北 京 邮 电 大 学

计算机学院

《操作系统》课程实验

Linux 操作系统内核
实 验 指 导 书

2019 年 10 月

目　录

1. 实验大纲

1.1 实验目的

在学习《操作系统》课程内容同时，以开放式源代码操作系统 Linux 为实验平台，同步完成 Linux 操作系统内核的代码分析和修改等 7 组基本课程实验。通过实验，熟悉 Linux 系统使用方法，掌握 Linux 内核系统结构，了解 Linux 进程管理、存储管理、设备管理、文件系统等资源管理功能的实现机理和典型算法。初步掌握运用内核开发环境对内核进行修改完善的能力。

通过本课程实验，使得学生熟悉 Linux 操作系统相关技术，并进一步巩固课堂所学有关操作系统人机界面和资源管理得相关知识；并通过 Linux 源代码分析和简单编程，培养学生对实际操作系统的基本系统分析能力。

1.2实验内容说明

Linux 基本实验由以下 8 组实验组成。

1.2.1 第1组 系统安装

实验 1.1 Linux 系统安装

从 CD-ROM 安装 Red Hat Linux 操作系统，如 Red Hat Linux7.2，建立后续各个实验的运行环境。

实验 1.2 虚拟机安装

在配备 Windows 操作系统 Host 机上，安装虚拟机软件 Virtual Box、Virtual PC for Windows、VMware For Windows 等，进行 BIOS 设定， 对硬盘进行分区和格式化，安装 Linux 操作系统，以便在一台机器上模拟出多种操作系统运行环境。

1.2.2 第2组Linux内核

实验 2.1 观察 Linux 行为

学习 linux 内核、进程、存储和其他资源的一些重要特性。通过使用/proc 文件系统接口，编写一个程序检查反映机器平衡负载、进程资源利用率方面的各种内核值，学会使用/proc 文件系统这种内核状态检查机制。

实验 2.2 内核定时器

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。

实验 2.3 内核模块

模块是 Linux 系统的的一种特有机制，可用于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。例如， 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。

实验 2.4 系统调用

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。

1.2.3 第3组 进程管理

实验 3.1 进程行为观察

1. 在 Linux 下，分别用 snice、skill、top 等命令和/proc 中的有关目录、文件观察系统中进程运行情况和 CPU 工作情况。
2. 在 Linux 下，用 ptrace()、gdb 跟踪一个进程的运行情况，用 strace 工具跟踪 fork() 过程，用 ltrace 工具跟踪 execl() 过程。观察并分析跟踪信息。

实验 3.2 Shell 编程（2 选 1）

1. 以超级用户身份编程，计算某一时段中所有程序平均运行时间。
2. 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。

1.2.4 第4组 存储管理

实验 4.1 观察实验

1. 在 Linux 下，使用 gdb 程序观察一个程序文件的内容和结构。启动该程序执行，再用 GDB 观察其内存映象的内容和结构。
2. 在 Linux 下，用 free 和 vmstat 命令观察内存使用情况。
3. 在 Linux 下，查看/proc 与内存管理相关的文件，并解释显示结果。
4. 在 Linux 下，用 malloc()函数实现 cat 或 copy 命令。

实验 4.2 虚拟存储器管理

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。

1.2.5 第5组 进程通信

实验 5.1 观察实验

在 Linux 下，用 ipcs()命令观察进程通信情况。

实验 5.2 进程同步实验

在学习 **linux** 内核的同步机制基础上，深入分析各种同步机制的实现方案，设计和编写一套同步原语。

1.2.6 第6组 I/O设备管理

实验 6.1. 观察实验

1．stat 命令查看机器上硬盘特别文件的 I 节点内容。

2．在 Linux 下，查看/proc 与内存管理相关的文件，解释显示结果。

实验 6.2 编程实验

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。

1.2.7 第7组 文件系统管理

实验 7.1 代码分析

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

实验 7.2 编程实验 1（二选一）

在 Linux 环境下，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

实验 7.3 编程实验 2（二选一）

在 Linux 环境下，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。

1.2.8 第8组 多核多线程编程（全都做）

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

实验内容包括：

实验 8.1 观察实验平台物理 cpu、CPU 核和逻辑 cpu 的数目

实验 8.2. 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

实验 8.3. 3 线程加锁 vs 3 线程不加锁 对比

实验 8.4. 针对 Cache 的优化

实验 8.5. CPU 亲和力对并行程序影响

8 种实现方案运行时间对比总结

!1.3实验要求

- 学生以小组为单位，每组人数不超过 3 人。
- 以上 8 组实验中，除了第一组系统安装实验外，第 2 到第 8 组实验中，每组至少完成 1 个实验，完成的编程实验总数不少于 5 个，每个小组成员至少完成 2 个实验。
- “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”、“多核多线程编程（全部 8 个）”为必做实验。
- 实验完成后提交课程实验报告文档，并验收程序代码和上机演示。
- **实验验收分为 2 次：第一次在期中考试后，第二次在期末考试之前**
- 课程实验报告要求

对于编程实验，报告应包括：题目，实验目的、实验内容、实验设计原理、实验步骤、实验结果及分析和人员任务分配等。

2. 系统安装实验

2.1 实验1.1 Linux系统安装

1、实验目的

从 CD-ROM 安装 Red Hat Linux 操作系统，如 Red Hat Linux7.2，建立后续各个实验的运行环境。

2、实验内容（以Red Hat Linux7.2为例）

Red Hat Linux7.2 安装光盘共有两张，第一张可直接从光盘启动，包含大部分的软件包和一些安装工具。第二张光盘包含许多附加软件包。以下为安装过程和注意事项。

（1）启动安装程序。用 Linux 的第一张光盘，从光驱引导启动程序，进入启动界面，显示提示符 ”boot: ”，选择图形模式进行安装。

（2）选择安装界面的使用语言

（3）选择默认的键盘设置

（4）选择默认的鼠标设置

（5）选择安装类型。Red Hat Linux 提供了个人桌面、工作站、服务器和定制等多种安装类型。本实验选择个人桌面或定制方式。

（6）进行硬盘分区。Red Hat Linux 采用了“装载”的处理方式，将 1 个分区和 1 个目录联系起来，每个分区都是整个文件系统的一部分。

Linux 最少需要 2 个分区：Linux native(文件)分区、Linux Swap（交换）分区。前者用于存放 Linux 系统文件，只能用 EXT2 分区类型，在分区时应将载入点设置为“/”目录; 后者用作交换空间，将主存内暂时不用的数据缓存起来。建议采用如下分区方案

- SWAP 分区
  SWAP 分区大小至少等于实际系统内存容量，一般可取为内存的 2 倍。
- /boot 分区
  包含操作系统内核和启动时所用文件。建立单独的/boot 分区后，即使主要根分区出了问题，系统仍然能够启动。此分区大小约在 50MB-100MB 之间
- /分区
  根目录挂载位置。系统运行所需要的其它文件都在该分区，大小约在 1.7GB 到 5GB 之间

初次安装系统时，最好选择自动安装方式。如果安装者对系统比较熟悉，可以用系统配置的营盘管理工具 Disk Druid 来订制所需分区。

（7）将文件系统设置为 EXT2

（8）配置引导装载程序。选择 LILO 作为引导安装程序。LILO 可以安装在第一硬盘的主引导区（MBR）或 Linux 分区的引导扇区。如果使用 LILO 来做双启动，须选择前者；如果用 Linux 启动软盘或其它系统引导器引导 Linux，选择后者，即将 LILO 安装在 Linux 分区的引导扇区。

（9）网络和防火墙配置

（10）选择默认的语言及其他语言支持

（11）时区配置

（12）设置 root 配置

（13）选择软件包组
(14) 筹建引导盘
(15) 配置显卡
(16) 进行安装

2.2 实验1.2 虚拟机VM软件安装

1、实验目的

在配备 Windows 操作系统 Host 机上，安装虚拟机软件， 进行 BIOS 设定， 对硬盘进行分区和格式化，安装 Linux 操作系统，以便在一台机器上模拟出多种操作系统运行环境。

可选的虚拟机软件有：**VirtualBox**，**VMware For Windows**，Virtual PC for Windows

通过本实验，进一步掌握课堂上所讲的虚拟机的概念。

2、实验内容

实验前的准备：

1、 获取安装介质

2、 熟悉虚拟机的操作

安装步骤（以 Vmware 为例）：

1、 虚拟机软件的安装及设置

（1）、安装 VMware,输入虚拟机序列号

（2）、创建一个新的虚拟机。

第一步：“File”->“New Virtual Machine”->“Custom”->“Next”->操作系统那栏选 Linux->“Next”。

第二步：设置虚拟机名以及配置文件 ->“Next”。

第三步：设置虚拟机的内存大小。

第四步：网络连接，使用默认设置->“Next”。

第五步：磁盘设定，如果你不是想让红旗 Linux 桌面 4.0 终生运行在虚拟机里，请选 “Use a physical disk”，让虚拟机与当前系统共用同一硬盘，而不是虚拟出一个硬盘。有一定风险，但是只要不胡乱操作，风险不大->“Next”。

第六步：指定要使用的硬盘->“Next”，设置配置文件的位置->“Finish”，忽略那个风险提示。

（3）、光驱软驱默认情况下也是和当前系统共用的，使用 iso 文件引导虚拟机，则“Edit virtual machine settings”，在左侧列表中选“DVD/CD-ROM”那项，再在右侧选“Use ISO image:”，指定安装红旗 Linux 桌面 4.1 的 iso 文件。

（4）、虚拟机默认不是从光盘引导的，要在它的 BIOS 里改，得先“Start this virtual machine”。如果出现“Do not forget to ...”的提示框，直接点“OK”。待 VMware 窗口中一大块变黑的时候，赶快用鼠标点那块黑，那块黑是虚拟机的屏幕。现在你的鼠标和键盘就转为控制虚拟机了。注意虚拟机屏幕下方的进度条，在走完之前，按“F2”键进行 BIOSs 设定。

（5）、用键盘的左右箭头键选中“Boot”标签，用上下箭头键选中“CD-ROM”。同时用“Shift”键和“+”键，把“CD-ROM”拎到顶上。用键盘的左右箭头键选中“Exit”标签，用上下箭头键选

中“Exit Saving Changes”，回车->“YES”，等待虚拟机重新启动。同时按“Ctrl”和“Alt”键，鼠标和键盘就从虚拟机中解脱出来了。看到进度条的时候按 VMware 窗口左上方的红方块，停掉虚拟机。

3. Linux 内核实验

3.1 实验2.1 观察Linux行为

1、实验目的

学习 linux 内核、进程、存储和其他资源的一些重要特性。通过使用/proc 文件系统接口，编写一个程序检查反映机器平衡负载、进程资源利用率方面的各种内核值，学会使用/proc 文件系统这种内核状态检查机制。

2、实验内容

编写一个默认版本的程序通过检查内核状态报告 Linux 内核行为。程序应该在 stdout 上打印以下值：

1，CPU 类型和型号；
2，所使用的 Linux 内核版本；
3，从系统最后一次启动以来已经经历了多长时间（天，小时和分钟）；
4，总共有多少 CPU 时间执行在用户态，系统态，空闲态；
5，配置内存数量；当前可用内存数，磁盘读写请求数；
6，内核上下文转换数；
7，系统启动到目前创建了多少进程。

设计思路：

首先得到命令行，再解释命令行，然后到**/proc** 查找文件，最后执行命令。

3、程序源代码清单（参考）

```
#include      <stdio.h>
#include      <sys/time.h>

int main( int argc, char *argv[])
{
   char repTypeName[16];
   char c1, c2, ch;
   int interval, duration;
   char *lineBuf;
```

```
    int LB_SIZE;
    FILE *thisProcFile;
    char *now;
    int iteration;

  //决定报告类型
  strcpy(repTypeName, "Standard");
    if (argc < 1)
    {
       sscanf(argv[1],"%c%c",&c1,&c2);
       if (c1 != '-')
       {
          fprintf(stderr, "usage:ksamp [-s][-1 int dur]\n");
          exit(1);
       }
       if (c2 == 's')
       {
          strcpy(repTypeName, "Short");
       }
       if (c2 == "1")
       {
          strcpy(repTypeName, "Long");
          interval = atoi(argv[2]);
          duration = atoi(argv[3]);
       }
    }

 //得到当前时间
 /*    gettimeofday(&now, NULL);       */
    printf("Status report type %s at %s\n", repTypeName, ctime(&(now)));
    thisProcFile = fopen("/proc/sys/kernel/hostname","r");    //打开文件
    fgets(lineBuf, LB_SIZE + 1, thisProcFile);                    //读取文件
    printf("Machine hostname: %s", lineBuf);                  //打印文件
    fclose(thisProcFile);                                             //关闭文件

 //内核版本
    thisProcFile = fopen("/proc/version","r");
    fgets(lineBuf,LB_SIZE + 1,thisProcFile);
    printf("The Version: %s\n", lineBuf);
    fclose(thisProcFile);

 //CPU 类型和型号
    thisProcFile = fopen("/proc/cpuinfo","r");
    fgets(lineBuf,LB_SIZE + 1,thisProcFile);
```

```
    printf("The CPU: %s\n", lineBuf);
    fclose(thisProcFile);

//当前时间
    thisProcFile = fopen("/proc/uptime","r");
    fgets(lineBuf,LB_SIZE + 1,thisProcFile);
    printf("The Running Time: %s\n", lineBuf);
    fclose(thisProcFile);

//内存情况
    printf("\bThe MemInfo:\n");
    thisProcFile = fopen("/proc/meminfo","r");
    while (! feof(thisProcFile)) {
        putchar(fgetc(thisProcFile));
    }
    fclose(thisProcFile);

    //  当前状态
    printf("\bThe Status:\n");
    thisProcFile = fopen("/proc/stat","r");
    while (! feof(thisProcFile)) {
        putchar(fgetc(thisProcFile));
    }
    fclose(thisProcFile);

iteration = 0;
interval = 2;
duration = 60;
    while(iteration < duration)
    {
        sleep(interval);
    thisProcFile = fopen("/proc/loadavg","r");
    while (! feof(thisProcFile)) {
        putchar(fgetc(thisProcFile));
    }
    fclose(thisProcFile);
        iteration += interval;
    }
    return(0);
}
```

3.2 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

A、 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。

3、程序源代码清单（参考）

A.3、shell 程序源代码清单

```
/****part1.c*****/
//part 1
#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);
}
```

B、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。

B.3、shell 程序源代码清单

```
/************part2.c**********/
//part2
#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);
    signal(SIGVTALRM,sighandle);
    signal(SIGPROF,sighandle);

    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);
    setitimer(ITIMER_VIRTUAL,&v,NULL);
    setitimer(ITIMER_PROF,&v,NULL);
        for(j= 0;j<1000;j++){
        for(i= 0;i<500;i++){printf("*********\r");fflush(stdout);}
    }
    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*10000;
    t2 = (10 - virtt.it_value.tv_sec)*1000 + (1000000 - virtt.it_value.tv_usec)/1000 +
vtsecond*10000;
    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;
        }
}
```

C、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。

C.3、shell 程序源代码清单

```
/**************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,moremse
c);

            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,mor
```

```
                emsec);
                   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;
        }
}
```

3.3 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

- **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。read_func()负责产生/proc/clock 被读时的动作。

- **编译内核模块 Makefile 文件**

```
Makefile
CC＝gcc
MODCFLAGS := -Wall -D__KERNEL__ -DMODULE –DLINUX
clock.o :clock.c /usr/include/linux//version.h
$(CC) $( MODCFLAGS) –c clock.c
    echo insmod clock.o to turn it on
    echo rmmod clock to turn ig off
    echo
```

编译完成之后生成 clock.o 模块文件。

- **内核模块源代码 clock.c**

```
#define MODULE
#define MODULE_VERSION          “1.0”
#define MODULE_NAME             “clock”

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>

int proc_read_clock(char* page, char** start, off_t off,
                                        int count,int* eof,void* data)
{
    int len;
    struct timeval xtime;
    do_gettimeofday(&xtime);
    len = sprintf(page,"%d %d\n",xtime.tv_sec,xtime.tv_usec);
    printk("clock: read_func()\n");
    return len;
}
// proc_dir_entry 数据结构
struct proc_dir_entry* proc_my_clock;

int init_module()
{
    printk("clock: init_module()\n");
     my_clock=create_proc_read_entry("clock",0,&proc_root, proc_read_clock, 0);
     printk(KERN_INFO“%s %s has initialized.\n”,
                            MODULE_NAME,MODULE_VERSION);
    return 0;
}

void cleanup_module()
```

```
{
        printk("clock: cleanup_module()\n");
         remove_proc_entry(proc_my_clock->name, &proc_root);
         printk(KERN_INFO“%s %s has removed.\n”,
                                        MODULE_NAME,MODULE_VERSION);
}

MODULE_DESCRIPTION(“clock module for gettimeofday of proc.”);
EXPORT_NO_SYMBOLS;
```

- **加载内核模块**

在系统 root 用户下运行用户态模块命令装载内核模块

#insmod clock.o

- **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",O_RDONLY);
        len = read(infile,procClockTime,256);
        close(infile);

        procClockTime[len] = '\0';

printf("SystemTime is %d %d\nProcClockTime is %s\n",
        getSystemTime.tv_sec ,
        getSystemTime.tv_usec,
        procClockTime);

        sleep(1);
}
```

- **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 clock.o

3.4 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. 添加新调用的源代码

在**/usr/src/linux-2.4.7-10/kernel/sys.c** 中添加相应的调用代码

```
asmlinkage int sys_mycall(struct timeval *tv)
{ struct timeval ktv;
  MOD_INC_USE_COUNT;

  do_gettimeofday(&ktv);
  if (copy_to_user(tv,&ktv,sizeof(ktv))){
    MOD_DEC_USE_COUNT;
    return -EFAULT;
  }

  MOD_DEC_USE_COUNT;

  return 0;
}
```

2. 连接系统调用

a、修改**/usr/src/linux-2.4.7-10/include/asm-i386/unistd.h，**

在系统调用列表后面相应位置添加一行

```
#define _NR_mycall 222
```

新增加的调用号位 222

b、修改**/usr/src/linux-2.4.7-10/arch/i386/kernel/entry.S**

在 **sys_call_table[]**清单最后添加一行

```
.long SYMBOL_NAME(sys_mycall)
```

3. 重建新的 Linux 内核

```
cd /usr/src/linux-2.4.7-10/
make mrproper
make oldconfig
made dep
    make clean
```

make bzImage
make modules
make modules_install
make install

4. 重建引导信息

a、 在**/boot/grub/grub.conf** 中自己添加一条新的启动选项，并使该选项指向**vimlinuz-2.4.7-10custom**

b、重新安装 **grub**

5. 重新引导从新的内核进入

6. 修改/usr/lib/bcc/include/unistd.h，在系统调用列表后面相应位置添加一行

#define _NR_mycall 222

7．编写程序测试 **test.c：**

```
#include <linux/unistd.h>
#include <linux/time.h>
_syscall1(int,mycall,struct timeval *,thetime)
main()

{     struct timeval gettime;
      struct timeval mycalltime;

      gettimeofday(&gettime,NULL);
      mycall(&mycalltime);
      printf("gettimeofday:%d%d\n",gettime.tv_sec,gettime.tv_usec);
      printf("mycall:%d%d\n",mycalltime.tv_sec,mycalltime.tv_usec);
}
```

4 进程管理实验

4.1 实验3.1 进程行为观察

1、实验目的

通过本实验了解并掌握 Linux 主要的进程管理命令。

2、实验内容

1. 在 Linux 下，分别用 snice、skill、top 等命令和/proc 中的有关目录、文件观察系统中进程运行情况和 CPU 工作情况。
2. 在 Linux 下，用 ptrace()、gdb 跟踪一个进程的运行情况，用 strace 工具跟踪 fork() 过程，用 ltrace 工具跟踪 execl() 过程。观察并分析跟踪信息。

4.2 实验3.2 Shell编程（2选1）

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1

设计一个简单的 shell 解释程序，能实现基本的 bsh 功能。

（1）设计思想：

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。**shell** 程序源代码清单：

（2）源代码清单

源代码 **xsh.c:**

```
#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;
        }
    }
}
```

（3）测试结果：

```
[root@localhost Work]# make xsh.c
make: Nothing to be done for `xsh.c'.
[root@localhost Work]# make xsh
make: `xsh' is up to date.
[root@localhost Work]# ./xsh
```

```
=> ps
  PID TTY          TIME CMD
29297 pts/2    00:00:00 bash
29344 pts/2    00:00:00 xsh
29345 pts/2    00:00:00 ps
=> ps | more
ps: error: Garbage option.
usage: ps -[Unix98 options]
       ps [BSD-style options]
       ps --[GNU-style long options]
       ps --help for a command summary
=> ps > ps.txt
ps: error: Garbage option.
usage: ps -[Unix98 options]
       ps [BSD-style options]
       ps --[GNU-style long options]
       ps --help for a command summary
=> ls
fibno_timer.c  ksamp.c  psTimeInfo.c  shell1.c  shell2.c  timer.c  xshbk  xsh.c
xshrb.c
ksamp          ls.txt   shell1        shell1.c~ shell3.c  xsh      xshbk.c  xshrb
=> chsh
Changing shell for root.
New shell [/bin/bash]:
```

3、实验内容2

编写一个带有重定向和管道功能的 Shell

（1）设计思路

通过 fork（）创建子进程，用 execvp（）更改子进程代码，用 wait（）等待子进程结束。这三个系统调用可以很好地创建多进程。另一方面，编写的 Shell 要实现管道功能，需要用 pipe 创建管道使子进程进行通信。

说明：

- 本程序允许同时有重定向和管道，也可以不带它们，但是管道数目不得超过一个
- 重定向的位置应该是在最后
- 设有一个内部命令 exit 用来退出 Shell

（2）源代码清单（参考）

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
```

```
main()
{
	int hd;
	char buf[256];
	char *buf2,*buf3,*cmd,*cmd2,*cmd3,*argv[64],*argv2[64],*argv3[64];
	int n,sv,fd[2];
	for(;;) {
		printf("=> ");
		if(fgets(buf,sizeof(buf),stdin)==NULL) exit(0);
		buf2=strstr(buf,"|");
		buf3=strstr(buf, ">");
		if (buf2)
			*buf2++='\0';
		if(buf3)
			*buf3++='\0';
		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"));}
		else exit(1);
		if (buf2){
			cmd2=strtok(buf2," \t\n");
			n=0;
			argv2[n++]=cmd2;
			while(argv2[n++]=strtok(NULL," \t\n"));}
		if (buf3){
			cmd3=strtok(buf3," \t\n");
			n=0;
			argv3[n++]=cmd3;
			while(argv3[n++]=strtok(NULL," \t\n"));}
		if (!cmd2){
			if (fork() == 0) {
			execvp(cmd,argv);
			fprintf(stderr," ******ERROR******:    %s\n",strerror(errno));
			exit(1);}
			wait(&sv);
		       }
		else {
			pipe(fd);
			if(fork()==0) {
				hd=-1;
				dup2(fd[0],0);
```

```
                    if (cmd3) hd=open(cmd3, O_CREAT|O_WRONLY,0666);
                    if (hd != -1 ) dup2(hd,1);
                    close(fd[0]);
                    close(fd[1]);
                    close(hd);
                    execvp(cmd2,argv2);
                    fprintf(stderr,"    ******ERROR******:      %s\n",strerror(errno)); exit(1);
                   } else if(fork()==0) {
                    dup2(fd[1],1);
                    close(fd[0]);
                    close(fd[1]);
                    execvp(cmd,argv);
                    fprintf(stderr,"    ******ERROR******:      %s\n",strerror(errno)); exit(1);
                   }
                close(fd[0]); close(fd[1]);
                wait(&sv);    wait(&sv);

            }
        }
}
```

5. 存储管理实验

5.1 实验4.1 存储管理代码分析

1、实验目的

了解 Linux 内核中存储管理部分的代码结构和主要功能。

2、实验内容

阅读 Linux/Minix 中以下模块的调用主线，并写出分析报告

- ~~exec 系统调用的内部实现模块调用主线~~
- ~~malloc 函数的内部实现模块调用主线~~

- 缺页中断处理程序

3、示例—缺页中断处理程序分析

当程序在运行中，访问到的无效的虚拟地址的时候，系统将激发出缺页中断。激发缺页中断的情况通常有四种：

1. C－O－W 型中断，当某个进程进行写操作时，如果写的页是多进程在使用时，为了不影响其它进程的正常运行，通常需要将该页复制一份，供执行写操作的进程单独使用。
2. 被访问的物理页由于太长时间没有访问而被 kswapd 置换到 swap file 中。
3. 被访问的物理页由于是第一次访问，所以还在磁盘上或由于访问后被置换时因为没有发生写操作，而未写到 swap file 中。
4. 当进程动态的访问一片存储区域时，如在程序中动态开辟的数组等，则该页不在 swap file 中，也不在磁盘上。

缺页中断服务入口程序是函数 do_page_fault。do_page_fault 首先进行各种错误情况判断，并作相应处理。然后根据 error_code 来判断缺页中断类型:

（1） 第一种情况采用 do_wp_page 函数来处理

（2） 第二、三、四种情况由 do_no_page 函数来处理。

下面将分别介绍这些函数的流程图。

void do_wp_page(struct task_struct * tsk, struct vm_area_struct * vma,
unsigned long address, int write_access)

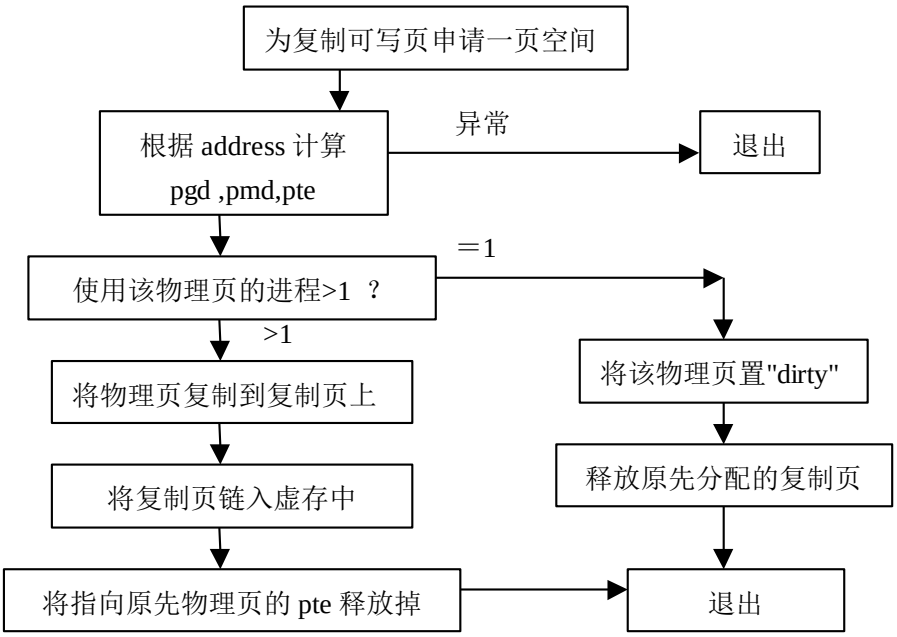


图 1　do_wp_page 示意图

void do_no_page(struct task_struct * tsk, struct vm_area_struct * vma,
unsigned long address, int write_access)

{

根据 address 计算三级页表中 pgd/pmd/pte 的值，如无对应的项，分配空间将其填上

if (pte_present(entry))

该页已在内存中，return; //可能是共享该页的其他进程已将该页读入

else if (!pte_none(entry)) //情况 2

pte 中该项非空，说明该页被 kswapd 置换到 swap file 中，

调用 do_swap_page 读入该页；

else if (!vma->vm_ops || !vma->vm_ops->nopage) //情况 4

该页对应的 VMA 块没有相应的 nopage 操作，说明该 VMA 块对应的是数据段内容

调用__get_free_page 操作为该页分配内存，并将该页赋值为 0,同时链入该进程的页表中

else //情况 3

调用虚拟块操作将磁盘中对应文件读入 page 中，

将 pte 表中对应的项指向该页。

将该页 dirty 位置位，如果该页有多进程使用，且非共享，置写保护位。

}

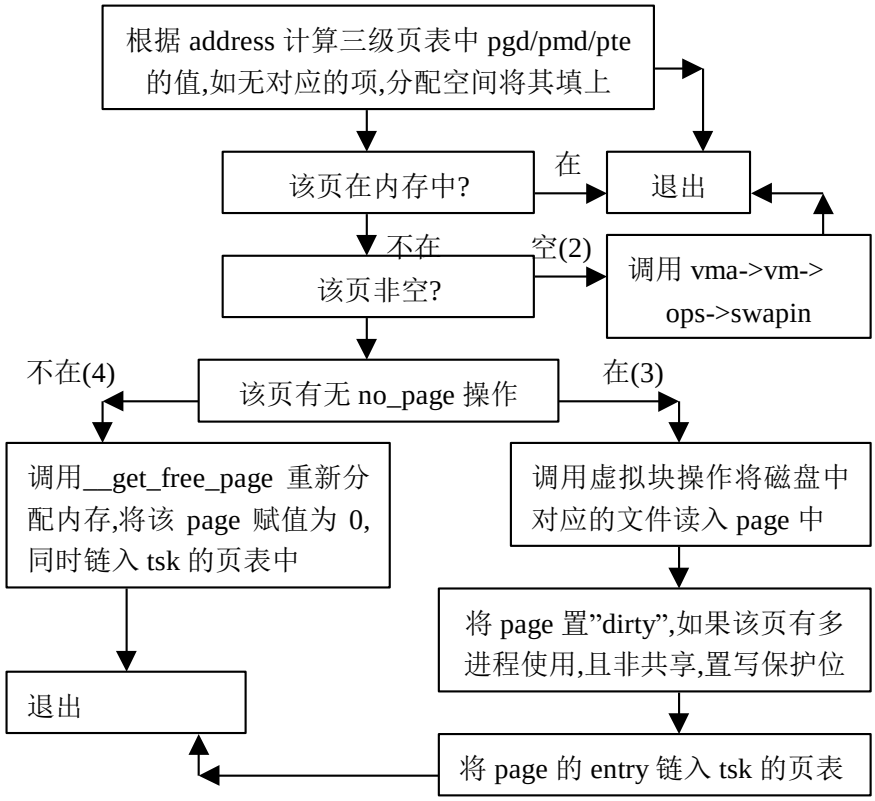


图 2 do_no_page 示意图(2、3、4 分别代表 2、3、4 种情况)

参考文献：《莱昂氏源代码分析》

5.3 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

- 预置缺页频率参数
- 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、试验步骤

（1）提供系统变量 pfcount 并编译内核

声明变量 pfcount

文件 linux/include/linux/mm.h

增加声明

extern unsigned long volatile pfcount;

定义变量

文件 linux/arch/i386/mm/fault.c

定义变量

unsigned long volatile pfcount;

变量操作

文件 linux/arch/i386/mm/fault.c

增加自增操作

在 do_page_fault 系统调用的全局范围内增加操作

pfcount++;

同时在/kernel/ksyms.c 中 export 变量
增加
EXPORT_SYMBOL(pfcount);
EXPORT_SYMBOL(jiffies);

之后编译内核

（2）提供一个读取模块文件 pf.c,在 proc 中添加访问接口
至于如何添加和编译模块就不再仔细说明，在第四个试验中我们就通过考擦 proc 伪文件系统学习 linux 独特的模块机制。用命令
gcc –D__KERNEL__ -c pf.c –o pf.o –I/user/src/linux/include 生成文件 pf.o，院后执行 insmod pf.o 加载模块。这样通过/proc 文件系统接口我们就可以轻松的读取我们所需要的两个变量数据了。使用命令
cat /proc/pf/pfcount /proc/pf/jiffies 就可以在终端中打印出系统至今为止的缺页次数和进来的 jiffies 数目了。

5、源程序代码清单（参考）

```
#define MODULE
#include <linux/proc_fs.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/sched.h>
#include <linux/string.h>
#include <linux/types.h>
#include <linux/ctype.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/module.h>

struct proc_dir_entry *proc_pf;
struct proc_dir_entry *proc_pfcount, *proc_jiffies;

extern unsigned long jiffies, pfcount;

static inline struct proc_dir_entry *proc_pf_create(
                    const char* name,
                   mode_t mode,
                    get_info_t *get_info)
                          return create_proc_info_entry(
                                              name,
                                              mode,
                                              proc_pf,
                                             get_info);
```

```
                }
int get_pfcount(char *buffer, char **start, off_f offset, int length)
{
        int length =0;
        length = sprintf(buffer, "%d\n",pfcount);
        return length;
}


int get_jiffies(char *buffer, char **start, off_f offset, int length)
{
        int length =0;
        length = sprintf(buffer, "%d\n",jiffies);
        return length;
}

int init_module(void)
{
        proc_pf =proc_mkdir("pf",0);
        proc_pf_create("pfcount", 0, get_pfcount);
        proc_pf_create("jiffies", 0, get_jiffies);
        return 0;

}

void cleanup_module(void)
{
        remove_proc_entry("pfcount",proc_pf);
        remove_proc_entry("jiffies", proc_pf);
        remove_proc_entry("pf", 0);
```

6. 进程通信

6.1 实验5.1 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.

~~6.2 实验5.2 代码分析（略）~~

1、实验目的

通过分析 Linux 中相关模块调用主线的代码结构，了解模块调用机制。

2、实验内容

阅读 Linux/Minix 中以下模块的调用主线，并写出分析报告

- kill 系统调用内部实现模块调用主线
- pipe 系统调用内部实现模块调用主线

3、实验结果示例

进程的创建：

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
    pid_t pid;
    pid = fork ();
    if (pid == 0)
    {
        printf ("%s", "This is child 1\n");
    } else {
        pid = fork ();
        if (pid == 0)
        {
            printf ("%s", "This is child2\n");
        } else {
            printf ("%s", "This is parent\n");
        }
    }
```

```
    return 0;
}
```

kill 命令调用主线：

先向 Linux 系统的内核发送一个系统操作信号和某个程序的进程标识号，然后系统内核就可以对进程标识号指定的进程进行操作。比如在 top 命令中，我们看到系统运行许多进程，有时就需要使用 kill 中止某些进程来提高系统资源。在讲解安装和登陆命令时，系统多个虚拟控制台的作用是当一个程序出错造成系统死锁时，可以切换到其它虚拟控制台工作关闭这个程序。

管道间通信：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
int main ()
{
    pid_t pid;
    int pfd[2];
    char send_buf[100], recv_buf[100];
    pipe (pfd);
    pid = fork ();
    if (pid == 0)
    {
        close (pfd[0]);
        strncpy (send_buf, "Child 1 is sending a message\n", sizeof (send_buf));
        lockf (pfd[1], F_LOCK, 0);
        write (pfd[1], send_buf, strlen(send_buf));
        sleep (1);
        lockf (pfd[1], F_ULOCK, 0);
        close (pfd[1]);
    }
    else
    {
        pid = fork ();
        if (pid == 0)
        {
            close (pfd[0]);
            strncpy (send_buf, "Child 2 is sending a message\n", sizeof (send_buf));
            lockf (pfd[1], F_LOCK, 0);
            write (pfd[1], send_buf, strlen(send_buf));
            sleep (1);
```

```
                lockf (pfd[1], F_ULOCK, 0);
                close (pfd[1]);
            }
            else
            {
                read (pfd[0], recv_buf, sizeof(recv_buf));
                printf ("%s", recv_buf);
                read (pfd[0], recv_buf, sizeof(recv_buf));
                printf ("%s", recv_buf);
                wait (0);
                wait (0);
                close (pfd[0]);
            }
        }
        return 0;
}
```

pipe 的调用主线：

pipe (pfd);pid = fork ();

如果 pid == 0，则调用：

close (pfd[0]);strncpy ();lockf ();write ();sleep ();lockf ();函数，通过进程向管道中写入数据，

如果调用不成功，调用了 close (pfd[0]);strncpy ();lockf ();write ();sleep ();lockf ();函数，创建一个进程，如果成功，通过该进程向管道中写入数据；如果不成功，则从管道中读出数据。

6.2 实验5.2 进程同步实验

1、实验目的

深入学习 **Linux** 内核，在学习 **linux** 内核的同步机制的同时，深入分析各种同步机制的实现方案，在此基础上设计和编写一套同步原语。

2、实验内容

设计并实现一个新的同步原语，该原语允许多个进程因一个事件而阻塞，直到其他进程产生这个信号为止。当一个进程产生一个事件的信号时，所有因这个事件而阻塞的进程都取消阻塞。如果信号产生时，没有进程因为这个信号而阻塞，那么这个信号无效。

实现下列系统调用：

```
int evntopen(int);
int evntclose(int);
int evntwait(int evenNum);
void evntsig(int evenNum);
```

3、实验原理

在深入学习软中断信号，信号量和管道的工作原理和实现机制后，我们知道，一个事件必须有一个事件号，一系列的进程等待这个事件发生，那么肯定需要一个等待队列，所以睡眠的进程就放到这个队列中去。通过考察 **linux** 中如 **wake_up、sleep_on** 等的实现我们将构建上述同步原语。

首先建立一个存放事件的队列，通过简单的链表即可实现。

```
//定义结构，保存事件队列
    typedef struct __eventStruct{
    int eventNum;
    wait_queue_head_t *p;
    struct __eventStruct *next;
}     __eventNode;
//两个全局变量记录链表的头尾
__eventNode *lpmyevent_head = NULL; //指向链表头
__eventNode *lpmyevent_end = NULL;   //指向链表尾
```

4、实验步骤及部分代码清单（参考）

（1）实现函数定义

定义 **Open** 同步原语，当 **Open** 一个事件的时候，有两种情况：一是事件已经存在，只需要返回 **Event** 事件号即可，第二种是事件链表中没有该事件，简单的处理办法直接返回-1，表示事件不存在。当传递参数 0 的时候，将产生一个新事件。

```
int   eventopen(int eventNum)
{
    __eventNode *newNode;
    __eventNode *prevNode;
    if(eventNum)
        if(!scheventNum,&prev))
                return -1;
        else return eventNum;
    else
    {
        newNode = (__eventNode *)kmalloc(
                sizeof(__eventNode),
                GFP_KERNEL);
        newNode->p =(wait_queue_head_t *)kmalloc(
                    sizeof(wait_queue_head_t));
        newNode->next = NULL;
        newNode->p->task_list.next =
                                    &newNode->p->task_list;
        newNode->p->task_list_prev =
                                    &newNode->p->task_list;
```

```
            if(!lpmyevent_head)
            {
                newNode->eventNum =2;
                lpmyevent_head =lpmyevent_end =newNode;
                return newNode->eventNum;
            }
            else
            {
           newNode->eventNum = lpmyevent_end->eventNum + 2;
                lpmyevent_end->next = newNode;
                lpmyevent_end = newNode;
            }
            return newNode->eventNum;
        }
        return 0;
    }
```

定义 **evntsig(eventNum),evntsig()**要完成的功能是唤醒所有等待该事件发生的进程。我们只需要在事件列表上找到该事件，然后调用 **wake_up()**。

```
int   evntsig(int eventNum)
{
    __eventNode *tmp = NULL;
    __eventNode *prev = NULL;

    if(!(tmp = scheventNum(eventNum,&prev))
        return;
        wake_up(tmp->p);
        return 1;
}
```

evntwait()将一个进程加到他要等待的事件队列中去。

```
int     evntwait(int eventNum)
{
    __eventNode *tmp;
    __eventNode *prev = NULL;
    wait_queue_t     wait;
    unsigned long flags;

    if(tmp =scheventNum(eventNum,&prev))
    {
        wait.task = current;
        current->state = TASK_UNINTERRUPTIBLE;
        write_lock_irqsave(&temp->p->lock,flags);
        __add_wait_queue(tmp->p,&wait);
        write_unlock(&tmp->p-lock);
```

```
            schedule();

            write_lock_irq(&tmp->p->lock);
            __remove_wait_queue(tmp->p,&wait);
            write_unlock_irqrestore(&tmp->p->lock,flags);
        }
    }
```

至于 **evntclose()**先在事件队列中找到该事件，然后根据事件在该链表中的位置进行特定的处理，之后唤醒所有睡眠在该事件上的事件，最后将这个事件从链表中删除，释放内存。

```
int evntclose(int eventNum)
{
    __eventNode *prev;
    __eventNode *releaseItem;

    evntsig(eventNum);
    if(releaseItem = scheventNum(eventNum,&prev))
    {
        if (releaseItem == lpmyevent_end)
                lpmyevent_end = prev;
        if (releaseItem == lpmyevent_head)
        {
            lpmyevent_head = lpmyevent_head->next;
            goto wake;
        }
        prev->next = releadeItem->next;
    }
    wake:
        if(releaseItem)
        {
            kfree(releaseItem);
            return 1;
        }
    return 0;
}
```

最后是一个辅助调度函数 **scheventNum()**函数。他有两个参数 e**ventNum** 和 **prev**，前者是事件号，后者是要返回的一个 **__eventNode** 类型的指向指针的指针，它将指向要要寻找节点的上一节点。函数返回要寻找的事件节点。

```
__eventNode * scheventNum(int eventNum,__eventNode **prev)
{
    __eventNode *tmp = lpmyevent_head;
    *prev = NULL;
    while(tmp)
    {
```

```
            if(tmp->eventNum == eventNum)
                    return tmp;
            *prev = tmp;
            tmp =tmp->next;
        }
        return NULLL;
}
```

（2）同步原语实现的函数转化为系统调用，修改系统初始化代码，编译内核。

7　I/O 设备管理实验

7.1 实验6.1. 观察实验

1、实验目的

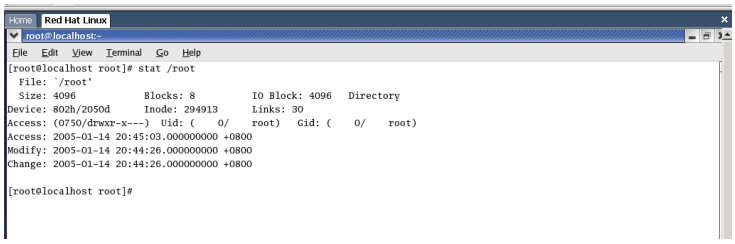
掌握与设备管理有关的 Linux 命令。

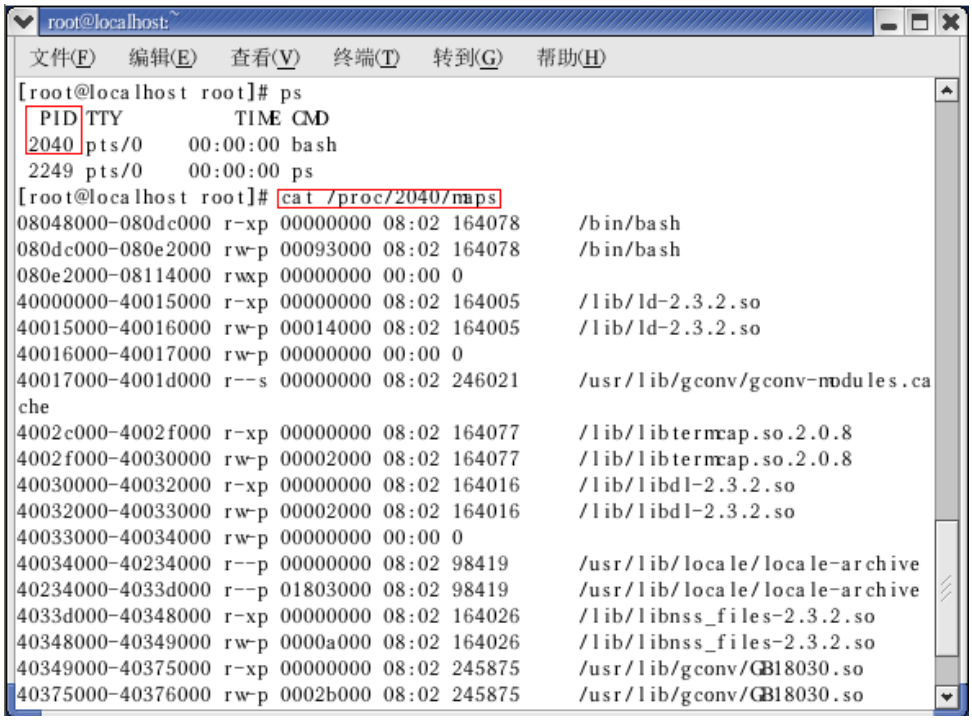
2、实验内容

- 用 stat 命令查看机器上硬盘特别文件的 I 节点内容
- 在 Linux 下，查看/proc 与内存管理相关的文件，解释显示结果

3、实验结果示例

参见以下截图





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。

3、程序源代码清单（参考）

```
main()
{    int p;
     p = fork();
     if(p > 0)
     {
```

```
            exit(0);
        }
        else if(p == 0)
        {
            for(i = 0; i < 100; i++)
            {
                sleep(100);
                system("ps > f1.txt");
            }
        }
        else
        {
            perror("Create new process!");
        }
}
```

8. 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、分析报告示例

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。

为了减少在程序或系统崩溃时遗留下来的临时文件所带来的问题，程序员应当遵守下列约定：

(1) 在打开临时文件后立即对其执行“ unlink”操作。

(2) 应在“tmp”目录下创建临时文件。在文件名中包括进程标识数就可构成一惟一文件名

参考文献：《莱昂氏源代码分析》

8.2 实验7.2 编程实验1（与7.3二选一）

1、实验目的与内容

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。

2、程序源代码清单（参考）

```
#include <sys/stat.h>
#include <stdio.h>
#include <dirent.h>

int func(char*path,int* AllFileNumber,int* AllDepthNumber,
                    int*    AllFilenameNumber,int* AllFoldNumber)
{
  struct stat statbuf;
  struct dirent *dirp;
  DIR *dp;
  char *str1;
```

```
  char *str2;
  int judge1;
  int judge2;
  int ret

  judge1=0;
  judge2=0;
  stat(path.&statbuf);              //将路径结点的信息放到 statbuf 结构体中

  //检查节点文件的类型
  switch(statbuf.st_mode&S_IFMF){
case S_IFREG:{                              //当为普通文件时
                    *AllfilenameNumber=sizeof(pathname)+1;
                    *AllFileNumber=*AllFileNumber+1;
                    *AllDepthNumber=*AllDepthNumber+1;
                    break;}

caseS_IFDIR:{*AllDepthNumber=*AllDepthNumber+1; //当为目录文件时，打开目录
                *AllFoldNumber=*AllFoldNumber+1;
                dp=opendir(path);
                while((dirp=readdir(dp))!=NULL){
                if((strcmp(dirp->d_name,".")==0)||(strcmp(dirp->d_name,"..")==0))
                    continue;                       //忽略文件 . 和 ..
                    if(judge1==0){
                    strcpy(str1,path);      //复制路径到字符串 str1 上
                    judge1=1;}
                    if (judge2==0){
                    str2=str1+strlen(str1);
                    judge2=1;}
                    *str2++='/';                    //在 str1 的后面加上'/'
                    *str2=0;
                    strcpy(str2,dirp->d_name); //在 str1 的后面加上某个文件名
                    ret=func(str1, AllFileNumber,AllDepthNumber,
                                    AllFilenameNumber,AllFoldNumber)
                    *str2--;
                    *str2=0;
                    closedir(dp);                   //关闭目录
                    break;}
                 default:          break;
   }
   return 1;
}

int main(int argc,char *argv[])
```

```
{
    int    ret;
    int    AllFileNumber=0;
    int    AllDepthNumber=0;
    int    AllFilenameNumber=0;
    int    AllFoldNuber=0;
    ret=func(argv[1],&AllFileNumber,&AllDepthNumber,&AllFilenameNumber,
              &AllFoldNumber);
    printf("The average filenumber is %d\n",AllFileNumber/AllFoldNumber);
    printf("The average fileDepthnumber is %d\n",
            AllDepthNumber/AllFoldNumber);
    Printf("The average filenameNumber is %d\n",
              AllFilenameNumber/AllFileNumber)
    Exit(0);
}
```

8.3 实验7.3 编程实验2（与7.2二选一）

1、实验目的与内容

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、程序源代码清单（参考）

下面是源程序：

```
#include <sys/types.h>
#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.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);
}
```

其中，系统调用 stat/fstat 用于从一个 i 节点获得文件的状态信息，stat 用给出的路径名从磁盘寻找指定文件的 i 节点，fstat 用给出的已打开文件句柄从核心活动 i 节点表中寻找，stat 和 fstat 将数据放入调用者提供的 stat 结构中。

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(path,sbuf)	/* get file status */
	char *path;		/* path name */
	struct stat *sbuf; /* status information */
/* return O on success or -1 on error */

int fstat(fd,sbuf)	/* get file sttus */
	int fd;			/* file descriptor    */
	struct stat *sbnf;
/* returns O on success or -1 on error */
```

9. 多核多线程编程

9.1 实验目的

参照参考文献“利用多核多线程进行程序优化”，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。

利用多核多线程进行程序优化. pdf

9.2 实验内容

实验1. 观察实验平台物理cpu、CPU核和逻辑cpu的数目

目的：观察实验所采用的计算机（微机、笔记本电脑）物理 cpu、CPU 核和逻辑 cpu 的数目

测试环境(示例)：
硬件：CPU(e. g. 戴尔, i5 Dual-core 双核)，主频2.4G，内存4G
软件：Suse Linunx Enterprise 10，内核版本：linux-2.6.16

- 物理 cpu 数目：主板上实际插入的 cpu 数量，可以数不重复的 physical id 有几个（physical id）。

  多路服务器、大型主机系统、集群系统一般可以配置多个物理 CPU；常规微机、笔记本电脑一般只配备 1 个物理 CPU；
- cpu 核（cpu cores）的数目：单块 CPU 上面能处理数据的芯片组的数量，如双核、四核等；
- 逻辑 cpu 数目：

  对不支持超线程 HT 的 CPU，逻辑 cpu 数目=物理 CPU 个数×每颗 CPU 核数，

  对支持超线程 HT 的 CPU，逻辑 cpu 数目=物理 CPU 个数×每颗 CPU 核数*2

方式 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

方式 2：Windows 环境下，通过下述程序获取 cpu 逻辑核的数量
(http://stackoverflow.com/questions/150355/programmatically-find-the-number-of-cores-on-a-machine)

```
#ifdef _WIN32
#include <windows.h>
#elif MACOS
#include <sys/param.h>
#include <sys/sysctl.h>
#else #include <unistd.h>
#endif
int getNumCores() {
#ifdef WIN32
       SYSTEM_INFO sysinfo;
       GetSystemInfo(&sysinfo);
       return sysinfo.dwNumberOfProcessors;
#elif MACOS
       int nm[2];
       size_t len = 4;
       uint32_t count;

       nm[0] = CTL_HW;
       nm[1] = HW_AVAILCPU;
       sysctl(nm, 2, &count, &len, NULL, 0);
       if(count < 1) {
              nm[1] = HW_NCPU;
              sysctl(nm, 2, &count, &len, NULL, 0);
              if(count < 1) { count = 1; }
        }
        return count;
#else   return sysconf(_SC_NPROCESSORS_ONLN);
#endif
```

}

实验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

程序功能：

求从1一直到 APPLE_MAX_VALUE (100000000) 相加累计的和，并赋值给 apple 的 a 和 b ；求 orange 数据结构中的 a[i]+b[i] 的和，循环 ORANGE_MAX_VALUE(1000000) 次。

步骤1. 单线程/进程样例程序

```
#define ORANGE_MAX_VALUE      1000000

#define APPLE_MAX_VALUE       100000000

#define MSECOND              1000000


struct apple

{

    unsigned long long a;

         unsigned long long b;

};


struct orange

{

         int a[ORANGE_MAX_VALUE];

         int b[ORANGE_MAX_VALUE];
```

```
};


int main (int argc, const char * argv[]) {
   // insert code here...
    struct apple test;
         struct orange test1;


         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
         {
                  test.a += sum;
                  test.b += sum;
         }
```

```
    sum=0;
          for(index=0;index<ORANGE_MAX_VALUE;index++)
          {
                    sum += test1.a[index]+test1.b[index];
          }


    return 0;
}
```

步骤2. 2线程并行程序

采用任务分解的方法 ，将互不相关的计算 apple 值和计算 orange 值的2部分代码分解为2个线程，实现线程级并行执行。

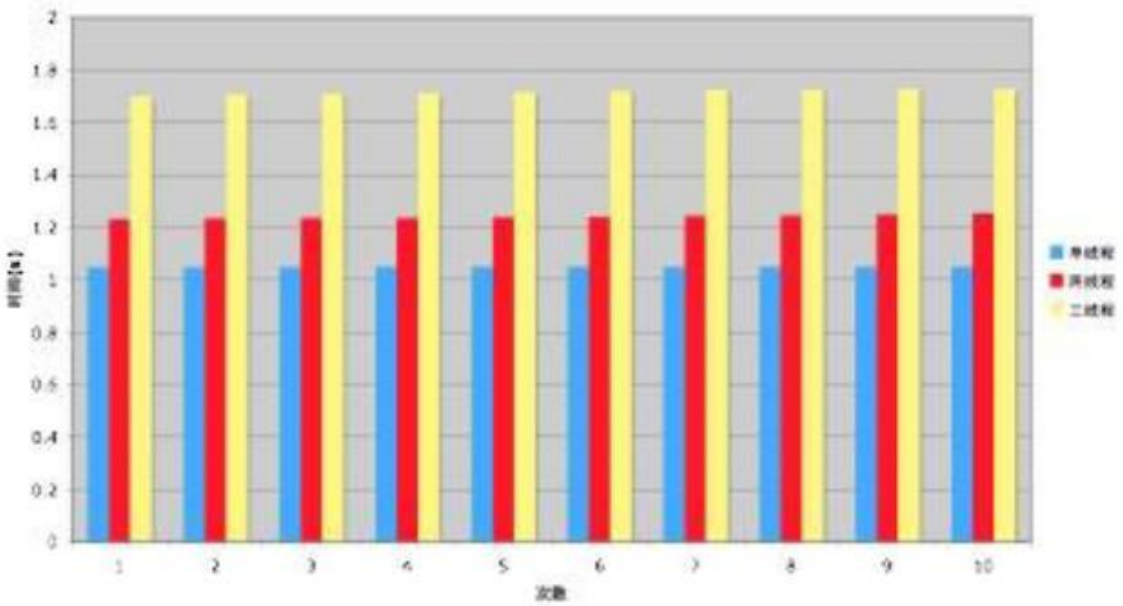
步骤3. 3线程加锁并行程序

通过数据分解的方法，还可以发现，计算 apple 的值可以分解为两个线程，一个用于计算 apple a 的值，另外一个线程用于计算 apple b 的值。

但两个线程存在同时访问 apple 的可能性，需要加锁访问该数据结构。

步骤4. 参照参考文献，采用K-Best 测量方法，对比分析单线程/进程、2线程、3线程加锁程序的运行时间差异，以表格或图形方式给出对比结果,并分析导致差异的原因，判断多线程并行编程是否达到预期。

e. g.



实验3. 3线程加锁 vs 3线程不加锁 对比

在前述3线程加锁方案，计算Apple的两个并行线程访问的是 apple 的不同元素，没有加锁的必要。所以修改 apple 的数据结构，删除读写锁代码，通过不加锁来提高性能。

比较分析3线程程序，在加锁与不加锁耗时的运行时间差异，以表格或图形方式给出对比结果,并分析导致差异的原因，判断加锁机制是否达到预期。。

实验4. 针对 Cache 的优化

在串行程序设计过程中，为了节约带宽或者存储空间，比较直接的方法，就是对数据结构做一些针对性的设计，将数据压缩（pack）的更紧凑，减少数据的移动，以此来提高程序的性能。但在多核多线程程序中，这种方法往往有时会适得其反。

数据不仅在执行核和存储器之间移动，还会在执行核之间传输。根据数据相关性，其中有两种读写模式会涉及到数据的移动：写后读和写后写 ，因为这两种模式会引发数据的竞争，表面上是并行执行，但实际只能串行执行，进而影响到性能。

处理器交换的最小单元是 cache 行，或称 cache 块。在多核体系中，对于不共享 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，导致执行核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。上文曾提到，不加锁的方案反而比加锁的方案更慢，就是互相竞争 cache 的原因。

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。

实验内容：

步骤1. 不加锁三线程程序的瓶颈在于 cache，针对不加锁的3线程程序，修改其部分代码如下：

清单 **4.** 针对**Cache**的优化

```
struct apple
{
unsigned long long a;
char c[128]; /*32,64,128*/
unsigned long long b;
};
```

，让 apple 的两个成员 a 和 b 位于不同的 cache 行中，观察增加Cache后的运行时间，分析是否达到预期要求。

步骤2. 针对加锁三线程程序，对apple 数据结构也增加一行类似功能的代码，判断效率是否提升？

预期结论：性能不会有所提升，其原因是加锁的三线程方案效率低下的原因不是 Cache 失效造成的，而是那把锁。

实验5. CPU亲和力对并行程序影响

CPU 亲和力可分为两大类：软亲和力和硬亲和力。

管理处理器的亲和性（affinity）.pd

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 硬亲和力是指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。

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。

实验内容：

步骤1.

在双核机器上，针对两线程的方案，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，分析对比与单线程/进程程序、2线程（无CPU亲和）程序的运行时间。

程序代码见参考文献“利用多核多线程进行程序优化”。

步骤2.

进一步分析不难发现，样例程序大部分时间都消耗在计算 apple 上。为此，将计算 a 和 b 的值，分布到不同的 CPU 上进行计算，同时考虑 Cache 的影响。观察程序运行时间，并与采用 Cache 的三线程方案进行对比。

8种实现方案运行时间对比总结

根据前面实验内容，总结对比单线程/进程串行程序和其它 7 种多线程并行程序的运行时间，并以图表方式给出具体结果。

E.g.