**操作系统实验报告**

目录

[一、组内成员及分工 1](#_Toc471426157)

[三、Linux内核实验 1](#_Toc471426158)

[实验2.2 内核定时器 1](#_Toc471426159)

[1、实验目的 1](#_Toc471426160)

[2、实验内容 1](#_Toc471426161)

[实验2.3内核模块 9](#_Toc471426162)

[1、实验目的 9](#_Toc471426163)

[2、实验内容 9](#_Toc471426164)

[3、实验原理 9](#_Toc471426165)

[4、实验步骤 10](#_Toc471426166)

[实验2.4 系统调用 13](#_Toc471426167)

[1、实验目的 13](#_Toc471426168)

[2、实验内容与步骤 13](#_Toc471426169)

[四、进程管理实验 18](#_Toc471426170)

[4.2 实验3.2 Shell编程 18](#_Toc471426171)

[1、实验目的 18](#_Toc471426172)

[2、实验内容 18](#_Toc471426173)

[3、实验设计原理 18](#_Toc471426174)

[4、实验步骤 18](#_Toc471426175)

[5、实验结果及分析 19](#_Toc471426176)

[五、存储管理实验 22](#_Toc471426177)

[实验4.1 观察实验(存储管理实验) 22](#_Toc471426178)

[1.实验步骤 22](#_Toc471426179)

[2.观测程序源代码 22](#_Toc471426180)

[3.实验结果及分析 23](#_Toc471426181)

[实验4.2虚拟存储器管理 26](#_Toc471426182)

[1、实验目的 26](#_Toc471426183)

[2、实验内容 26](#_Toc471426184)

[3、实验原理 26](#_Toc471426185)

[4、实验步骤 26](#_Toc471426186)

[六、进程通信 30](#_Toc471426187)

[实验5.1 通信进程观察实验 30](#_Toc471426188)

[1、实验目的与内容 30](#_Toc471426189)

[2、实验原理 30](#_Toc471426190)

[3、实验步骤 30](#_Toc471426191)

[七、I/O设备管理实验 32](#_Toc471426192)

[7.2 实验6.2 编程实验 32](#_Toc471426193)

[1、实验目的 32](#_Toc471426194)

[2、实验内容 32](#_Toc471426195)

[3、实验原理 32](#_Toc471426196)

[4、实验步骤 32](#_Toc471426197)

[5、实验结果及分析 33](#_Toc471426198)

[八、文件系统管理实验 33](#_Toc471426199)

[8.1 实验7.1 代码分析 33](#_Toc471426200)

[1、实验目的 33](#_Toc471426201)

[2、实验内容 33](#_Toc471426202)

[3、实验结果（代码分析） 33](#_Toc471426203)

[实验7.3 编程实验2 36](#_Toc471426204)

[1、实验目的 36](#_Toc471426205)

[2、实验内容 36](#_Toc471426206)

[3、实验原理 37](#_Toc471426207)

[4、实验步骤 37](#_Toc471426208)

[5、实验结果及分析 41](#_Toc471426209)

[九、多核多线程编程 41](#_Toc471426210)

[实验9.1~9.5 实验报告 41](#_Toc471426211)

[1.实验目的 41](#_Toc471426212)

[2.实验内容 41](#_Toc471426213)

[3.实验原理 42](#_Toc471426214)

[4.实验步骤 42](#_Toc471426215)

[5.实验总结 67](#_Toc471426216)

# 组内成员及分工

|  |  |  |
| --- | --- | --- |
| **班级** | **姓名** | **学号** |
| 2014211302 | 周尧棋 | 2014211168 |
| 2014211303 | 高增诚 | 2014211190 |
| 2014211303 | 徐俊雄 | 2014211191 |

分工如下：

|  |  |  |
| --- | --- | --- |
| 周尧棋 | 高增诚 | 徐俊雄 |
| 实验2.2 内核定时器  实验2.3 内核模块  实验2.4 系统调用  实验4.2 虚拟存储器管理  实验5.1观察实验（进程通信） | 9. 多核多线程编程实验1~实验5  实验4.1观察实验（存储管理实验）  实验7.3编程实验2（文件系统管理实验） | 实验3.2 Shell编程（进程管理实验）  实验6.2编程实验（I/O设备管理实验）  实验7.1代码分析（文件系统管理实验） |

# 三、Linux内核实验

## 实验2.2 内核定时器

### 1、实验目的

学习掌握内核定时器的实现原理和方法，建立一种用户空间机制来测量多线程程序的执行时间。

### 2、实验内容

（1）用定时器**ITIMER\_REAL**实现**gettimeofday**的功能。使其一秒钟产生一个信号，计算已经过的秒数。

设计思路：

设置定时器**ITIMER\_REAL**间隔为一秒钟。并为计时到时设定信号处理程序，即**singal(SIGALRM,…)**，使其输出当前所记时间。

源代码part1.c

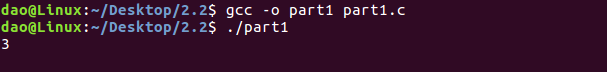
//part 1

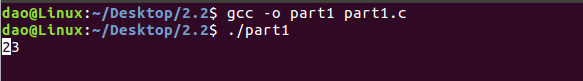
/\*用定时器ITIMER\_REAL实现gettimeofday的功能。使其一秒钟产生一个信号，

计算已经过的秒数。\*/

|  |
| --- |
| #include<sys/time.h>  #include<stdio.h>  #include<signal.h>  #include<sys/time.h>  #include<stdio.h>  #include<signal.h>  static void sighandle(int);  static int second = 0;  int main()  {  struct itimerval v;  signal(SIGALRM,sighandle);  v.it\_interval.tv\_sec = 1;  v.it\_interval.tv\_usec = 0;  v.it\_value.tv\_sec = 1;  v.it\_value.tv\_usec = 0;  setitimer(ITIMER\_REAL,&v,NULL);  for(;;);  }  static void sighandle(int s)  {  second++;  printf("%d\r",second);  fflush(stdout);  } |

将源代码编译后运行程序，结果显示每个一秒显示的数字变化一次，下为3s和23s时的截图





（2）记录一个进程运行时所占用的**real time, cpu time,user time ,kernel time**。

设计思路：

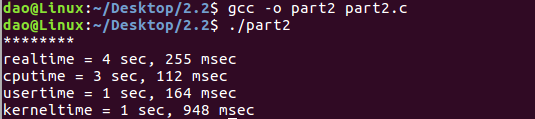
任务开始前设置好定时器**ITIMER\_REAL，ITIMER\_VIRTUAL，ITIMER\_PROF**,即其相应的信号处理程序。在任务执行过程中内核定时器通过产生等间隔的信号来记录进程所需的各种时间参量，并在任务结束后打印出来。

源代码part2.c

//part2.c

|  |
| --- |
| #include <sys/time.h>  #include <stdio.h>  #include <signal.h>  static void sighandle(int);  static long realsecond = 0;  static long vtsecond = 0;  static long profsecond = 0;  static struct itimerval realt,virtt,proft;  int main()  {  struct itimerval v;  int i,j;  long moresec,moremsec,t1,t2;  signal(SIGALRM,sighandle); //SIGALRM，计算实际的时间  signal(SIGVTALRM,sighandle); //SIGVTALM，计算该进程占用CPU的时间  signal(SIGPROF,sighandle); //SIGPROF，包括该进程用的CPU时间和系统调用的时间  v.it\_interval.tv\_sec = 10;  v.it\_interval.tv\_usec = 0;  v.it\_value.tv\_sec = 10;  v.it\_value.tv\_usec = 0;  setitimer(ITIMER\_REAL,&v,NULL); //以系统真实的时间来计算，它送出SIGALRM信号。  setitimer(ITIMER\_VIRTUAL,&v,NULL); //以该进程在用户态下花费的时间来计算，它送出SIGVTALRM信号。  setitimer(ITIMER\_PROF,&v,NULL); //以该进程在用户态下和内核态下所费的时间来计算，它送出SIGPROF信号。  for (j = 0; j < 1000; j ++)  {  for (i = 0; i < 500; i ++)  {  printf("\*\*\*\*\*\*\*\*\r");  fflush(stdout);  }  }  //int getitimer(int which, struct itimerval \*curr\_value);  //getitimer获取which指定的定时器的值并填入curr\_value字段中  getitimer(ITIMER\_PROF,&proft);  getitimer(ITIMER\_REAL,&realt);  getitimer(ITIMER\_VIRTUAL,&virtt);  printf("\n");  moresec = 10 - realt.it\_value.tv\_sec;  moremsec = (1000000 - realt.it\_value.tv\_usec)/1000;  printf("realtime = %ld sec, %ld msec\n",realsecond+moresec,moremsec);    moresec = 10 - proft.it\_value.tv\_sec;  moremsec = (1000000 - proft.it\_value.tv\_usec)/1000;  printf("cputime = %ld sec, %ld msec\n",profsecond+moresec,moremsec);    moresec = 10 - virtt.it\_value.tv\_sec;  moremsec = (1000000 - virtt.it\_value.tv\_usec)/1000;  printf("usertime = %ld sec, %ld msec\n",vtsecond+moresec,moremsec);  t1 = (10 - proft.it\_value.tv\_sec)\*1000 + (1000000 - proft.it\_value.tv\_usec)/1000 + profsecond\*1000;  t2 = (10 - virtt.it\_value.tv\_sec)\*1000 + (1000000 - virtt.it\_value.tv\_usec)/1000 + vtsecond\*1000;  moresec = (t1 - t2)/1000;  moremsec = (t1 - t2) % 1000;  printf("kerneltime = %ld sec, %ld msec\n",moresec,moremsec);  fflush(stdout);  }  static void sighandle(int s)  {  switch(s)  {  case SIGALRM:realsecond+=10;break;  case SIGVTALRM:vtsecond+=10;break;  case SIGPROF:profsecond+=10;break;  default :break;  }  } |

实验结果：下图为测试程序进程运行时所花的各项时间



（3）编写一个主程序产生两个子进程，分别低轨计算**N =20，30，36**的**Fibonacci**序列。分别对三个进程计算相应的**real time, cpu time,user time ,kernel time**。

设计思路：

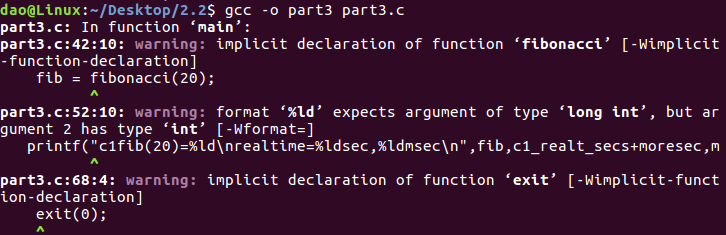
与（2）原理基本相同，不同的只是在任务开始前要分别设定好每个进程的定时器，而且其最终的实验结果也由相应进程自身打印出来。

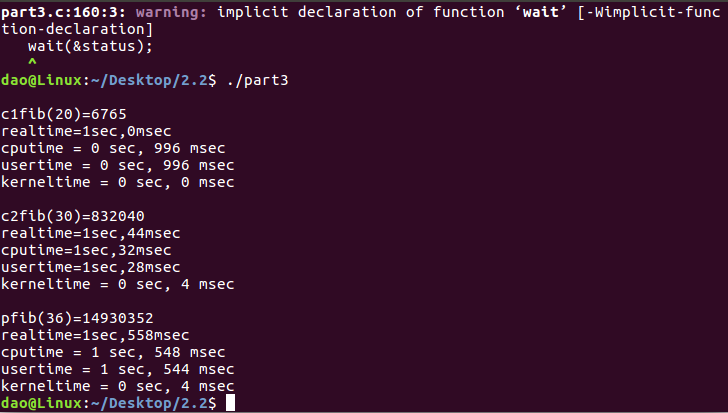
源代码part3.c

//part3

|  |
| --- |
| #include <sys/time.h>  #include <stdio.h>  #include <signal.h>  #include <unistd.h>  static void c1\_sighandle(int s);  static void c2\_sighandle(int s);  static void p\_sighandle(int s);  static long p\_realt\_secs = 0,c1\_realt\_secs = 0,c2\_realt\_secs = 0;  static long p\_virtt\_secs = 0,c1\_virtt\_secs = 0,c2\_virtt\_secs = 0;  static long p\_proft\_secs = 0,c1\_proft\_secs = 0,c2\_proft\_secs = 0;  static struct itimerval p\_realt,c1\_realt,c2\_realt;  static struct itimerval p\_virtt,c1\_virtt,c2\_virtt;  static struct itimerval p\_proft,c1\_proft,c2\_proft;  static struct itimerval ini\_value;  int main()  {  int fib = 0;  int pid1,pid2;  int status;  long moresec,moremsec,t1,t2;  pid1 = fork();  if (pid1 == 0)  {  //c1  //set c1 signal handle  signal(SIGALRM,c1\_sighandle);  signal(SIGVTALRM,c1\_sighandle);  signal(SIGPROF,c1\_sighandle);  ini\_value.it\_interval.tv\_sec = 10;  ini\_value.it\_interval.tv\_usec = 0;  ini\_value.it\_value.tv\_sec = 10;  ini\_value.it\_value.tv\_usec = 0;  //set c1 timer  setitimer(ITIMER\_REAL,&ini\_value,NULL);  setitimer(ITIMER\_VIRTUAL,&ini\_value,NULL);  setitimer(ITIMER\_PROF,&ini\_value,NULL);  fib = fibonacci(20);  //get timer of c1 and print  getitimer(ITIMER\_REAL,&c1\_realt);  getitimer(ITIMER\_VIRTUAL,&c1\_virtt);  getitimer(ITIMER\_PROF,&c1\_proft);  printf("\n");  moresec = 10 - c1\_realt.it\_value.tv\_sec;  moremsec = (1000000 - c1\_realt.it\_value.tv\_usec)/1000;  printf("c1fib(20)=%ld\nrealtime=%ldsec,%ldmsec\n",fib,c1\_realt\_secs+moresec,moremsec);  moresec = 10 - c1\_proft.it\_value.tv\_sec;  moremsec = (1000000 - c1\_proft.it\_value.tv\_usec)/1000;  printf("cputime = %ld sec, %ld msec\n",c1\_proft\_secs+moresec,moremsec);  moresec = 10 - c1\_virtt.it\_value.tv\_sec;  moremsec = (1000000 - c1\_virtt.it\_value.tv\_usec)/1000;  printf("usertime = %ld sec, %ld msec\n",c1\_virtt\_secs+moresec,moremsec);  t1=(10-c1\_proft.it\_value.tv\_sec)\*1000+(1000000-c1\_proft.it\_value.tv\_usec)/1000 + c1\_proft\_secs\*10000;  t2=(10-c1\_virtt.it\_value.tv\_sec)\*1000+(1000000-c1\_virtt.it\_value.tv\_usec)/1000 + c1\_virtt\_secs\*10000;  moresec = (t1 - t2)/1000;  moremsec = (t1 - t2) % 1000;  printf("kerneltime = %ld sec, %ld msec\n",moresec,moremsec);  fflush(stdout);  exit(0);  }//end c1  else  {  pid2 = fork();  if (pid2 == 0)  {  //c2  //set c2 signal handle  signal(SIGALRM,c2\_sighandle);  signal(SIGVTALRM,c2\_sighandle);  signal(SIGPROF,c2\_sighandle);  ini\_value.it\_interval.tv\_sec = 10;  ini\_value.it\_interval.tv\_usec = 0;  ini\_value.it\_value.tv\_sec =10;  ini\_value.it\_value.tv\_usec = 0;  //set c2 timer  setitimer(ITIMER\_REAL,&ini\_value,NULL);  setitimer(ITIMER\_VIRTUAL,&ini\_value,NULL);  setitimer(ITIMER\_PROF,&ini\_value,NULL);  fib = fibonacci(30);  //get timer of c2 and print  getitimer(ITIMER\_PROF,&c2\_proft);  getitimer(ITIMER\_REAL,&c2\_realt);  getitimer(ITIMER\_VIRTUAL,&c2\_virtt);  printf("\n");  moresec = 10 - c2\_realt.it\_value.tv\_sec;  moremsec = (1000000 - c2\_realt.it\_value.tv\_usec)/1000;  printf("c2fib(30)=%ld\nrealtime=%ldsec,%ldmsec\n",fib,c2\_realt\_secs+moresec,moremsec);  moresec = 10 - c2\_proft.it\_value.tv\_sec;  moremsec = (1000000 - c2\_proft.it\_value.tv\_usec)/1000;  printf("cputime=%ldsec,%ldmsec\n",c2\_proft\_secs+moresec,moremsec);  moresec = 10 - c2\_virtt.it\_value.tv\_sec;  moremsec = (1000000 - c2\_virtt.it\_value.tv\_usec)/1000;  printf("usertime=%ldsec,%ldmsec\n",c2\_virtt\_secs+moresec,moremsec);  t1=(10-c2\_proft.it\_value.tv\_sec)\*1000+(1000000-c2\_proft.it\_value.tv\_usec)/1000 + c2\_proft\_secs\*10000;  t2=(10-c2\_virtt.it\_value.tv\_sec)\*1000+(1000000-c2\_virtt.it\_value.tv\_usec)/1000 + c2\_virtt\_secs\*10000;  moresec = (t1 - t2)/1000;  moremsec = (t1 - t2) % 1000;  printf("kerneltime = %ld sec, %ld msec\n",moresec,moremsec);  fflush(stdout);  exit(0);  }//endc2  }  //parent  //setparent signal handle  signal(SIGALRM,p\_sighandle);  signal(SIGVTALRM,p\_sighandle);  signal(SIGPROF,p\_sighandle);  ini\_value.it\_interval.tv\_sec = 10;  ini\_value.it\_interval.tv\_usec = 0;  ini\_value.it\_value.tv\_sec = 10;  ini\_value.it\_value.tv\_usec = 0;  //set parent timer  setitimer(ITIMER\_REAL,&ini\_value,NULL);  setitimer(ITIMER\_VIRTUAL,&ini\_value,NULL);  setitimer(ITIMER\_PROF,&ini\_value,NULL);  fib = fibonacci(36);  getitimer(ITIMER\_PROF,&p\_proft);  getitimer(ITIMER\_REAL,&p\_realt);  getitimer(ITIMER\_VIRTUAL,&p\_virtt);  printf("\n");  moresec = 10 - p\_realt.it\_value.tv\_sec;  moremsec = (1000000 - p\_realt.it\_value.tv\_usec)/1000;  printf("pfib(36)=%ld\nrealtime=%ldsec,%ldmsec\n",fib,p\_realt\_secs+moresec,moremsec);  moresec = 10 - p\_proft.it\_value.tv\_sec;  moremsec = (1000000 - p\_proft.it\_value.tv\_usec)/1000;  printf("cputime = %ld sec, %ld msec\n",p\_proft\_secs+moresec,moremsec);  moresec = 10 - p\_virtt.it\_value.tv\_sec;  moremsec = (1000000 - p\_virtt.it\_value.tv\_usec)/1000;  printf("usertime = %ld sec, %ld msec\n",p\_virtt\_secs+moresec,moremsec);    t1= (10 - p\_proft.it\_value.tv\_sec)\*1000 + (1000000 - p\_proft.it\_value.tv\_usec)/1000 + p\_proft\_secs\*10000;  t2 = (10 - p\_virtt.it\_value.tv\_sec)\*1000 + (1000000 - p\_virtt.it\_value.tv\_usec)/1000 + p\_virtt\_secs\*10000;  moresec = (t1 - t2)/1000;  moremsec = (t1 - t2) % 1000;  printf("kerneltime = %ld sec, %ld msec\n",moresec,moremsec);  fflush(stdout);  //wait c1,c2 terminal  wait(&status);  wait(&status);  return 0;  }//end main  int fibonacci(int n)  {  if( n == 0 ) return 0;  else if( n == 1 || n == 2) return 1;  else return(fibonacci(n-1)+fibonacci(n-2) );  }  static void c1\_sighandle(int s)  {  switch(s)  {  case SIGALRM:c1\_realt\_secs+=10;break;  case SIGVTALRM:c1\_virtt\_secs+=10;break;  case SIGPROF:c1\_proft\_secs+=10;break;  default :break;  }  }  static void c2\_sighandle(int s)  {  switch(s)  {  case SIGALRM:c2\_realt\_secs+=10;break;  case SIGVTALRM:c2\_virtt\_secs+=10;break;  case SIGPROF:c2\_proft\_secs+=10;break;  default :break;  }  }  static void p\_sighandle(int s)  {  switch(s)  {  case SIGALRM:p\_realt\_secs+=10;break;  case SIGVTALRM:p\_virtt\_secs+=10;break;  case SIGPROF:p\_proft\_secs+=10;break;  default :break;  }  } |

实验结果：下图为N=20，30，36时计算三个斐波那契数列进程所花时间的记录





## 实验2.3内核模块

### 1、实验目的

模块是Linux系统的一种特有机制，可用以动态扩展操作系统内核功能。编写实现某些特定功能的模块，将其作为内核的一部分在管态下运行。本实验通过内核模块编程在/porc文件系统中实现系统时钟的读操作接口。

### 2、实验内容

设计并构建一个在/proc文件系统中的内核模块clock，支持read()操作，read()返回值为一字符串，其中包块一个空各分开的两个子串，为别代表xtime.tv\_sec和xtime.tv\_usec。

### 3、实验原理

Linux模块是一些可以作为独立程序来编译的函数和数据类型的集合。在装载这些模块式，将它的代码链接到内核中。Linux模块可以在内核启动时装载，也可以在内核运行的过程中装载。如果在模块装载之前就调用了动态模块的一个函数，那么这次调用将会失败。如果这个模块已被加载，那么内核就可以使用系统调用，并将其传递到模块中的相应函数。

### 4、实验步骤

（1）**编写内核模块**

文件中主要包含init\_module()，cleanup\_module()，proc\_read\_clock()三个函数。其中init\_module()，cleanup\_module()负责将模块从系统中加载或卸载，以及增加或删除模块在/proc中的入口。proc\_read\_clock()负责产生/proc/clock/my\_clock被读时的动作。

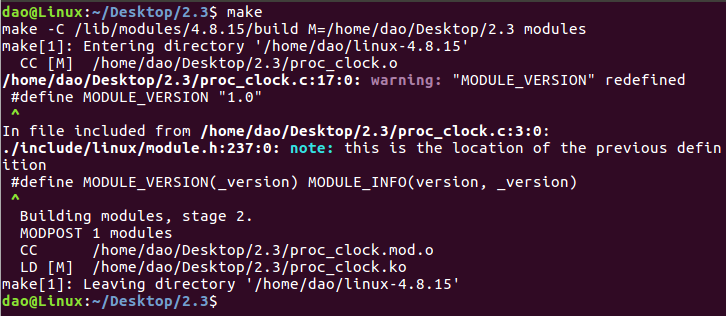
**内核模块源代码proc\_clock.c**

|  |
| --- |
| #include <linux/init.h>  #include <linux/kernel.h>  #include <linux/module.h>  #include <linux/proc\_fs.h>  #include <linux/fs.h>  #include <linux/vmalloc.h>  #include <asm/uaccess.h>  #include <linux/seq\_file.h>  //Prototypes  static int init\_clock(void);  static void cleanup\_clock(void);  static ssize\_t proc\_read\_clock(struct file \*file,char \_\_user \*pszPage,size\_t size,loff\_t \*off);  //#define MODULE  #define MODULE\_VERSION "1.0"  #define USER\_ROOT\_DIR "clock"  #define MODULE\_NAME "my\_clock"  //user defined directory  struct proc\_dir\_entry \*my\_clock\_dir;  struct proc\_dir\_entry \*my\_clock\_file;  static const struct file\_operations my\_clock\_fops =  {  .owner = THIS\_MODULE,  .read = proc\_read\_clock,  };  //Functions  static int init\_clock(void)  {  //Create user root dir under /proc  printk("clock: init\_module()\n");  my\_clock\_dir= proc\_mkdir(USER\_ROOT\_DIR,NULL);  my\_clock\_file = proc\_create(MODULE\_NAME,0,my\_clock\_dir, &my\_clock\_fops);  printk(KERN\_INFO"%s %s has initialized.\n",MODULE\_NAME,MODULE\_VERSION);  return 0;  }  static void cleanup\_clock(void)  {  printk("clock: cleanup\_module()\n");  remove\_proc\_entry(MODULE\_NAME,my\_clock\_dir);  remove\_proc\_entry(USER\_ROOT\_DIR,NULL);  printk(KERN\_INFO"%s %s has removed.\n",MODULE\_NAME,MODULE\_VERSION);  }  static ssize\_t proc\_read\_clock(struct file \*file,char \_\_user \*pszPage,size\_t size,loff\_t \*off)  {  int len = 0;  struct timeval xtime;  do\_gettimeofday(&xtime);  len = sprintf(pszPage,"%ld %ld\n",xtime.tv\_sec,xtime.tv\_usec);  printk("clock: read\_func()\n");  if (len <= \*off)  {  return 0;  }  len -= \*off;  \*off += len;  return len;  }  module\_init(init\_clock);  module\_exit(cleanup\_clock);  MODULE\_DESCRIPTION("clock module for gettimeofday of proc.");  MODULE\_LICENSE("GPL");  //EXPORT\_NO\_SYMBOLS; |

（2）**编译内核模块Makefile文件**

Makefile文件

|  |
| --- |
| ifneq ($(KERNELRELEASE),)  obj-m := proc\_clock.o  else  PWD := $(shell pwd)  KVER := $(shell uname -r)  KDIR := /lib/modules/$(KVER)/build  all:  $(MAKE) -C $(KDIR) M=$(PWD) modules  clean:  rm -rf \*.ko \*.o \*.mod.o \*.mod.c \*.symvers  endif |

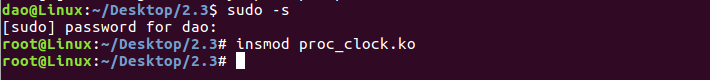


输入make指令，编译完成之后生成proc\_clock.ko模块文件。

（4）**加载内核模块**

输入命令sudo -s转为root用户，在系统root用户下运行用户态模块命令装载内核模块

#insmod proc\_clock.ko

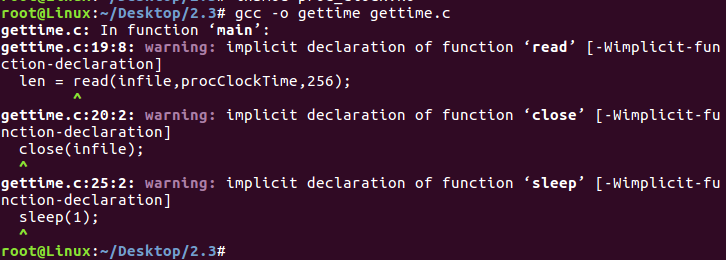


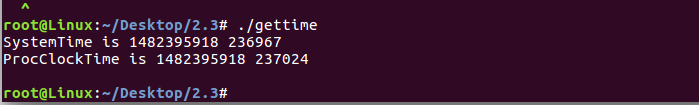
（5）**测试**

测试源代码gettime.c

|  |
| --- |
| #include <stdio.h>  #include <sys/time.h>  #include <fcntl.h>  int main(void)  {  struct timeval getSystemTime;  char procClockTime[256];  int infile,len;  gettimeofday(&getSystemTime,NULL);  infile = open("/proc/clock/my\_clock",O\_RDONLY);  if (infile < 0)  {  printf("error!");  return 0;  }  len = read(infile,procClockTime,256);  close(infile);  procClockTime[len] = '\0';  printf("SystemTimeis %ld %ld\nProcClockTime is %s\n",getSystemTime.tv\_sec ,getSystemTime.tv\_usec,procClockTime);  sleep(1);  } |

将测试代码编译后执行进行检验。





使用系统调用模块输出的时间与直接用gettimeofday()函数输出的时间大致相同

（6）**卸载内核模块**

在系统root用户下运行用户态模块命令卸载内核模块

#rmmod proc\_clock.ko

E:\学习\作业\操作系统\课程实验\2.3\screencut\rmmod.png

**5、实验总结**

**周尧棋：**

编写模块的机制和具体函数在Linux内核版本3.10以后出现了重大的变化，在以前版本中的某些函数在新的内核版本已经不存在或者不在使用，这使得实验进行过程中出现了很多问题和困难。由于我的Linux版本较高，所以自己上网查阅了新版本下内核模块编写的方法，对内核模块的添加和/proc文件系统中的操作函数有了更深的认识。其中一开始proc\_read\_clock()调用后输出的总是乱码，不能输出正确的当前系统时间。上网查询资料后将函数的参数和其中的一些实现进行了修改，修改为新的Linux版本使用的操作后终于实现了正确的输出，自己也挺有收获的。

## 实验2.4 系统调用

### 1、实验目的

向现有Linux内核加入一个新的系统调用从而在内核空间中实现对用户空间的读写。

例如，设计并实现一个新的内核函数mycall()，此函数通过一个引用参数的调用返回当前系统时间，功能上基本与gettimeofday()相同。也可以实现具有其它功能的系统调用。

### 2、实验内容与步骤

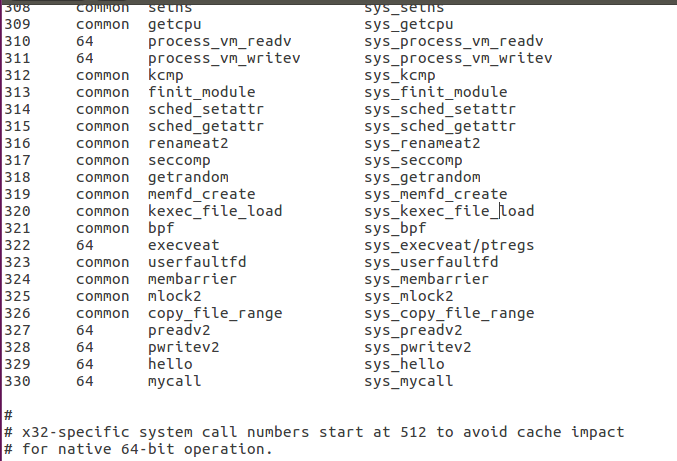
（1）**下载Linux内核源代码**

在<https://www.kernel.org/>上下载了linux-4.8.15.tar.xz放在了home文件夹中，并解压

tar -xvf linux-4.8.15.tar.xz

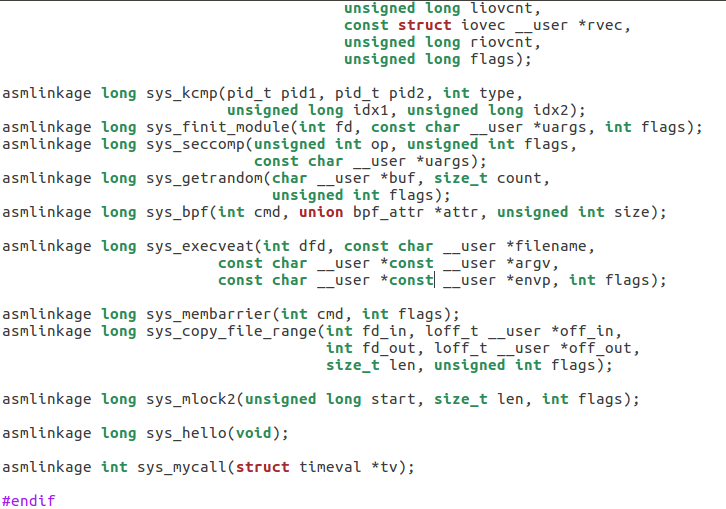
（2）**添加系统调用号**

在/home/linux-4.8.15/arch/x86/entry/syscalls/syscall\_64.tbl添加新的系统调用号



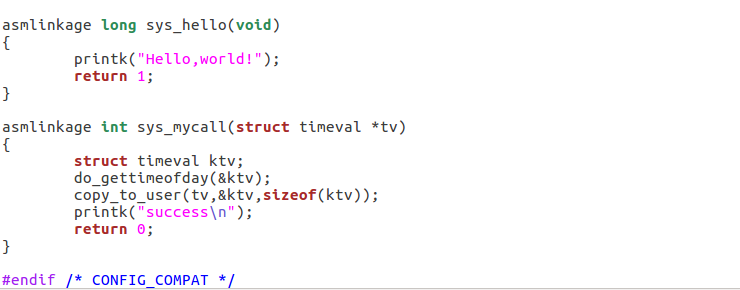
**（3）声明系统调用函数模型**

在/home/linux-4.8.15/include/linux/syscalls.h添加函数声明



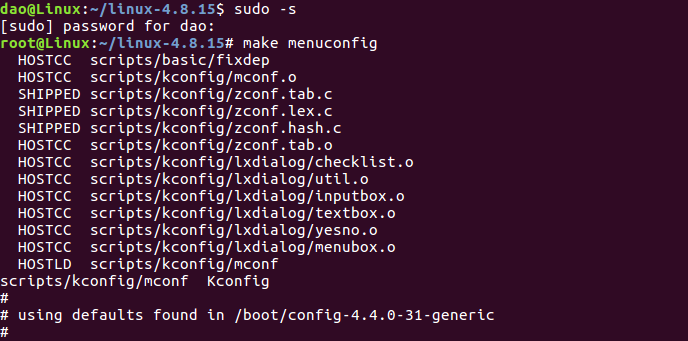
**（4）添加系统调用函数的定义**

在/home/linux-4.8.15/kernel/sys.c中添加相应的调用代码

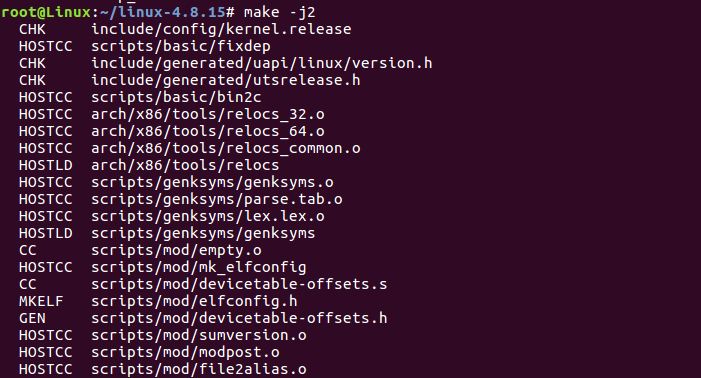


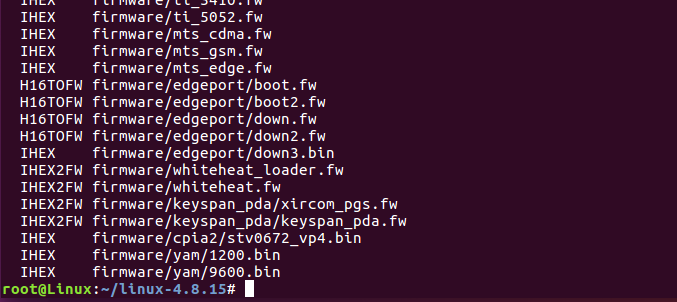
（5）**编译内核**

a、输入命令sudo -s转为root用户，再输入命令make menuconfig 直接选择save，然后退出即可，生成.config



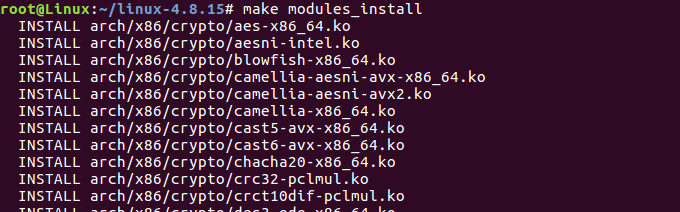
b、配置完毕后，输入命令make -j2使用双线程编译内核

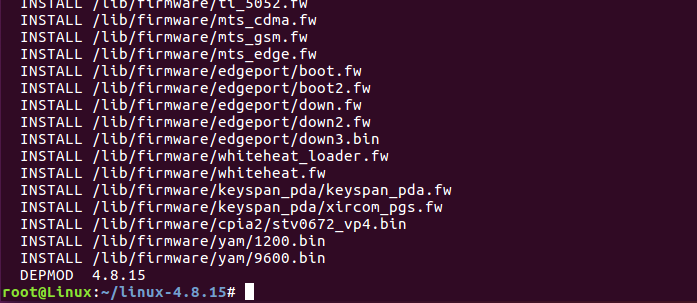




内核编译完毕

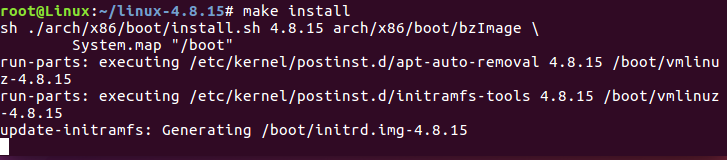
c、输入指令make modules\_install 安装模块

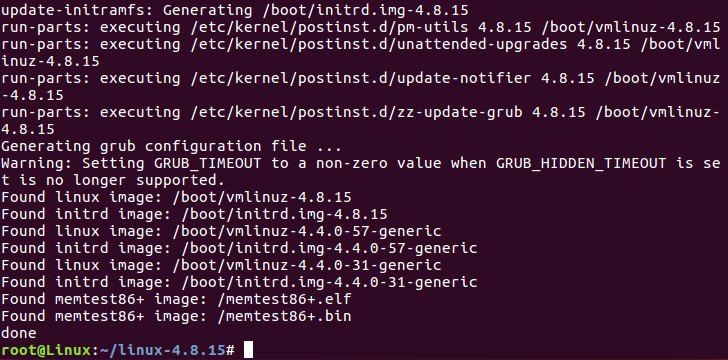




模块安装完毕

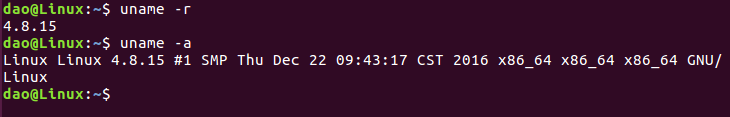
d、输入指令make install 安装内核





安装完毕

e、重启后检查内核版本，变为刚编译的内核了！



（6）测试系统调用

sys\_hello和sys\_mycall

源代码如下：

**//sys\_hello.c**

|  |
| --- |
| #include <stdio.h>  #include <linux/kernel.h>  #include <sys/syscall.h>  #include <linux/unistd.h>  int main()  {  long int amma = syscall(329);  printf("System call sys\_hello returned %ld\n",amma);  return 0;  } |

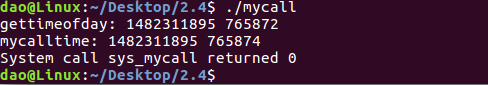
**//sys\_mycall.c**

|  |
| --- |
| #include <stdio.h>  #include <linux/kernel.h>  #include <sys/syscall.h>  #include <linux/time.h>  #include <linux/unistd.h>  int main()  {  struct timeval gettime;  struct timeval mycalltime;  gettimeofday(&gettime,NULL);  long int amma = syscall(330,&mycalltime);  printf("gettimeofday: %d %d\n",gettime.tv\_sec,gettime.tv\_usec);  printf("mycalltime: %d %d\n",mycalltime.tv\_sec,mycalltime.tv\_usec);  printf("System call sys\_mycall returned %ld\n",amma);  return 0;  } |

测试程序运行截图：

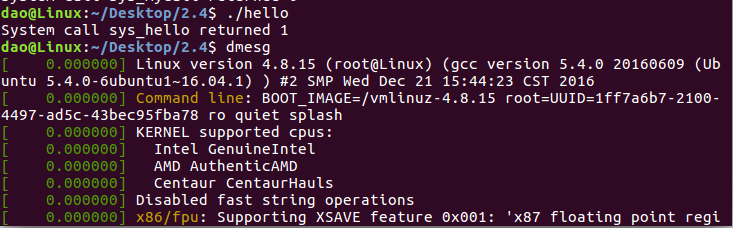
a、sys\_mycall

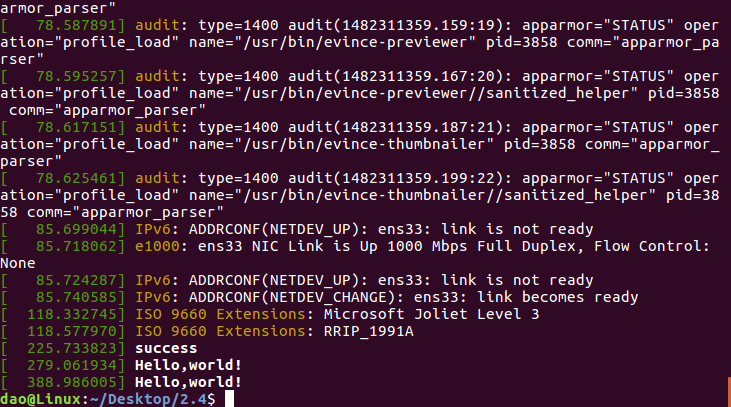
syscall的参数为系统调用号。通过系统调用打印出了系统时间，功能上基本与gettimeofday( )相同。



b、sys\_hello

syscall的参数为系统调用号。该语句会将此输出作为系统的日志，通过dmesg命令查看。由于日志并没有立即输出，我又执行了一次测试程序，日志便将两次的Hello,world!都输出了，并且发现还输出了前面sys\_call系统调用中的打印语句，打印出了success。





**3、实验总结**

**周尧棋：**

在添加系统调用的实验中，关键的一步便是编译内核。由于下载的Linux内核版本较高，在一开始并没有找到实验指导书上所说的那些文件来进行修改，上网查询了资料后才找到了正确的需要修改的文件，向其中添加了自己的系统调用。在编译内核尝试的前一两次，总是中途报错，原因是缺少了一些必要的文件如curses.h和openssl.h等文件，输入指令**sudo apt-get install libncurses5-dev** 和**sudo apt-get install libssl-dev**安装完必要的文件后终于能正常的开始编译了。在编译内核的第三次尝试时，编译了三个小时后，途中出现了空间不足的错误，让人有些绝望。重装系统并分配了50G硬盘后的我卷土重来，然而第四次的内核编译时间长达了七个小时，让人痛不欲生，查阅资料发现原因可能是Linux版本过新或者Ubuntu系统内核过大，需要安装很多相关drivers。用Red Hat系统和低版本Linux编译内核的时长没有那么长。并且在输入make指令时是使用了单线程进行编译，第五次内核编译使用了双线程后速度明显缩短。经过漫长的努力后终于完成了内核版本的更新，让我十分兴奋！此次实验还是让我对Linux操作系统有了更深的认识，与内核打交道确实不是一件容易的事。

# 四、进程管理实验

## 4.2 实验3.2 Shell编程

### 1、实验目的

通过编写shell程序，了解子进程的创建和父进程与子进程间的协同，获得多进程程序的编程经验。

### 2、实验内容

设计一个简单的shell解释程序，能实现基本的bsh功能。

### 3、实验设计原理

将每一条命令分子段压入argv栈。然后在子进程中调用execvp()来实现该命令的功能。

### 4、实验步骤

（1）编写代码

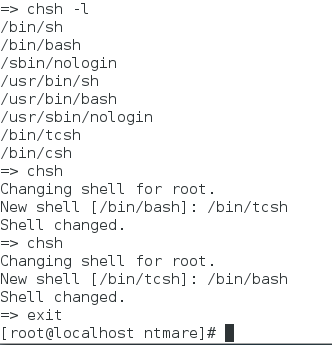
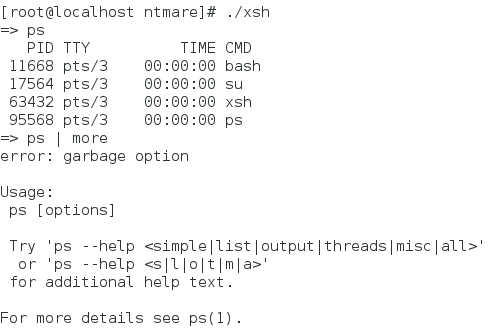
|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #define BUFFERSIZE 256  //最简单的shell，只是简单的执行命令调用，没有任何的其他功能  int main()  {  char buf[BUFFERSIZE],\*cmd,\*argv[100];  char inchar;  int n,sv,buflength;  buflength = 0;  for(;;) {  printf("=> ");  //处理过长的命令；  inchar = getchar();  while (inchar != '\n' && buflength < BUFFERSIZE ){  buf[buflength++] = inchar;  inchar = getchar();  }  if (buflength > BUFFERSIZE){  printf("Command too long,please enter again!\n");  buflength = 0;  continue;  }  else  buf[buflength] = '\0';  //解析命令行，分成一个个的标记  cmd=strtok(buf," \t\n");  if(cmd) {  if(strcmp(cmd,"exit")==0) exit(0);  n=0;  argv[n++]=cmd;  while(argv[n++]=strtok(NULL," \t\n"));  if(fork()==0) {  execvp(cmd,argv);  fprintf(stderr,"sxh:%s:command not found.\n",buf);  exit(1);  }  wait(&sv);  buflength = 0;  }  }  } |

（2）编译运行



### 5、实验结果及分析

执行结果如下：



文字版：

[root@localhost ntmare]# ./xsh

=> ps

PID TTY TIME CMD

11668 pts/3 00:00:00 bash

17564 pts/3 00:00:00 su

63432 pts/3 00:00:00 xsh

95568 pts/3 00:00:00 ps

=> ps | more

error: garbage option

Usage:

ps [options]

Try 'ps --help <simple|list|output|threads|misc|all>'

or 'ps --help <s|l|o|t|m|a>'

for additional help text.

For more details see ps(1).

=> ps > ps.txt

error: garbage option

Usage:

ps [options]

Try 'ps --help <simple|list|output|threads|misc|all>'

or 'ps --help <s|l|o|t|m|a>'

for additional help text.

For more details see ps(1).

=> ls

Desktop hello Pictures shiyan6.2.cpp xsh.c

Documents hello.cpp Public Templates

Downloads kernel-3.10.0-327.36.3.el7.src.rpm rpmbuild Videos

f1.txt Music shiyan6.2 xsh

=> chsh -l

/bin/sh

/bin/bash

/sbin/nologin

/usr/bin/sh

/usr/bin/bash

/usr/sbin/nologin

/bin/tcsh

/bin/csh

=> chsh

Changing shell for root.

New shell [/bin/bash]: /bin/tcsh

Shell changed.

=> chsh

Changing shell for root.

New shell [/bin/tcsh]: /bin/bash

Shell changed.

=> exit

[root@localhost ntmare]#

结果分析：

1. ps命令能够给出当前系统中进程的快照。它能捕获系统在某一事件的进程状态。

不加参数执行ps命令，结果默认会显示4列信息。

PID: 运行着的命令(CMD)的进程编号

TTY: 命令所运行的位置（终端）

TIME: 运行着的该命令所占用的CPU处理时间

CMD: 该进程所运行的命令

上述的运行结果与之相符。

1. ls命令用于列出文件和目录。默认上，他会列出当前目录的内容。

不带参数运行ls会只列出文件或者目录。看不到其他信息输出。

上述的运行结果与之相符。

1. chsh命令用来修改设定用户的shell。在上面的执行过程中，我先执行chsh -l列出当前系统所有可用shell。而后执行chsh修改shell，可以看到，修改之前显示的当前shell为[/bin/bash]，修改后为[/bin/tcsh]，实现该命令功能后，我又将shell修改回原样。
2. 最后，输入”exit”退出程序。

# 五、存储管理实验

## 实验4.1 观察实验(存储管理实验)

### 1.实验步骤

(1)、安装GDB

(2)、编写观测程序

(3)、按照指令手册进行观察操作

### 2.观测程序源代码

#include<stdio.h>

#include<stdlib.h>

char str[50] = "Hello Linux.";

int main()

{

int num = 10;

while(num--){

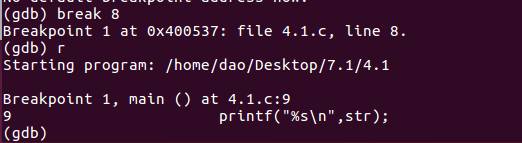
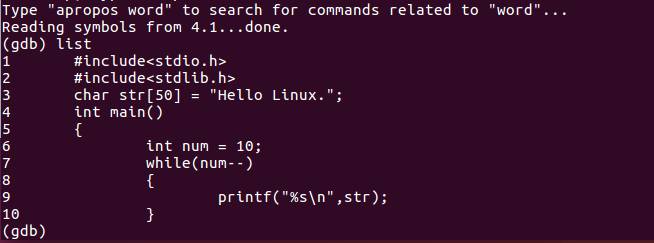
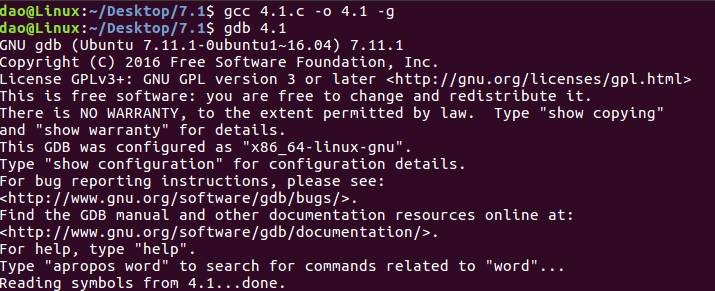
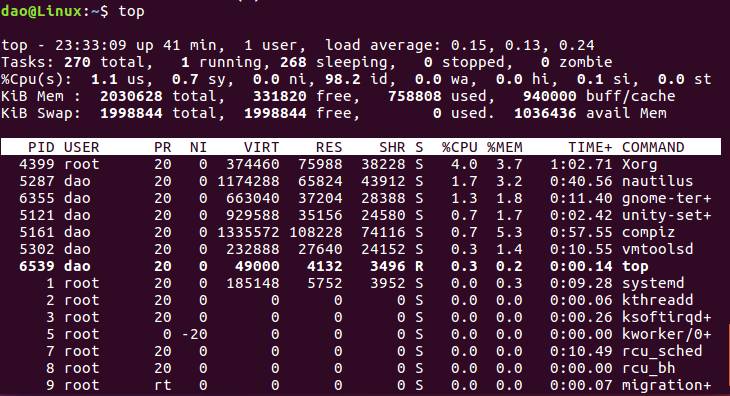
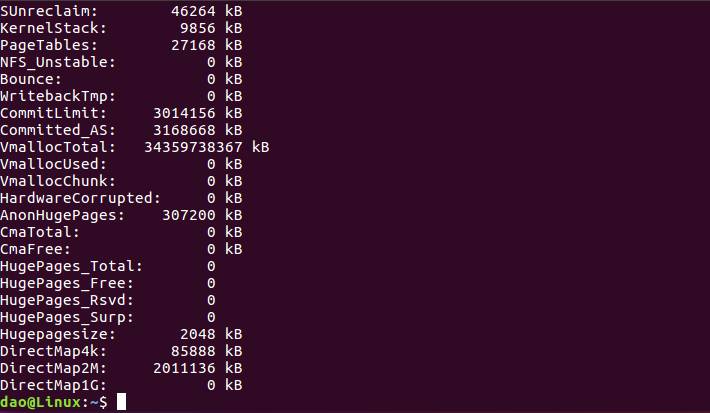
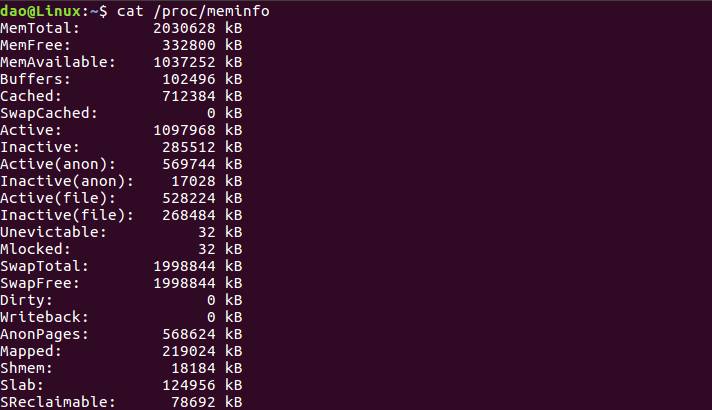
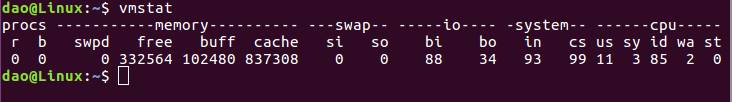
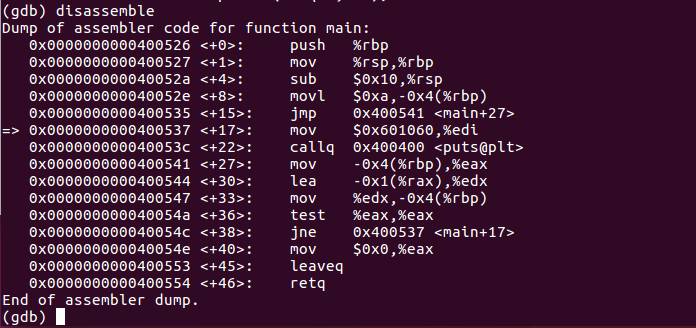
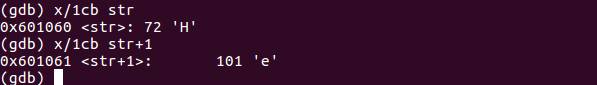
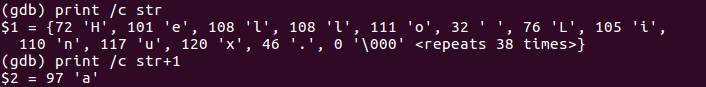
printf("%s\n",str);

}

}

### 3.实验结果及分析

结果截图：

9

## 实验4.2虚拟存储器管理

### 1、实验目的

学习Linux虚拟存储实现机制；编写代码，测试虚拟存储系统的缺页错误（缺页中断）发生频率。

### 2、实验内容

修改存储管理软件以便可以判断特定进程或者整个系统产生的缺页情况，达到一下目标

（1）预置缺页频率参数

（2）报告当前缺页频率

### 3、实验原理

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的变量pfcount作为计数变量，在执行do\_page\_fault时，该变量加1。系统经历的时间可以利用原有系统的变量jiffies，这是一个系统计时器。在内核加载以后开始计时，以10ms为计时单位

实现施可采用2种方案

1. 通过提供一个系统调用来访问内核变量pfcount和jiffies。但是增加系统变量存在居多的不便，如重新编译内核等，而且容易出错以致系统崩溃。
2. 通过/proc文件系统以模块的方式提供内核变量的访问接口。在/proc文件系统下建立目录pf以及在该目录下的文件Pfcount和Jiffies。

### 4、实验步骤

**（1）下载Linux内核源代码**

在<https://www.kernel.org/>上下载了linux-4.8.15.tar.xz放在了home文件夹中，并解压

tar -xvf linux-4.8.15.tar.xz

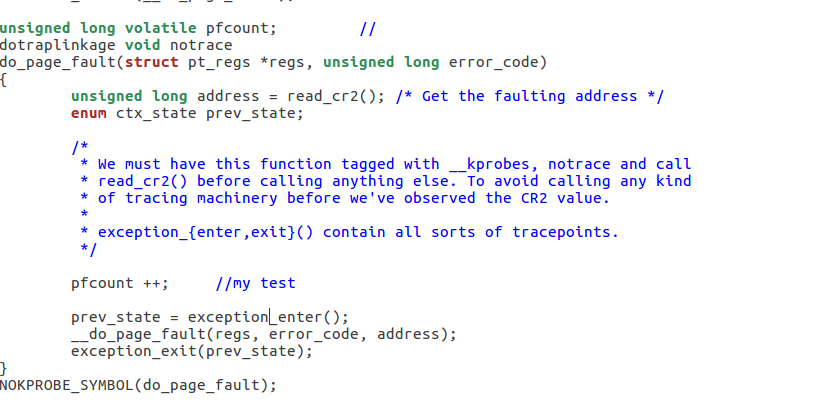
**（2）修改内核源代码**

a、添加统计变量

在/home/linux-4.8.15/arch/x86/mm/fault.c中的do\_page\_fault函数的上一行定义统 计缺页次数的全局变量pfcount，加上unsignedlong volatilepfcount;

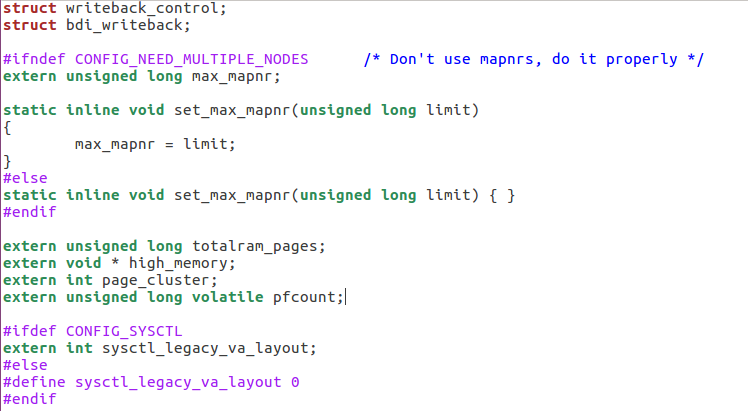
将pfcount加入到do\_page\_fault全局范围，用以统计缺页次数

pfcount++;



b、修改内存管理代码

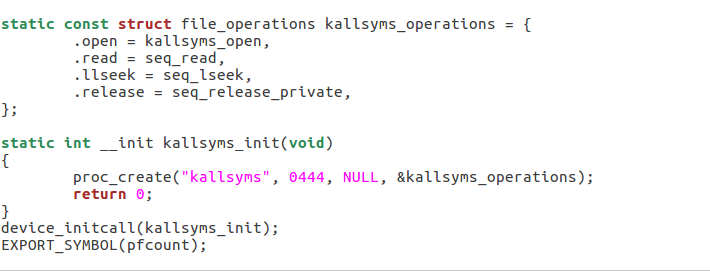
在/home/linux-4.8.15/include/linux/mm.h中加入全局变量pfcount的声明，代码加 在extern int page\_cluster;语句之后，写入extern unsigned long volatile pfcount;



c、导出pfcount全局变量

为让整个内核（包括模块）都可以访问pfcount，在 /home/linux-4.8.15/kernel/kallsyms.c中加入

EXPORT\_SYMBOL(pfcount);



**（3）重新编译内核**

**（4）编写内核模块代码pf.c**

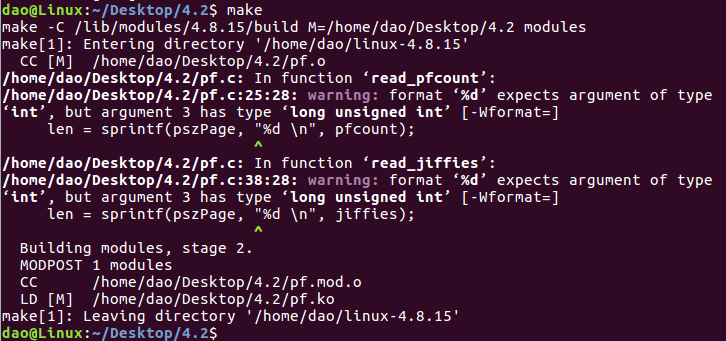
a、pf.c源代码

|  |
| --- |
| #include <linux/init.h>  #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/mm.h>  #include <linux/proc\_fs.h>  #include <linux/fs.h>  #include <linux/string.h>  #include <asm/uaccess.h>  //#define MODULE  #define USER\_ROOT\_DIR "pf"  #define MODULE1 "Pfcount"  #define MODULE2 "Jiffies"  extern unsigned long volatile pfcount;  extern unsigned long volatile jiffies;  struct proc\_dir\_entry \*proc\_pfcount;  struct proc\_dir\_entry \*proc\_jiffies;  struct proc\_dir\_entry \*my\_dir;  static ssize\_t read\_pfcount(struct file \*file, char \_\_user \*pszPage, size\_t size, loff\_t \*off)  {  int len = 0;  len = sprintf(pszPage, "%d \n", pfcount);  if (len <= \*off)  {  return 0;  }  len -= \*off;  \*off += len;  return len;  }  static ssize\_t read\_jiffies(struct file \*file, char \_\_user \*pszPage, size\_t size, loff\_t \*off)  {  int len = 0;  len = sprintf(pszPage, "%d \n", jiffies);  if (len <= \*off)  {  return 0;  }  len -= \*off;  \*off += len;  return len;  }  static const struct file\_operations proc\_fops1=  {  .owner = THIS\_MODULE,  .read = read\_pfcount,  };  static const struct file\_operations proc\_fops2=  {  .owner = THIS\_MODULE,  .read = read\_jiffies,  };  int pf\_init(void)  {  printk("init\_module()\n");  my\_dir = proc\_mkdir(USER\_ROOT\_DIR, NULL);  proc\_pfcount = proc\_create(MODULE1,0,my\_dir,&proc\_fops1);  proc\_jiffies = proc\_create(MODULE2,0,my\_dir,&proc\_fops2);  return 0;  }  void pf\_exit(void)  {  remove\_proc\_entry(MODULE1,my\_dir);  remove\_proc\_entry(MODULE2,my\_dir);  remove\_proc\_entry(USER\_ROOT\_DIR,NULL);  }  module\_init(pf\_init);  module\_exit(pf\_exit);  MODULE\_LICENSE("GPL"); |

b、编译内核模块Makefile文件

|  |
| --- |
| ifneq ($(KERNELRELEASE),)  obj-m := pf.o  else  PWD := $(shell pwd)  KVER := $(shell uname -r)  KDIR := /lib/modules/$(KVER)/build  all:  $(MAKE) -C $(KDIR) M=$(PWD) modules  clean:  rm -rf \*.ko \*.o \*.mod.o \*.mod.c \*.symvers  endif |

编译完成之后生成pf.ko模块文件。



c、加载内核模块

输入sudo insmod pf.ko模块命令装载内核模块

E:\学习\作业\操作系统\课程实验\4.2\screencut\insmod.png

d、测试

输入命令cat /proc/pf/Pfcount查看pfcount的数量

E:\学习\作业\操作系统\课程实验\4.2\screencut\pfcount.png

输入命令cat /proc/pf/Jiffies查看jiffies的数量

E:\学习\作业\操作系统\课程实验\4.2\screencut\jiffies.png

# 六、进程通信

## 实验5.1 通信进程观察实验

### 1、实验目的与内容

在Linux下，用ipcs()命令观察进程通信情况，了解Linux基本通信机制。

### 2、实验原理

Linux IPC继承了Unix System V及DSD等，共有6种机制：信号(signal)、管道(pipe和命名管道(named piped)、消息队列（message queues）、共享内存（shared memory segments）、信号量（semaphore）、套接字（socket）。

本实验中用到的几种进程间通信方式：

（1）共享内存段（shared memory segments）方式

– 将２个进程的虚拟地址映射到同一内存物理地址，实现内存共享

– 对共享内存的访问同步需由用户进程自身或其它IPC机制实现（如信号量）

– 用户空间内实现，访问速度最快。

– Linux利用shmid\_ds结构描述所有的共享内存对象。

（2）信号量（semaphore）方式

– 实现进程间的同步与互斥

– P/V操作， Signal/wait操作

– Linux利用semid\_ds结构表示IPC信号量

（3）消息队列（message queues）方式

–消息组成的链表，进程可从中读写消息。

– Linux维护消息队列向量表msgque，向量表中的每个元素都有一个指向msqid\_ds结构的指针，每个msqid\_ds结构完整描述一个消息队列

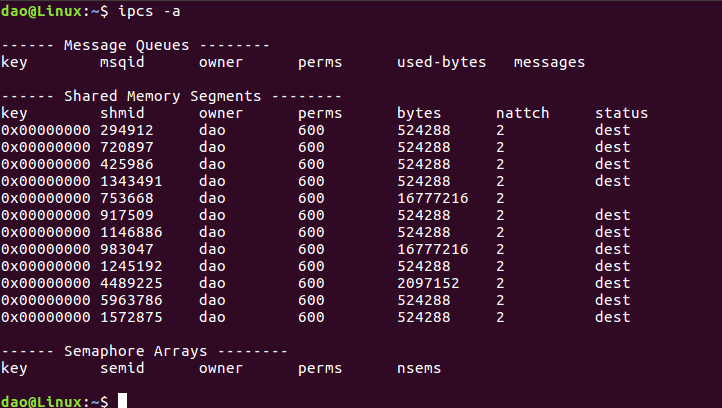
LINUX系统提供的IPC函数有：

* msgget(关键字，方式）：创建或打开一个消息队列
* msgsnd(消息队列标志符，消息体指针，消息体大小，消息类型）：　向队列传递消息
* msgrcv(消息队列标志符，消息体指针，消息体大小，消息类型）：　从队列中取消息
* msgctl(消息队列标志符，获取／设置／删除，maqid\_ds缓冲区指针）：　获取或设置某个队列信息，或删除某消息队列

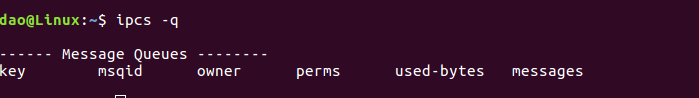
Linux系统中，内核，I/O任务，服务器进程和用户进程之间采用消息队列方式，许多微内核OS中，内核和各组件间的基本通信也采用消息队列方式.

### 3、实验步骤

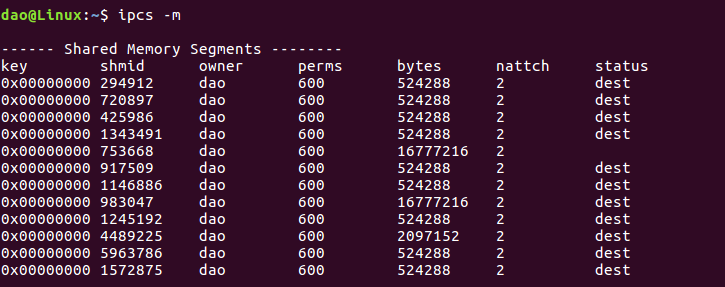
在命令行中使用ipcs -a就可以看到所有进程的情况。



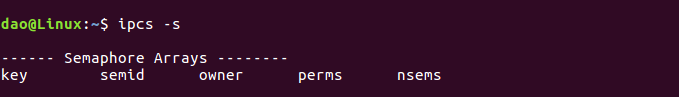
使用-q打印出使用消息队列进行进程间通信的信息



使用-m参数打印出使用共享内存进行进程间通信的信息



使用-s打印出使用信号进行进程间通信的信息



# 七、I/O设备管理实验

## 7.2 实验6.2 编程实验

### 1、实验目的

编写一个daemon进程，该进程定时执行 ps命令，然后将该命令的输出写至文件F1尾部。通过此实验，掌握Linux I/O系统相关内容。

### 2、实验内容

编写一个daemon进程，该进程定时执行 ps命令，然后将该命令的输出写至文件F1尾部。

### 3、实验原理

在这个程序中，首先fork一个子程序，然后，关闭父进程，这样，新生成的子进程被交给init进程接管，并在后台执行。

新生成的子进程里，使用system系统调用，将ps的输出重定向，输入到f1.txt里面。

关于fork()函数:

1）在父进程中，fork返回新创建子进程的进程ID；

2）在子进程中，fork返回0；

3）如果出现错误，fork返回一个负值；

据此，可以根据fork的返回值判断该进程为子进程或父进程，若为父进程，则退出该进程。

### 4、实验步骤

（1）编写代码

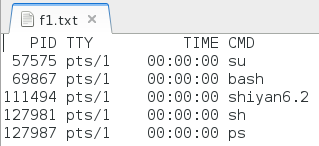
|  |
| --- |
| #include<unistd.h>  #include<stdio.h>  #include<stdlib.h>  int main()  { int p;  p = fork();  if(p > 0)  {  exit(0);  }  else if(p == 0)  {  for(int i = 0; i < 100; i++)  {  sleep(100);  system("ps > f1.txt");  }  }  else  {  perror("Create new process!");  }  return 0;  } |

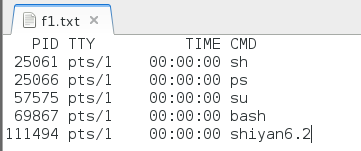
（2）编译运行



### 5、实验结果及分析

执行该程序，休眠100s后，子进程会将第一个ps的内容输出到f1.txt文件中，该文件位于程序所在的同一目录下。其内容如下：



此后每sleep（100），该文件会更新一次，重复100次。某一次更新后的内容如下：

# 八、文件系统管理实验

## 8.1 实验7.1 代码分析

### 1、实验目的

了解与文件管理有关的Linux内核模块的代码结构。

### 2、实验内容

阅读 Linux/Minix中有关文件模块的调用主线,并写出分析报告，包括

* 文件建立模块，即系统调用create()
* 文件删除模块，即系统调用rm()
* 读/写模块，即 read/write

### 3、实验结果（代码分析）

**A. 创建文件模块分析**

|  |
| --- |
| 5780 /\*creat system call \*/  5781 Creat()  5782 {  5783 resister \*ip;  5784 extern uchar;  5785  5786 ip = namei(&uchar,1);  5787 if(ip == NULL){  5788 if(u.u\_error)  5789 return;  5790 ip = maknode(u.u\_arg[1]&07777&(~ISVTX));  5791 if (ip == NULL)  5792 return;  5793 open1(ip,FWRITE,2);  5794 }else  5795 open1(ip,FWRITE,1);  5796 } |

第5 7 8 6：“namei”( 7 5 1 8 )将一路径名变换成一个“inode”指针。“uchar”是一个过程的名字，它从用户程序数据区一个字符一个字符地取得文件路径名。

5 7 8 7：一个空“inode”指针表示出了一个错，或者并没有具有给定路径名的文件存在。

5 7 8 8：对于出错的各种条件，请见U P M的C R E AT ( I I )。

5 7 9 0：“maknode”( 7 4 5 5 )调用“ialloc”创建一内存“ inode”，然后对其赋初值，并使其进入适当的目录。注意，显式地清除了“粘住”位( I S V T X )。

**B. 删除文件rm模块分析**

|  |
| --- |
| 3510 unlink()  3511 {  3512 resister \*ip,\*pp;  3513 extern uchar;  3514  3515 pp = namei(&uchar,2);  3516 if (pp ==NULL)  3517 return;  3518 prele(pp);  3519 ip = iset(pp ->dev,u.u\_dent.u\_ino);  3520 if (ip == NULL)  3521 panic (\*unlink – iset \*);  3522 if ((ip ->i\_mode%IFMT) == IFDIR && !suser())  3523 goto out;  3524 u.u\_offset[1] = - DIRSIZ+2;  3525 u.ubase = &u.u\_dent;  3526 u.ucount = DIRSIZE +2;  3527 u.u\_dent.u\_ino = 0;  3528 writei(pp);  3529 ip ->i\_nlink--;  3530 ip->i\_flag =! IUPD;  3531  3532 out:  3533 iput(pp);  3534 iput(ip);  3535 } |

新文件作为永久文件自动进入文件目录。关闭文件不会自动地造成文件被删除。当内存“ inode”项中的“ i \_ nlink”字段值为0并且相应文件未被打开时，将删除该文件。在创建文件时，该字段由“ m a k n o d e”赋初值为1。系统调用“ link”( 5 9 4 1 )可将其值加1，系统调用“unlink”( 3 5 2 9 )则可将其值减1。创建临时“工作文件”的程序应当在其终止前执行“ unlink”系统调用将这些文件删除。

注意，“unlink”系统调用本身并没有删除文件。当引用计数( i \_ count )被减为0时( 7 3 5 0、7 3 6 2 )，才删除该文件。

**C. 读写模块分析**

|  |
| --- |
| 5711 Read( )  5712{  5713 rdwr(FREAD);  5714 }  5720 Write( )  5721 {  5722 rdwr(FWRITE);  5723 }  5731 rdwr(mode)  5732{  5733 resister \*fp,m;  5734  5735 m = mode;  5736 fp = setf(u.u\_arg[R0]);  5737 if (fp ==NILL)  5738 return;  5739 if ((fp ->f\_flag&m ==0) {  5740 u.u\_error = EBADF;  5741 return;  5742 }  5743 u.u\_base = u.u\_arg[0];  5744 u.u\_count = u.u\_arg[1];  5745 u.u\_segflg = 0;  5746 if(fp ->f\_flag&FPIPE) {  5747 if(m == FREAD)  5748 readp(fp);else  5749 writep(fp);  5750 }else{  5751 u.u\_offset[1] = fp ->f\_offset[1];  5752 u.u\_offset[0] = fp ->f\_offset[0];  5753 if (m == FREAD)  5754 readi(fp ->f\_inode);else  5755 writei(fp ->f\_inode);  5756 dpadd(fp ->f\_offset,u.u\_arg[1] – u.u\_count;  5757 }  5758 u.u\_ar0[R0] = u.u\_arg[1] – u.u\_count;  5759 } |

“read”系统调用的基本工作过程为：

……read ( f , b , n ) ;／\*用户程序\*／

｛发生陷入｝

2693 trap｛＃3 系统调用｝

5711 read() ；

5713 rdwr(FREAD);

用户进程执行系统调用激活运行在核心态的“ trap”。“trap”识别＃３系统调用，然后通过“trap l”调用例程“read”，它又调用“rdwr”。

“rdwr”包含了很多“read”和“write”操作共用的代码。它调用“ getf”( 6 6 1 9 )将用户进程提供的文件标识变换成“file”数组中一项的地址。

注意，该系统调用的第 1 个参数是以不同于另外 2 个参数的方式传送的。

将“u.u\_segflg”设置为0，这表示此操作的目的地址在用户地址空间中。在以一个 inode指针参数调用“ read i”后，将要求传送的字符数减去剩余未传输字符数(在 u.u \_ count 中)，加至文件位移量中。

“read”和“ write”执行的操作有很多相似之处，两者共享很多代码。

系统调用“ read”(5711)和“write”( 5720)，然后立即调用“rdwr”，它执行下列操作：

5736：将用户程序文件标识变换成指向相应文件表项的指针。

5739：检查所要求的操作(读或写)是否与文件打开时的读/写方式符合。

5743：用各参数在“u”中设置几个标准单元。

5746：从此开始对“管道”文件进行特殊处理。

5755：按读、写要求分别调用“ read i”或“write i”。

5756：更新文件位移量，使其增加实际传送的字符数，同时也将实际传送的字符数返回。

## 实验7.3 编程实验2

### 1、实验目的

编写一个利用Linux系统调用删除文件的程序，加深对文件系统和文件操作的理解。

### 2、实验内容

在Linux下，编写 Shell程序，计算磁盘上所有目录下平均文件个数、所有目录平均深度、所有文件名平均长度。

通过此实验，了解Linux系统文件管理相关机制。

### 3、实验原理

### 4、实验步骤

* **修改7-3程序，即文件删除程序如下：**

#include <sys/types.h>

#include <stdio.h>

#include <dirent.h>

#include <sys/stat.h>

#include <string.h>

#include <stdlib.h>

/\* selective options about the myrm \*/

int iflag, rflag, vflag;

int myrm1(char \*);

int myrm\_file(char \*);

int myrm\_dir(char \*);

int usage(void);

char getyn(char \*);

int main(int argc, char \*\*argv)

{

char \*path, \*opt;

/\* if input only contains program name, then give some informations \*/

argv++;

argc--;

if (!argc)

usage();

/\* read options \*/

while (argc != 0 && \*\*argv == '-')

{

opt = ++\*argv;

argc--;

argv++;

/\* if the option is "--" then end reading options and goto next argument \*/

if (opt[0] == '-' && opt[1] == '\0')

break;

/\* obtain the options \*/

for (; \*opt != '\0'; opt++)

switch (\*opt)

{

case 'i':

iflag = 1;

break;

case 'v':

vflag = 1;

break;

case 'r':

rflag = 1;

break;

default:

usage();

}

}

if (!argc)

usage();

/\* remove files \*/

while ((path = \*argv++) != NULL)

myrm1(path);

return(0);

}

int myrm1(char \*path)

{

struct stat st;

/\* open the inode of file \*/

if (lstat(path, &st))

{

fprintf(stderr, "Error: File %s not exist!\n", path);

return(-1);

}

/\* judge whether the file is a file or a directory \*/

if (S\_ISDIR(st.st\_mode))

if (rflag)

myrm\_dir(path);

else

fprintf(stderr, "Error: %s is a directory, and can't be removed.\n", path);

else /\* not a directory, remove this file \*/

myrm\_file(path);

return(0);

}

int myrm\_dir(char \*path)

{

DIR \*dp;

struct dirent \*dirp;

char \*dirname, ch, \*msg;

/\* if iflag is 1, ask whether go into this directory \*/

msg = (char \*)malloc(strlen(path) + 30);

if (iflag)

{

msg = strcat(strcpy(msg, path), ": chdir this directory?(Y/N)");

if (ch = getyn(msg), ch == 'n' || ch == 'N')

return(0);

}

/\* open the directory \*/

if ((dp = opendir(path)) == NULL)

{

fprintf(stderr, "Error: can't open directory %s!.\n", path);

return(-1);

}

chdir(path);

/\* read each entry in the directory and remove it\*/

dirp = (struct dirent \*)malloc(sizeof(struct dirent));

while ((dirp = readdir(dp)) != NULL)

{

if (strcmp(dirp->d\_name, ".") == 0 || strcmp(dirp->d\_name, "..") == 0)

continue;

myrm1(dirp->d\_name);

}

/\* close the directory \*/

if (closedir(dp)) {

fprintf(stderr, "Error: can't close the directory %s!\n", path);

return(-1);

}

chdir("..");

if ((dirname = strrchr(path, '/')) != NULL)

dirname++;

else dirname = path;

/\* remove this empty directory \*/

if (rmdir(dirname))

fprintf(stderr, "Error: failed to remove the directory %s!\n", path);

else if (vflag)

printf("directory %s has been removed successfully.\n", path);

}

int myrm\_file(char \*path)

{

char ch, \*msg;

if (iflag)

{

msg = (char \*)malloc(strlen(path) + 25);

msg = strcat(strcpy(msg, path), ": delete it?(Y/N)");

if (ch = getyn(msg), ch == 'n' || ch == 'N')

return(0);

}

if (unlink(path))

{

fprintf(stderr, "Error: file %s can't be removed!\n", path);

return(-1);

}

if (vflag)

printf("file %s has been removed successfully.\n", path);

}

/\* get yes or no from the keyboard \*/

char getyn(char \*msg)

{

char first,ch;

/\* give out some tips \*/

printf("%s ",msg);

while (1)

{

first = ch = getchar();

while (first != '\n' && (ch = getchar()) != '\n')

;

if (first == 'y' || first == 'Y' || first == 'n' || first == 'N')

return(first);

printf("Please: make sure what you input is 'Y/y/N/n!'\n%s ",msg);

}

}

int usage(void)

{

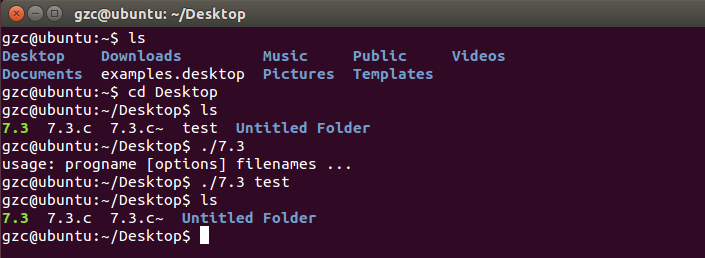
printf("usage: progname [options] filenames ...\n");

exit(0);

}

### 5、实验结果及分析

实验前后对比（test文件被删除）



# 九、多核多线程编程

## 实验9.1~9.5 实验报告

### 1.实验目的

在Linux环境下，编写多线程程序，分析以下几个因素对程序运行时间的影响：

* 程序并行化
* 线程数目
* 共享资源加锁
* CPU亲和
* cache优化

掌握多CPU、多核硬件环境下基本的多线程并行编程技术。

### 2.实验内容

*-9.1*观察实验平台物理cpu、CPU核和逻辑cpu的数目

*-9.2*单线程/进程串行 vs 2线程并行 vs 3线程加锁并行程序对比

*-9.3*单线程/进程串行 vs 2线程并行 vs 3线程加锁并行程序对比

*-9.4*针对 Cache 的优化

*-9.5*CPU亲和力对并行程序影响

### 3.实验原理

*-9.1* 通过Linux指令查看平台信息

*-9.2*通过数据分解的方法，将程序改造为单线程，双线程和加锁三线程三种方式，并对结果进行对比

*-9.3* 针对加锁的三线程方案，由于两个线程访问的是 apple 的不同元素，根本没有加锁的必要，所以修改 apple 的数据结构（删除读写锁代码），通过非加锁来提高性能。

*-9.4*处理器交换的最小单元是 cache 行，或称 cache 块。在多核体系中，对于非共享 cache 的架构来说，两个独立的 cache 在需要读取同一 cache 行时，会共享该 cache 行，如果在其中一个 cache 中，该 cache 行被写入，而在另一个 cache 中该 cache 行被读取，那么即使读写的地址不相交，也需要在这两个 cache 之间移动数据，这就被称为 cache 伪共享，导致必须在存储总线上来回传递这个cache行，这种现象被称为“乒乓效应”。同样地，当两个线程写入同一个 cache 的不同部分时，也会互相竞争该 cache 行，也就是写后写的问题。在 X86 机器上，某些处理器的一个 cache 行是64字节，具体可以参看 Intel 的参考手册。既然非加锁三线程方案的瓶颈在于 cache，那么让 apple 的两个成员 a 和 b 位于不同的 cache 行中，观察效率是否会有所提高。

*-9.5* CPU 亲和力可分为两大类：软亲和力和硬亲和力。

Linux 内核进程调度器天生就具有被称为 CPU 软亲和力（affinity）的特性，这意味着进程通常不会在处理器之间频繁迁移。这种状态正是我们希望的，因为进程迁移的频率小就意味着产生的负载小。但不代表不会进行小范围的迁移。CPU 硬亲和力是进程固定在某个处理器上运行，而不是在不同的处理器之间进行频繁的迁移。这样不仅改善了程序的性能，还提高了程序的可靠性。从以上不难看出，在某种程度上硬亲和力比软亲和力具有一定的优势。但在内核开发者不断的努力下，2.6内核软亲和力的缺陷已经比2.4的内核有了很大的改善。在双核机器上，针对两线程的方案，如果将计算 apple 的线程绑定到一个 CPU 上，将计算 orange 的线程绑定到另外一个 CPU 上，观察效率是否会有所提高。

*-K-Best 测量方法*在检测程序运行时间这个复杂问题上，将采用 Randal E.Bryant 和 David R. O’Hallaron 提出的 K 次最优测量方法。假设重复的执行一个程序，并纪录 K 次最快的时间，如果发现测量的误差 ε 很小，那么用测量的最快值表示过程的真正执行时间，称这种方法为“ K 次最优（K-Best）方法”，要求设置三个参数：

K: 要求在某个接近最快值范围内的测量值数量。

ε 测量值必须多大程度的接近，即测量值按照升序标号 V1, V2, V3, … , Vi, … ，同时必须满足（1+ ε）Vi >= Vk

M: 在结束测试之前，测量值的最大数量。

按照升序的方式维护一个 K个最快时间的数组，对于每一个新的测量值，如果比当前 K 处的值更快，则用最新的值替换数组中的元素 K ，然后再进行升序排序，持续不断的进行该过程，并满足误差标准，此时就称测量值已经收敛。如果 M 次后，不能满足误差标准，则称为不能收敛。

在接下来的所有试验中，采用 K=10，ε=2%，M=200 来获取程序运行时间，同时也对 K 次最优测量方法进行了改进，不是采用最小值来表示程序执行的时间，而是采用 K 次测量值的平均值来表示程序的真正运行时间。由于采用的误差 ε 比较大，在所有试验程序的时间收集过程中，均能收敛，但也能说明问题。

为了可移植性，采用 gettimeofday() 来获取系统时钟（system clock）时间，可以精确到微秒。

### **4.实验步骤**

（程序内部采用k-best算法，同时生成记录文件，文件在实验报告目录下可见）

*-9.1*

通过下列命令查看cpu相关信息

物理cpu数：

[XXXX@server ~]$ grep 'physical id' /proc/cpuinfo|sort|uniq|wc -l

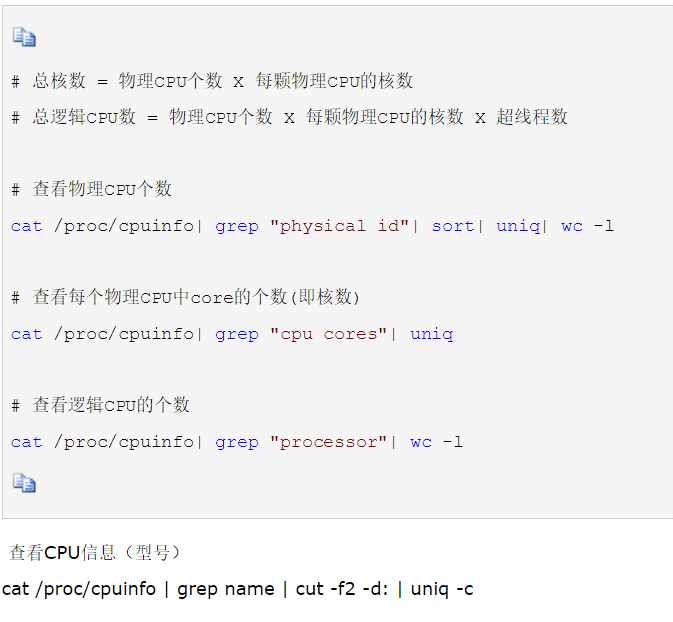
cpu核数：

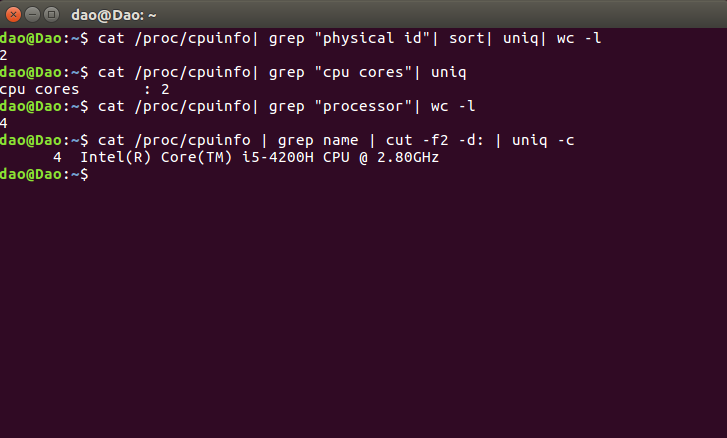
[XXXX@server ~]$ grep 'cpu cores' /proc/cpuinfo|uniq|awk -F ':' '{print $2}'

逻辑cpu：

[XXXX@server ~]$ cat /proc/cpuinfo| grep "processor"|wc -l

执行结果:





*-9.2*（代码内融入k-best算法）

步骤一：单线程代码

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #define ORANGE\_MAX\_VALUE 1000000  #define APPLE\_MAX\_VALUE 100000000  #define MEXCOND 1000000  #include<stdio.h>  #include<sys/time.h>  #include<unistd.h>  struct apple  {  unsigned long long a;  unsigned long long b;  };  struct orange  {  int a[ORANGE\_MAX\_VALUE];  int b[ORANGE\_MAX\_VALUE];  };  void InsertSort(long int a[], int n)  {  int i;  for(i= 1; i<n; i++){  if(a[i] < a[i-1]){ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入  int j= i-1;  int x = a[i]; //复制为哨兵，即存储待排序元素  a[i] = a[i-1]; //先后移一个元素  while(x < a[j]){ //查找在有序表的插入位置  a[j+1] = a[j];  j--; //元素后移  }  a[j+1] = x; //插入到正确位置  } //打印每趟排序的结果  }  }  int main(int argc, const char \*argv[])  {//insert code here...  FILE \*fp;  fp = fopen("8.2step1","w");  struct apple test;  test.a = 0;  test.b = 0;  struct orange test1;  int index;  int sum;  long int time\_record[200] = {0};  int count = 0;  struct timeval tv;  struct timeval tv2;  struct timezone tz;  long int fore\_time\_usec;  long int fore\_time\_sec;  long int after\_time\_usec;  long int after\_time\_sec;  long int during;  long int average;  int j;    average = 0;  while(count <= 199)  {  gettimeofday(&tv,NULL);  fore\_time\_usec = tv.tv\_usec;  fore\_time\_sec = tv.tv\_sec;  for(sum = 0; sum<APPLE\_MAX\_VALUE;sum++)  {  test.b+=sum;  test.a+=sum;  }  sum = 0;  for(index = 0;index<ORANGE\_MAX\_VALUE;index++)  {  test1.a[index] = 0;  test1.b[index] = 0;  sum+= test1.a[index]+test1.b[index];  }  gettimeofday(&tv2,NULL);  after\_time\_sec = tv2.tv\_sec;  after\_time\_usec = tv2.tv\_usec;  during = after\_time\_usec - fore\_time\_usec + (after\_time\_sec - fore\_time\_sec)\*1000000;    time\_record[count] = during;  fprintf(fp,"tz\_dsttime:%ld\n",during);  count ++;  }    InsertSort(time\_record,200);  for(j= 0; j<10; j++)  {  fprintf(fp,"%ld\n", time\_record[j]);  average = average + time\_record[j];  }  average = average / 10;  fprintf(fp,"average = %ld",average);  fclose(fp);  return 0;  } | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 946576 | 949795 | 949850 | 949899 | 951194 | 951286 | 951417 | 951748 | 951967 | 952659 |
| Average = 950639 ns | | | | | | | | | |

步骤二：双线程代码

|  |
| --- |
| #define ORANGE\_MAX\_VALUE 1000000  #define APPLE\_MAX\_VALUE 100000000  #define MEXCOND 1000000  #include <pthread.h>  #include<stdio.h>  #include<sys/time.h>  #include<unistd.h>  struct apple  {  unsigned long long a;  unsigned long long b;  };  struct orange  {  int a[ORANGE\_MAX\_VALUE];  int b[ORANGE\_MAX\_VALUE];  };  void InsertSort(long int a[], int n)  {  for(int i= 1; i<n; i++){  if(a[i] < a[i-1]){ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入  int j= i-1;  int x = a[i]; //复制为哨兵，即存储待排序元素  a[i] = a[i-1]; //先后移一个元素  while(x < a[j]){ //查找在有序表的插入位置  a[j+1] = a[j];  j--; //元素后移  }  a[j+1] = x; //插入到正确位置  } //打印每趟排序的结果  }  }  void\* add(void\* x)  {  int sum;  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)x)->a += sum;  ((struct apple \*)x)->b += sum;  }  //printf("used");  return NULL;  }    int main (int argc, const char \* argv[]) {  // insert code here...  FILE \*fp;  fp = fopen("8.2step2","w");  struct apple test;  struct orange test1;//={{0},{0}};  pthread\_t ThreadA;  int index;  long int time\_record[200] = {0};  int count = 0;  struct timeval tv;  struct timeval tv2;  // struct timezone tz;  // long int fore\_time\_usec;  // long int fore\_time\_sec;  // long int after\_time\_usec;  // long int after\_time\_sec;  long int during;  long int average;  int j;  int sum=0;    while(count <= 199)  {  gettimeofday(&tv,NULL);    pthread\_create(&ThreadA,NULL,add,&test);    for(index=0;index<ORANGE\_MAX\_VALUE;index++)  {  sum += test1.a[index]+test1.b[index];  }  pthread\_join(ThreadA,NULL);    gettimeofday(&tv2,NULL);  time\_record[count] = tv2.tv\_usec - tv.tv\_usec + (tv2.tv\_sec - tv.tv\_sec )\*1000000;    // if(during < 0)  // during = 0 - during;  fprintf(fp,"tz\_dsttime:%ld\n",time\_record[count]);    count ++;  }  InsertSort(time\_record,200);  for(j= 0; j<10; j++)  {  fprintf(fp,"%ld\n", time\_record[j]);  average = average + time\_record[j];  }  average = average / 10;  fprintf(fp,"average = %ld",average);  fclose(fp);  return 0;  } |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 850851 | 886853 | 1164608 | 1166008 | 1166137 | 1166990 | 1167461 | 1167930 | 1168310 | 1168622 |
| Average = 1107377ns | | | | | | | | | |

步骤三：三线程加锁代码

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #define ORANGE\_MAX\_VALUE 1000000  #define APPLE\_MAX\_VALUE 100000000  #define MEXCOND 1000000  #include<pthread.h>  #include<stdio.h>  #include<sys/time.h>  #include<unistd.h>  struct apple  {  unsigned long long a;  unsigned long long b;  pthread\_rwlock\_t rwLock;  };  struct orange  {  int a[ORANGE\_MAX\_VALUE];  int b[ORANGE\_MAX\_VALUE];  };  void InsertSort(long int a[], int n)  {  int i;  for(i= 1; i<n; i++){  if(a[i] < a[i-1]){ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入  int j= i-1;  int x = a[i]; //复制为哨兵，即存储待排序元素  a[i] = a[i-1]; //先后移一个元素  while(x < a[j]){ //查找在有序表的插入位置  a[j+1] = a[j];  j--; //元素后移  }  a[j+1] = x; //插入到正确位置  } //打印每趟排序的结果  }  }  void\* addx(void\* x)  {  int sum;  // pthread\_rwlock\_init(&((struct apple \*)x)->rwLock ,NULL);  pthread\_rwlock\_wrlock(&((struct apple \*)x)->rwLock);  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)x)->a += sum;  }  pthread\_rwlock\_unlock(&((struct apple \*)x)->rwLock);  // pthread\_rwlock\_destroy(&((struct apple \*)x)->rwLock);  return NULL;  }  void\* addy(void\* y)  {  int sum;  // pthread\_rwlock\_init(&((struct apple \*)y)->rwLock ,NULL);  pthread\_rwlock\_wrlock(&((struct apple \*)y)->rwLock);  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)y)->b += sum;  }  pthread\_rwlock\_unlock(&((struct apple \*)y)->rwLock);  // pthread\_rwlock\_destroy(&((struct apple \*)y)->rwLock);  return NULL;  }  int main (int argc, const char \* argv[]) {  // insert code here...  FILE \*fp;  fp = fopen("8.2step3","w");  struct apple test;  struct orange test1={{0},{0}};  pthread\_t ThreadA,ThreadB;  int index;  long int time\_record[200] = {0};  int count;  struct timeval tv;  struct timeval tv2;  int j;  int sum;  long int average;  average = 0;  sum = 0;  count = 0;    while(count <= 199)  {  gettimeofday(&tv,NULL);    pthread\_rwlock\_init(&test.rwLock,NULL);    pthread\_create(&ThreadA,NULL,addx,&test);  pthread\_create(&ThreadB,NULL,addy,&test);  for(index=0;index<ORANGE\_MAX\_VALUE;index++)  {  sum+=test1.a[index]+test1.b[index];  }    pthread\_join(ThreadA,NULL);  pthread\_join(ThreadB,NULL);    gettimeofday(&tv2,NULL);  time\_record[count] = tv2.tv\_usec - tv.tv\_usec + (tv2.tv\_sec - tv.tv\_sec)\*1000000;    // if(during < 0)  // during = 0 - during;  // time\_record[count] = during;    fprintf(fp,"tz\_dsttime:%ld\n",time\_record[count]);  // printf("%d\n",count);  count ++;  }  InsertSort(time\_record,200);  for(j= 0; j<10; j++)  {  fprintf(fp,"%ld\n", time\_record[j]);  average = average + time\_record[j];  }  average = average / 10;  fprintf(fp,"average = %ld",average);  fclose(fp);  return 0;  } | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1647291 | 1804195 | 1891160 | 1894830 | 1894941 | 1895059 | 1895080 | 1896100 | 1896609 | 1896861 |
| Average = 1861212ns | | | | | | | | | |

*-9.3*三线程不加锁代码

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #include<sys/time.h>  #include<stdio.h>  #include<stdlib.h>  #include<pthread.h>  #define ORANGE\_MAX\_VALUE 1000000  #define APPLE\_MAX\_VALUE 100000000  #define MSECOND 1000000  struct apple  {  unsigned long long a;  unsigned long long b;  pthread\_rwlock\_t rwlock;  };  struct orange  {  int a[ORANGE\_MAX\_VALUE];  int b[ORANGE\_MAX\_VALUE];  };  void\* addx(void\* x)  {  int sum;  //pthread\_rwlock\_wrlock(&((struct apple \*)x)->rwlock);  for( sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)x)->a+=sum;  }  // pthread\_rwlock\_unlock(&((struct apple \*)x)->rwlock);  return NULL;  }  void\* addy(void\* y)  {  int sum;  // pthread\_rwlock\_wrlock(&((struct apple \*)y)->rwlock);  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)y)->b+=sum;  }  // pthread\_rwlock\_unlock(&((struct apple \*)y)->rwlock);  return NULL;  }  void InsertSort(long int a[], int n)  {  int i;  for( i= 1; i<n; i++){  if(a[i] < a[i-1]){ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入  int j= i-1;  int x = a[i]; //复制为哨兵，即存储待排序元素  a[i] = a[i-1]; //先后移一个元素  while(x < a[j]){ //查找在有序表的插入位置  a[j+1] = a[j];  j--; //元素后移  }  a[j+1] = x; //插入到正确位置  } //打印每趟排序的结果  }  }  int main(int argc,const char \* argv[])  {  long int timeuse;//int  int index;  int sum;  int i=0;  struct apple test;  struct orange test1={{0},{0}};  pthread\_t ThreadA,ThreadB;  struct timeval tv1,tv2;  int count = 0;  long int time\_record[200] = {0};  FILE \*fp = fopen("os9.3.txt","w");  long int average;  int j;    while(count <= 199)  {  gettimeofday(&tv1,NULL);    pthread\_create(&ThreadA,NULL,addx,&test);  pthread\_create(&ThreadB,NULL,addy,&test);  for(index=0;index<ORANGE\_MAX\_VALUE;index++)  {  sum+=test1.a[index]+test1.b[index];  }  pthread\_join(ThreadA,NULL);  pthread\_join(ThreadB,NULL);  gettimeofday(&tv2,NULL);  time\_record[count]=1000000\*(tv2.tv\_sec-tv1.tv\_sec)+tv2.tv\_usec-tv1.tv\_usec;  fprintf(fp,"tz\_dsttime:%ld\n",time\_record[count]);  //printf("tz\_dsttime:%ld\n",time\_record[count]);  count++;  }  InsertSort(time\_record,200);  average = 0;  for(j= 0; j<10; j++)  {  fprintf(fp,"%ld\n", time\_record[j]);  //printf("%ld\n", time\_record[j]);  average = average + time\_record[j];  }  average = average / 10;  fprintf(fp,"average = %ld",average);  fclose(fp);  return 0;  } | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2203208 | 2261895 | 2289107 | 2289348 | 2297267 | 2297751 | 2307596 | 2309388 | 2316954 | 2317591 |
| Average = 2289010 ns | | | | | | | | | |

*-9.4*

1）针对不加锁三线程的cache优化

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #define ORANGE\_MAX\_VALUE 1000000  #define APPLE\_MAX\_VALUE 100000000  #define MEXCOND 1000000  #include <pthread.h>  #include<stdio.h>  #include<sys/time.h>  #include<unistd.h>  struct apple  {  unsigned long long a;  char c[128]; /\*32,64,128\*/  unsigned long long b;  pthread\_rwlock\_t rwLock;  };  struct orange  {  int a[ORANGE\_MAX\_VALUE];  int b[ORANGE\_MAX\_VALUE];  };  void InsertSort(long int a[], int n)  {  int i;  for(i= 1; i<n; i++){  if(a[i] < a[i-1]){ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入  int j= i-1;  int x = a[i]; //复制为哨兵，即存储待排序元素  a[i] = a[i-1]; //先后移一个元素  while(x < a[j]){ //查找在有序表的插入位置  a[j+1] = a[j];  j--; //元素后移  }  a[j+1] = x; //插入到正确位置  } //打印每趟排序的结果  }  }  void\* addx(void\* x)  {  int sum;  // pthread\_rwlock\_init(&((struct apple \*)x)->rwLock ,NULL);  pthread\_rwlock\_wrlock(&((struct apple \*)x)->rwLock);  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)x)->a += sum;  }  pthread\_rwlock\_unlock(&((struct apple \*)x)->rwLock);  // pthread\_rwlock\_destroy(&((struct apple \*)x)->rwLock);  return NULL;  }  void\* addy(void\* y)  {  int sum;  // pthread\_rwlock\_init(&((struct apple \*)y)->rwLock ,NULL);  pthread\_rwlock\_wrlock(&((struct apple \*)y)->rwLock);  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)y)->b += sum;  }  pthread\_rwlock\_unlock(&((struct apple \*)y)->rwLock);  // pthread\_rwlock\_destroy(&((struct apple \*)y)->rwLock);  return NULL;  }  int main (int argc, const char \* argv[]) {  // insert code here...  FILE \*fp;  fp = fopen("8.4cache\_lock\_file","w");  struct apple test;  struct orange test1={{0},{0}};  pthread\_t ThreadA,ThreadB;  int index;  long int time\_record[200] = {0};  int count;  struct timeval tv;  struct timeval tv2;  int j;  int sum;  long int average;  average = 0;  sum = 0;  count = 0;    while(count <= 199)  {  gettimeofday(&tv,NULL);    pthread\_rwlock\_init(&test.rwLock,NULL);    pthread\_create(&ThreadA,NULL,addx,&test);  pthread\_create(&ThreadB,NULL,addy,&test);  for(index=0;index<ORANGE\_MAX\_VALUE;index++)  {  sum+=test1.a[index]+test1.b[index];  }    pthread\_join(ThreadA,NULL);  pthread\_join(ThreadB,NULL);    gettimeofday(&tv2,NULL);  time\_record[count] = tv2.tv\_usec - tv.tv\_usec + (tv2.tv\_sec - tv.tv\_sec)\*1000000;    // if(during < 0)  // during = 0 - during;  // time\_record[count] = during;    fprintf(fp,"tz\_dsttime:%ld\n",time\_record[count]);  // printf("%d\n",count);  count ++;  }  InsertSort(time\_record,200);  for(j= 0; j<10; j++)  {  fprintf(fp,"%ld\n", time\_record[j]);  average = average + time\_record[j];  }  average = average / 10;  fprintf(fp,"average = %ld",average);  fclose(fp);  return 0;  } | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 728573 | 760398 | 772413 | 784171 | 786204 | 807014 | 811395 | 824517 | 843766 | 850171 |
| Average = 796862 ns | | | | | | | | | |

2）针对加锁三线程的cache优化

|  |
| --- |
| #define ORANGE\_MAX\_VALUE 1000000  #define APPLE\_MAX\_VALUE 100000000  #define MEXCOND 1000000  #include <pthread.h>  #include<stdio.h>  #include<sys/time.h>  #include<unistd.h>  struct apple  {  unsigned long long a;  char c[128]; /\*32,64,128\*/  unsigned long long b;  // pthread\_rwlock\_t rwLock;  };  struct orange  {  int a[ORANGE\_MAX\_VALUE];  int b[ORANGE\_MAX\_VALUE];  };  void InsertSort(long int a[], int n)  {  int i;  for(i= 1; i<n; i++){  if(a[i] < a[i-1]){ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入  int j= i-1;  int x = a[i]; //复制为哨兵，即存储待排序元素  a[i] = a[i-1]; //先后移一个元素  while(x < a[j]){ //查找在有序表的插入位置  a[j+1] = a[j];  j--; //元素后移  }  a[j+1] = x; //插入到正确位置  } //打印每趟排序的结果  }  }  void\* addx(void\* x)  {  int sum;  // pthread\_rwlock\_init(&((struct apple \*)x)->rwLock ,NULL);  // pthread\_rwlock\_wrlock(&((struct apple \*)x)->rwLock);  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)x)->a += sum;  }  // pthread\_rwlock\_unlock(&((struct apple \*)x)->rwLock);  // pthread\_rwlock\_destroy(&((struct apple \*)x)->rwLock);  return NULL;  }  void\* addy(void\* y)  {  int sum;  // pthread\_rwlock\_init(&((struct apple \*)y)->rwLock ,NULL);  // pthread\_rwlock\_wrlock(&((struct apple \*)y)->rwLock);  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)y)->b += sum;  }  // pthread\_rwlock\_unlock(&((struct apple \*)y)->rwLock);  // pthread\_rwlock\_destroy(&((struct apple \*)y)->rwLock);  return NULL;  }  int main (int argc, const char \* argv[]) {  // insert code here...  FILE \*fp;  fp = fopen("8.4cache\_nolock\_file","w");  struct apple test;  struct orange test1={{0},{0}};  pthread\_t ThreadA,ThreadB;  int index;  long int time\_record[200] = {0};  int count;  struct timeval tv;  struct timeval tv2;  int j;  int sum;  long int average;  average = 0;  sum = 0;  count = 0;    while(count <= 199)  {  gettimeofday(&tv,NULL);    // pthread\_rwlock\_init(&test.rwLock,NULL);    pthread\_create(&ThreadA,NULL,addx,&test);  pthread\_create(&ThreadB,NULL,addy,&test);  for(index=0;index<ORANGE\_MAX\_VALUE;index++)  {  sum+=test1.a[index]+test1.b[index];  }    pthread\_join(ThreadA,NULL);  pthread\_join(ThreadB,NULL);    gettimeofday(&tv2,NULL);  time\_record[count] = tv2.tv\_usec - tv.tv\_usec + (tv2.tv\_sec - tv.tv\_sec)\*1000000;    // if(during < 0)  // during = 0 - during;  // time\_record[count] = during;    fprintf(fp,"tz\_dsttime:%ld\n",time\_record[count]);  // printf("%d\n",count);  count ++;  }  InsertSort(time\_record,200);  for(j= 0; j<10; j++)  {  fprintf(fp,"%ld\n", time\_record[j]);  average = average + time\_record[j];  }  average = average / 10;  fprintf(fp,"average = %ld",average);  fclose(fp);  return 0;  } |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1871314 | 1873843 | 1874271 | 1878498 | 1879018 | 1879966 | 1879971 | 1880138 | 1880707 | 1883921 |
| Average = 1878164 ns | | | | | | | | | |

*-9.5*

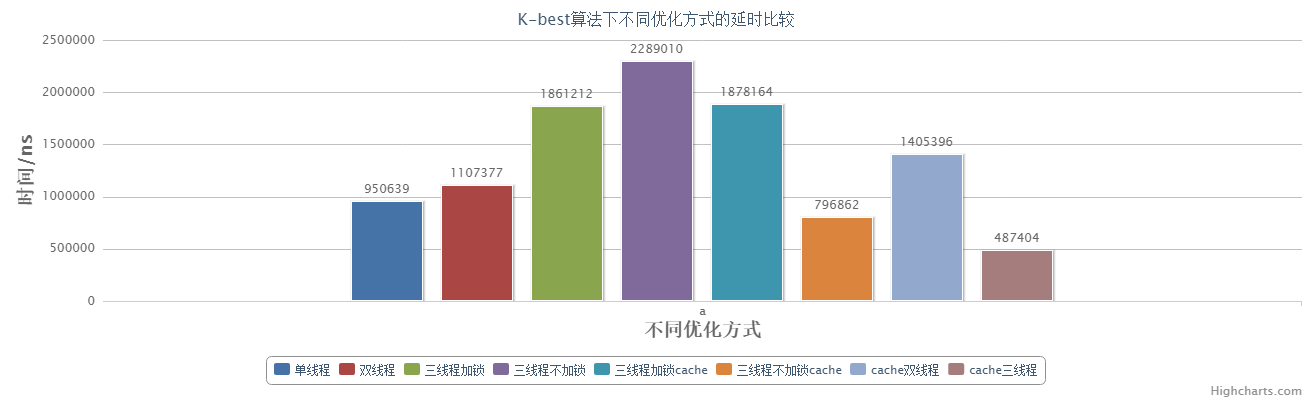
1）cache双线程

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #define ORANGE\_MAX\_VALUE 1000000  #define APPLE\_MAX\_VALUE 100000000  #define MEXCOND 1000000  #define \_GNU\_SOURCE  #include <pthread.h>  #include<stdio.h>  #include<sys/time.h>  #include<unistd.h>  #include <sched.h>  #include <stdlib.h>  #include <sys/syscall.h>  #define gettid() syscall(\_\_NR\_gettid)  struct apple  {  unsigned long long a;  unsigned long long b;  };    struct orange  {  int a[ORANGE\_MAX\_VALUE];  int b[ORANGE\_MAX\_VALUE];  };  int set\_cpu(int i)  {  cpu\_set\_t mask;  CPU\_ZERO(&mask);  int cpu\_nums = sysconf(\_SC\_NPROCESSORS\_CONF);  if(2 <= cpu\_nums)  {  CPU\_SET(i,&mask);  if(-1 == sched\_setaffinity(gettid(),sizeof(&mask),&mask))  {  return -1;  }  }  return 0;  }    void\* add(void\* x)  {  int sum=0;  if(-1 == set\_cpu(1))  {  return NULL;  }  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)x)->a += sum;  ((struct apple \*)x)->b += sum;  }  return NULL;  }  void InsertSort(long int a[], int n)  {  int i;  for(i= 1; i<n; i++){  if(a[i] < a[i-1]){  int j= i-1;  int x = a[i];  a[i] = a[i-1];  while(x < a[j]){  a[j+1] = a[j];  j--;  }  a[j+1] = x;  }  }  }  int main (int argc, const char \* argv[]) {  // insert code here...  struct apple test;  struct orange test1={{0},{0}};  //int sum;  int index;  pthread\_t ThreadA;  long int time\_record[200] = {0};  int count;  struct timeval tv;  struct timeval tv2;  int j;  int sum;  long int average;  FILE \*fp;  fp = fopen("8.5\_output.txt","w");  average = 0;  sum = 0;  count = 0;  index = 0;  average = 0;    if(-1 == set\_cpu(0))  {  return -1;  }  while(count <= 199)  {  gettimeofday(&tv,NULL);      pthread\_create(&ThreadA,NULL,add,&test);    for(index=0;index<ORANGE\_MAX\_VALUE;index++)  {  sum+=test1.a[index]+test1.b[index];  }    pthread\_join(ThreadA,NULL);    gettimeofday(&tv2,NULL);  time\_record[count] = tv2.tv\_usec - tv.tv\_usec + (tv2.tv\_sec - tv.tv\_sec)\*1000000;      fprintf(fp,"tz\_dsttime:%ld\n",time\_record[count]);  count ++;  }  InsertSort(time\_record,200);  for(j= 0; j<10; j++)  {  fprintf(fp,"%ld\n", time\_record[j]);  average = average + time\_record[j];  }  average = average / 10;  fprintf(fp,"average = %ld",average);    return 0;  } | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 954574 | 1327533 | 1328074 | 1485626 | 1488565 | 1489662 | 1492433 | 1492920 | 1495086 | 1499492 |
| Average = 1405396 | | | | | | | | | |

2）cache三线程

|  |
| --- |
| #define ORANGE\_MAX\_VALUE 1000000  #define APPLE\_MAX\_VALUE 100000000  #define MEXCOND 1000000  #define \_GNU\_SOURCE  #include <pthread.h>  #include<stdio.h>  #include<sys/time.h>  #include<unistd.h>  #include <sched.h>  #include <stdlib.h>  #include <sys/syscall.h>  #define gettid() syscall(\_\_NR\_gettid)  struct apple  {  unsigned long long a;  char c[128]; /\*32,64,128\*/  unsigned long long b;  };  struct orange  {  int a[ORANGE\_MAX\_VALUE];  int b[ORANGE\_MAX\_VALUE];  };  int set\_cpu(int i)  {  cpu\_set\_t mask;  CPU\_ZERO(&mask);  int cpu\_nums = sysconf(\_SC\_NPROCESSORS\_CONF);  if(2 <= cpu\_nums)  {  CPU\_SET(i,&mask);  if(-1 == sched\_setaffinity(gettid(),sizeof(&mask),&mask))  {  return -1;  }  }  return 0;  }  void\* addx(void\* x)  {  int sum=0;  if(-1 == set\_cpu(1))  {  return NULL;  }  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)x)->a += sum;  }  return NULL;  }  void\* addy(void\* y)  {  int sum=0;  if(-1 == set\_cpu(0))  {  return NULL;  }  for(sum=0;sum<APPLE\_MAX\_VALUE;sum++)  {  ((struct apple \*)y)->b += sum;  }  return NULL;  }  void InsertSort(long int a[], int n)  {  int i;  for(i= 1; i<n; i++){  if(a[i] < a[i-1]){  int j= i-1;  int x = a[i];  a[i] = a[i-1];  while(x < a[j]){  a[j+1] = a[j];  j--;  }  a[j+1] = x;  }  }  }  int main (int argc, const char \* argv[]) {  // insert code here...  struct apple test;  struct orange test1={{0},{0}};  pthread\_t ThreadA,ThreadB;  int index;  long int time\_record[200] = {0};  int count;  FILE \*fp;  fp = fopen("8.5\_2\_output.txt","w");  struct timeval tv;  struct timeval tv2;  int j;  int sum;  long int average;  average = 0;  sum = 0;  count = 0;  index = 0;  while(count <= 199)  {  if(-1 == set\_cpu(0))  {  return -1;  }  gettimeofday(&tv,NULL);  pthread\_create(&ThreadA,NULL,addx,&test);  pthread\_create(&ThreadB,NULL,addy,&test);  for(index=0;index<ORANGE\_MAX\_VALUE;index++)  {  sum+=test1.a[index]+test1.b[index];  }  pthread\_join(ThreadA,NULL);  pthread\_join(ThreadB,NULL);  gettimeofday(&tv2,NULL);  time\_record[count] = tv2.tv\_usec - tv.tv\_usec + (tv2.tv\_sec - tv.tv\_sec)\*1000000;  fprintf(fp,"tz\_dsttime:%ld\n",time\_record[count]);  count ++;  }  InsertSort(time\_record,200);  for(j= 0; j<10; j++)  {  fprintf(fp,"%ld\n", time\_record[j]);  average = average + time\_record[j];  }  average = average / 10;  fprintf(fp,"average = %ld",average);  return 0;  } |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 408118 | 427570 | 434523 | 459738 | 463939 | 496435 | 525271 | 535243 | 550899 | 572310 |
| Average = 487404ns | | | | | | | | | |

****

### 5.实验总结

实验过程中碰到的困难有几个，第一个是由于局部变量sum被我声明为全局变量，所以实验结果很不稳定，甚至可能出现duration为零的情况，最后经过分析得知，由于不同进程都会对此变量进行操作，而且此变量没有上锁，所以会造成冲突导致的异常，以至于有些进程还没有进行操作就结束。第二个困难是由于机器的问题，众所周知，不同机器处理数据的能力不同，所表现出cpu的能力也不同，因此延时也会差距很大，我先后试了五台电脑，最终找到一台和实验指导书中性能相近的电脑，跑出了以上表格中的结果。第三就是k-best算法的理解，猛一看上去k-best算法非常复杂，其实在试验所给环境下实现起来非常容易，第一是默认收敛，因此收敛条件不需要判断，第二是计算最快十个的平均值即可，所以我们将程序跑两百次，将其升序排列，并把其中最快的十次计算平均值我们就得到了我们想要的结果。

我们对实验的改进有以下几点，第一是把k-best算法写入代码中，使其自动生成200次数据并排序计算平均值，第二是把实验的结果写入文件中，因此可以记录下正确的实验结果，同时实验初期检查代码变得更为方便，总之在此试验中我学到了很多对cpu进行优化的方法，也学习了一部分linux基本操作，非常满足开心。