb/line6/snd-usb-variax.ko
LD [M]  sound/usb/snd-usb-audio.ko
LD [M]  sound/usb/misc/snd-ua101.ko
LD [M]  sound/usb/snd-usbmidi-lib.ko
LD [M]  sound/usb/usx2y/snd-usb-us122l.ko
LD [M]  sound/usb/usx2y/snd-usb-usx2y.ko
LD [M]  sound/x86/snd-hdmi-lpe-audio.ko
```

```
INSTALL sound/usb/misc/snd-ua101.ko
INSTALL sound/usb/snd-usb-audio.ko
INSTALL sound/usb/snd-usbmidi-lib.ko
INSTALL sound/usb/usx2y/snd-usb-us122l.ko
INSTALL sound/usb/usx2y/snd-usb-usx2y.ko
INSTALL sound/x86/snd-hdmi-lpe-audio.ko
DEPMOD  4.16.10myKernel
```

```
Found linux image: /boot/vmlinuz-4.16.10
Found initrd image: /boot/initrd.img-4.16.10
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

（5）编写测试程序 readpfcount.c。测试程序的功能是以内核模块的形式读取 pfcount 的值并输出。完整代码详见附件/pfcount/readpfcount.c。

同时编写相应的 Makefile 文件，完整代码详见附件/pfcount/Makefile。

（6）编译测试程序并加载内核。命令如下：

```
# sudo make
# sudo insmod readpfcount.ko
```

```
root@ubuntu:/home/kh/test/test7/pfcount# make
make -C /usr/src/linux-4.16.10 M=/home/kh/test/test7/pfcount modules
make[1]: 进入目录“/usr/src/linux-4.16.10”
Makefile:976: "Cannot use CONFIG_STACK_VALIDATION=y, please install libelf-dev, libelf-devel or elfutils-libelf-devel"
  CC [M]  /home/kh/test/test7/pfcount/readpfcount.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/kh/test/test7/pfcount/readpfcount.mod.o
  LD [M]  /home/kh/test/test7/pfcount/readpfcount.ko
make[1]: 离开目录“/usr/src/linux-4.16.10”
root@ubuntu:/home/kh/test/test7/pfcount# insmod readpfcount.ko
```

2. （统计一段时间内的缺页次数）

/proc/vmstat 文件：

这个文件中包含了从内核导出的虚拟内存的相关信息，例如：

pgdeactivate 放到inactive链表上的页数
pglazyfree Amount of pages postponed to
pgfault 缺页数量
pgmajfault 需通过读磁盘解决的缺页数量

nr_free_pages 空闲页数量
nr_zone_inactive_anon 所有zone统计的非活跃的匿名页数之和
nr_zone_active_anon 所有zone统计的活跃的匿名页数之和
nr_zone_inactive_file 所有zone统计的非活跃的文件页数之和

其中，pgfault 代表缺页数量，可以间隔一定的时间分别读取 pgfault 的数字来统计这段时间里的缺页次数。

教务处制

程序的大致步骤为：读取缺页次数--计时--再次读取缺页次数。

（1） 读取缺页次数

编写 get_page_fault()函数读取缺页次数并将相关提示打印在屏幕上。首先先打开指定文件 proc/vmstat,如打开失败则输出提示；如打开成功则利用 strcmp()函数，逐字符查找 pgfault 字段,将其中统计的缺页次数读取出来,转成整型后保存在全局变量 page_fault 中,并将 page_fault 打印在屏幕提示中，关键代码如下：

```
do{ ...
}while(strcmp("pgfault",string));
printf("find intr!\n");
printf("Now the number of page fault is %s\n",d);
return atoi(d);
```

其中，atio()(表示 ascii to integer)是把字符串转换成整型数的一个函数

（2） 计时函数

setitimer 函数为设置定时器（闹钟），可替代 alarm 函数，比 alarm 函数精确度更高，精度为微秒，可以实现周期定时。利用 setitimer 函数，当到达设定的时间间隔之后，将全局变量 exit_flag 设置为 1，主进程便可以继续执行下一步，再次利用 get_page_fault()函数读取缺页次数，关键代码如下：

```
setitimer(ITIMER_REAL,&v,NULL);
while(!exit_flag) ;
```

完整代码详见附件/pfintr/pfintr.c

四、 实验结果

1. （实验 9.1：统计系统缺页次数）

内核加载成功后，输入以下命令：

```
# cat /proc/readpfcount
```

运行结果如下：

```
root@ubuntu:/home/kh/test/test7/pfcount# cat /proc/readpfcount
The pfcount is 2079082 and jiffies is 4295914339!
```

发现成功输出缺页次数，实验成功。

2. （实验 9.2：统计一段时间内的缺页次数）

使用 gcc 编译 pfintr/pfintr.c 文件并运行，结果如下：

```
kh@ubuntu:~/test/test7/pfintr$ ./pfintr
find intr!
Now the number of page fault is 6450713
Use default time!
find intr!
Now the number of page fault is 6451966
In 5 seconds,system calls 1253 page fault!
```

发现成功读取缺页次数并统计出了 5 秒内的缺页次数为 1253，实验成功。

值得注意的是：如需获得较多的缺页次数，可以在执行 test 文件的同时，在另一终端执行一个较大的任务。

五、 实验思考与总结

1. （实验 9.1：统计系统缺页次数）

（1）说明本实验中统计缺页次数的原理，并阐述其合理性。

本实验是通过修改内核源代码来实现的，基本原理是增加一个长整型变量 pfcount（初值为 0），用来统计缺页次数，在每次缺页时，对该变量的值增加 1，输出该变量的值，即为缺页次数。这是从内核层面统计缺页次数，结果是合理的。

2. （实验 9.2：统计一段时间内的缺页次数）

（1）如何验证实验结果的准确性？

验证该实验的结果可以借助于实验 9.1，即可通过编写程序的方式，在某个固定时间输出系统缺页次数和/proc/vmstat 中的 pgfault 字段的值，然后再过一段时间再输出系统缺页次数和/proc/vmstat 中的 pgfault 字段的值，取两次的差值，相互验证实验结果是否一致。

（2）尝试使用更方便的方法读取/proc/vmstat 中的字段，如使用Python 编程或Shell 编程。

python 程序详见附件 pfintr/pfintr.py，运行结果如下：

```
kh@ubuntu:~/test/test7/pfintr$ python pfintr.py
Use time: 5s
Now the number of page fault is 9147195
Now the number of page fault is 9153736
In 5 seconds,system calls 6541 page fault!
```

sh 脚本详见附件 pfintr/pfintr.sh，运行结果如下：

```
kh@ubuntu:~/test/test7/pfintr$ bash pfintr.sh
Use time: 5s
Now the number of page fault is 9321588
Now the number of page fault is 9322970
In 5 seconds,system calls 1382 page fault!
```

可以发现，均成功输出了 5 秒内的缺页次数。

3. 实验总结

本次实验使用了两种方法来统计 Linux 系统的缺页次数。在第一个实验中，我们在编译 Linux 系统源代码之前添加了一个长整型变量 pfcount 来统计缺页次数，并进入编译安装好的系统中成功进行了验证。在这过程中，学习了 echo 命令的使用，学会了通过修改内核统计缺页次数的方法，进一步加深了对 Linux 系统、操作系统中缺页中断的理解。

在第二个实验中，我们用 C 语言、Python、Shell 脚本三种方式实现了：读取 proc/vmstat 中表示缺页次数的字段，并以此统计出一段时间内系统的缺页次数。在此过程中，学会了如何观察 /proc 中有关虚拟内存的内容，同时也掌握了查询缺页次数相关工具的使用，对 Linux 系统的虚拟内存管理有了更深的理解。