苏州大学实验报告

院、系	计算机学院	年级专业	19 计科图灵	姓名	张昊	学号	1927405160
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	2022 年 5 月 19 日	

实 验 名 称　　实验 8　Linux 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. （**实验 9.1: 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. （**实验 9.2: 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. 操作方法和实验步骤

1. 统计系统缺页次数

（1）在内核源代码 include/linux/mm.h 文件中声明变量 pfcount 用于统计缺页次数：

```
extern unsigned long volatile pfcount;
```

```
extern int page_cluster;
extern unsigned long volatile pfcount;

#ifdef CONFkIG_SYSCTL
"include/linux/mm.h" 2680L, 85818C written
```

（2）在 arch/x86/mm/fault.c 文件中定义变量 pfcount：

```
unsigned long volatile pfcount;
```

```
#define CREATE_TRACE_POINTS
#include <asm/trace/exceptions.h>

unsigned long volatile pfcount;

/*
 * Returns 0 if mmiotrace is disabled, or if the fault is not
 * handled by mmiotrace:
 */
static nokprobe_inline int
kmmio_fault(struct pt_regs *regs, unsigned long addr)
-- INSERT --
```

并在 do_page_fault()函数中找到并修改 good_area，以使变量 pfcount 递增 1：

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

```
        /*
         * Ok, we have a good vm_area for this memory access, so
         * we can handle it..
         */
good_area:
        if (unlikely(access_error(error_code, vma))) {
                bad_area_access_error(regs, error_code, address, vma);
                return;
        }
        pfcount++;
"arch/x86/mm/fault.c" 1503L, 38682C written
```

（3）在 kernel/kallsyms.c 文件最后插入：

```
EXPORT_SYMBOL(pfcount);
```

```
root@hao-zhang:/usr/src/linux-4.16.10# echo 'EXPORT_SYMBOL(pfcount);' >> kernel/kallsyms.c
root@hao-zhang:/usr/src/linux-4.16.10# tail kernel/kallsyms.c
        .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);
root@hao-zhang:/usr/src/linux-4.16.10#
```

（4）重新编译内核。

```
# cd /usr/src/linux-4.16.10
# make mrproper
# make clean
# make menuconfig    # Local version 修改为：-HaoZhang-5160-220520
# make -j8
# make modules -j8
# make modules_install
# make install
# reboot
```

```
  ()  Cross-compiler tool prefix
  [ ] Compile also drivers which will not load
  (-HaoZhang-5160-220520) Local version - append to kernel release
  [ ] Automatically append version information to the version string
      Kernel compression mode (Gzip)  --->
```

```
holger@hao-zhang:~$ uname -r
4.16.10-HaoZhang-5160-220520
holger@hao-zhang:~$
```

（5）编写内核模块进行测试：

```
/* readpfcount.c */
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/proc_fs.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/seq_file.h>
#include <linux/slab.h>

//extern unsigned long pfcount;

static int my_proc_show(struct seq_file* m, void* v){
   seq_printf(m, "The pfcount is %ld and jiffies is %ld!\n", pfcount,
jiffies);
   return 0;
}
static int my_proc_open(struct inode* inode, struct file* file){
   return single_open(file, my_proc_show, NULL);
}
static struct file_operations my_fops = {
   .owner = THIS_MODULE,
   .open = my_proc_open,
   .release = single_release,
   .read = seq_read,
   .llseek = seq_lseek,
};
static int __init my_init(void) {
   struct proc_dir_entry* file = proc_create("readpfcount", 0x0644, NULL,
&my_fops);
   if (!file) {
      printk("proc_create failed.\n");
      return -ENOMEM;
   }
   return 0;
}
static void __exit my_exit(void) {
   remove_proc_entry("readpfcount", NULL);
}
MODULE_LICENSE("GPL");
module_init(my_init);
module_exit(my_exit);
```

（6）编写相应的 Makefile 文件：

```
obj-m := readpfcount.o
KDIR := /usr/src/linux-$(shell uname -r | cut -d '-' -f1)
PWD := $(shell pwd)
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

编译并加载内核：

```
$ sudo make
$ sudo insmod readpfcount.ko
```

```
holger@hao-zhang:/codes/exp08/total$ sudo make
make -C /usr/src/linux-4.16.10 M=/codes/exp08/total modules
make[1]: Entering directory '/usr/src/linux-4.16.10'
  CC [M]  /codes/exp08/total/readpfcount.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /codes/exp08/total/readpfcount.mod.o
  LD [M]  /codes/exp08/total/readpfcount.ko
make[1]: Leaving directory '/usr/src/linux-4.16.10'
holger@hao-zhang:/codes/exp08/total$ sudo insmod readpfcount.ko
```

2. 统计一段时间内的缺页次数

/proc/vmstat 文件是一个用来查看虚拟内存使用状况的工具，可以通过读取其中 pgfault 字段的变化来统计一段时间内的缺页次数。

编写 C 程序 pfintr.c，通过读取/proc/vmstat 文件相关内容来统计一段时间内的缺页次数：

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>
#define FILENAME "/proc/vmstat"
#define DEFAULT_TIME 5

long get_page_fault(void);

bool isnumber(const char *str) {
  for (int i = 0; i < strlen(str); ++i) {
    if (!isdigit(str[i])) {
      return false;
    }
  }
  return true;
}

int main(int argc, const char **argv) {
```

```
  int calc_time = DEFAULT_TIME;
  long page_fault;  //休眠时间
  if (argc >= 2 && isnumber(argv[1])) {
    calc_time = atoi(argv[1]);
  }
  printf("Use time: %ds\n", calc_time);
  page_fault = get_page_fault();
  sleep(calc_time);  //使用进程休眠的方法
  page_fault = get_page_fault() - page_fault;
  printf("In %d seconds,system calls %ld page fault!\n",
        calc_time, page_fault);
  return 0;
}

/* 该函数对 /proc/stat 文件内容进行读操作，读取指定项的值到 data */
ssize_t readfile(char *data) {
  int i, offset = 0, count = 0;
  char c, string[50];
  int fd = open(FILENAME, O_RDONLY);
  ssize_t retval;
  if (fd < 0) {
    printf("Open file /proc/stat failed!\n");
    return -1;
  }
  /* 查找 vmstat 文件中的关键字 pgfault */
  do {
    i = 0;
    do {
      lseek(fd, offset, SEEK_SET);
      retval = read(fd, &c, sizeof(char));
      if (retval < 0) {
        printf("read file error!\n");
        return retval;
      }
      offset += sizeof(char);
      if (c == ' ' || c == '\n') {
        string[i] = 0;
        break;
      }
      if ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z')
          || (c >= 'A' && c <= 'Z'))
        string[i++] = c;
    } while (1);
  } while (strcmp("pgfault", string));
  /* 读取缺页次数 */
```

```
  i = 0;
  do {
    lseek(fd, offset, SEEK_SET);
    retval = read(fd, &c, sizeof(char));
    if (retval < 0) {
      printf("read file error!\n");
      return retval;
    }
    offset += sizeof(char);
    if (c == ' ' || c == '\n') {
      string[i] = 0;
      i = 0;
      count++;
    }
    if ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z')
        || (c >= 'A' && c <= 'Z'))
      string[i++] = c;
  } while (count != 1);
  close(fd);
  strcpy(data, string);
  return 0;
}

/* 该函数通过调用文件操作函数 readfile, 得到当前系统的缺页中断次数 */
long get_page_fault(void) {
  char pgfault[50], *buff;
  /* 读取缺页中断次数 */
  if (readfile(pgfault) < 0) {
    printf("read data from file failed!\n");
    exit(0);
  }
  printf("Now the number of page fault is %s\n", pgfault);
  return strtol(pgfault, &buff, 10);
}
```

对其进行编译，得到可执行文件 pfintr。

```
$ gcc pfintr.c -o pfintr
```

编写 Python 脚本 pfintr.py 来更方便地完成上述工作：

```
#!/bin/python3
import re
import sys
import time

def get_page_fault():
    """ 得到当前系统的缺页中断次数 """
    pgfault = -1
```

```
    with open('/proc/vmstat', 'r') as vmstat:
        # 通过正则表达式提取 pgfault 的值
        match = re.search(r'pgfault\s(\d+)', vmstat.read())
        if match:
            pgfault = int(match.group(1))
    return pgfault

calc_time = 5  # 休眠时间
if len(sys.argv) >= 2 and sys.argv[1].isdigit():
    calc_time = int(sys.argv[1])
print("Use time: {}s".format(calc_time))
pgfault1 = get_page_fault()
if pgfault1 < 0:
    print('read file error!')
    exit(1)
print("Now the number of page fault is {}".format(pgfault1))
time.sleep(calc_time)  # 进程休眠
pgfault2 = get_page_fault()
if pgfault2 < 0:
    print('read file error!')
    exit(1)
print("Now the number of page fault is {}".format(pgfault2))
page_fault = pgfault2 - pgfault1
print("In {} seconds, system calls {} page fault!"
      .format(calc_time, page_fault))
```

同样可以编写 Shell 脚本 pfintr.sh 来更方便地完成上述工作：

```
#!/bin/bash

calc_time=5  # 休眠时间
if [ "$1" ]; then
  if grep '^[[:digit:]]*$' <<<"$1" &>/dev/null; then
    calc_time=$1
  fi
fi

echo "Use time: ${calc_time}s"
pgfault1=$(cat /proc/vmstat | grep pgfault | cut -d ' ' -f2)
echo "Now the number of page fault is $pgfault1"
sleep "$calc_time"
pgfault2=$(cat /proc/vmstat | grep pgfault | cut -d ' ' -f2)
echo "Now the number of page fault is $pgfault2"
((page_fault=$pgfault2-$pgfault1))
echo "In $calc_time seconds, system calls $page_fault page fault!"
```

四. 实验结果和分析

1. 统计系统缺页次数

首先查看内核模块是否加载成功：

```
holger@hao-zhang:/codes/exp08/total$ lsmod | grep readpfcount
readpfcount            16384  0
holger@hao-zhang:/codes/exp08/total$
```

内核加载成功后测试实验结果：

```
cat /proc/readpfcount
```

```
holger@hao-zhang:/codes/exp08/total$ cat /proc/readpfcount
The pfcount is 925091 and jiffies is 4295115997!
holger@hao-zhang:/codes/exp08/total$ cat /proc/readpfcount
The pfcount is 925447 and jiffies is 4295159621!
holger@hao-zhang:/codes/exp08/total$
```

最后卸载内核模块：

```
holger@hao-zhang:/codes/exp08/total$ sudo rmmod readpfcount
holger@hao-zhang:/codes/exp08/total$ lsmod | grep readpfcount
holger@hao-zhang:/codes/exp08/total$
```

2. 统计一段时间内的缺页次数

为了获得较多的缺页中断，在执行 pfintr 文件的同时，在另一终端执行一个较大的任务：

```
/* large-job.c */
#include <time.h>
#include <stdlib.h>
#define N 10000
int **matrix;
int main(void) {
  srand(time(NULL));
  for (int i = 0; i < N; ++i) {
    // 申请内存空间
    matrix = (int **) malloc(sizeof(int *) * N);
    for (int j = 0; j < N; ++j) {
      matrix[j] = (int *) malloc(sizeof(int) * N);
    }
    // 尽量避免顺序访问数组，从而造成缺页
    for (int j = 0; j < N; ++j) {
      for (int k = N-1; k >= 0; --k) {
        matrix[k][j] = (i + j + k) * rand();  //随机产生数字写入
      }
    }
    for (int j = 0; j < N; ++j) {
      free(matrix[j]);
    }
    free(matrix);
  }
  return 0;
}
```

该文件不断申请大量内存空间，尽量避免顺序访问数组，从而造成缺页。

将该文件编译，并在测试时运行：

```
holger@hao-zhang:/codes/exp08/period$ gcc large-job.c -o large-job
holger@hao-zhang:/codes/exp08/period$ ./large-job &
[1] 6929
```

执行 pfintr 文件后的结果为：

```
holger@hao-zhang:/codes/exp08/period$ ./pfintr
Use time: 5s
Now the number of page fault is 120681668
Now the number of page fault is 120886721
In 5 seconds,system calls 205053 page fault!
```

执行 pfintr.py 脚本后的结果为：

```
holger@hao-zhang:/codes/exp08/period$ python3 pfintr.py
Use time: 5s
Now the number of page fault is 122265475
Now the number of page fault is 122474009
In 5 seconds, system calls 208534 page fault!
```

执行 pfintr.sh 脚本后的结果为：

```
holger@hao-zhang:/codes/exp08/period$ bash pfintr.sh
Use time: 5s
Now the number of page fault is 121272504
Now the number of page fault is 121476596
In 5 seconds, system calls 204092 page fault!
```

五. 讨论、心得

1. 通过本次实验，我理解了操作系统中缺页中断的工作原理，并尝试修改内核实现了统计系统缺页次数的方法，并利用了多中工具统计一段时间内的缺页次数。
2. 统计系统缺页次数实验中统计缺页次数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。
3. 验证统计一段时间内的缺页次数实验的结果可以借助于上一个实验中的 readpfcount，在某个固定时间段前后两次输出 pfcount 和/proc/vmstat 中的 pgfault 字段的值，并取差值，相互验证实验结果。
4. 验证统计一段时间内的缺页次数实验中，在系统较为空闲的情况下，若不运行较大的任务会是如下结果。

   C 语言版本：

```
holger@hao-zhang:/codes/exp08/period$ ./pfintr
Use time: 5s
Now the number of page fault is 159249874
Now the number of page fault is 159249877
In 5 seconds,system calls 3 page fault!
```

Python 脚本：

```
holger@hao-zhang:/codes/exp08/period$ python3 pfintr.py
Use time: 5s
Now the number of page fault is 159250948
Now the number of page fault is 159250957
In 5 seconds, system calls 9 page fault!
```

Shell 脚本：

```
holger@hao-zhang:/codes/exp08/period$ bash pfintr.sh
Use time: 5s
Now the number of page fault is 159251552
Now the number of page fault is 159252163
In 5 seconds, system calls 611 page fault!
```

可见，Shell 脚本的缺页次数要比其他编程语言实现的版本高出几百次。多次运行后排除了偶然现象后结果仍为此。根据 Shell 编程的特点，结合 bash 源代码初步分析原因在于 Shell 编程中读文件、查找对应行等操作是依赖于其他命令实现的，这些命令在执行时 bash 会通过 fork 创建子进程，从而导致程序的局部性不是很好；此外我在 Shell 脚本中还用到了管道来重定向命令的输出，这也是依赖于进程间管道通信实现的，进一步影响了内存页面的命中率。可做如下修改来避免使用管道重定向：

```
#!/bin/bash

calc_time=5  # 休眠时间
if [ "$1" ]; then
  if grep '^[[:digit:]]*$' <<<"$1" &>/dev/null; then
    calc_time=$1
  fi
fi

echo "Use time: ${calc_time}s"
pgfault1=$(awk '$1=="pgfault" {print $2}' /proc/vmstat)
echo "Now the number of page fault is $pgfault1"
sleep "$calc_time"
pgfault2=$(awk '$1=="pgfault" {print $2}' /proc/vmstat)
echo "Now the number of page fault is $pgfault2"
((page_fault=$pgfault2-$pgfault1))
echo "In $calc_time seconds, system calls $page_fault page fault!"
```

修改后的 Shell 脚本在不运行较大的任务时结果如下：

```
holger@hao-zhang:/codes/exp08/period$ bash pfintr2.sh
Use time: 5s
Now the number of page fault is 159263612
Now the number of page fault is 159263870
In 5 seconds, system calls 258 page fault!
```

可见情况稍有改善。但是对于统计缺页次数还是建议使用传统的编程语言，因为其程序的局部性更好，在系统缺页率不是很高的时候可以减少误差。